



CHALMERS

Models and languages of concurrent computation

Lecture 13 of TDA384/DIT391
(Principles of Concurrent Programming)

Carlo A. Furia

Chalmers University of Technology – University of Gothenburg
SP1 2017/2018

Today's menu

Classifying approaches to concurrency

Message passing models and languages

Ada

Go

SCOOB

MPI

Shared memory models and languages

Linda

OpenMP

Cilk

X10

Other languages for concurrency

Part of today's presentation is based on material developed by S. Nanz for the Concepts of Concurrent Computation course given at ETH Zurich in 2015.

Classifying approaches to concurrency

Concurrency approaches galore

This course mainly focused on two representative approaches to concurrent programming:

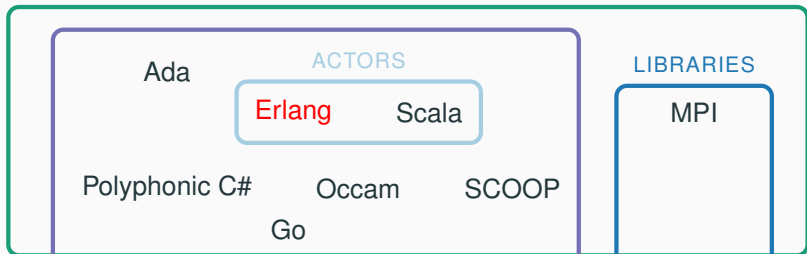
- **shared-memory** concurrency with **Java threads**
- **message-passing** concurrency with **Erlang processes**

There are many other variants of concurrency models and languages. This class gives an overview of a few approaches that are **historically** and **practically significant**.

We try to **classify** languages according to the features of their **concurrency models**. The categories are necessarily **fuzzy**, because actual languages often combine different models, but they are useful to navigate the rich landscape of concurrent and parallel programming.

Concurrency languages galore

MESSAGE PASSING



SHARED MEMORY

Message passing: synchronous vs. asynchronous

A natural classification of **message passing** primitives is between synchronous and asynchronous:

- **synchronous**: sender and receiver of a message block at the same time until they have exchanged the message
Examples: phone calls, live TV, Ada (rendezvous)
- **asynchronous**: sender and receiver of a message need not block
– sending and receiving can occur at different times
Examples: email, movies, Erlang (messages)

Message passing: synchronous vs. asynchronous

A natural classification of **message passing** primitives is between synchronous and asynchronous:

- **synchronous**: sender and receiver of a message block at the same time until they have exchanged the message
Examples: phone calls, live TV, Ada (rendezvous)
- **asynchronous**: sender and receiver of a message need not block – sending and receiving can occur at different times
Examples: email, movies, Erlang (messages)

Erlang uses asynchronous message passing with mailboxes:

- sending is non-blocking
- receiving is blocking (when no suitable message is available in the recipient's mailbox)

Shared memory: synchronous vs. asynchronous

For shared-memory models, the distinction between synchronous and asynchronous is less sharp, and mainly applies to **synchronization primitives**:

- **synchronous** primitives require all synchronizing parties to convene at a common time
Examples: traditional lectures, barrier synchronization
- **asynchronous** primitives share information between synchronizing parties without requiring them to access the information at the same time
Examples: take-home lab assignments, Linda

Addressing

Communication requires **addresses**: identifiers to match senders and receivers. A natural classification of **communication** primitives is according to their usage of addresses:

- **symmetric**: the sender specifies the receiver, and the receiver specifies the sender
Examples: email with filtering, communication with channels
- **asymmetric**: either the sender specifies the receiver, or the receiver specifies the sender – but not both
Examples: regular mail, phone calls (without caller identification), Erlang, sockets
- **indirect**: sender and receiver do not refer to each other directly, but communicate through an intermediary
Examples: communication with channels, Linda

Addressing

Communication requires **addresses**: identifiers to match senders and receivers. A natural classification of **communication** primitives is according to their usage of addresses:

- **symmetric**: the sender specifies the receiver, and the receiver specifies the sender
Examples: email with filtering, communication with channels
- **asymmetric**: either the sender specifies the receiver, or the receiver specifies the sender – but not both
Examples: regular mail, phone calls (without caller identification), Erlang, sockets
- **indirect**: sender and receiver do not refer to each other directly, but communicate through an intermediary
Examples: communication with channels, Linda

Erlang uses **asymmetric** message passing: sending `To ! Message` specifies the recipient `To`; receiving `receive Msg -> Msg end` need not specify the sender but only the message content.

