Parallelism

Master 1 International

Andrea G. B. Tettamanzi
Université de Nice Sophia Antipolis
Département Informatique
andrea.tettamanzi@unice.fr

Lecture 7, Part a

Languages and Libraries, Performance Measures

Table of Contents

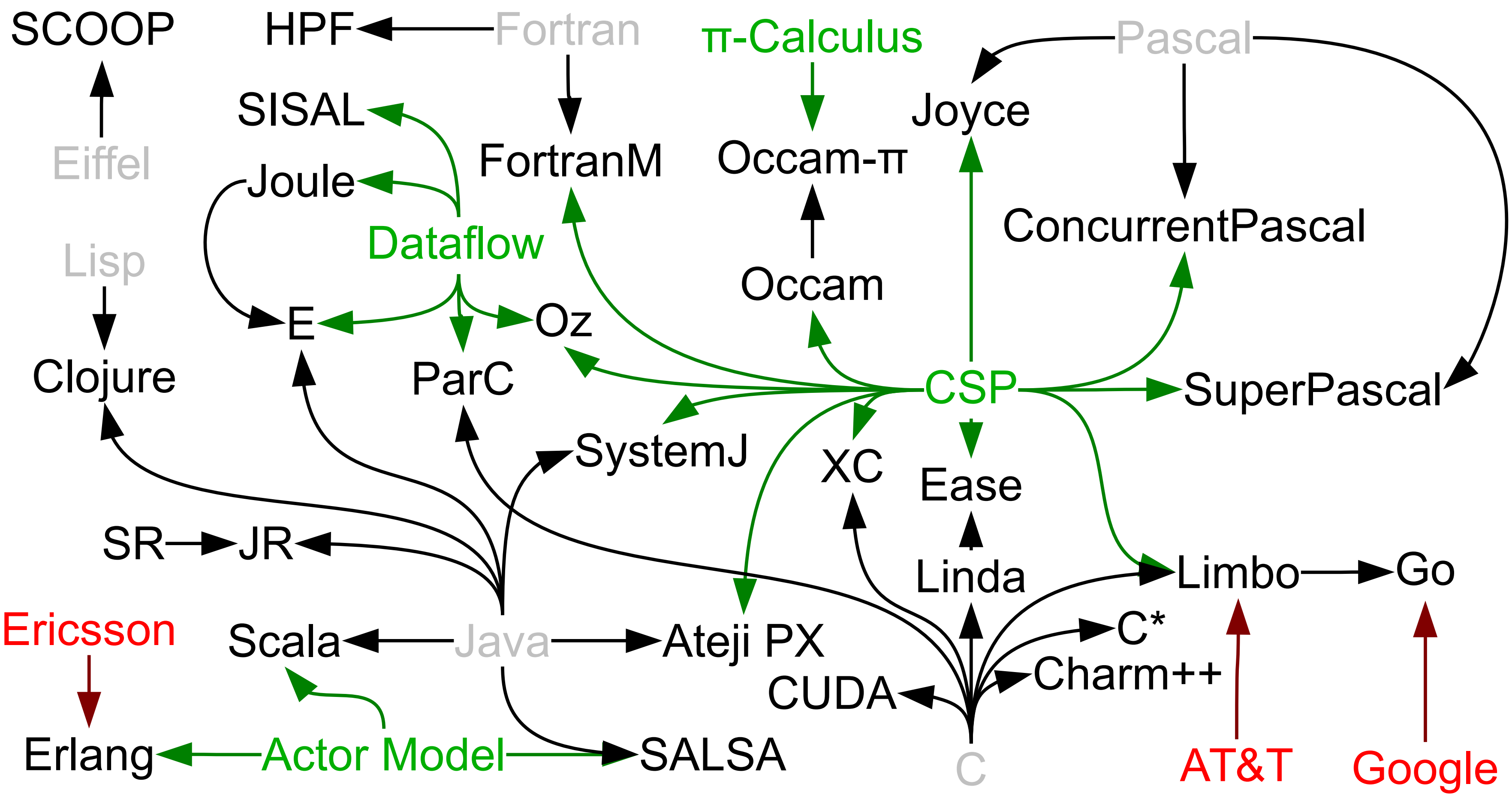
Languages or Libraries?

