Introduction to Parallel computing and Distributed systems

Adam Belloum
Institute of Informatics
University of Amsterdam
a.s.z.belloum@uva.nl

High Performance computing Curriculum, Jan 2015 http://www.hpc.uva.nl/

UvA-SURFsara
If you know these concepts you are attending the wrong class ...

- Supercomputing / High Performance Computing (HPC)
- Node
- CPU / Socket / Processor / Core
- Task
- Pipelining
- Shared Memory
- Symmetric Multi-Processor (SMP)
- Distributed Memory
- Communications
- Synchronization
- Granularity
- Observed Speedup
- Parallel Overhead
- Massively Parallel
- Embarrassingly Parallel
- Scalability
Topic covered in these lectures

- The basics of parallel programming *
- Grid computing
- Cloud computing *
- Workflow Management systems
- BigData/Data Science *
- GPU programming *

(*) Topic will be further developed in the context of the following workshops more details on http://hpc.uva.nl
Content

• Computer Architectures
• High Performance Computing (HPC)
• Speed up
• Parallel programming models
Computer Architecture

• supercomputers use many CPUs to do the work
• All supercomputing architectures have
  – processors and some combination cache
  – some form of memory and IO
  – the processors are separated from every other processors by some distance
• there are major differences in the way that the parts are connected

some problems fit into different architectures better than others
• How CPU works [http://www.youtube.com/watch?v=cNN_tTXABUA]
• How Computers Add Numbers In One Lesson: [http://www.youtube.com/watch?v=VBDooT8o4q00&feature=fvwp]
• Computer Architecture Lesson 1: Bits and Bytes [http://www.youtube.com/watch?v=UmSelKbP4sc]
• Computer Architecture Lesson 2: Memory addresses [http://www.youtube.com/watch?v=yF_txERujps&NR=1&feature=episodic]
• Richard Feynman Computer Heuristics Lecture [http://www.youtube.com/watch?v=EKWGGDXe5MA]
Architectures: Michael J. Flynn (1972)

- Flynn’s taxonomy distinguishes multiprocessor computer according to independent dimensions
  - Instruction
  - Data

- Each dimension
  - Single
  - Multiple
Parallel Computer Memory Architectures

• we can also **classify** supercomputers according to how the **processors** and **memory** are connected

  – **couple of processors** to a single large memory address space

  – **couple of computers**, each with its own memory address space
Parallel Computer Memory Architectures

Shared Memory
- Uniform Memory Access (UMA)
- Non-Uniform Memory Access (NUMA)

Distributed Memory Multiprocessor
- Processors have their own local memory
- Changes it makes to its local memory have no effect on the memory of other processors.
High Performance Computing

- increasing *computing power* available allows
  - increasing the *problem dimensions*
  - adding more *particles* to a system
  - increasing the *accuracy* of the result
  - improve experiment turnaround time
  - ...
Why Use Parallel Computing?

• Solve larger problems
• Limits to serial computing
• Provide concurrency
• Use of non-local resources
• Save time and/or money

DreamWorks Presents the Power of Supercomputing
http://www.youtube.com/watch?v=TGSRvV9u32M&feature=fvwp

https://computing.llnl.gov/tutorials/parallel_comp/
High Performance Computing

• **What does High-Performance Computing (HPC) mean?**
  – High-performance computing (HPC) is the use of **super computers** and **parallel processing techniques** for solving complex computational problems.
  – *HPC technology* focuses on **developing** parallel processing systems by incorporating both **administration** and **parallel computational** techniques.

The terms high-performance computing and supercomputing are sometimes used interchangeably.

http://www.techopedia.com/definition/4595/high-performance-computing-hpc
Content

• High Performance Computing
• Computer Architectures
• Speed up
• Parallel programming models
• Example of Parallel programs
Speedup

• how can we measure how much faster our program runs when using more than one processor?

• define **Speedup** $S$ as:
  
  – The ratio of 2 program execution times

  * $T_1$ is the execution time for the problem on a single processor (use the “best” serial time)
  * $T_P$ is the execution time for the problem on $P$ processors

  – constant problem size
Speedup: Limit of Parallel programming

- A program **always** has a **sequential** part and a parallel part

```
(1) A = B+C;
(2) D = A + 1;
(3) E = D + A;
(4) For (I=0; I<E; I++)
(5) M(I) = 0;
```

- the best you can do is to sequentially execute 4 instructions no matter how many processors you get

http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming
Speedup: Implication

• Parallel programming is **great** for programs with a lot of parallelism
  – Jacobi, scientific applications (weather prediction, DNA sequencing, etc)

• Parallel programming **may not be that great** some traditional applications:
  • Computing Fibonacci series \( F(K+2)=F(k+1) + F(k) \)
Speedup: Amdahl’s Law (1967)

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

\[
\text{speedup} = \frac{1}{1 - P}
\]

Jon Johansson
Academic ICT
Copyright © 2006
University of Alberta
Speedup: Amdahl’s Law (1967)

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

\[
\text{speedup} = \frac{1}{P + S N}
\]
Speedup

- Linear speedup
- Sublinear speedup
- Superlinear speedup

- why do a speedup test?