Message passing models and languages

Message passing models and languages

Ada

Ada

Ada is an **object-oriented** programming language first developed in 1977 by the US Department of Defense to have one unified language for all software development. It still is under **active development**.

Ada's **design goals** include:

- suitable to build highly reliable systems
- reusable, modular components
- concurrency supported at language level

It introduced several **features** to programming language practice:

- strong typing and safe pointers
- modularity mechanisms (packages)
- exceptions
- high-level concurrency primitives

Ada

Ada is an **object-oriented** programming language first developed in 1977 by the US Department of Defense to have one unified language for all software development. It still is under **active development**.



Ada is named after **Ada Lovelace** (1815–1852), credited with being the first computer programmer.

Ada is an **object-oriented** programming language first developed in 1977 by the US Department of Defense to have one unified language for all software development.

Featured in:



```

nfc
  gui_mainfcn(gui_State, varargin{:});
end
* End initialization code - DO NOT EDIT

global Nr dair thetad nbits

Executes just before linear_array_gui is made visible.
function linear_array_gui_OpeningFcn(hObject, eventdata, handles, varargin)
* This function has no output args, see OutputFcn.
* hObject handle to figure
* eventdata reserved - to be defined in a future version of MATLAB
* handles structure with handles and user data (see GUIDATA)
* varargin command line arguments to linear_array_gui (see VARARGIN)

* Choose default command line output for linear_array_gui
handles.output = hObject;

* Update handles structure
guidata(hObject, handles);
101110 0010000 0110000 01101100 task body Airplane is
  huy : Runway_Access:
  begin
    Controller.Request_Takeoff (ID, huy);
    Put_Line (Airplane_ID' Inage (ID) & " taking off...");
    delay 2.0;
    huy.Cleared_Runway (ID); --huy); -- this is a blocking call - will run on
    controller reaching accept block and return on completion
    exit; -- in following block
    Put_Line (Airplane_ID' Inage (ID) & " in holding pattern"); -- simply print a
message
  end select;

```

Matlab code

Ada code

(The Matlab code is an anachronism)

Concurrency in Ada

Ada's support for concurrency includes both synchronous and asynchronous mechanisms:

- Concurrent execution uses **tasks**, which communicate via **synchronous message passing** using **rendezvous**
- **Protected objects** are a shared-memory model which extends monitors with waiting **guards**
- The language includes **nondeterministic** statements

Tasks

Procedures in Ada may include **tasks**:

- Each task has an **interface** and an **implementation**
- When a procedure starts executing, its tasks are implicitly activated

```
procedure Meetings is  
  
    task Point is  
        -- task interface  
end;  
  
    task body Point is  
        -- task implementation  
end Point;  
  
    -- rest of procedure  
end Meetings;
```

Rendezvous points

Task coordinate using **synchronous primitives** called **rendezvous**:

- Task interfaces declare **entry points**: actions they synchronize on
- Task implementations use **accept** statements to indicate where **rendezvous** take place

```
task Point is  
  entry Meet;  
end;
```

```
task body Point is  
begin  
  -- before meeting  
  accept Meet do  
    -- during meeting  
  end Meet;  
  -- after meeting  
end Point;
```

Synchronization using rendezvous

Rendezvous are a **synchronous mechanism**, where clients synchronize with tasks by invoking an entry point:

- the client **blocks** waiting for the task to reach the corresponding entry point; and resumes after the task has executed the rendezvous' body
- a task at an **accept** statement matches a client that is invoking the corresponding entry point

```
task body Point is  
begin  
  -- before meeting  
  accept Meet do  
    -- during meeting  
  end Meet;  
  -- after meeting  
end Point;
```

```
-- client  
declare  
  P: Point;  
begin  
  -- wait until P reaches Meet  
  P.Meet;  
  -- continue after P executes accept  
end;
```

Rendezvous with parameters

Entry points can include **parameters**, whose actual values are exchanged during rendezvous:

```
task Buffer is                                -- client synchronizing  
  entry Put (Item: in Integer);                B.Put(42);  
  entry Get (Item: out Integer);  
end;
```

