B4M35PAG - Paralelní algoritmy

https://cw.fel.cvut.cz/wiki/courses/b4m35pag/start

Přemysl Šůcha
email: premyssl.sucha@cvut.cz
homepage: http://people.ciirc.cvut.cz/suchap
Organization and Contents

**Introduction to Parallel Computing**

**Second Edition**

- **Fundamentals**
  - 1. Introduction to Parallel Computing
  - 2. Parallel Programming Platforms
  - 3. Principles of Parallel Algorithm Design
  - 4. Basic Communication Operations
  - 5. Analytical Modeling of Parallel Programs

- **Parallel Programming**
  - 6. Programming Using the Message Passing Paradigm
  - 7. Programming Shared Address Space Platforms
  - 8. Dense Matrix Algorithms
  - 9. Sorting
  - 10. Graph Algorithms
  - 12. Dynamic Programming
  - 13. Fast Fourier Transform

- **Non-Numerical Algorithms**

# Organization and Contents

<table>
<thead>
<tr>
<th>#</th>
<th>Title</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction to Parallel Computing</td>
<td>Chapter 2</td>
</tr>
<tr>
<td>2</td>
<td>Principles of Parallel Algorithms Design</td>
<td>Chapter 3</td>
</tr>
<tr>
<td>3</td>
<td>Basic Communication Operations</td>
<td>Chapter 4</td>
</tr>
<tr>
<td>4</td>
<td>Analytical Modeling of Parallel Algorithms</td>
<td>Chapter 5</td>
</tr>
<tr>
<td>5</td>
<td>Sorting</td>
<td>Chapter 9</td>
</tr>
<tr>
<td>6</td>
<td>Matrix Algorithms</td>
<td>Chapter 8</td>
</tr>
<tr>
<td>7</td>
<td>Algorithms for Linear Algebra</td>
<td>Chapter 8</td>
</tr>
<tr>
<td>8</td>
<td>Parallel Accelerators</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Graph Algorithms I.</td>
<td>Chapter 10</td>
</tr>
<tr>
<td>10</td>
<td>Graph Algorithms II, Test</td>
<td>Chapter 10</td>
</tr>
<tr>
<td>11</td>
<td>Combinatorial Algorithms</td>
<td>Chapter 11</td>
</tr>
<tr>
<td>12</td>
<td>Dynamic Programming</td>
<td>Chapter 12</td>
</tr>
<tr>
<td>13</td>
<td>Fast Fourier Transform</td>
<td>Chapter 13</td>
</tr>
</tbody>
</table>
## Motivation

- **TOP500** ([www.top500.org](http://www.top500.org)) - June 2018

<table>
<thead>
<tr>
<th>Rank</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Summit</strong> - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States</td>
<td>2,282,544</td>
<td>122,300.0</td>
<td>187,659.3</td>
<td>8,806</td>
</tr>
<tr>
<td>2</td>
<td><strong>Sunway TaihuLight</strong> - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China</td>
<td>10,649,600</td>
<td>93,014.6</td>
<td>125,435.9</td>
<td>15,371</td>
</tr>
<tr>
<td>3</td>
<td><strong>Sierra</strong> - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/NNSA/LLNL United States</td>
<td>1,572,480</td>
<td>71,610.0</td>
<td>119,193.6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><strong>Tianhe-2A</strong> - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China</td>
<td>4,981,760</td>
<td>61,444.5</td>
<td>100,678.7</td>
<td>18,482</td>
</tr>
<tr>
<td>5</td>
<td><strong>AI Bridging Cloud Infrastructure (ABCI)</strong> - PRIMERGY CX2550 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR, Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan</td>
<td>391,680</td>
<td>19,880.0</td>
<td>32,576.6</td>
<td>1,649</td>
</tr>
</tbody>
</table>
Recent Highlights in Parallel Computing

• In March 2016, AlphaGo beat Lee Sedol, a 9-dan professional. AlphaGo ran on 48 CPUs and 8 GPUs.

