1 Introduction

1.1 Why Use Parallel Computing?

A simple computational problem:
A professor and his n assistants have to calculate the dot product of two vectors $x,\ y \in \mathbb{R}^n$ $\vec{x} \cdot \vec{y} = \sum_{i=1}^{n} x_i y_i$ by hand.

A parallel solution:
Professor sets $p = 0$. Assistant #i calculates the product $x_i y_i$ and when ready forwards the result to professor who adds it to p. When all assistants are forwarded their products to the professor and professor has finished his addition task the problem is solved.

Reasons for using parallel computing

- Save (wall clock) time $\implies$ Solve larger problems.
- Overcome memory constraints of single computers by using the memories of multiple computers $\implies$ Solve larger problems.
- Do multiple things at the same time (concurrency).
- Take advantage of computational resources available on a local network (or Internet).
- Use multiple ”cheap” computing resources instead of an expensive supercomputer.

Flyygelin pakkaaminen sievästi sataan kenkälaatikkoon jätetään toisten ongelmaksi...

Classical von Neumann Architecture

- The CPU executes a stored program that specifies a sequence of read and write operations on the memory.
- Basic design:

- Memory is used to store both program and data instructions
- Program instructions are coded data which tell the computer to do something
- CPU gets instructions and/or data from memory, decodes the instructions and then *sequentially* performs them.

Physical and practical limits to serial computers

- Transmission speeds: Absolute limits are the speed of light (30 cm/ns) and the transmission limit of copper wire (9 cm/ns). Increasing speeds means increasing proximity of processing elements.
- Limits to miniaturization: Even with molecular or atomic-level components, a limit will be reached on how small components can be.
- Economic limitations: It is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive ("multicores").

1.2 Flynn's Classical Taxonomy

Flynn's Classical Taxonomy

- Widely used classifications of parallel computers (M.J. Flynn, 1966).
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of *Instruction* and *Data*.
- Each of these dimensions has one of two possible states: *Single* or *Multiple* resulting four possible classifications:

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Single Instruction, Single Data (SISD)

- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- Examples: most PCs (before 2005), single CPU workstations and mainframes

---

Single Instruction, Multiple Data (SIMD)

- Single instruction: All processing units execute the same instruction at any given clock cycle.
- Multiple data: Each processing unit can operate on a different data element.
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing, dense linear algebra,...
- Synchronous and deterministic execution.

Multiple Instruction, Multiple Data (MIMD)

- Currently, the most common type of parallel computer.
- Multiple Instruction: every processor may be executing a different instruction stream.
- Multiple Data: every processor may be working with a different data stream.
- Execution can be synchronous or asynchronous, deterministic or non-deterministic.
- Examples: most current supercomputers, parallel computers formed by cluster of PCs, "grids",...

Multiple Instruction, Single Data (MISD)

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- Few actual examples of MISD computer have ever existed?
- Applications? Multiple cryptography algorithms attempting to crack a single coded message?

1.3 Some parallel computing jargon...

Jargon

Task: A logically discrete section of computational work. (Typically a program or program-like set of instructions.)

Parallel Task: A task that can be executed by multiple processors safely (yields correct results)

Serial Execution: Execution of a program sequentially, one statement at a time.

Parallel Execution: Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.

Jargon...

Shared Memory: From a *hardware point of view*, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory.

From a *programmer's point of view*, describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

Jargon...

Distributed Memory: From a *hardware point of view*, refers to network based memory access for physical memory that is not common.

From a *programmer's point of view*, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

Jargon...

Communications: Parallel tasks typically need to exchange data. The actual event of data exchange is commonly referred to as communications regardless of the technical details of the method employed.

Synchronization: The coordination of parallel tasks in real time. Implemented by establishing a synchronization point where a task may not proceed further until another task(s) reaches the same point.

Synchronization usually involves waiting by at least one task (overhead!).

Jargon...

Granularity: In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

- Coarse: relatively large amounts of computational work are done between communication events
- Fine: relatively small amounts of computational work are done between communication events

---

Jargon...

Speedup: Refers to how much a parallel algorithm to solve a problem is faster than (the best) sequential algorithm to solve the same problem.