Nondeterminism

Nondeterministic select statements group multiple **guarded accept** statements. During a rendezvous, any **accept** statement whose guard evaluates to true gets nondeterministically executed.

```
task body Buffer is  
  begin loop select  
    when Count < SIZE => -- buffer not full  
      accept Put (Item: in Integer) do  
        -- add Item to buffer  
      end;  
    or -- when neither full nor empty: nondeterministic choice  
      when Count > 0 => -- buffer not empty  
        accept Get (Item: out Integer) do  
          -- remove item from buffer  
        end;  
    end select; end loop;  
end Buffer;
```

Protected objects

Protected objects provide high-level shared-memory concurrency with features similar to **monitors**:

- all attributes are private
- procedures, functions, and entries may be public

In a protected object:

- **Functions** do not modify object state, therefore functions calls may execute concurrently on the same protected object
- **Procedures** and **entries** may modify object state, therefore they execute with exclusive access to protected objects
- **Entries** may include **guards**, which provide a synchronization mechanism similar to **conditional critical regions**

Conditional critical regions

A **conditional critical region** is a critical region with a guard B .

In Java pseudo-code:

```
synchronized (object) when (B) {  
    // critical region guarded by B  
}
```

Threads **queue** to enter a conditional critical region:

- when the lock protecting the region becomes available, the first thread P in the queue tries to acquire it
- if P 's guard B to the region evaluates to true, P holds the lock and enters the critical region
- if P 's guard B to the region evaluates to false, P releases the lock immediately and goes back into the queue

Guarded entries – implementing semaphores in Ada

Ada's protected objects provide conditional critical regions using **guards on entries**.

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

```
protected body Semaphore is
  procedure Up is
  begin
    Count := Count + 1;
  end Up;
```

```
-- conditional critical region
entry Down when Count > 0 is
  begin
    Count := Count - 1;
  end Down;
```

```
function Get_count
  return Natural is
  begin
    return Count;
  end Count;
end Semaphore;
```

Message passing models and languages

Go

Go (also called **golang**) is a procedural programming language first developed at Google in 2009. It is available as **open source**.

Go's **main features** include:

- static typing with type inference
- a packaging system that integrates with the build system
- memory safety checks performed by the compiler
- concurrency supported at language level with **goroutines** and **channels**

Go's creators include Ken Thompson and Rob Pike, who were also main developers of C and Unix in the 1970s.

Concurrency in Go

Go's support for **concurrency** includes:

- **Goroutines**: lightweight processes, similar to Erlang's processes but supporting both shared memory and message-passing communication
- **Channels**: a buffered mechanism to exchange messages between goroutines

Goroutines

A **goroutine** is a lightweight process executing an arbitrary function.

Using the keyword **go** in front of a regular function call spawns a goroutines executing that call:

```
func hello(who string) {  
    fmt.Println("Hello %s!", who)  
}  
  
// spawn two instances of hello,  
// printing "Hello world!" and "Hello class!" in any order:  
go hello("world")  
go hello("class")
```

Keyword **go** produces a behavior similar to as Erlang's **spawn**.

The name goroutine is a play on the name coroutine, a generalization of subroutines for concurrency (goroutines can implement coroutines).

Channels

Goroutines can exchange messages through **channels**.

Each **channel**:

- is declared using the modifier `channel`
- has a **type**, denoting the type of its messages
- is accessible to goroutines that have a **reference** to it
- has a finite **capacity** – zero by default

```
// a channel `stringMsg' with capacity 0 (default)  
// to exchange messages consisting of strings  
var stringMsg chan string = make(chan string)
```

```
// a channel `ints' with capacity 30  
// to exchange messages consisting of integers  
var ints chan int = make(chan int, 30)
```

Message passing through channels

Addressing is **indirect** in Go: messages are exchanged through channels, and hence senders and receivers communicate indirectly, without being explicitly aware of each other.

Sending and receiving are primitives of the language using the **<- operator**:

SENDING

```
ch <- v
```

send the value of expression `v` to channel `ch`

RECEIVING

```
m := <- ch
```

receive a message from channel `ch` and store it in variable `m`

Message passing: synchronous vs. asynchronous

Synchronization using message passing behaves differently according to the **capacity** of channels:

Channels with capacity zero (**unbuffered**) are **synchronous**: a receiver evaluating `<- ubch` blocks until a value is available in unbuffered channel `ubch`; and a sender evaluating `ubch <- v` blocks until a receiver is available to receive message `v` on `ubch`.