- To write concurrent or parallel programs, there are two options:
  - Use a programming language specifically designed for that
  - Use a standard library which extends the functionality of a sequential programming language
- Why concurrent languages?
  - Concurrent languages are designed around a theoretical model of concurrency
  - They lead to clean, well-structured, and efficient coding
- Why libraries?
  - You don't have to learn a new language

Concurrent Languages

- A large number of programming languages have been proposed to support concurrent programming natively
- I have personally counted more than 80 of them, but probably there are more than 100
- Most of these language never got a large user base
  - Some of them are research prototypes based on a particular theoretical model
  - Some are proprietary/domain-specific languages, developed within industry or publicly funded research projects
- Most of them extend a popular sequential programming language by adding parallel and concurrent constructs
- A few are built from scratch, but inspired by existing languages

An (Incomplete) Chart



Brief Overview of Some Languages

- It would be impossible to cover all these languages
- We will give a brief overview of a handful of them
  - Go, because it is backed by Google Inc.
  - Ateji PX, because it is an extension of Java made in France
  - Clojure, functional, based on JVM, with a Lisp-like syntax
  - C*, of historical interest, but also useful as an introduction to contemporary languages for GP-GPUs
- Of course, this is an arbitrary selection
- You are welcome to explore, discover, and try out other languages, following your taste and inclination

Go

- Go has been designed for Google Inc. by R. Griesemer, Rob Pike, and Ken Thompson since 2007; announced in 2009.
- Syntax of Go broadly similar to that of C:
  - blocks of code are surrounded with curly braces;
  - control flow structures: for, switch, if.
- Unlike C,
  - line-ending semicolons are optional,
  - variable declarations are different and usually optional
  - type conversions must be explicit,
  - New concurrent control keywords: go, select
- New types: maps, UTF-8 strings, array slices, channels

Concurrency in Go

- Go provides “goroutines” (an allusion to coroutines)
- Goroutines = small lightweight threads
- Created from functions with the **go** statement
- Goroutines are executed in parallel
- Groups of goroutines are multiplexed over multiple threads

Ateji PX

- An extension of Java to facilitate parallel computing on multi-core processors, GPUs, Grids and the Cloud
- Introduces the new construct ||, which introduces a parallel branch
- Data parallelism is obtained by || followed by a quantification
  - e.g.: `|| (int i : array.length) array[i]++;`
- Communication among parallel branches:
  - Through shared variables
  - Explicit, through named channels (à la pi-calculus)
    - <channel> ! <expr> (send a value on a channel)
    - <channel> ? <expr> (receive a value from a channel)

Ateji PX: Example 1

```
int fib(int n) {
	if(n <= 1) return 1;
	int fib1, fib2;
	// recursively create parallel branches
	[
			|| fib1 = fib(n-1);
			|| fib2 = fib(n-2);
	]
	return fib1 + fib2;
}
```

Ateji PX: Example 2 (demonstrating data flow programming)

```
void adder(Chan<Integer> in1, Chan<Integer> in2, Chan<Integer> out) {
      for( ; ; ) {
                  int value1, value2;
                  [ in1 ? value1; || in2 ? Value2; ];
                  out ! value1 + value2;
      }
}
// …
[
      || source(c1); // generates values on c1
      || source(c2); // generates values on c2
      || adder(c1, c2, c3);
      || sink(c3); // read values from c3
]
```

Clojure

- A dialect of Lisp created by Rich Hickey
- Runs on the JVM, the Common Language Runtime, and the JavaScript interpreter
- Purely functional; immutable core data structures
- Offers various mechanisms to coordinate the concurrent execution of threads:
  - Software transactional memory (synchronous state sharing)
    - Keywords: dosync, ref, set, alter, …
  - An agent system (asynchronous independent state sharing)
  - An atoms system (synchronous independent state sharing)
  - A dynamic var system (isolating changing state)
    - Keywords: def, binding, …

Clojure Refs and Dynamic Vars

- Refs are mutable references to objects
  - They can be ref-set or altered to refer to different objects within a transaction
  - Transactions are delimited by “dosync” blocks
  - Reads of refs provide a snapshot at a particular point in time
- Dynamic vars are also mutable references to objects
  - They have a thread-shared root binding
  - Any modification to those bindings are scoped to local thread
  - Nested bindings obey a stack protocol and unwind as control exits the binding block

C*

- C* is an OO superset of ANSI C with synchronous semantics
- Developed by Thinking Machines Corporation, 1987–1993
- A C* program can consist of:
  - Standard sequential C code
  - C* code
  - Header files
  - Calls to the CM timing utility, library functions, and CM Fortran subroutines
- Source file extension: *.cs

