Concepts of Concurrent Computation
Spring 2015
Lecture 13: Concurrent Languages

Sebastian Nanz
Chris Poskitt
Classification
Concurrent and parallel languages

- Developers today have the choice among a multitude of different approaches to concurrent and parallel programming.
Message-passing approaches
Ada
Ada

- Object-oriented language, influenced by Pascal, developed from 1975 by US Department of Defence, standards: Ada83, Ada95, Ada 2005

- Design goals: highly reliable systems, reusable components, concurrency part of the language

- Named after Ada Lovelace (1815–1852), “the first computer programmer”

- Supports concurrent execution via tasks, which can have entries for synchronous message-passing communication

- Ada also offers shared memory synchronization via protected objects, a monitor-like mechanism where condition variables are replaced with guards
Ada tasks

- Tasks are declared within procedures
- Two parts: task specification, task implementation
- Tasks are activated when the procedure starts executing

```ada
procedure SimpleProc is
  task type SimpleTask;
  task body SimpleTask is
  begin
    ...
  end SimpleTask;

  taskA, taskB: SimpleTask;
  begin
    null;
  end SimpleProc;
```
Process communication: Rendezvous (1)

- Uses synchronous communication, called the “rendezvous”
- Entry points (declared in the type declaration) specify the actions a task can synchronize on

```plaintext
task type SimpleTask is
  entry MyEntry;
end SimpleTask;
```
Process communication: Rendezvous (2)

- **accept**-statements (within the task body) indicate program points where rendezvous can take place

- Clients invoke an entry point to initiate a rendezvous, and wait for the accepting task to reach a corresponding entry point

```plaintext
task body SimpleTask is
  begin
    ...
    accept MyEntry do
      -- body of rendezvous
      end MyEntry;
    ...
  end SimpleTask;

declare
  T: SimpleTask;
begin
  ...
  T.MyEntry;
  -- wait until T reaches MyEntry
  ...
end SimpleTask;
```

- Upon establishing a rendezvous, the client waits for the accepting task to execute the body of the rendezvous and resumes afterward
Process communication: Rendezvous (3)

- Entry points can have parameters to pass on values

```plaintext
accept append(x : in integer) do
  ...
end append;
```

- **select**-statement allows for waiting for multiple entries

- Within a **select**, alternatives may be guarded by boolean expressions

- Only if the guard evaluates to true the **accept** is permitted

```plaintext
select
  when count < n =>
    accept append(x : in integer) do
      ...
    end append;
  or
  when ...
end
```
task body Buffer is
  count, in, out: integer := 0;
  buff: array(0..n-1) of integer;
begin
  loop
    select
      when count < n =>
        accept append(x : in integer) do
          buff(in) := x;
        end append;
        in := (in + 1) mod n; count := count + 1;
      or
      when count > 0 =>
        accept remove(y : out integer) do
          y := buff(out);
        end remove;
        out := (out - 1) mod n; count := count - 1;
    end select;
  end loop;
end buffer;
Protected objects

- Monitor-like concept
  - All data private
  - Exports only procedures, functions, and entries
- Functions may **only read data**, therefore multiple function calls may be active on the same object
- Procedures and entries may **read and write data**, and exclude other procedures and functions
- Invocation of entries with **guards**, similar to Hoare’s conditional critical regions
Conditional critical regions

- Conditional critical regions provide condition synchronization without condition variables.

- If $S$ is a critical region for variable $x$, then the following is a conditional critical region with guard $B$:

  ```
  region x when B do S
  ```

- If a process wants to enter a conditional critical region, it must obtain the mutex lock; otherwise it is queued.

- When the lock is acquired, the boolean expression $B$ is tested. If $B$ evaluates to true, the process proceeds into the critical region. Otherwise it releases the lock and is queued. Upon re-acquisition of the lock, the process must retest $B$. 
Example: Protected objects

```vhdl
protected type Semaphore is
  entry Down;
  procedure Up;
  function Get_Count return Natural;
private
  Count: Natural := 0;
end Semaphore;

protected body Semaphore is
  entry Down when Count > 0 is
  begin
    Count := Count - 1;
  end Down;

  procedure Up is
  begin
    Count := Count + 1;
  end Up;

  function Get_Count return Natural is
  begin
    return Count;
  end Count;
end Semaphore;
```
Ada: Discussion