Channels with positive capacity (**buffered**) are potentially **asynchronous**: a receiver evaluating `<- bch` does not block unless buffered channel `bch` is empty; and a sender evaluating `bch <- v` does not block unless `bch` is full.

Channels store and deliver messages in **FIFO** (first-in first-out) order; messages sent to a channel are enqueued in order, and are received starting from the one that has been in the channel the longest.

Message passing: synchronous vs. asynchronous

Synchronization using message passing behaves differently according to the **capacity** of channels:

Channels with capacity zero (**unbuffered**) are **synchronous**: a receiver evaluating `<- ubch` blocks until a value is available in unbuffered channel `ubch`; and a sender evaluating `ubch <- v` blocks until a receiver is available to receive message `v` on `ubch`.

Channels with positive capacity (**buffered**) are potentially **asynchronous**: a receiver evaluating `<- bch` does not block unless buffered channel `bch` is empty; and a sender evaluating `bch <- v` does not block unless `bch` is full.

Message passing with unbuffered channels is similar to Ada's rendezvous; message passing with buffered channels is similar to Erlang's send/receive but with finite capacity.

Producer-consumer with bounded buffer in Go

The features of Go buffered channels make it very easy to implement a **producer-consumer bounded buffer**:

```
const Size = 10 // capacity of buffer
var buffer = make(chan int, Size) // channel with capacity `Size`

func put(item int) {
    buffer <- item // send item to buffer
} // block if full

func get() (item int) {
    return <- buffer // receive item from buffer
} // block if empty
```

Nondeterminism

The `select` statement support `nondeterministic` receiving from multiple channels. In the following example:

- receive the first message that is available on channels `greetingsCh` and `goodbyeCh`
- block if no messages are available on either channel
- receive from one nondeterministically chosen channel if messages are available on both channels
- time out after 3 seconds of wait

```
select {  
  case m := <- greetingsCh:  
    fmt.Println("Greetings from %s", m)  
  case m := <- goodbyeCh:  
    fmt.Println("Goodbye from %s", m)  
  case <- time.After(3 * time.Second)  
    fmt.Println("No messages for 3 seconds")  
}
```

Nondeterminism

The **select** statement support **nondeterministic** receiving from multiple channels.

Blocking and waiting can be **avoided** completely by adding a **default** case, which executes immediately if none of the channels are ready:

```
select {  
  case m := <- greetingsCh:  
    fmt.Println("Greetings from %s", m)  
  case m := <- goodbyeCh:  
    fmt.Println("Goodbye from %s", m)  
  default:  
    fmt.Println("I'm not waiting for messages!")  
}
```

Message passing models and languages

SCOOP

SCOOP

SCOOP (Simple Concurrent Object-Oriented Programming) is the part of the Eiffel programming language that deals with concurrency.

Each object is associated at runtime to a **single thread** (called “processor” in SCOOP jargon), which is the sole responsible for executing **calls on the object**.

Objects that are associated with different threads are called **separate**.

The type modifier **separate** indicates that calls to objects of that type **may** be handled by a different thread.

```
cnt: INTEGER           -- run by Current object's thread
shared_counter: separate COUNTER -- may be run by different thread
```

Method calls

Method calls are implicit **synchronization** events in SCOOP.

The call `o.m` of **procedure** method `m` on object `o`:

- is **synchronous** if `o` has type **not separate**
- may be **asynchronous** if `o` has type **separate**

If `m` is a **function** (returning a value), the call blocks until the result is computed – that is it behaves like a non-separate call.

A call `o.m` is executed by the single **thread handling** object `o`:

- the client's thread sends a **message** to `o`'s handler, requesting to execute `m`
- the client request is added to a **queue** in `o`'s handler
- if `m` is a function, after `o`'s handler gets to execute `m`, it sends the **results** back to the client

This explains how SCOOP uses **message-passing** concurrency.