It is defined as: $s = W_S/W_P$, where W_S = wall-clock time of execution of serial algorithm W_P = wall-clock time of execution of parallel algorithm

---

Jargon...

Observed Speedup: Observed speedup of a code which has been parallelized is defined as: $s = W_S/W_P$, where W_S = wall-clock time of serial execution W_P = wall-clock time of parallel execution

One of the simplest and most widely used indicators for a parallel program's performance.

Jargon...

Amdahl's Law: quantifies the potential speedup from converting serial code to parallel. Let σ be the code that is inherently serial, p the fraction that can be parallelized $p + \sigma = 1$, and N the number of processors. The speedup is then

$$s = \frac{1}{\sigma + p/N}.$$

The maximum speedup achived is thus $s = 1/\sigma$.

Jargon...

Parallel Overhead: The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:

- Task start-up and termination times
- Data communications
- Synchronizations
- Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.

Jargon...

Massively Parallel: Refers to the hardware that comprises a given parallel system having very many processors. (Very many = ??)

Grid Computing: "A service for sharing computer power and data storage capacity over the Internet." (CERN)

Ian Foster's checklist: "Grid" 1. coordinates resources that are not subject to centralized control

2. uses standard, open, general-purpose protocols and interfaces

3. Deliver nontrivial qualities of service.

Jargon...

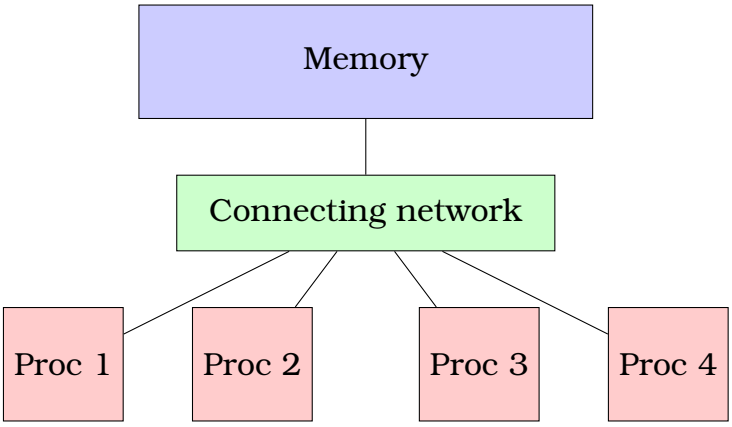
Scalability: Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:

- Characteristics of your specific application, solution algorithm, and coding
- Hardware - particularly memory-cpu bandwidths and network communications
- Parallel overhead related

2 Parallel Computer Memory Architectures

2.1 Shared memory architecture

Shared memory



Shared memory

- Shared memory parallel computers (*Symmetric Multiprocessors, SMP*) have the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: *Uniform Memory Access (UMA)* and *Non-Uniform Memory Access (NUMA)*.

Shared memory

Advantages:

- Global address space provides a user-friendly programming perspective to memory.
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.

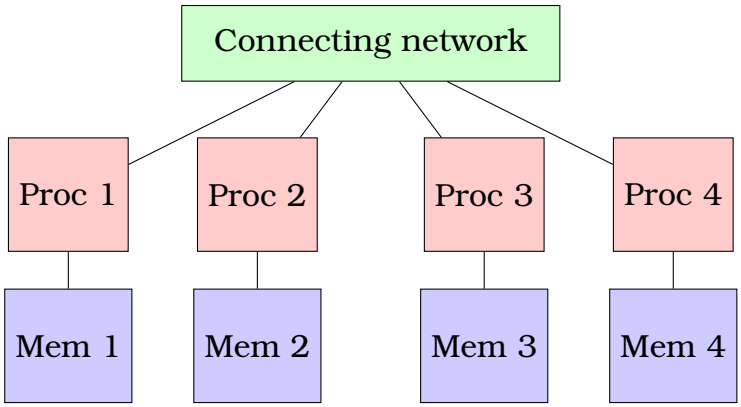
Shared memory

Disadvantages:

- Lack of scalability between memory and CPUs. Adding more CPUs can exponentially increase traffic on the shared memory–CPU path.
- Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
- It becomes increasingly difficult and expensive to design and produce shared memory machines with increasing numbers of processors.