• In June 2016, Ford Using Deep Learning for Lane Detection - new sub-centimeter accurate approach to estimate a moving vehicle’s position within a lane in real-time
Parallel Computing Platforms

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

To accompany the text "Introduction to Parallel Computing", Addison Wesley, 2003.
Topic Overview

• Parallel Computing Platforms
• Communication Model of Parallel Platforms
• Physical Organization of Parallel Platforms
• Interconnection Networks for Parallel Computers
• Communication Costs in Parallel Machines
• Routing Mechanisms for Interconnection Networks
• Mapping Techniques
Parallel Computing Platforms

- An explicitly parallel program must specify concurrency and interaction between concurrent subtasks.
- The former is sometimes also referred to as the control structure and the latter as the communication model.
Control Structure of Parallel Programs

- Parallelism can be expressed at various levels of **granularity** - from instruction level to processes.
- Between these extremes exist a range of **models**, along with corresponding architectural support.
Control Structure of Parallel Programs

• Processing units in parallel computers either operate under the centralized control of a single control unit or work independently.

• If there is a single control unit that dispatches the same instruction to various processors (that work on different data), the model is referred to as single instruction stream, multiple data stream (SIMD).

• If each processor has its own control unit, each processor can execute different instructions on different data items. This model is called multiple instruction stream, multiple data stream (MIMD).
SIMD and MIMD Processors

A typical SIMD architecture (a) and a typical MIMD architecture (b).

A typical SIMD architecture (a) and a typical MIMD architecture (b).
SIMD Processors

• Variants of this concept have found use in co-processing units such as the MMX, SSE, AVX, … units in Intel processors and DSP chips such as the Sharc.

• SIMD relies on the regular structure of computations (such as those in image processing).

• It is often necessary to selectively turn off operations on certain data items. For this reason, most SIMD programming paradigms allow for an ``activity mask'', which determines if a processor should participate in a computation or not.
Conditional Execution in SIMD Processors

Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.
MIMD Processors

- In contrast to SIMD processors, MIMD processors can execute **different programs on different processors**.
- A variant of this, called **single program multiple data streams (SPMD)** executes the same program on different processors.
- It is easy to see that SPMD and MIMD are closely related in terms of programming flexibility and underlying architectural support.
- Single instruction, multiple thread (**SIMT**) is an execution model where SIMD is combined with multithreading.
SIMD-MIMD Comparison

- SIMD computers require less hardware than MIMD computers (single control unit).
- However, since SIMD processors are specially designed, they tend to be expensive and have long design cycles.
- Not all applications are naturally suited to SIMD processors.
- In contrast, platforms supporting the SPMD paradigm can be built from inexpensive off-the-shelf components with relatively little effort in a short amount of time.
Communication Model of Parallel Platforms

• There are **two primary forms of data exchange** between parallel tasks - accessing a shared data space and exchanging messages.

• Platforms that provide a shared data space are called **shared-address-space machines** or multiprocessors.

• Platforms that support messaging are also called **message passing platforms** or multicomputers.
Shared-Address-Space Platforms

- Part (or all) of the memory is accessible to all processors.
- Processors interact by modifying data objects stored in this shared-address-space.
- If the time taken by a processor to access any memory word in the system global or local is identical, the platform is classified as a uniform memory access (UMA), else, a non-uniform memory access (NUMA) machine.
NUMA and UMA Shared-Address-Space Platforms

Typical shared-address-space architectures: (a) Uniform-memory access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.
The distinction between NUMA and UMA platforms is important from the point of view of algorithm design. NUMA machines require locality from underlying algorithms for performance.

Programming these platforms is easier since reads and writes are implicitly visible to other processors.

However, read-write data to shared data must be coordinated (this will be discussed in greater detail when we talk about threads programming).

Caches in such machines require coordinated access to multiple copies. This leads to the cache coherence problem.
Message-Passing Platforms

- These platforms comprise of a set of processors and their own (exclusive) memory.

- Instances of such a view come naturally from clustered workstations and non-shared-address-space multicomputers.

- These platforms are programmed using (variants of) send and receive primitives.

