Programming Paradigms

Concurrency (Part 1)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart
Summer 2022
Overview

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

**Why do we care about concurrency?**

- To capture the *logical structure of a problem*
  - Inherently concurrent problems, e.g., server handling multiple requests
- To exploit *parallel hardware* for speed
  - Since around 2005: Multi-core processors are the norm
- To cope with *physical distribution*
  - Local or global groups of interacting machines
Terminology

- **Concurrent**
  - Two or more running tasks whose execution may be at some unpredictable point

- **Parallel**
  - Two or more tasks are actively executing at the same time
  - Requires multiple processor cores

- **Distributed**
  - Physically separated processors
Levels of Parallelism

- Signals propagating through circuits and gates
- Instruction-level parallelism
  - E.g., load from memory while another instruction executes
- Vector parallelism
  - E.g., GPUs execute a single instruction on a vector of data
- Thread-level parallelism
Levels of Parallelism

- Signals propagating through circuits and gates
- Instruction-level parallelism
  - E.g., load from memory while another instruction executes (Handled implicitly by hardware)
- Vector parallelism
  - E.g., GPUs execute a single instruction on a vector of data
- Thread-level parallelism
Levels of Parallelism

- **Signals propagating through circuits and gates**
- **Instruction-level parallelism**
  - E.g., load from memory while another instruction executes
- **Vector parallelism**
  - E.g., GPUs execute a single instruction on a vector of data
- **Thread-level parallelism**
Example: Independent Tasks

// Task Parallel Library in C#
Parallel.For(0, 100, i => {
    A[i] = foo(A[i]);
});
Example: Independent Tasks

```csharp
// Task Parallel Library in C#
Parallel.For(0, 100, i => {
    A[i] = foo(A[i]);
});
```

Array of data Function that updates each element independently
Example: Independent Tasks

```csharp
// Task Parallel Library in C#
Parallel.For(0, 100, i => {
    A[i] = foo(A[i]);
});
```

- Array of data
- Function that updates each element independently
- No need to synchronize tasks
- Uses as many cores as possible (up to 100)
Example: Dependent Tasks

// As before, but foo now is:
int zero_count;
public static int foo(int n) {
    int rtn = n - 1;
    if (rtn == 0) zero_count ++;
    return rtn;
}
Example: Dependent Tasks

// As before, but foo now is:
int zero_count;
public static int foo(int n) {
    int rtn = n - 1;
    if (rtn == 0) zero_count ++;
    return rtn;
}

Count how many zeros written to the array
Problem: Data Race

---

Thread 1

\[ r1 := \text{zero\_count} \]
\[ r1 := r1 + 1 \]
\[ \text{zero\_count} := r1 \]

---

Thread 2

\[ r1 := \text{zero\_count} \]
\[ r1 := r1 + 1 \]
\[ \text{zero\_count} := r1 \]

---

data races
Data Races

- Definition of data race
  - Two accesses to the same shared memory location
  - At least one access is a write
  - Ordering of accesses is non-deterministic
Another Example

// code to transfer money between accounts
// written in a toy language

fun transfer(amount, account_from, account_to) {
  if (account_from.balance < amount) return NOPE;
  atomic {
    account_to.balance += amount;
    account_from.balance -= amount;
  }
  return YEP;
}

Another Example

// code to transfer money between accounts
// written in a toy language
fun transfer(amount, account_from, account_to) {
  if (account_from.balance < amount) return NOPE;
  atomic {
    account_to.balance += amount;
    account_from.balance -= amount;
  }
  return YEP;
}

Ensures that code block is executed atomically, i.e., no other thread executes it at the same time
Another Example

// code to transfer money between accounts
// written in a toy language
fun transfer(amount, account_from, account_to) {
    if (account_from.balance < amount) return NOPE;
    atomic {
        account_to.balance += amount;
        account_from.balance -= amount;
    }
    return YEP;
}

Ensures that code block is executed atomically, i.e., no other thread executes it at the same time

Quiz: Could a program invoking transfer multiple times concurrently have a data race?
Another Example

// code to transfer money between accounts
// written in a toy language
fun transfer(amount, account_from, account_to) {
    if (account_from.balance < amount) return NOPE;
    atomic {
        account_to.balance += amount;
        account_from.balance -= amount;
    }
    return YEP;
}

Yes, there still is a data race:
Concurrent and unsynchronized read and write of account_from.balance
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>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>
</tr>
<tr>
<td><strong>Extension</strong></td>
<td>OpenMP</td>
<td>MPI</td>
</tr>
<tr>
<td><strong>Library</strong></td>
<td>pthreads, Windows threads</td>
<td>MPI</td>
</tr>
</tbody>
</table>
Quiz: Terminology

Which of the following sentences are true?

- Concurrency means different machines perform computations at the same time.
- A data race can occur only when two threads execute concurrently.
- Instruction-level parallelism should be avoided to ensure correctness.
- In PLs with message passing, messages typically exchange pointers to shared memory.
Quiz: Terminology

Which of the following sentences are true?

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

■ A data race can occur only when two threads execute concurrently.

■ Instruction level parallelism should be avoided to ensure correctness.

■ In PLs with message passing, messages typically exchange pointers to shared memory.
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"); }
}
```
Example: C with OpenMP

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

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

Pragmas: Compiler directives
(# sign must be in first column)
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

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

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 B and C
Example: Fortran 95

### Fortran 95 Code

```fortran
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

\begin{verbatim}
forall (i=1:n-1)
  A(i) = B(i) + C(i)
  A(i+1) = A(i) + A(i+1)
end forall
\end{verbatim}

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 [2, 2, 3]
- C is [3, 1, 2]
- n is 3

(Note: Arrays indices start at 1 in Fortran)
Thread 1
\[ i := 1 \]
\[ 2 + 3 \text{ gives } 5 \]
\[ A(1) = 5 \]
\[ 5 + 3 \text{ gives } 8 \]
\[ A(2) = 8 \]

\[ \rightarrow A \text{ is } [5, 8, 3] \]

Thread 2
\[ i := 2 \]
\[ 2 + 1 \text{ gives } 3 \]
\[ A(2) = 3 \]
\[ 3 + 0 \text{ gives } 3 \]
\[ A(3) = 3 \]

\[ \text{All threads wait for each other} \]
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
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

```c
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
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
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