Advanced Algorithmics (6EAP)
Parallel algorithms

Jaak Vilo
2011 Spring
• $67 + 56 = ?$
\[
\text{SUM( 10 7 98 82 84 66 62 94 6 25 73 62 65 5 36 983 73 93 77 10 7 98 82 84 66 62 94 6 25 73 62 65 5 36 1 983 1 11 173 93 77 10 7 98 82 84 66 62 94 6 25 73 62 65 5 36 983 73 93 77 10 7 98 1 82 66 62 1 1 94 6 25 73 62 65 5 36 983 73 1 93 77 10 7 98 82 84 66 62 94 6 25 73 62 65 5 36 1 73 93 77 10 7 98 82 84 66 62 94 6 25 73 1 65 5 36 983 73 93 77 ) } = ?
\]
University of Illinois researchers have taken several small steps toward developing new parallel programming models to tap the many-core processors of the future. The DeNovo project attempts to define a new and more rigorous way of utilizing shared memory. It is working concurrently with a separate effort to define a deterministic, parallel language primarily based on a parallel version of Java and eventually migrating to a parallel version of C++. The chip project that is nearest to testing is the 1,024-core Rigel processor architecture targeting high density, high throughput computing, which would be programmed through a task-level applications programming interface aimed at chores in imaging, computer vision, physics, and simulations. The Bulk Architecture chip design is testing the notion of atomic transactions.
NASA, Air Force define cutting-edge next-generation space computer

The processor will be applicable to a broad variety of military and civil space missions

By Michael Cooney | Network World US | Published: 21:19, 11 April 2013

It’s not every day a customer wants a computer that can do everything from direct a landing on the surface of Mars, to control descent onto a speeding asteroid or coordinate a flight of satellites - but NASA and the US Air Force’s space group aren’t everyday customers.

NASA and the Air Force Research Laboratory’s Space Vehicles Directorate put out a call today for research and development of what likely will become the next computer system - what the agencies call the Next Generation Space Processor (NGSP) - to fly onboard a variety of future spacecraft.

The processor is envisioned to be a radiation-hardened, general-purpose multicore chip and associated software, applicable to a broad variety of military and civil space missions, and a broad range of spacecraft sizes and power/mass/volume constraints, NASA stated.
Titan has AMD Opteron CPUs in conjunction with Nvidia Tesla GPUs to improve energy efficiency while providing an order of magnitude increase in computational power over Jaguar. It uses 18,688 CPUs paired with an equal number of GPUs to perform at a theoretical peak of 27 petaFLOPS; however, in the LINPACK benchmark used to rank supercomputers' speed, it performed at 17.59 petaFLOPS. This was enough to take first place in the November 2012 list by the TOP500 organisation.
One can calculate FLOPS using this equation:[1]

\[
\text{FLOPS} = \text{chassis} \times \frac{\text{nodes}}{\text{chassis}} \times \frac{\text{sockets}}{\text{node}} \times \frac{\text{cores}}{\text{socket}} \times \text{clock} \times \frac{\text{FLOPs}}{\text{cycle}}
\]

or

\[
\text{FLOPS} = \text{cores} \times \text{clock} \times \frac{\text{FLOPs}}{\text{cycle}}
\]

Most microprocessors today can do 4 FLOPs per clock cycle. Therefore, a single-core 2.5 GHz processor has a theoretical performance of 10 billion FLOPS = 10 GFLOPS.

Note: In this context, *sockets* is referring to chip sockets on a motherboard, in other words, how many computer chips are in use, with each chip having one or more cores on it. This equation only applies to one very specific (but common) hardware architecture and it ignores limits imposed by memory bandwidth and other constraints. In general, GigaFLOPS are not determined by theoretical calculations such as this one; instead, they are measured by actual benchmarks of actual performance/throughput. Because this equation ignores all sources of overhead, in the real world, you will never get actual performance that is anywhere near to what this equation predicts.
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.