- Libraries such as MPI and PVM provide such primitives.
Message Passing vs. Shared Address Space Platforms

- Message passing requires **little hardware support**, other than a network.
- **Shared address space platforms can easily emulate message passing.** The reverse is more difficult to do (in an efficient manner).
Physical Organization of Parallel Platforms

We begin this discussion with an ideal parallel machine called **Parallel Random Access Machine**, or **PRAM**.
Architecture of an Ideal Parallel Computer

• A natural **extension** of the Random Access Machine (RAM) serial architecture is the Parallel Random Access Machine, or PRAM.

• PRAMs consist of *p* **processors** and a **global memory of unbounded size** that is **uniformly accessible** to all processors.

• Processors share a **common clock** but may execute **different instructions** in each cycle.
Architecture of an Ideal Parallel Computer

- Depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.
  - Exclusive-read, exclusive-write (EREW) PRAM.
  - Concurrent-read, exclusive-write (CREW) PRAM.
  - Exclusive-read, concurrent-write (ERCW) PRAM.
  - Concurrent-read, concurrent-write (CRCW) PRAM.
Architecture of an Ideal Parallel Computer

• What does concurrent write mean, anyway?
  – Common: write only if all values are identical.
  – Arbitrary: write the data from a randomly selected processor.
  – Priority: follow a predetermined priority order.
  – Sum: Write the sum of all data items.
Interconnection Networks for Parallel Computers

- Interconnection networks carry **data between processors** and to memory.
- Interconnects are made of **switches** and **links** (wires, fiber).
- Interconnects are classified as static or dynamic.
- **Static networks** consist of point-to-point communication links among processing nodes and are also referred to as **direct** networks.
- **Dynamic networks** are built using switches and communication links. Dynamic networks are also referred to as **indirect** networks.
Network Topologies: Completely Connected Network

- Each processor is connected to every other processor.
- The **number of links** in the network scales as $O(p^2)$.
- While the **performance scales very well**, the hardware complexity is **not realizable** for large values of $p$.
- In this sense, these networks are **static counterparts of crossbars**.
Network Topologies: Completely Connected and Star Connected Networks

Example of an 8-node completely connected network.

(a) A completely-connected network of eight nodes; (b) a star connected network of nine nodes.
Network Topologies:  
Star Connected Network

- Every node is connected only to a common node at the center.
- **Distance** between any pair of nodes is $O(1)$. However, the central node becomes a bottleneck.
- In this sense, star connected networks are static counterparts of buses.
Network Topologies: Linear Arrays, Meshes, and $k$-$d$ Meshes

- In a **linear array**, each node has two neighbors, one to its left and one to its right. If the nodes at either end are connected, we refer to it as a **1-D torus** or a ring.
- A **generalization to 2 dimensions** has nodes with 4 neighbors, to the north, south, east, and west.
- A further generalization to $d$ dimensions has nodes with $2d$ neighbors.
- A special case of a $d$-dimensional mesh is a **hypercube**. Here, $d = \log p$, where $p$ is the total number of nodes.
Network Topologies: Linear Arrays

Linear arrays: (a) with no wraparound links; (b) with wraparound link.
Network Topologies:
Two- and Three Dimensional Meshes

Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.
Network Topologies: Hypercubes and their Construction

Construction of hypercubes from hypercubes of lower dimension.
Network Topologies: Properties of Hypercubes

- The **distance** between any two nodes is at most $\log p$.
- Each node has $\log p$ **neighbors**.
- The distance between two nodes is given by the number of bit positions at which the two nodes differ.
Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.
Evaluating Static Interconnection Networks

- **Diameter**: The distance between the farthest two nodes in the network. The diameter of a linear array is $p - 1$, that of a mesh is $2(\sqrt{p} - 1)$, that of a tree and hypercube is $\log p$, and that of a completely connected network is $O(1)$.

- **Bisection Width**: The minimum number of wires you must cut to divide the network into two equal parts. The bisection width of a linear array and tree is $1$, that of a mesh is $\sqrt{p}$, that of a hypercube is $p/2$ and that of a completely connected network is $p^2/4$.

