Advanced Algorithmics (4AP)  
Parallel algorithms

Jaak Vilo  
2009 Spring

Slides based on materials from

• Joe Davey
• Matt Maul
• Ashok Srinivasan
• Edward Chrzanoski
• Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar
• Dr. Amitava Datta and Prof. Dr. Thomas Ottmann
• http://en.wikipedia.org/wiki/Parallel_computing
• ... and many others

TU HPC

• 42 nodes, 2x4-core = 336 core
• 32GB RAM / node
• Infiniband fast interconnect
• http://www.hpc.ut.ee/
• Job scheduling (Torque)
• MPI

Textbooks

### EXCS - krokodill
- 32-core
- 256GB RAM

### Drivers for parallel computing
- Multi-core processors (2, 4, …, 64 …)
- Computer clusters (and NUMA)
- Specialised computers (e.g. vector processors, massively parallel, …)
- Distributed computing: GRID, cloud, …
- Need to create computer systems from cheaper (and weaker) components, but many …
- One of the major challenges of modern IT

### What is a Parallel Algorithm?
- Imagine you needed to find a lost child in the woods.
- Even in a small area, searching by yourself would be very time consuming
- Now if you gathered some friends and family to help you, you could cover the woods in much faster manner…

### Parallel complexity
- Schedule on p processors
- schedule depth – max /longest path in schedule/  
- N – set of nodes on DAG
- time ti – assigned for each DAG node
- \[ T_p(n) = \min_{i \in N} \{ \max_{i \in N} t_i \} \]
- Length of the longest (critical) path!
Bit-level parallelism

- Adding of 64-bit numbers on 8-bit architecture...
- Sum, carry over, sum, carry over, ...

Boolean circuits

- (Micro)processors
- Basic building blocks,
- bit-level operations
- combine into word-level operations

Boolean circuits diagrams

- 8-bit adder
- Circuit complexity
- Longest path (= time)
- Nr or elements (= size)
Most of the available algorithms to compute \( \pi \), on the other hand, can not be easily split up into parallel portions. They require the results from a preceding step to effectively carry on with the next step. Such problems are called inherently serial problems.

**Instruction-level parallelism**

A canonical five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

A five-stage pipelined **superscalar processor**, capable of issuing two instructions per cycle. It can have two instructions in each stage of the pipeline, for a total of up to 10 instructions (shown in green) being simultaneously executed.

**HPC: 2.5GHz, 4-op-parallel**

**Background**

- Terminology
  - Time complexity
  - Speedup
  - Efficiency
  - Scalability
- Communication cost model

**Time complexity**

- Parallel computation
  - A group of processors work together to solve a problem
  - Time required for the computation is the period from when the first processor starts working until when the last processor stops

Sequential  Parallel - bad  Parallel - ideal  Parallel - realistic
Other terminology

- Speedup: \( S = \frac{T_1}{T_P} \)
- Efficiency: \( E = \frac{S}{P} \)
- Work: \( W = P T_P \)
- Scalability
  - How does \( T_P \) decrease as we increase \( P \) to solve the same problem?
  - How should the problem size increase with \( P \) to keep \( E \) constant?

- Sometimes a speedup of more than \( N \) when using \( N \) processors is observed in parallel computing, which is called super linear speedup. Super linear speedup rarely happens and often confuses beginners, who believe the theoretical maximum speedup should be \( N \) when \( N \) processors are used.

One possible reason for a super linear speedup is the cache effect

Super linear speedups can also occur when performing backtracking in parallel: One thread can prune a branch of the exhaustive search that another thread would have taken otherwise.

- e.g. 5% non parallelizable => speedup can not be more than 20x!

http://en.wikipedia.org/wiki/Amdahl%27s_law

Amdahl law

Is used to find the maximum expected improvement to an overall system when only part of the system is improved

Sequential programme

<table>
<thead>
<tr>
<th>Two independent parts</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Make</td>
<td>5x faster</td>
<td></td>
</tr>
<tr>
<td>Make</td>
<td>2x faster</td>
<td></td>
</tr>
</tbody>
</table>
Communication cost model

- Processes spend some time doing useful work, and some time communicating
- Model communication cost as
  - $T_c = t_s + L \
  - $L$ time
  - Independent of location of processes
  - Any process can communicate with any other process
  - A process can simultaneously send and receive one message