- One of the first languages to introduce high-level concurrency constructs into the language
- Both message-passing and shared-memory concepts available: good to fit the approach to the problem at hand and performance requirements
- Ada is still actively developed
The Actor model: Erlang
The Actor model

- A mathematical model of concurrent computation, introduced by (Hewitt, 1973) and refined by (Agha, 1985) and others

- Actor metaphor: “active agent which plays a role on cue according to a script”

- Process communication through asynchronous message passing

- No shared state between actors
Actor

- An actor is an entity which in response to a message it receives can
  - send finitely many messages to other actors
  - determine new behavior for messages it receives in the future
  - create a finite set of new actors
- Communication via asynchronous message passing
- Recipients of messages are identified by addresses
- A message consists of
  - the target to whom the communication is addressed
  - the content of the message
Erlang

- **Erlang**: functional language, developed by Ericsson since 1986
- Erlang implements the Actor model
Erlang syntax for concurrency

- When processes (≈ actors) are created using `spawn`, they are given unique process identifiers (PIDs)
  
  \[
  \text{PID} = \text{spawn(Module, Function, Arguments)}
  \]

- Messages are sent by passing tuples to a PID with the `!` syntax
  
  \[
  \text{PID}!\{\text{message}\}
  \]

- Messages are retrieved from the mailbox using the `receive` function with pattern matching
  
  ```erlang
  receive
      Message1  -> Actions1 ;
      Message2  -> Actions2 ;
      ...;
  end
  ```
Example: A simple counter

Interface

start() ->
   spawn(counter, counter_loop, [0]).

increment(Counter) ->
   Counter ! inc.

value(Counter) ->
   Counter ! {self(), value},
   receive
      {Counter,Value} -> Value
   end.

Counter

counter_loop(Val) ->
   receive
      inc ->
         counter_loop(Val + 1);
      {From, value} ->
         From ! {self(),Val},
         counter_loop(Val);
      Other ->
         counter_loop(Val)
   end.
Actors: Discussion

- Influential model for asynchronous message passing
- Also implemented in various other languages, e.g. Scala
- Success story: Ericsson AXD301 switch for telecommunication systems with very high reliability – more than one million lines of Erlang
Message Passing Interface (MPI)
Message Passing Interface (MPI)

- **Message Passing Interface (MPI):** API specification for process communication via messages, developed in 1993/94.
- For parallel programs on distributed memory systems.
“Hello, World!” in MPI

- Processes involved in an MPI execution are identified by ranks, i.e. integer numbers 0, 1, ..., numproc – 1
- In the following program, Process 0 gets and prints messages from all other processes

```c
MPI_Init(&argc,&argv);           // Initialize MPI
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  // My identifier
MPI_Comm_size(MPI_COMM_WORLD, &numproc);  // Total number of processes
if (my_rank != 0) {
    sprintf(message, "Greetings from process %d!", my_rank);
    dest = 0;
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
} else {
    for (source = 1; source < numproc; source++) {
        MPI_Recv(message, sizeof(message), MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
        printf("%s\n", message);
    }
}
MPI_Finalize();                     // Shut down MPI
```
SPMD in MPI

- As seen in the previous program, the most common paradigm used in MPI is **SPMD**
- Within each process, we take branches based on its rank
- At startup, processes are mapped to processors by the MPI runtime
MPI: Discussion

- Dominant model used in high-performance computing
- Good portability: implemented for many distributed memory architectures
- Available as library in many languages, in particular Fortran, C, C++
Polyphonic C#

(Based on slides by C.A. Furia)
Polyphonic C#

- Polyphonic C# is an extension of C# with a few high-level primitives for concurrency, appeared in 2004
  - Based on the Join calculus (Fournet & Gonthier, 1996)
  - Taken up by Microsoft’s Cω project
  - JoinJava is a similar extension for Java

- Based on two basic notions
  - Asynchronous methods
  - Chords

(M. Mussorgsky, Pictures at an exhibition)
Asynchronous methods

- Calls to asynchronous methods return immediately without returning any result
  - The callee is scheduled for execution in a different thread
  - Similar to sending a message or raising an event
  - Declared using `async` keyword (instead of `void`)

```csharp
public async startComputation () {
  // computation
}
```