- **Cost**: The number of links or switches (whichever is asymptotically higher) is a meaningful measure of the cost. **Arc Connectivity**: is the minimum number of arcs that must be removed from the network to break it into two disconnected networks.
# Evaluating Static Interconnection Networks

<table>
<thead>
<tr>
<th>Network</th>
<th>Diameter</th>
<th>Bisection Width</th>
<th>Arc Connectivity</th>
<th>Cost (No. of links)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completely-connected</td>
<td>1</td>
<td>(p^2/4)</td>
<td>(p - 1)</td>
<td>(p(p - 1)/2)</td>
</tr>
<tr>
<td>Star</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>(p - 1)</td>
</tr>
<tr>
<td>Complete binary tree</td>
<td>(2 \log((p + 1)/2))</td>
<td>1</td>
<td>1</td>
<td>(p - 1)</td>
</tr>
<tr>
<td>Linear array</td>
<td>(p - 1)</td>
<td>1</td>
<td>1</td>
<td>(p - 1)</td>
</tr>
<tr>
<td>2-D mesh, no wraparound</td>
<td>(2(\sqrt{p} - 1))</td>
<td>(\sqrt{p})</td>
<td>2</td>
<td>(2(p - \sqrt{p}))</td>
</tr>
<tr>
<td>2-D wraparound mesh</td>
<td>(2\lfloor\sqrt{p}/2\rfloor)</td>
<td>(2\sqrt{p})</td>
<td>4</td>
<td>(2p)</td>
</tr>
<tr>
<td>Hypercube</td>
<td>(\log p)</td>
<td>(p/2)</td>
<td>(\log p)</td>
<td>((p \log p)/2)</td>
</tr>
<tr>
<td>Wraparound k-ary d-cube</td>
<td>(d\lfloor k/2\rfloor)</td>
<td>(2k^{d-1})</td>
<td>2(d)</td>
<td>(dp)</td>
</tr>
</tbody>
</table>
Communication Costs in Parallel Machines

- Along with idling and contention, communication is a major overhead in parallel programs.
- The cost of communication is dependent on a variety of features including the programming model semantics, the network topology, data handling and routing, and associated software protocols.
• The **total time to transfer a message** over a network comprises of the following:
  
  – *Startup time* ($t_s$): Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).
  
  – *Per-hop time* ($t_h$): This time is a function of number of hops and includes factors such as **switch latencies**, network delays, etc.
  
  – *Per-word transfer time* ($t_w$): This time includes all **overheads that are determined by the length of the message**. This includes **bandwidth** of links, error checking and correction, etc.
Store-and-Forward Routing

- A message traversing multiple hops is completely **received** at an intermediate hop before being forwarded to the next hop.
- The total communication cost for a message of size $m$ words to traverse $l$ communication links is
  \[ t_{\text{comm}} = t_s + (mt_w + t_h)l. \]
- In most platforms, $t_h$ is small and the above expression can be approximated by
  \[ t_{\text{comm}} = t_s + mlt_w. \]
Routing Techniques

Passing a message from node \( P_0 \) to \( P_3 \) (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.
Cut-Through Routing

• Takes the concept of packet routing to an extreme by further dividing messages into basic units called **flits**.
• Since flits are typically **small**, the header information must be minimized.
• This is done by forcing all flits to take the **same path**, in sequence.
• A tracer message first programs all intermediate routers. All flits then take the same route.
• Error **checks are performed on the entire message**, as opposed to flits.
• No **sequence numbers** are needed.
Simplified Cost Model for Communicating Messages

• The cost of communicating a message between two nodes $l$ hops away using cut-through routing is given by

$$t_{comm} = t_s + lt_h + t_w m.$$ 

• In this expression, $t_h$ is typically smaller than $t_s$ and $t_w$. For this reason, the second term in the RHS does not show, particularly, when $m$ is large.

• Furthermore, it is often not possible to control routing and placement of tasks.