2.2 Distributed memory architecture

Distributed memory architecture



Distributed memory architecture

- *Distributed memory systems* require a communication network to connect inter-processor memory.
- Processors have their own local memory. There is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors.

Distributed memory architecture

- When a processor needs access to data in another processor, it is the task of the programmer to explicitly define how and when data is communicated.
- Synchronization between tasks is the programmer's responsibility, too.
- The technology (and the cost!) to realize the network used for data transfer varies.

Distributed memory architecture

Advantages:

- Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference.
- Cost effectiveness: standard processors (PCs) and networking (Ethernet) can be used.

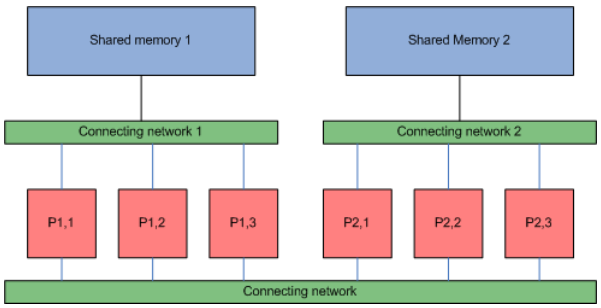
Distributed memory architecture

Disadvantages:

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to distributed memory.
- Non-uniform memory access times.

2.3 Hybrid memory architecture

Hybrid memory architecture



- The most powerful supercomputers employ both shared and distributed memory architectures.
- Advantages and disadvantages: whatever is common to both architectures.

3 Parallel Programming Models

3.1 Overview

Overview of parallel programming models

- Parallel programming models exist as an *abstraction* above hardware and memory architectures.

  Most commonly used parallel programming models:

  - Shared Memory
  - Threads
  - Message Passing
  - Data Parallel
  - Hybrid

Overview of parallel programming models

- PP models are *not* specific to a particular type of machine or memory architecture.
- Any of these models can (theoretically) be implemented on any underlying hardware.
- Examples: “Virtual shared memory” on a distributed memory machine, Message passing model on a shared memory machine.
- There is no ”best” model, although there certainly are better implementations of some models over others.

3.2 Shared Memory Model

Shared memory model

- Tasks share a common address space, which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- No notion of “data ownership” $\Longrightarrow$ the programmer does not need to specify explicitly the communication of data between tasks.
- On shared memory platforms, the native compilers translate user program variables into actual memory addresses, which are global.
- No portable shared memory implementations for distributed memory platform currently exist.

3.3 Threads Model

Threads model

- In the threads model of parallel programming, a single process "`a.out`" can have multiple, concurrent execution paths.
- Each thread has local data, but also, shares the entire resources of `a.out`.
- Saves the overhead associated with replicating a program's resources for each thread.
- Each thread also benefits from a global memory view because it shares the memory space of `a.out`.

Threads model

- Threads communicate with each other through global memory.
- Synchronization constructs are required to insure that more than one thread is not updating the same global address at any time.
- Threads are commonly associated with shared memory architectures and operating systems.

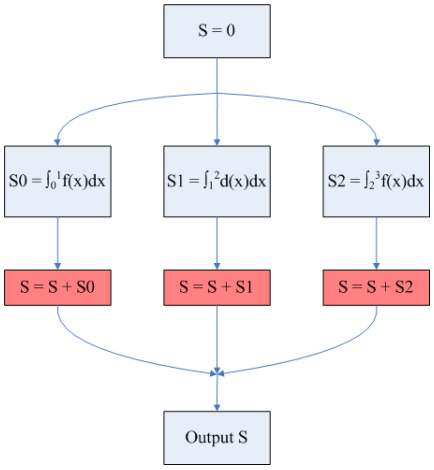
Threads – implementations

- From a programming perspective, threads implementations commonly comprise: A library of subroutines and/or a set of compiler directives. The programmer is responsible for determining all parallelism.
- Threaded implementations are not new in computing. Hardware vendors have implemented their own proprietary versions of threads.
- Unrelated standardization efforts have resulted in two very different implementations of threads: *POSIX Threads* and *OpenMP*.