Atomic method calls

Before a method starts executing, it gets exclusive access to **all its separate arguments**. This makes it easy to ensure that methods **execute atomically**:

```
-- transfer 'amount' from 'source' to 'target'  
transfer (source, target: separate BANK_ACCOUNT; amount: INTEGER)  
do -- lock both 'source' and 'target' before proceeding  
  source.withdraw(amount)  
  target.deposit(amount)  
end
```


Assertions: preconditions and postconditions

Eiffel supports **assertions** such as pre- and postconditions:

preconditions: a call `o.m (a)` to method `m` is valid only if `o` and `a` satisfy `m`'s precondition

postconditions: the implementation of a method `m` is correct only if every valid call `o.m (a)` terminates in a state where `o` satisfies `m`'s postcondition

```
withdraw (amount: NATURAL)
```

```
  require -- precondition: cannot withdraw more than 'balance'  
    amount <= balance
```

```
  do
```

```
    balance := balance - amount
```

```
  ensure -- postcondition: 'balance' is decreased by 'amount'  
    balance = old balance - amount
```

```
  end
```

Preconditions as waiting conditions

Preconditions that refer to separate arguments double as **waiting conditions** in SCOOP:

```
class PRODUCER
```

```
  put (b: separate BUFFER;  
       item: INTEGER)  
  -- wait until b not full  
  require  
    not b.is_full  
  do  
    b.append (item)  
  end  
end
```

```
class CONSUMER
```

```
  get (b: separate BUFFER)  
       : INTEGER  
  -- wait until b not empty  
  require  
    not b.is_empty  
  do  
    Result := b.remove  
  end  
end
```

Message passing models and languages

MPI

MPI (Message Passing Interface) is an **API specification** for inter-process **message-passing**, initially developed in the early 1990s.

MPI is the dominant standard in **high-performance computing**. MPI mainly targets **parallel** programs on **distributed-memory** systems – multi-processor systems where each processor has its own memory – although it is routinely used on shared-memory architectures as well.

MPI is **portable**:

- available as a library for many languages
- high-performance implementations in C/C++ and Fortran
- implementations for many different computer architectures

MPI supports multiple styles of programs; the most common one is **SPMD** (single program, multiple data): multiple processes execute the same program on different processors and **different** portions of the **input**.

Each process has a **rank**, which is an integer identifier ranging from 0 to $\text{num_procs} - 1$, where num_procs is the total number of processes. MPI programs assign **tasks** to different processes according to their rank.

Hello world in MPI for C

Process with rank 0 prints messages received from other processes.

```
char message[256]; int my_rank, num_procs, other_rank;
MPI_Init(&argc, &argv); // initialize MPI
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // get my rank
MPI_Comm_size(MPI_COMM_WORLD, &num_procs); // number of processes
// different behavior according to rank
if (my_rank != 0) { // send message to process 0
    sprintf(message, "Hello from #%d!", my_rank);
    MPI_Send(message, sizeof(message), MPI_CHAR, 0,
              0, MPI_COMM_WORLD);
} else { // proc 0: receive messages from other processes
    for (other_rank = 1; other_rank < num_procs; other_rank++) {
        { MPI_Recv(message, sizeof(message), MPI_CHAR, other_rank,
                  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
          printf("%s\n", message); }
    }
}
MPI_Finalize(); // shut down MPI
```

Shared memory models and languages

Shared memory models and languages

Linda

Linda

Linda is a **coordination** language developed in 1985 by Gelernter and Carriero. Using a coordination language means splitting the features of a concurrent programming language in two parts:

- a **computation language**, which executes computations sequentially
- a **coordination language**, which supports creating concurrent processes and synchronizing them

Linda is a set of primitives that define a coordination language based on the notion of **tuple space**: shared memory is organized in tuples which can be stored and retrieved by processes

Tuple spaces

A **tuple space** is a shared collection of tuples of any size and type:

$$\left[\langle \text{count}, 3 \rangle \langle \text{birthday}, 20, \text{January} \rangle \langle 1, 1, 2, 3, 5, 8 \rangle \right]$$

Multiple copies of the same tuple may exist in a tuple space.

The metaphor of the **message board** helps us understand the **primitives** used to access a tuple space:

<code>out(a_1, \dots, a_n)</code>	write tuple	post message to board
<code>in(a_1, \dots, a_n)</code>	read and remove tuple	remove message from board
<code>read(a_1, \dots, a_n)</code>	read tuple	read message from board
<code>eval(P)</code>	start new process	—

Operations `in` and `read` **pattern-match** on their arguments, and block until a matching tuple is in the space (similarly to Erlang's **receive** but on shared memory); when multiple tuples match, one is chosen **nondeterministically**.