• For these reasons, we can approximate the cost of message transfer by

$$t_{comm} = t_s + t_w m.$$
Simplified Cost Model for Communicating Messages

• It is important to note that the original expression for communication time is valid for only **uncongested networks**.

• If a link takes multiple messages, the corresponding $t_w$ term **must be scaled up by the number of messages**.

• **Different communication patterns congest different networks to varying extents.**

• It is important to understand and account for this in the communication time accordingly.
Routing Mechanisms for Interconnection Networks

• How does one compute the route that a message takes from source to destination?
  – **Routing must prevent deadlocks** - for this reason, we use dimension-ordered or **e-cube routing**.
  – **Routing must avoid hot-spots** - for this reason, two-step routing is often used. In this case, a message from source $s$ to destination $d$ is first sent to a randomly chosen intermediate processor $i$ and then forwarded to destination $d$. 
Routing Mechanisms for Interconnection Networks

Routing a message from node $P_s$ (010) to node $P_d$ (111) in a three-dimensional hypercube using E-cube routing.
Mapping Techniques for Graphs

- Often, we need to embed a known communication pattern into a given interconnection topology.
- We may have an algorithm designed for one network, which we are porting to another topology.

For these reasons, it is useful to understand mapping between graphs.
Mapping Techniques for Graphs: Metrics

- When mapping a graph $G(V,E)$ into $G'(V',E')$, the following metrics are important:
- The maximum number of edges mapped onto any edge in $E'$ is called the *congestion* of the mapping.
- The maximum number of links in $E'$ that any edge in $E$ is mapped onto is called the *dilation* of the mapping.
- The ratio of the number of nodes in the set $V'$ to that in set $V$ is called the *expansion* of the mapping.
Embedding a Linear Array into a Hypercube

• A linear array (or a ring) composed of $2^d$ nodes (labeled 0 through $2^d - 1$) can be embedded into a $d$-dimensional hypercube by mapping node $i$ of the linear array onto node $G(i, d)$.

• $G(i, d)$ of the hypercube. The function $G(i, x)$ is defined as follows:

\[
G(0, 1) = 0 \\
G(1, 1) = 1 \\
G(i, x + 1) = \begin{cases} 
G(i, x), & i < 2^x \\
2^x + G(2^{x+1} - 1 - i, x), & i \geq 2^x 
\end{cases}
\]
Embedding a Linear Array into a Hypercube

The function $G$ is called the binary reflected Gray code (RGC).

Since adjoining entries ($G(i, d)$ and $G(i + 1, d)$) differ from each other at only one bit position, corresponding processors are mapped to neighbors in a hypercube. Therefore, the congestion, dilation, and expansion of the mapping are all 1.
Embedding a Linear Array into a Hypercube: Example

(a) A three-bit reflected Gray code ring; and (b) its embedding into a three-dimensional hypercube.
Embedding a Mesh into a Hypercube

A $2^r \times 2^s$ wraparound mesh can be mapped to a $2^{r+s}$ node hypercube by mapping node $(i, j)$ of the mesh onto node $G(i, r-1) || G(j, s-1)$ of the hypercube (where $||$ denotes concatenation of the two Gray codes).
Embedding a Mesh into a Hypercube

(a) A 4 × 4 mesh illustrating the mapping of mesh nodes to the nodes in a four-dimensional hypercube; and (b) a 2 × 4 mesh embedded into a three-dimensional hypercube.

Once again, the congestion, dilation, and expansion of the mapping is 1.
Embedding a Mesh into a Linear Array

• Since a **mesh has more edges than a linear array**, we will not have an optimal congestion/dilation mapping.
• We first examine the mapping of a linear array into a mesh and then invert this mapping.
• This gives us an optimal mapping (in terms of congestion).
Embedding a Mesh into a Linear Array: Example

(a) Embedding a 16 node linear array into a 2-D mesh (congestion 1).

(b) Inverting the mapping - mapping a 2D mesh into a linear array (congestion 5)

(a) Embedding a 16 node linear array into a 2-D mesh; and (b) the inverse of the mapping. Solid lines correspond to links in the linear array and normal lines to links in the mesh.