- Aynchronous methods do not return any value
Chords: syntax

- A chord extends the notion of a method definition
  - The signature of a chord is a collection of (traditional) method declarations joined by &
  - The body of a chord is all similar to the body of a traditional method

```java
public String get() & public async put(String i) {
  return i;
}
```

- Within a chord at most one method can be non-async
- Within a class the same method can appear in more than one chord
Chords: semantics

- A chord is only executed once all the methods in its signature have been called
  - Calls are buffered until there is a matching chord
    - the implicit buffer supports complex synchronization patterns with little code (see Producers/Consumers later)
  - If multiple matches are possible, nondeterminism applies
  - Execution returns a value to the only non-asynchronous method in the chord (if any)
Chords semantics: example

```csharp
public class Buffer()
{
    public String get() & public async put(String i) {
        return i;
    }
}

Buffer b = new Buffer();
b.put(“A”)
Console.WriteLine(b.get()); // prints “A”

b.put(“A”); b.put(“B”);
Console.WriteLine(b.get() + b.get()); // prints “AB”

b.get(); // blocks until some other thread calls put
```
Polyphonic C#: Discussion

- Combination of two ideas: asynchronous methods and chords
- Asynchronous methods also appear in earlier languages such as Cilk
- **Chords**: novel idea for message passing communication among more than two threads
- Cω project is discontinued
Shared Memory Approaches
OpenMP

(Some slides adapted from Intel teaching material)
OpenMP

- OpenMP (Open Multi-Processing) API for shared memory multithreaded programming, appeared in 1997

- Using preprocessor directives (pragmas) to mark parallel code, may be ignored by other compilers

```c
#pragma omp construct [clause [clause]...]
```
Programming model

- Fork-join parallelism
  - Master thread spawns a team of threads as needed
  - Parallelism is added incrementally: that is, the sequential program evolves into a parallel program
Work sharing: data parallelism

- **parallel** construct forks additional threads
- **for** and **do** constructs distribute loop iterations within the threads that encounter the construct

```c
// assume N = 100000
#pragma omp parallel
{
    #pragma omp for
    {
        for(i = 0, i < N, i++)
            c[i] = a[i] + b[i];
    }
}
```
Work sharing: task parallelism

- The sections construct can be used to compute tasks in parallel

```c
#pragma omp parallel sections
{
    #pragma omp section /* Optional */
    a = taskA();
    #pragma omp section
    b = taskB();
    #pragma omp section
    c = taskC();
}

x = combine(a, b);
y = combine(x, c);
```
OpenMP clauses

- OpenMP constructs can be further refined by clauses
- **private**: make variables local to each thread (shared by default)
- **critical section**: the enclosed block is executed by at most one thread at a time
- **schedule(type, chunk)**: define the type of scheduling used for work sharing
  - type static: divide work equally between threads (each gets *chunk* iterations)
  - type dynamic: threads may request more iterations when finished (for load balancing)
  - type guided: chunk size decreases exponentially, but won’t be smaller than *chunk*
OpenMP: Discussion

- Library approach, no language integration
- Implemented for C, C++, Fortran, available on many platforms
- Supports incremental development of parallel programs, starting with a sequential one
- Some support for load balancing
Coordination Languages: Linda
Linda

- Coordination languages are based on the assumption that a concurrent programming language has two parts
  - A computation language, in which single-threaded execution is defined
  - A coordination language, for creation of computations and process communication

- The coordination features are based on the idea of a tuple space, which holds data tuples that can be stored and retrieved by the processes

- Linda is the original coordination language, appeared around 1985
Tuple spaces

- A tuple space is a collection of tuples such as
  \{ ("test", 11, true), ("test", 3, false), ("b", 23), ... \}

- Tuple spaces can be read and modified via the following operations:
  - `out(a_1, ..., a_n)` write tuple
  - `in(a_1, ..., a_n)` read and remove matching tuple
  - `read(a_1, ..., a_n)` read matching tuple
  - `eval(P)` start a new process P

- Pattern matching for `in` and `read`
  - \( (a_1, ..., a_n) \) can contain both actual and formal parameters
  - If no matching tuple is found, the operation blocks
Example: Tuple spaces

- Assume we have the following tuple space:
  \[
  \{(\text{"test"}, 11, \text{true}), (\text{"test"}, 3, \text{false}), (\text{"b"}, 23)\}
  \]