Simple model for analyzing algorithm performance:

$$t_{\text{comm}} = t_{\text{startup}} + (\text{words} \times t_{\text{data}})$$

$$= \text{latency} + (\text{words} \times 1/(\text{words/second}))$$

$$= \alpha + w \beta$$

I/O model

- We will ignore I/O issues, for the most part
- We will assume that input and output are distributed across the processors in a manner of our choosing
- Example: Sorting
  - Input: $x_1, x_2, \ldots, x_n$
  - Initially, $x_i$ is on processor $i$
  - Output $x_{p1}, x_{p2}, \ldots, x_{pn}$
  - $x_i$ on processor $i$
  - $x_i < x_{i+1}$

Important points

- Efficiency
  - Increases with increase in problem size
  - Decreases with increase in number of processors
- Aggregation of tasks to increase granularity
  - Reduces communication overhead
- Data distribution
  - 2-dimensional may be more scalable than 1-dimensional
  - Has an effect on load balance too
- General techniques
  - Divide and conquer
  - Pipelining

Parallel Architectures

- Single Instruction Stream, Multiple Data Stream (SIMD)
  - One global control unit connected to each processor
- Multiple Instruction Stream, Multiple Data Stream (MIMD)
  - Each processor has a local control unit

Architecture (continued)

- Shared-Address-Space
  - Each processor has access to main memory
  - Processors may be given a small private memory for local variables
- Message-Passing
  - Each processor is given its own block of memory
  - Processors communicate by passing messages directly instead of modifying memory locations
MPP – Massively Parallel

• Each node is an independent system having its own:
  – Physical memory
  – Address space
  – Local disk and network connections
  – Operating system

SMP - Symmetric multiprocessing

• Shared Memory
  – All processes share the same address space
  – Easy to program; also easy to program poorly
  – Performance is hardware dependent; limited memory bandwidth can create contention for memory

• MIMD (multiple instruction multiple data)
  – Each parallel computing unit has an instruction thread
  – Each processor has local memory
  – Processors share data by message passing
  – Synchronization must be explicitly programmed into a code

MPP

• Short for *massively parallel processing*, a type of computing that uses many separate CPUs running in parallel to execute a single program. MPP is similar to symmetric multiprocessing (SMP), with the main difference being that in SMP systems all the CPUs share the same memory, whereas in MPP systems, each CPU has its own memory. MPP systems are therefore more difficult to program because the application must be divided in such a way that all the executing segments can communicate with each other. On the other hand, MPP don't suffer from the bottleneck problems inherent in SMP systems when all the CPUs attempt to access the same memory at once.

Interconnection Networks

• Static
  – Each processor is hard-wired to every other processor

• Dynamic
  – Processors are connected to a series of switches

Why Do Parallel Computing

• Time: Reduce the turnaround time of applications
• Performance: Parallel computing is the only way to extend performance toward the TFLOP realm
• Cost/Performance: Traditional vector computers become too expensive as one pushes the performance barrier
• Memory: Applications often require memory that goes beyond that addressable by a single processor
Whole classes of important algorithms are ideal for parallel execution. Most algorithms can benefit from parallel processing such as Laplace equation, Monte Carlo, FFT (signal processing), image processing. Life itself is a set of concurrent processes. Scientists use modelling so why not model systems in a way closer to nature.

Some Misconceptions

Requires new parallel languages?
- No. Uses standard languages and compilers (Fortran, C, C++, Java, Occam)
- However, there are some specific parallel languages such as Qlisp, Multi-T and others – check out:

Requires new code?
- No. Most existing code can be used. Many production installations use the same code base for serial and parallel code.

Requires confusing parallel extensions?
- No. They are not that bad. Depends on how complex you want to make it. From nothing at all (letting the compiler do the parallelism) to installing semaphores yourself.

Parallel computing is difficult:
- No. Just different and subtle. Can be akin to assembler language programming 😊

Parallel Computing Architectures

Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Single Instruction Stream</th>
<th>Single Data Stream</th>
<th>Multiple Data Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD uniprocessors</td>
<td>SIMD Processor arrays</td>
<td></td>
</tr>
<tr>
<td>MISP Systolic arrays</td>
<td>MIMD Multicomputers</td>
<td></td>
</tr>
</tbody>
</table>

Memory Model

<table>
<thead>
<tr>
<th></th>
<th>Shared Address Space</th>
<th>Individual Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>SMP (Symmetric Multiprocessor)</td>
<td>N/A</td>
</tr>
<tr>
<td>Distributed</td>
<td>NUMA (Non-Uniform Memory Access)</td>
<td>MPP (Massively Parallel Processors)</td>
</tr>
</tbody>
</table>