Instructions can be grouped together only if there is no **data dependency** between them.
TÜ HPC – v1, Feb 2, 2009

- 42 nodes, 2x4-core = 336 core
- 32GB RAM / node
- Infiniband fast interconnect

- [http://www.hpc.ut.ee/](http://www.hpc.ut.ee/)

- Job scheduling (Torque)
- MPI
Aurumasin IntiBandi võrgu skeem
<table>
<thead>
<tr>
<th></th>
<th>Single (SDR)</th>
<th>Double (DDR)</th>
<th>Quad (QDR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1X</td>
<td>2 Gbit/s</td>
<td>4 Gbit/s</td>
<td>8 Gbit/s</td>
</tr>
<tr>
<td>4X</td>
<td>8 Gbit/s</td>
<td>16 Gbit/s</td>
<td>32 Gbit/s</td>
</tr>
<tr>
<td>12X</td>
<td>24 Gbit/s</td>
<td>48 Gbit/s</td>
<td>96 Gbit/s</td>
</tr>
</tbody>
</table>

**Latency**
The single data rate switch chips have a latency of 200 nanoseconds, and DDR switch chips have a latency of 140 nanoseconds. The end-to-end latency range is from 1.07 microseconds MPI latency to 1.29 microseconds MPI latency to 2.6 microseconds.

**Compare:**
The speed of a normal ethernet ping/pong (request and response) is roughly 350us (microseconds) or about .35 milliseconds or .00035 seconds.
Anno 2013

- HPC: 2.5GHz, 4-op-parallel (v1.)

- **Vedur: klots1-klots80 (ProLiant DL165 G7):**
  - 2 x AMD Opteron(TM) Processor 6276 CPU (32 cores)
  - 48GB RAM
  - 500GB HDD
  - 4x QDR Infiniband

- $2.5E9 \times 80 \times 32 \times 4 = 2.56e+13$  \Rightarrow 25$ TFLOP?
EXCS - krokodill

- 32-core
- 256GB RAM
Update

• HPC: 512GB,

• ETAIS – Estonian scientific calculations infrastructure project
  – Many thousands of cores

• EXCS – Center of Excellence
  – To buy **8x10-core cpu, 1TB RAM** machines fall 2011
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
Slides based on materials from

- Joe Davey
- Matt Maul
- Ashok Srinivasan
- Edward Chrzanowski
- Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar
- Dr. Amitava Datta and Prof. Dr. Thomas Ottmann

- ... and many others
textbooks


• CLRS: (3rd edition) pp 772-812 (Chapter 27)
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...
A simple parallel algorithm

Adding n numbers in parallel
Terms (CLRS)

- work, span
Parallel complexity

- Schedule on $p$ processors
- Schedule depth – $\max$ /longest path in schedule/
- $N$ – set of nodes on DAG
- Time $t_i$ – assigned for each DAG node

$T_p(n) = \min\{ \max_{i \in N} t_i \}$

- Length of the longest (critical) path!
Boolean circuits, bit-level parallelism

- (Micro)processors
- Basic building blocks,
- bit-level operations
- combine into word-level operations
This is the NOR gate as seen from above.
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.
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
Other terminology

- **Speedup**: $S = T_1 / T_P$
- **Efficiency**: $E = S / P$
- **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?

**Notation**

- $P = \text{Number of processors}$
- $T_1 = \text{Time on one processor}$
- $T_P = \text{Time on P processors}$
• 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.
Amdahl law

is used to find the maximum expected improvement to an overall system when only part of the system is improved.
• e.g. 5% non parallelizable => speedup can not be more than 20x!