Jon Johansson Academic ICT Copyright © 2006 University of Alberta
Content

• High Performance Computing
• Computer Architectures
• Speed up
• How to design Parallel programs
• Parallel programming models
• Example of Parallel programs
Design Parallel programs

• Domain decomposition and functional decomposition
  – **Domain decomposition**: DATA associate with a problem is decomposed.
    • Each parallel task then **works** on a portion of data
  – **Functional deposition**: focus on the computation that is be performed. The problem is decomposed according to the work that must be done.
    • Each task then **performs a portion of the overall work**
Design Parallel programs

Domain decomposition:

• Also Called data parallelism

• DATA associate with a problem is decomposed.

• Each parallel task then works on a portion of data

• Example: MapReduce

http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming
Design Parallel programs

Domain decomposition methods:

- Same datum may be needed by multiple tasks
- Decompose the data in such a manner that the required communication is minimized
- Ensure that the computational loads on processes are balanced

Domain decomposition methods

http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming
Functional deposition

- The focus is on the **computation** that is to be performed rather than on the data manipulated by the computation.

- The problem is decomposed according to the **work that must be done**.

- **Each task** then performs a portion of the overall work.

http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming
Data Dependence

• A **dependence** exists between programs when the **order** of statement execution **affects** the results of the program.

• A data dependence results from multiple use of the same location(s) in storage by different tasks.

  (task 1) – (task2)
  – True dependence: Write X – Read X
  – Output dependence: Write X – Write X
  – Anti dependence: Read X – Write X

**Dependencies:** are important to parallel programming because they are one of the inhibitors to parallelism.

http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming
Data Dependence

• The value of \( a(I-1) \) must be computed before the value of \( a(I) \).

• \( A(I) \) exhibits a **data dependency** on \( a(I-1) \).

• Parallelism is inhibited.

Data dependency examples

\[
\begin{align*}
&\text{For } (I=0; I<500; i++) \\
&a(I) = 0; \\
&\text{For } (I=0; I<500; i++) \\
&a(I) = a(I-1) + 1;
\end{align*}
\]
Load balancing

• Distribute the *computation/communication* such that all the processor are busy all the time.

• At a synchronization point, the worst case performance is the real performance
Communications

• Parallel applications that do not need communications are called **embarrassingly parallel programs**
  – Monte carlo method, Seti at home
  – Most programs (e.g. Jacobi) are not like that
  – Communication is inherent to exploit parallelism in a program
Communications

• Factors to consider:
  – Cost of the communication
  – Latency and bandwidth
  – Synchronous and asynchronous
  – Point to point or collective
Overlapping communication and computation

• Make processors busy when waiting for communication results
  – Usually achieved by using non-blocking communicating primitives

Loading balancing, minimizing communication and overlapping communication with computation are keys to develop efficient parallel applications
Some basic load balancing techniques

• **Equally partition the work each task receives**
  – For *array/matrix* operations where each task performs similar work, evenly **distribute the data** set among the tasks.
  – For *loop iterations* where the work done in each iteration is similar, evenly **distribute the iterations** across the tasks.

• **Use dynamic work assignment**
  – Sparse arrays
  – Adaptive grid method
  – If a *heterogeneous mix of machines* with *varying performance*
    – *scheduler - task pool* approach
Granularity

• Computation/ Communication
  – In parallel programming, granularity is a qualitative measure of the ratio of the computation to communication.

  – **Periods of computation** are typically separated form **periods of communication** by synchronization events
    • Computation phase and communication phase
Granularity

- Fine-grain parallelism
  - Relatively small amount of computational work are done between communication events
  - Low computation to communication ratio
  - Implies high commutation overhead and less opportunity for performance enhancement

- Coarse-grain parallelism
  - Relatively large amounts of computation work are done between communication/synchronization events
  - High computation to communication ratio
  - Implies more opportunity for performance increase
  - Harder to load balance efficiently
Deadlock/Livelock

- Deadlock appears when two or more programs are waiting and none can make progress.

- Livelock results from indefinite loop.
content

• High Performance Computing
• Computer Architectures
• Speed up
• How to design parallel applications
• Parallel programming models
• Example of Parallel programs
Parallel programming

• need to do something to your program to use multiple processors
• need to incorporate instructions into your sequential program which allow multiple instances of the program to run
  – One per processor
  – Each gets a piece of the work

• Several methods to do this …
Parallel programming models

- Shared Memory (without threads)
- Threads
- Distributed Memory / Message Passing
- Data Parallel
- Hybrid
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)
Parallel programming

Message Passing Interface (MPI)

- Interprocess communication which have separate address spaces

- Data is explicitly sent by one process and received by another
  
  - Data transfer usually requires cooperative operations to be performed by each process.
  