- Operations
  - `in(\text{"a"}, x)` blocks, no matching tuple
  - `in(\text{"test"}, x, b)` removes tuple (\text{"test"}, 11, true) and binds 11 to x and true to b (could have also selected tuple (\text{"test"}, 3, false))
  - `read(\text{"test"}, x, b)` reads tuple (\text{"test"}, 3, false)
  - `out(\text{"a"}, 14)` puts (\text{"a"}, 14) into the tuple space
  - The last action unblocks `in(\text{"a"}, x)`, which will remove the inserted tuple
Simulating semaphores in Linda

- Semaphores can be implemented in Linda
  - Initialization: tuple space with $k$ tuples ("token")
  - Implement $\text{down}$ with $\text{in}$("token")
  - Implement $\text{up}$ with $\text{out}$("token")

- Solution to the mutual exclusion problem:

```java
while true do
  in("token")
  critical section
  out("token")
  non-critical section
end
```
Linda: Discussion

- Communicating processes in Linda are only *loosely coupled*, processes need not know about other processes
- The coordination language is completely *orthogonal* to computation
  - Distribution of processes is easy
  - Potentially processes written in different languages can cooperate
- Implementations of Linda can be found in several languages such as Java (JavaSpaces) and C
Cilk

- Cilk is a language extension to C/C++, appeared in 1994
- For shared-memory multiprocessing
Cilk extends C/C++ with only few keywords:

- **cilk**: the routine may be spawned off in parallel
- **spawn**: the routine may execute in parallel with the parent caller
- **sync**: wait until all child threads have returned

```cilk
int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;
        x = spawn fib (n-1);
        y = fib (n-2);
        sync;
        return (x+y);
    }
}
```
Work stealing

- Each processor maintains a queue of threads that are ready to execute
- If the queue of a processor is empty, the processor may steal threads from a random processor’s queue
Cilk: Discussion

- Programmer indicates what can be executed in parallel
- The runtime environment decides how the work is divided among processors
- Hence it is automatic to map Cilk programs to new architectures
- When removing all Cilk keywords from a Cilk program, the result is a valid serial C program
- Cilk is commercially implemented and distributed by Intel
X10
Partitioned global address spaces (PGAS)

- Each processor has its own local memory, but the address space is unified.
- This allows processes on other processors to access remote data via simple assignment or dereference operations.

![Partitioned global memory diagram]

Diagram: Each processor has its own local memory segment, and these segments are logically connected to form a unified address space. Processes on different processors can access remote data by simple assignment or dereference operations.
X10

- Object-oriented language based on the PGAS model, appeared in 2004, developed by IBM
- New threads can be spawned asynchronously: Asynchronous PGAS model
- A memory partition and the threads operating on it are called a place
X10 operations (1)

- **async** $S$
  - Asynchronously spawns a new child thread executing $S$ and returns immediately

- **finish** $S$
  - Executes $S$ and waits until all asynchronously spawned child threads have terminated

```python
def fib(n: int) -> int:  
  if (n < 2) return 1;  
  val n1: int;  
  val n2: int;  
  finish {  
    async n1 = fib(n - 1);  
    n2 = fib(n - 2);  
  }  
  return n1 + n2;  
```
X10 operations (2)

- **when (E) S**
  - Conditional critical region: suspends the thread until E is true, then executes S atomically
  - E must be nonblocking, sequential, only access local data, and be side-effect free

```java
when (!buffer.full) {
    buffer.insert(item);
}
```
X10 operations (3)

- **at (p) S**
  - Executes S at place p
  - Blocks current thread until completion of S

```java
class C {
    var x: int;
    def this(n: int) { x = n; }
}

def increment(c: GlobalRef[C]) {
    at (c.home) c().x++; 
}
```
X10: Discussion

- Developed as part of the High Productivity Computing Systems initiative of the US Department of Defense: novel languages for supercomputing

- Very similar (in the same project)
  - Chapel, developed by Cray
  - Fortress (Fortran-based)

- More traditional PGAS languages
  - UPC (Unified Parallel C)
  - Co-array Fortran
  - Titanium (Java extension)
Conclusion

- Developers have a wide choice of languages for concurrency and parallelism
  - Important to know which languages target which applications
- Many are based on very innovative language concepts
  - Adoption can be low because of the learning curve
- No dominant innovative language for concurrency yet
  - Interesting field for research