Implementing semaphores in Linda

A counting semaphore can be implemented by putting as many copies of tuple `<"semaphore", id>` as the semaphore's count, where `id` is a unique identifier of the semaphore instance.

```
public class TupleSemaphore implements Semaphore {  
    // initialize with capacity tuples  
    TupleSemaphore(int capacity)  
    { for (int i = 0; i < capacity; i++) up(); }  
    // add a copy of the tuple; do not block  
    void up()  
    { out("semaphore", hashCode()); }  
    // remove a copy of the tuple; block if no tuples are in the space  
    void down()  
    { in("semaphore", hashCode()); }  
}
```

Shared memory models and languages

OpenMP

OpenMP (Open Multi-Processing) is an **API specification** for **shared-memory** multi-threaded programming, initially developed in the late 1990s.

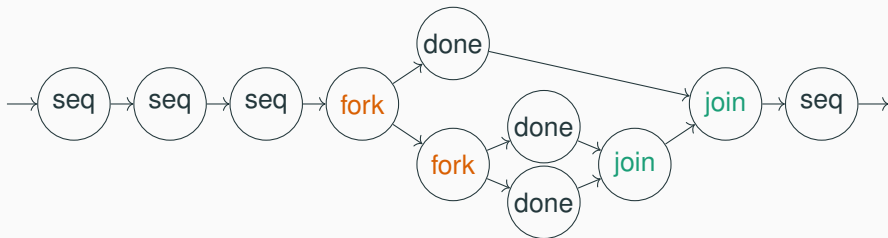
OpenMP mainly targets **fork/join** parallelism and numerical programming. It includes **parallel** extensions of **loops**, which support incrementally adding parallelism to sequential programs. The extensions are available as **pre-processor macros**, which are ignored by a compiler without OpenMP support.

Implementations of the OpenMP API are available in C/C++ and Fortran.

Fork/join parallelism

OpenMP's programs follow the **fork/join** model of parallelism:

- a **master thread** spawns parallel threads when needed, waits for them to terminate, and combines their results
- the overall program **alternates** sequential and parallel phases

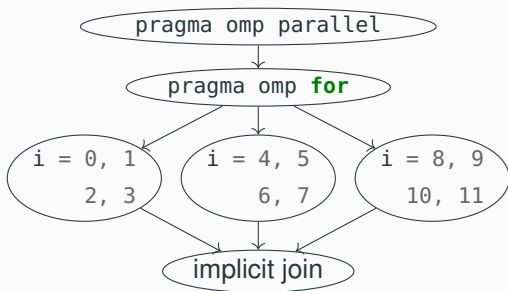


Data parallelism

With data parallelism, parallel threads process **disjoint portions** of the input data – typically stored in arrays or matrices:

- parallel **forks** new threads
- parallel **for** and **do** loops assign work to the running threads and **join** their results

```
#pragma omp parallel  
{  
#pragma omp for  
{  
  for (i = 0; i < 12; i++)  
    c[i] = a[i] + b[i];  
}  
}
```



Task parallelism

With task parallelism, parallel threads compute different functions of the input:

- parallel sections **forks** new threads
- section assigns a piece of code to one of the parallel threads

```
#pragma omp parallel sections  
{ // compute a, b, c in parallel  
#pragma omp section  
    a = computeA();  
#pragma omp section  
    b = computeB();  
#pragma omp section  
    c = computeC();  
} // implicit join  
return combine(a, b, c);
```


Clauses

Synchronization between parallel threads can be specified indirectly using **clauses**:

- `private(x)`: each thread has a **private** copy of variable `x` (counter loop variables are private by default; other variables are shared by default)
- `critical`: the block is executed by **at most one** thread at a time
- `schedule(type, chunk)`: set the way **work is split** and assigned to threads:
 - `type static`: each thread gets `chunk` iterations per round
 - `type dynamic`: threads that finish earlier may get more iterations
 - `type guided`: chunk size decrements exponentially (but won't be smaller than `chunk`)

Shared memory models and languages

Cilk

Cilk is a **language extension** to C/C++ for **shared-memory** multi-threaded programming, initially developed in the mid 1990s.

Cilk targets **fork/join** parallelism by extending C/C++ with just **few constructs**:

- programmers indicate what can be executed in parallel
- the runtime environment allocates work to threads using work stealing
- a Cilk program stripped of all Cilk keywords is a valid sequential C/C++ program

A commercial Cilk **implementation** is distributed by Intel.