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

Two independent parts

Original process

Make B 5x faster

Make A 2x faster
Communication cost model

• Processes spend some time doing useful work, and some time communicating
• Model communication cost as
  – $T_C = t_s + L \cdot t_b$
  – $L =$ message size
  – 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:

\[ t_{\text{comm}} = \text{communication time} \]
\[ = t_{\text{startup}} + (\#\text{words} \times t_{\text{data}}) \]
\[ = \text{latency} + (\#\text{words} \times \frac{1}{\text{words/second}}) \]
\[ = \alpha + w*\beta \]

**latency** and **bandwidth**
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 $x_{p_1}, x_{p_2}, \ldots, x_{p_n}$
    • $x_{p_i}$ on processor $i$
    • $x_{p_i} \leq x_{p_{i+1}}$
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
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>MISD Systolic arrays</td>
<td>MIMD Multiprocessors multicomputers</td>
<td></td>
</tr>
</tbody>
</table>
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
## Parallel Computing Architectures

### Memory Model

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

Distributed shared memory network with directory
AMD reveals potent parallel processing breakthrough

Upcoming Kaveri processor will drink from shared-memory Holy Grail

By Rik Myslewski in San Francisco, 1st May 2013

AMD has released details on its implementation of The Next Big Thing in processor evolution, and in the process has unleashed the TNBT of acronyms: the AMD APU (CPU+GPU) HSA hUMA.

Before your eyes glaze over and you click away from this page, know that if this scheme is widely adopted, it could be of great benefit to both processor performance and developer convenience — and to you.

Simply put, what AMD’s heterogeneous Uniform Memory Access (hUMA) does is allow central processing units (CPUs) and graphics processing units (GPUs) — which AMD places on a single die in their accelerated processing units (APUs) — to seamlessly share the same memory in a heterogeneous system architecture (HSA). And that’s a very big deal, indeed.
INTRODUCING hUMA

UMA

CPU Memory

CPU  CPU  CPU  CPU  CPU

NUMA

CPU Memory    GPU Memory

CPU  CPU  CPU  CPU  GPU

hUMA

Memory

CPU  CPU  CPU  CPU  GPU

APU with HSA
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

• NUMA (non-uniform memory access)
  – Distributed memory in hardware, shared memory in software, with hardware assistance (for performance)
  – Better scalability than SMP, but not as good as a full distributed memory architecture
MPP – Massively Parallel

• Each node is an independent system having its own:
  – Physical memory
  – Address space
  – Local disk and network connections
  – Operating system
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 processing (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
Adding $n$ numbers on the mesh

Adding $n$ numbers in $\sqrt{n}$ steps
Adding $n$ numbers on the hypercube

Adding $n$ numbers in $\log n$ steps
128-way fat tree

http://en.wikipedia.org/wiki/Hypercube
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
Cont...

• 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, Mul-T and others – check out:
      http://ceu.fi.udc.es/SAL/C/1/index.shtml

• 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 😊
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

- If concurrent write is allowed we must decide which value to accept
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 \( \lg n \) steps for the largest key to be placed in \( S[1] \)
Example: Finding the Largest key in an array

```
Example:
Finding the
Largest
key in
an
array
```

```
3  12  7  6  8  11  19  13
```

```
P1
3  12  7  6  8  11  19  13
P2
6  3  7  11  8  13  19
P3
7  6  7  11  8  13  19
P4
11  8  11  13  19
```

```
Read
12  3  6  7  11  8  13  19
First  Second  First  Second  First  Second  First  Second
```

```
Write
12  12  7  6  11  11  19  13
```

Example: Finding the Largest key in an array

12 12 7 6 11 11 19 13

Read
P1
7 - 12
P3
19 - 11

Write

12 12 7 6 19 11 19 13
Example: Finding the Largest key in an array

12 12 7 6 19 11 19 13

Read

19 12

Write

19 12 7 6 19 11 19 13
Example: Merge Sort

```
Example:
  Merge
  Sort

P1
P2
P3
P4

P1
P2
P3
P4

P1
P2
P3
P4
```
Merge Sort Analysis

- Number of compares
  - $1 + 3 + ... + (2^i-1) + ... + (n-1)$
  - $\sum_{i=1}^{\lg(n)} 2^i - 1 = 2n - 2\lg n = \Theta(n)$

- We have improved from $n\lg(n)$ to $n$ simply by applying the old algorithm to parallel computing, by altering the algorithm we can further improve merge sort to $(\lg n)^2$
CLRS