NUMA architecture
The PRAM Model

- Parallel Random Access Machine
  - Theoretical model for parallel machines
  - p processors with uniform access to a large memory bank
  - MIMD
  - UMA (uniform memory access) – Equal memory access time for any processor to any address

Memory Protocols

- Exclusive-Read Exclusive-Write
- Exclusive-Read Concurrent-Write
- Concurrent-Read Exclusive-Write
- Concurrent-Read Concurrent-Write

Example: Finding the Largest key in an array

- In order to find the largest key in an array of size n, at least n-1 comparisons must be done.
- A parallel version of this algorithm will still perform the same amount of compares, but by doing them in parallel it will finish sooner.

Example: Finding the Largest key in an array

- Assume that n is a power of 2 and we have n / 2 processors executing the algorithm in parallel.
- Each Processor reads two array elements into local variables called first and second
- It then writes the larger value into the first of the array slots that it has to read.
- Takes \( \log n \) steps for the largest key to be placed in \( S[1] \)
Example: Finding the Largest key in an array

Example: Merge Sort

Merge Sort Analysis

- Number of compares
  - $1 + 3 + \ldots + (2^i - 1) + \ldots + (n-1)$
  - $\sum_{i=1}^{\log_2 n} 2^i - 1 = 2n - 2\log_n n = \Theta(n)$
- We have improved from $\log(n)$ to $n$ simply by applying the old algorithm to parallel computing, by altering the algorithm we can further improve merge sort to $(\log n)^2$

Parallel Design and Dynamic Programming

- Often in a dynamic programming algorithm a given row or diagonal can be computed simultaneously
- This makes many dynamic programming algorithms amenable for parallel architectures

Current Applications that Utilize Parallel Architectures

- Computer Graphics Processing
- Video Encoding
- Accurate weather forecasting
- Scientific computing, modelling
- …
Parallel addition features

- If \( n \gg P \)
  - Each processor adds \( n/P \) distinct numbers
  - Perform parallel reduction on \( P \) numbers
  - \( T_P = nP + (1 + t_s + t_b) \log P \)
  - Optimal \( P \) obtained by differentiating wrt \( P \)
    - \( P_{opt} \approx n/(1 + t_s + t_b) \)
  - If communication cost is high, then fewer processors ought to be used
    - \( E = \left[ 1 + (1 + t_s + t_b) P \log P/n \right]^{-1} \)
      - As problem size increases, efficiency increases
      - As number of processors increases, efficiency decreases

Some common collective operations

- **Broadcast**
  - \( T \approx (t_s + L t_b) \log P \)
    - \( L \): Length of data

- **Gather/Scatter**
  - Gather: Data move towards the root
  - Scatter: Review question
  - \( T \approx t_s \log P + P L t_b \)

- **All gather**
  - Equivalent to each processor broadcasting to all the processors
All gather

Matrix-vector multiplication

- \( c = A \cdot b \)
  - Often performed repeatedly
  - \( b_i = A \cdot b_{i-1} \)
  - We need same data distribution for \( c \) and \( b \)
- One dimensional decomposition
  - Example: row-wise block striped for \( A \)
  - \( b \) and \( c \) replicated
  - Each process computes its components of \( c \) independently
  - Then all-gather the components of \( c \)

1-D matrix-vector multiplication

- Each process computes its components of \( c \) independently
  - Time = \( \Theta(n^2/P) \)
- Then all-gather the components of \( c \)
  - Time = \( t_s \log P + P L t_b \)
- Note: \( P \leq n \)

2-D matrix-vector multiplication

- Processes \( P_0 \) sends \( B_0 \) to \( P_0 \)
  - Time: \( t_s \cdot n^{P/0.5} \)
- Processes \( P_0 \) broadcast \( B_0 \) to all \( P_i \)
  - Time = \( t_s \cdot \log P^{0.5} + t_b \cdot n \cdot \log P^{0.5} / P^{0.5} \)
- Processes \( P_0 \) compute \( c_0 = A \cdot B_0 \)
  - Time = \( \Theta(n^2/P) \)
- Processes \( P_i \) reduce \( C_i \) on to \( P_0, 0 \leq i \leq P^{0.5} \)
  - Time = \( t_s \cdot \log P^{0.5} + t_b \cdot n \cdot \log P^{0.5} / P^{0.5} \)
- Total time = \( \Theta(n^2/P + t_s \cdot \log P + t_b \cdot n \cdot \log P / P^{0.5}) \)
  - \( P \leq n^{0.5} \)
  - More scalable than 1-dimensional decomposition
31.2 Strassen’s algorithm for matrix multiplication