Cilk keywords

Cilk adds only few keywords to C/C++:

- **spawn** `f()`: the call `f()` can be run in parallel
- **sync**: wait until all parallel calls have completed
- **cilk**: declares a function that may use Cilk constructs (not used in recent versions of Cilk)

```
cilk int fibonacci(int n) { // compute n-th fibonacci number
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fibonacci(n - 1); // can fork
    y = spawn fibonacci(n - 2); // can fork
    sync;                       // wait for x and y
    return x + y;
  }
}
```

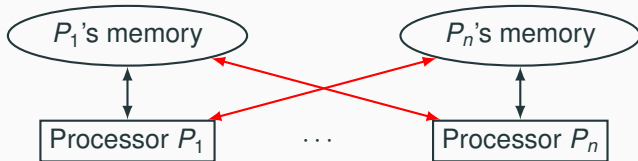
Shared memory models and languages

X10

PGAS (Partitioned Global Address Space) is a **shared-memory** computer architecture:

- each processor has its **local memory**
- all local memories share a **global address** space

Thus, every processor can read and write any other processor's memory in the **same way** it accesses its local memory.



X10 is an **object-oriented** language for multi-threaded programming on **PGAS**, developed in 2004 by IBM.

X10 supports **asynchronously** creating threads that work on **places** — a memory partition where a bunch of threads operate.

Fork/join operations

Operations **async** and **finish** support fork/join parallelism:

- **async** { B }: asynchronously spawn a thread executing block B
- **finish** { B }: execute block B and wait until all threads spawned in B have terminated

```
class Fibonacci {  
  public static def fibonacci(n: Int): Int {  
    if (n < 2) return n;  
    val x: Int; val y: Int;  
    finish {  
      async x = fibonacci(n - 1);  
      async y = fibonacci(n - 2);  
    } // x and y are available  
    return x + y;  
  }  
}
```


Critical regions

Other operations define **critical regions**:

- **atomic** { B }: execute block B atomically; B must be nonblocking, sequential, and only access local data
- **when** (C) { B }: define B as a critical region with blocking condition C; B and C must be nonblocking, sequential, and only access local data (C should also be without side effects)

```
class Buffer[T] {  
  private var buffer: List[T];  
  private var count: Int;  
  
  def put(item: T)  
  { atomic {buffer.add(item); count += 1;}}  
  
  def get(): T  
  { when (!buffer.isEmpty()) {count -= 1; return buffer.remove();}}  
}
```

Code mobility

The construct `at (p) { B }` supports executing code **at a different location**:

- suspend execution in the current place
- transmit code B and the data it uses to place p
- execute B at place p and wait for termination
- if B is an expression, transmit result back

```
class Counter {                               // somewhere else in the code
  private var count: Int;                       def increment(cnt: GlobalRef[Counter]) {
  def this(n: Int)                               // increment cnt by 1 at its place
  { count += n; }                               at (cnt.home) cnt(1);
}                                               }
```

Other PGAS languages

X10 was developed in a US Department of Defense project about novel languages for supercomputing. Other **similar languages** were developed in the same project:

- Chapel by Cray
- Fortress by Sun – based on Fortran

Other preexisting PGAS languages follow more **standard models** of parallelism:

- UPC (Unified Parallel C)
- CAF (Coarray Fortran)
- Titanium – a dialect of Java

Other languages for concurrency

But wait – there's more!

Developing models and languages for concurrent programming that are **practical** and **efficient** is still a very active **research** area.

A few other **widely used** languages/libraries:

- **Pthreads** (POSIX threads) are the standard API for shared-memory thread programming in C/C++. They provide features similar to Java/C# threads.
- **C# threads** are very close to Java's. C# also includes features for fork/join parallelism (**async** and **await** keywords).

But wait – there's more!

Some other **niche/experimental** approaches:

- Occam is a language using message-passing communication with **channels**; it follows the theoretical model of the process algebra CSP (Communicating Sequential Processes)
- Polyphonic C# is an extension of C# with asynchronous methods and **chords** – a feature to combine the results of asynchronous methods
- The developers of C# experimented with several high-level concurrency models, including a form of **transactions** using **LINQ** (Language Integrated Query, which introduced functional features in C#). If you are interested in some details: <http://joeduffyblog.com/2016/11/30/15-years-of-concurrency/>.