C New Features*

- A method for describing the size and shape of parallel data and for creating parallel variables
- New operators and expressions for parallel data
- New meanings for standard operators that allow them to work with parallel data
- Methods for choosing the parallel variables, and the specific data points within them, upon which C* code is to act
- Pointers to parallel data and to shapes
- Changes to the way functions work, so that, eg., parallel variables can be used as arguments
- Methods for communication among parallel variables

Example

```
shape [2][32768]ShapeA;        // shape declaration
int:ShapeA p1, p2, p3;         // declaration of parallel variables
int sum = 0;

main() {
    with(ShapeA) {
            p1 = 1;        p2 = 2;        // parallel assignments
            p3 = p1 + p2;       // parallel sum
            printf("The sum in one element is %d.\n", [0][1]p3);
            sum += p3;   // reduction assignment
            printf("The sum of all elements is %d.\n", sum);
    }
}
```

Libraries

- As there are plenty of concurrent programming languages, there are plenty of libraries that support concurrency and parallelism
- Some time-honored parallel libraries, like
  - PVM
  - MPI
- More recent libraries, to tap into the power of multicore CPUs, GP-GPUs, grids, and the Cloud, like
  - GPars, a library for Groovy
  - SystemC, an event-driven simulation kernel in C++
  - C++ AMP and OpenCL
  - …

PVM

- Parallel Virtual Machine, released in 1989
- Designed to allow a network of heterogeneous machines to be used as a single distributed parallel processor
- Very portable, open source, mature and stable
- PVM consists of
  - a run-time environment
  - A library for message-passing, task and resource management, and fault notification
- Supports the C, C++, and Fortran programming languages
- Supports broadcasting and multicasting, in addition to process-to-process message passing

PVM: Some Primitives

- pvm_mytid(): gets the id of the calling process
- pvm_send(): sends a message to the process with the given id
- pvm_probe(): checks whether a message has arrived
- pvm_recv(): receives a message from the process w/ the given id
- pvm_bcast(): broadcasts a message to a group of processes
- pvm_joingroup(): enrolls the calling process in a group
- pvm_lvgroup(): leaves the specified group
- pvm_insert(): stores data into the PVMD database
- pvm_lookup(): retrieves data from the PVMD database
- pvm_exit(): terminates local process
- …

MPI

- Message-Passing Interface, version 1.0 released in 1995
- Both a library and a standard
- The MPI standard defines the syntax and semantics of a core of library routines useful to write portable message-passing programs in Fortran and C
- MPI supports both point-to-point and collective communication
- MPI-1 had no shared memory support; MPI-2 has a limited one

Pthreads and OpenMP

- Threaded shared memory programming models
- Pthreads = POSIX Threads
  - Defines an API for creating and manipulating threads
  - Available on all POSIX-conformant operating systems
- OpenMP = Open Multiprocessing
  - API supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran
  - Core elements of OpenMP: constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables

ProActive

- A Java grid middleware for parallel, distributed, and multi-threaded computing.
- Developed by the OW2 Consortium, including INRIA, CNRS, University of Nice Sophia Antipolis, and ActiveEon.
- Open-source software released under the GPL license.
- Comprehensive framework and parallel programming model for
  - multi-core processors
  - distributed on Local Area Network (LAN)
  - on clusters and data centers
  - on intranets
  - on Internet grids
- Programming model: Active Objects

The Active Objects Model

- Active object ↔ thread
- A thread may contain zero or more passive objects
- Only references to active objects are shared in the system
- Passive objects are referenced only inside their thread
- In RMI
  - active objects are passed by reference
  - Passive objects are passed by deep copy
- All RMI are made asynchronous whenever possible
- They immediately return “future objects”
- Future object: a placeholder for an object still to come
- As long as future objects are not invoked, a process doesn't block

Performance Measures

- Motivation and Introduction

Speedup

- How much a parallel algorithm is faster than a corresponding sequential algorithm.
- Defined as the ratio:

$$S_n = \frac{T_1}{T_n}$$

  where:
  - T_1 is the execution time of the sequential algorithm (i.e., the algorithm executed by 1 processor)
  - T_n is the execution time of the parallel algorithm, executed by n processors
- The linear (or ideal) speedup is n;
- A ratio greater than n is called "superlinear speedup"