  - For example, a send operation must have a matching receive operation
Parallel programming

Message Passing Interface (MPI)

• What is MPI?
  – A message-Passing Library specification
  – Not a language or compiler specification
  – Not a specific implementation or product

• For parallel computers, clusters, and heterogeneous networks.
  – Designed to provide access to advanced parallel hardware for:
    • End users, library writers, tools developers
Parallel programming

Message Passing Interface (MPI)

- Why use MPI?
  - Optimized for performance
  - Will take advantage of fastest transport found
    - Shared memory (within a computer)
    - Fast cluster interconnects (Infiniband, Myrinet..) between computers (nodes)
    - TCP/IP if all else fails

Basic Message Passing

What is message passing?

- Data transfer plus synchronization
- Requires cooperation of sender and receiver
- Cooperation not always apparent in code
Parallel programming

Message Passing Interface (MPI)

Deadlocks?

• Send a large message from proc A to proc B
  – If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

• What will happen? (unsafe)
  – Process 0
    Send(1)
    Recv(1)
  – Process 1
    Send(0)
    Recv(0)

What is message passing?

• Data transfer plus synchronization
• Requires cooperation of sender and receiver
• Cooperation not always apparent in code
Parallel programming

Message Passing Interface (MPI)

- Very good for distributing large computations across reliable network
- Would be terrible for a distributed internet chat client or BitTorrent server
Example MPI Hello World

```c
#include <mpi.h>

int main(int argc, char** argv) {
  // Initialize the MPI environment
  MPI_Init(NULL, NULL);

  // Get the number of processes
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  // Get the rank of the process
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  // Get the name of the processor
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int name_len;
  MPI_Get_processor_name(processor_name, &name_len);

  // Print off a hello world message
  printf("Hello world from processor %s, rank %d "
         "out of %d processors\n",
       processor_name, world_rank, world_size);

  // Finalize the MPI environment.
  MPI_Finalize();
}
```
Threads

- threads model of parallel programming, a **single** process can have **multiple**, concurrent execution paths
- Each thread has local data, but also, shares the entire resources of program.
- Threads **communicate** with each other through **global memory**
Parallel programming

Open MultiProcessing (OpenMP)

- What is OpenMP?
  - is a library that supports parallel programming in **shared-memory** parallel machines.
  - allows for the parallel execution of code (**parallel DO loop**), the definition of shared data (**SHARED**), and synchronization of processes.
Parallel programming

- Open MultiProcessing (OpenMP)
  - What is the programming model?
    - All threads have access to the same, globally shared, memory
    - Data can be shared or private
      - Shared data is accessible by all threads
      - Private data can be accessed only by the threads that owns it
    - Data transfer is transparent to the programmer
    - Synchronization takes place, but it is mostly implicit
Parallel programming

Open MultiProcessing (OpenMP)

OpenMP language extensions

- parallel control structures
- work sharing
- data environment
- synchronization
- runtime functions, env. variables

parallel directive

do/parallel do and section directives

shared and private clauses

coordinates thread execution

critical and atomic directives

barrier directive

omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE
Example OpenMP Hello World

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {

  int nthreads, tid;

  /* Fork a team of threads giving them their own copies of variables */
  #pragma omp parallel private(nthreads, tid)
  {

    /* Obtain thread number */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    /* Only master thread does this */
    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }

  } /* All threads join master thread and disband */
}
```

```
$ icc -o omp_hello -openmp omp_hello.c
omp_hello.c(22): (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
$ export OMP_NUM_THREADS=3
$ ./omp_hello
Hello World from thread = 0
Hello World from thread = 2
Hello World from thread = 1
Number of threads = 3
```
Parallel programming

Pros/Cons of OpenMP
✓ easier to program and debug than MPI
✓ directives can be added incrementally - gradual parallelization
✓ can still run the program as a serial code
✓ serial code statements usually don't need modification
✓ code is easier to understand and maybe more easily maintained

- can only be run in shared memory computers
- requires a compiler that supports OpenMP
- mostly used for loop parallelization

Pros/Cons of MPI
✓ runs on either shared or distributed memory architectures
✓ can be used on a wider range of problems than OpenMP
✓ each process has its own local variables
✓ distributed memory computers are less expensive than large shared memory computers

- requires more programming changes to go from serial to parallel version
- can be harder to debug
- performance is limited by the communication network between the nodes
Parallel programming

Shared State Models

- Views an application as a collection of processes communicating by putting/getting objects into one or more spaces

- A space is a shared and persistent object repository that is accessible via network

- The processes use the repository as an exchange mechanism to get coordinated, instead of communicating directly with each other
Parallel programming

**Shared State Models:** Publish/Subscribe

- Publish/subscribe systems are programming capability provided by associative matching

- Allows the **producers** and **consumers** of data to coordinate in a way where they can be **decoupled** and may not even know each other’s identity

  SOA, Web service etc.
Parallel programming

RPC and RMI Models

- **Structure the interaction** between sender and receiver as:
  - a language construct, rather than a library function call that simply transfers an uninterpreted data.

- **provide a simple and well understood mechanism** for managing remote computations
content

• High Performance Computing
• Computer Architectures
• Speed up
• How to design parallel applications
• Parallel programming models
• Example of Parallel programs
calculations on 2-dimensional array elements

• The serial program calculates one element at a time in sequential order.

• Serial code could be of the form:

```plaintext
do j = 1,n
  do i = 1,n
    a(i,j) = fcn(i,j)
  end do
end do
```

https://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch
calculations on 2-dimensional array elements: solution 1

• Implement as a Single Program Multiple Data (SPMD) model.

• each task executes the portion of the loop corresponding to the data it owns.

```plaintext
do j = mystart, myend
  do i = 1,n
    a(i,j) = fcn(i,j)
  end do
end do
```

https://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch
calculations on 2-dimensional array elements: implementation

• Implement as a Single Program Multiple Data (SPMD) model.

• **Master process** initializes array, sends info to worker processes and receives results.

• **Worker process** receives info, performs its share of computation and sends results to master.

https://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch
calculations on 2-dimensional array elements: implementation

```plaintext
find out if I am MASTER or WORKER

if I am MASTER

    initialize the array
    send each WORKER info on part of array it owns
    send each WORKER its portion of initial array

    receive from each WORKER results

else if I am WORKER

    receive from MASTER info on part of array I own
    receive from MASTER my portion of initial array

    # calculate my portion of array
    do j = my first column, my last column
    do i = 1, n
        a(i,j) = fcn(i,j)
    end do
    end do

    send MASTER results

endif
```

https://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch
calculations on 2-dimensional array elements: solution 2

• Solution1: demonstrated **static load** balancing:
  – Each task has a fixed amount of work to do
  – May be significant idle time for faster or more lightly loaded processors - **slowest tasks determines overall performance**.

• If you have a **load balance problem** (some tasks work faster than others),
  – you may benefit by using a "**pool of tasks**" scheme.

https://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch
calculations on 2-dimensional array elements: solution 2

• Master Process:
  – Holds pool of tasks for worker processes to do
  – Sends worker a task when requested
  – Collects results from workers

• Worker Process: repeatedly does the following
  – Gets task from master process
  – Performs computation
  – Sends results to master
calculations on 2-dimensional array elements: solution 2

find out if I am MASTER or WORKER

if I am MASTER

  do until no more jobs
    if request send to WORKER next job
    else receive results from WORKER
  end do

else if I am WORKER

  do until no more jobs
    request job from MASTER
    receive from MASTER next job

    calculate array element: \( a(i,j) = fcn(i,j) \)

    send results to MASTER
  end do
endif
References

1. Introduction to Parallel Computing
   https://computing.llnl.gov/tutorials/parallel_comp/\#MemoryArch
2. Intro to Parallel Programming. Lesson 2, pt. 1- Shared Memory and threads
   http://www.youtube.com/watch?v=6sL4C2SwszM
3. Intro to Parallel Programming. Lesson 2, pt. 2- Shared Memory and threads
   http://www.youtube.com/watch?v=ydG8cOzJjLA
4. Intro to Parallel Programming. Lesson 2, pt. 3- Shared Memory and threads
   http://www.youtube.com/watch?v=403LWbrA5oU
How to score 6 EC?

Core modules

• Lectures (6 hours)
  – Introduction to distributed systems
  – BigData
• Introduction to Linux (3.30 hours)

Workshops

• openMP / MPI (4 hours)
• Hadoop (8 hours)
• HPC Cloud (8 hours)
• Local and Remote visualization (4 hours)
• GPU
How to score 6 EC?

Grading

• Literature study: read 2 papers and summarize

• Following workshops you will have to do a assignment (none-supervised assignment)
  – Hadoop
  – HPCCloud
  – Local and Remote visualization
  – GPU programming
  – MPI/OpenMP
## TODO (for Students)

<table>
<thead>
<tr>
<th>Topics</th>
<th>Organizers</th>
<th>Type/duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intro to distributed sys &amp; BigData</td>
<td>(Adam Belloum, UvA)</td>
<td>Lectures/6 hours</td>
</tr>
</tbody>
</table>