This section presents Strassen’s remarkable recursive algorithm for multiplying \( n \times n \) matrices that runs in \( (n^{\log_2 7}) = O(n^{2.81}) \) time. For sufficiently large \( n \), therefore, it outperforms the naive \( (n^3) \) matrix-multiplication algorithm \textsc{Matrix-Multiply} from Section 26.1.

\[ C = AB \]

\[
\begin{pmatrix}
  r & s \\
  t & u
\end{pmatrix} =
\begin{pmatrix}
  a & b \\
  c & d
\end{pmatrix}
\begin{pmatrix}
  e & g \\
  f & h
\end{pmatrix}
\]

- \( r = ae + bf \)
- \( s = ag + bh \)
- \( t = ce + df \)
- \( u = cg + dh \)
- \( T(n) = 8T(n/2) + O(n^2) \) = \( O(n^3) \)

Matrix operations are parallelizable

- Problems that are expressed in forms of matrix operations are often easy to automatically parallelise
- Fortran, etc – programming languages can achieve that at no extra effort

The Process

- A running executable of a (compiled and linked) program written in a standard sequential language (i.e. F77 or C) with library calls to implement the message passing
- A process executes on a processor
- All processes are assigned to processors in a one-to-one mapping (simple model of parallel programming)
- Other processes may execute on other processors
- A process communicates and synchronizes with other processes via messages
- A process is uniquely identified by:
  - The node on which it is running
  - Its process id (PID)
- A process does not migrate from node to node (though it is possible for it to migrate from one processor to another within a SMP node).
Processors vs. Nodes

Once upon a time...
- When distributed-memory systems were first developed, each computing element was referred to as a node.
- A node consisted of a processor, memory, (maybe I/O), and some mechanism by which it attached itself to the interprocessor communication facility (switch, mesh, torus, hypercube, etc.).
- The terms processor and node were used interchangeably.
- But lately, nodes have grown fatter...
- Multi-processor nodes are common and getting larger.
- Nodes and processors are no longer the same thing as far as parallel computing is concerned.
- Old habits die hard.
- It is better to ignore the underlying hardware and refer to the elements of a parallel program as processes or (more formally) as MPI tasks.

Solving Problems in Parallel

It is true that the hardware defines the parallel computer. However, it is the software that makes it usable.
Parallel programmers have the same concern as any other programmer:
- Algorithm design,
- Efficiency,
- Debugging ease,
- Code reuse, and
- Lifecycle.

Choose Wisely

- How do I select the right parallel computing model/language/libraries to use when I am writing a program?
- How do I make it efficient?
- How do I save time and reuse existing code?

Fosters Four step Process for Designing Parallel Algorithms

1. Partitioning – process of dividing the computation and the data into many small pieces – decomposition.
2. Communication – local and global (called overhead) minimizing parallel overhead is an important goal and the following check list should help the communication structure of the algorithm:
   - The communication operations are balanced among tasks.
   - Each task communicates with only a small number of neighbours.
   - Tasks can perform their communications concurrently.
   - Tasks can perform their computations concurrently.
3. Agglomeration is the process of grouping tasks into larger tasks in order to improve the performance or simplify programming. Often in using MPI this is one task per processor.
4. Mapping is the process of assigning tasks to processors with the goal to maximize processor utilization.

Cont.

However, they are also concerned with:
- Concurrency and communication
- Need for speed (nee high performance), and
- Plethora and diversity of architecture.