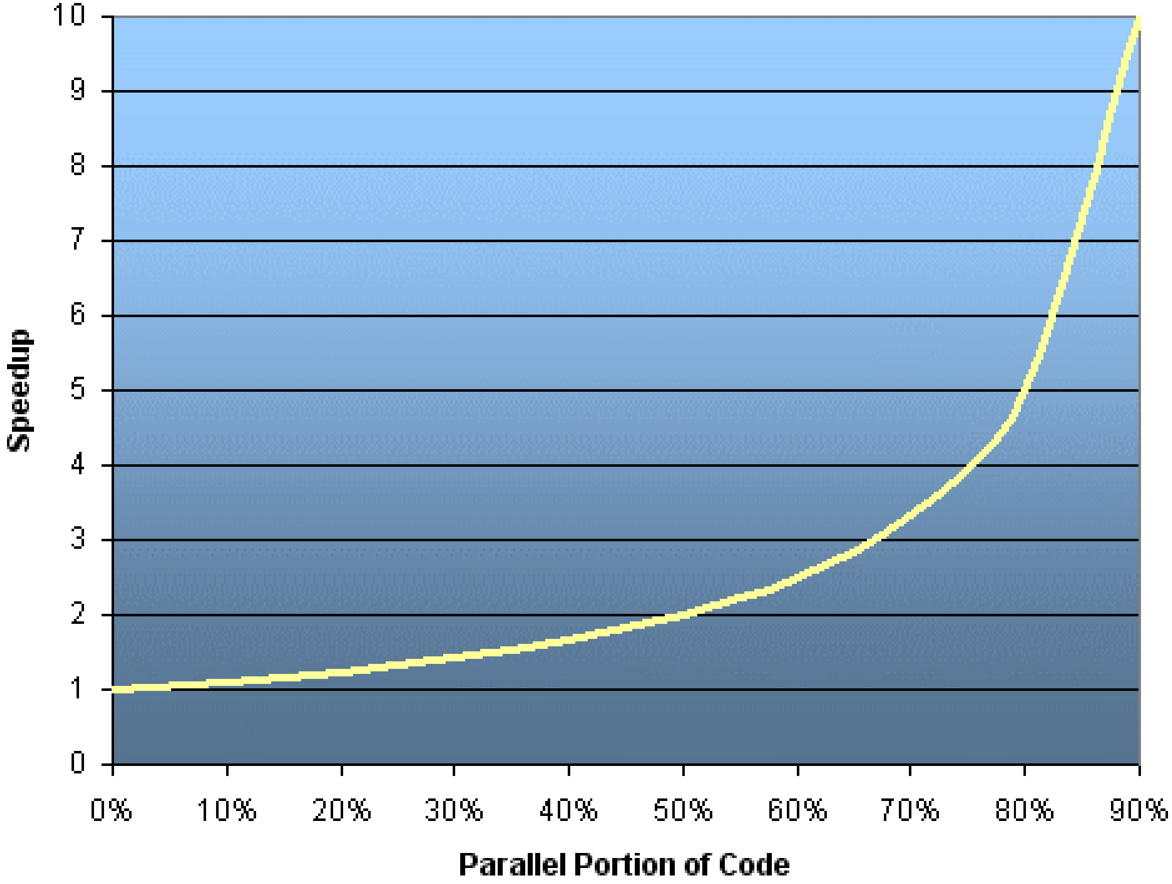
Amdahl's Law

- Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

$$S_{max} = \frac{1}{1 - P}$$

- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup).
- If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.
- **Note:** hypothesis of infinitely many processors available!

Amdahl's Law


10
9
8
7
6
5
4
3
2
1
0
Speedup
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
Parallel Portion of Code