POSIX Threads ("Pthreads")

- IEEE standard, 1995
- Library based; requires parallel coding
- C Language only (UNIX)
- Very explicit parallelism; requires significant programmer attention to detail.
- Too low level for most scientific computing applications ?

Threads example - numerical integration



Pthreads example

```
#include <pthread.h>
#define NUM_THREADS 3
pthread_mutex_t reduction_mutex; pthread_t *tid;
double result=0.0;

double f(double x) { return 1/(2+sin(x));}

double simpson( double a, double b )
{ double h3, ab;
  h3 = (b-a)/6; ab = 0.5*(a+b);
  return h3*(f(a)+4*f(ab)+f(b));}

void *worker( void *arg )
{
  int my_id;
  double my_result;

  my_id = *((int*)arg);
  my_result=simpson((double)my_id,(double)my_id+1.0);
  printf("Worker #%d got %lf \n", my_id, my_result);

  pthread_mutex_lock( &reduction_mutex );
  result += my_result;
  pthread_mutex_unlock( &reduction_mutex );
}

main()
{
  pthread_t tid[NUM_THREADS];
  int i, t_num[NUM_THREADS]={0,1,2};
```

```
  pthread_mutex_init( &reduction_mutex, NULL );

  for ( i=0; i<NUM_THREADS; i++ )
    pthread_create( &tid[i], NULL,
                    worker, (void*)&t_num[i] );

  for ( i=0; i<NUM_THREADS; i++ )
    pthread_join( tid[i], NULL );

  printf("Approximate integral = %lf\n", result);
  return(0);
}
```

OpenMP

- Compiler directive based; can use serial code
- Portable (Unix, Windows,...)
- Available in C/C++ and Fortran languages.
- *Can* be very easy and simple to use.
- Enables ”incremental parallelism”.

3.4 Message Passing Model

Message Passing Model

The message passing model demonstrates the following characteristics:

- A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process e.g. SEND/RECEIVE.

Message Passing Model

Implementations:

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are imbedded in source code.
- The programmer is responsible for determining all parallelism.

- A variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.

Message Passing Model – MPI

- In 1992, the MPI Forum was formed to establish a standard interface for message passing implementations. MPI-1 was released in 1994.
- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. Most parallel computing platforms offer an implementation of MPI.
- For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons.

3.5 Data Parallel Model

Data Parallel Model

- Most of the parallel work focuses on performing operations on a data set organized into a common structure, such as an array or cube.
- A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
- Tasks perform the same operation on their partition: e.g. `A(1:n,1:n)=A(1:n,1:n)+3`.
- On shared memory architectures, all tasks may have access to the data structure through global memory. On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task.

Data Parallel Model - HPF

- High Performance Fortran (HPF): Extensions to Fortran 95 to support data parallel programming.
- Directives to tell compiler how to distribute data etc. added.

```
PROGRAM hpf_example
  IMPLICIT NONE
  REAL, DIMENSION(100,100)    :: a, b, c
!HPF$ PROCESSORS, DIMENSION(2,2) :: P
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO p :: a, b, c
  b = 1; c = 1; a = b + c
END PROGRAM hpf_example
```

3.6 Other Models

Hybrid model(s)

- Two or more parallel programming models are combined.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) the shared memory model (OpenMP).
- Suits well to the increasingly common hardware environment of networked SMP machines.

Single Program Multiple Data (SPMD)

- A ”high level” programming model that can be built upon any combination of the previously mentioned parallel programming models.
- A single program is executed by all tasks simultaneously.
- At any moment in time, tasks can be executing the same or different instructions within the same program.
- Tasks do not necessarily have to execute the entire program, only their “own” portion of it.
- All tasks may use different data.

Multiple Program Multiple Data (MPMD)

- A ”high level” programming model that can be built upon any combination of the previously mentioned parallel programming models.
- MPMD applications typically have multiple executable object files (programs).
- While the application is being run in parallel, each task can be executing the same or different program as other tasks.
- All tasks may use different data.