Cont…
Solving Problems in Parallel

- Decomposition determines:
  - Data structures
  - Communication topology
  - Communication protocols
- Must be looked at early in the process of application development
- Standard approaches

Decomposition methods

- Perfectly parallel
- Domain
- Control
- Object-oriented
- Hybrid/layered (multiple uses of the above)

For the program

- Choose a decomposition
  - Perfectly parallel, domain, control etc.
- Map the decomposition to the processors
- Ignore topology of the system interconnect
- Use natural topology of the problem
- Define the inter-process communication protocol
  - Specify the different types of messages which need to be sent
  - See if standard libraries efficiently support the proposed message patterns

Perfectly parallel

- Applications that require little or no inter-processor communication when running in parallel
- Easiest type of problem to decompose
- Results in nearly perfect speed-up

Domain decomposition

- In simulation and modelling this is the most common solution
  - The solution space (which often corresponds to the real space) is divided up among the processors. Each processor solves its own little piece
  - Finite-difference methods and finite-element methods lend themselves well to this approach
  - The method of solution often leads naturally to a set of simultaneous equations that can be solved by parallel matrix solvers
  - Sometimes the solution involves some kind of transformation of variables (i.e. Fourier Transform). Here the domain is some kind of phase space. The solution and the various transformations involved can be parallelized

Cont...

- Example: the client-server model
  - The server is an object that has data associated with it (i.e. a database) and a set of procedures that it performs (i.e. searches for requested data within the database)
  - The client is an object that has data associated with it (i.e. a subset of data that it has requested from the database) and a set of procedures it performs (i.e. some application that massages the data)
  - The server and client can run concurrently on different processors: an object-oriented decomposition of a parallel application
  - In the real-world, this can be large scale when many clients (workstations running applications) access a large central database - kind of like a distributed supercomputer
Output Data Decomposition: Example
Consider the problem of multiplying two \( n \times n \) matrices \( A \) and \( B \) to yield matrix \( C \). The output matrix \( C \) can be partitioned into four tasks as follows:

\[
\begin{pmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{pmatrix}
\begin{pmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{pmatrix}
= \begin{pmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{pmatrix}
\]

Task 1: \( C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \)
Task 2: \( C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \)
Task 3: \( C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \)
Task 4: \( C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \)

Control decomposition
- If you cannot find a good domain to decompose, your problem might lend itself to control decomposition
  - Good for:
    - Unpredictable workloads
    - Problems with no convenient static structures
    - One set of control decomposition is functional decomposition
    - Problem is viewed as a set of operations. It is among operations where parallelization is done
    - Many examples in industrial engineering (i.e. designing an assembly line, a chemical plant, etc.)
    - Many examples in data processing where a series of operations is performed on a continuous stream of data

Object-oriented decomposition
- Object-oriented decomposition is really a combination of functional and domain decomposition
  - Rather than thinking about a dividing data or functionality, we look at the objects in the problem
  - The object can be decomposed as a set of data structures plus the procedures that act on those data structures
  - The goal of object-oriented parallel programming is distributed objects
  - Although conceptually clear, in practice it can be difficult to achieve good load balancing among the objects without a great deal of fine tuning
  - Works best for fine-grained problems and in environments where having functionally ready-at-hand code is more important than worrying about under-worked processors (i.e. satisfied simulation)
  - Message passing is still explicit (no standard C++ compiler automatically parallelizes over objects)
Partitioning the Graph of Lake Superior

Random Partitioning

Partitioning for minimum edge-cut.

Decomposition summary

- A good decomposition strategy is
  - Key to potential application performance
  - Key to programmability of the solution
- There are many different ways of thinking about decomposition
  - Decomposition models (domain, control, object-oriented, etc.) provide standard templates for thinking about the decomposition of a problem
  - Decomposition should be natural to the problem rather than natural to the computer architecture
  - Communication does no useful work; keep it to a minimum
- Always wise to see if a library solution already exists for your problem
- Don't be afraid to use multiple decompositions in a problem if it seems to fit

Summary of Software

- Compilers
  - Moderate 10-20% parallelism
  - Size/complexity of parallelism
  - Platform has a parallelizing compiler
  - Some quality implementation exists on the platform
  - Not portable
- OpenMP
  - Some quality implementation exists on the platform
  - Not portable
  - Doesn't concern us with portability
  - Moderate (20-50?) parallelism
- MPI
  - Portable between platforms
  - Needs some type of message passing platform
  - An application parallelizing effort is required
  - Less MPI; conditions plus fault tolerance
- High Performance Fortran (HPF)
  - Needs some type of message passing platform
  - A substantive coding effort is required
  - All MPI conditions plus fault tolerance
  - Still provides better functionality in some settings
- P-Threads
  - High level libraries
  - Like OpenMP, but new language constructs provide a data parallel/implicit programming model
  - Not recommended
  - Difficult to extend and maintain program
  - Not suitable to large number of processors
  - POOMA and HPC++
  - Library is available and it addresses a specific problem