Amdahl's Law Revisited

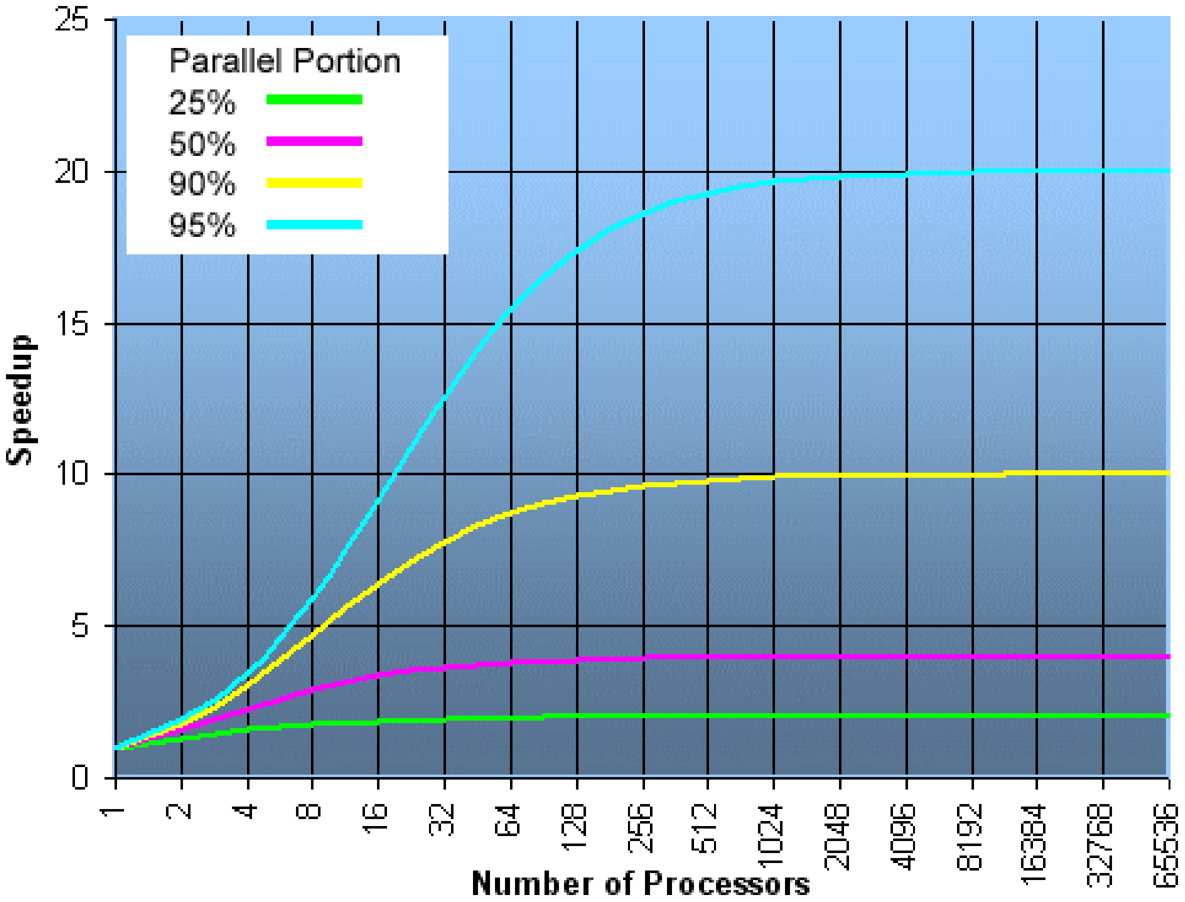
- Introducing the number of processors, *n*, available for performing the parallel fraction of work, the relationship can be modeled by:

$$S_{max} = \frac{1}{\frac{P}{n} + S}$$

  where *S* = 1 – *P* is the sequential fraction of code.

- It is quite obvious that there are limits to the scalability of parallelism

Amdahl's Law Revisited



Amdahl's Law and Scalability

- Certain problems demonstrate increased performance by increasing the problem size. For example:
  - 2D Grid Calculations: 85 seconds 85%
  - Serial fraction *S*: 15 seconds 15%
- We can increase the problem size by doubling the grid dimensions and halving the time step.
- This results in four times the number of grid points and twice the number of time steps. The timings then may look like:
  - 2D Grid Calculations 680 seconds 97.84%
  - Serial fraction *S*: 15 seconds 2.16%
- Problems that increase the percentage *P* of parallel time with their size are *more scalable* than problems with a fixed *P*.

Efficiency and Cost

- Efficiency: E = Speedup/n
  - The efficiency in case of linear speedup would be $E = 1$
- Processing units do not come for free
- Idea: let's weigh the performance by the cost of processing equipment (in processor cost units: cost of one processor = 1)
- Cost: $C_n = n\, T_n$
- Cost-optimal formulation of a parallel algorithm
  - Given speedup S_n, $C_n = n\, T_1/S_n$
  - Find best compromise between speedup and cost
  - In the case of linear speedup, $T_n = T_1/n$: therefore, $C_n = T_1$, i.e., adding more processing units comes for free!

Thank you for your attention