- **parallel** (to execute for-loop iterations in parallel)
- **spawn** (create a new task/thread), and
- **sync** (wait for all spawned children)

<table>
<thead>
<tr>
<th>Fib(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if n &lt;= 1</td>
</tr>
<tr>
<td>2 return n</td>
</tr>
<tr>
<td>3 else</td>
</tr>
<tr>
<td>4 x = Fib(n-1)</td>
</tr>
<tr>
<td>5 y = Fib(n-2)</td>
</tr>
<tr>
<td>6 return x+y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fib(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if n &lt;= 1</td>
</tr>
<tr>
<td>2 return n</td>
</tr>
<tr>
<td>3 else</td>
</tr>
<tr>
<td>4 x = spawn Fib(n-1)</td>
</tr>
<tr>
<td>5 y = Fib(n-2)</td>
</tr>
<tr>
<td>6 sync</td>
</tr>
<tr>
<td>7 return x+y</td>
</tr>
</tbody>
</table>
Parallel loops

Mat-Vec( A, x )
1  n= A.rows
2  let y be a new vector of length n
3  parallel for i=1 to n
4      yi = 0
5  parallel for i=1 to n
6      for j=1 to n
7          yi = yi + aij xj
8  return y
Parallel loop (5-7) divide & conquer

Mat-Vec-Main-Loop( A, x,y,n,i,i’ )
1   if i== i’
2     for j=1 to n
3       yi = yi + aij xj
4     else  mid = [(i+i’)/2]
5        spawn Mat-Vec-Main-Loop( A, x,y,n,i,mid )
6        Mat-Vec-Main-Loop( A, x,y,n,mid+1, i’ )
7       sync
• deterministic – always the same result
• non-deterministic – may depend on order, runto run
• race

Race-Example()
  x=0
  parallel for i=1 to 2
    x = x+1
  print x
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 $n \gg P$
  - $\ast$ Each processor adds $n/P$ distinct numbers
  - Perform parallel reduction on $P$ numbers
  - $T_P \sim n/P + (1 + t_s + t_b) \log P$
  - Optimal $P$ obtained by differentiating wrt $P$
    - $P_{opt} \sim n/(1 + t_s + t_b)$
    - If communication cost is high, then fewer processors ought to be used
  - $E = [1 + (1 + t_s + t_b) P \log P/n]^{-1}$
    - $\ast$ As problem size increases, efficiency increases
    - $\ast$ As number of processors increases, efficiency decreases
Some common collective operations

- **Broadcast**
  - Input: A, B, C, D
  - Output: A, A, A, A

- **Gather**
  - Input: A, B, C, D
  - Output: A, B, C, D

- **Scatter**
  - Input: A, B, C, D
  - Output: A, B, C, D

- **All Gather**
  - Input: A, B, C, D
  - Output: A, B, C, D
Broadcast

- $T \sim (t_s + L t_b) \log P$
  - $L$: Length of data
• Gather: Data move towards the root
• Scatter: Review question
• $T \sim t_s \log P + PLt_b$

Note: $\sum_{i=0}^{\log P-1} 2^i = \frac{2^{\log P} - 1}{2-1} = P-1 \sim P$
All gather

• Equivalent to each processor broadcasting to all the processors
All gather
All gather
All gather

- $T_n \sim t_s \log P + PLt_b$
Matrix-vector multiplication

- $c = A \ b$
  - Often performed repeatedly
    - $b_i = A \ b_{i-1}$
    - 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 = $t_s \log P + t_b n$
- Note: $P \leq n$
2-D matrix-vector multiplication

- Processes $P_{i0}$ sends $B_i$ to $P_{0i}$
  - Time: $t_s + t_b n / \sqrt{P}$

- Processes $P_{0j}$ broadcast $B_j$ to all $P_{ij}$
  - Time = $t_s \log \sqrt{P} + t_b n \log \sqrt{P} / \sqrt{P}$

- Processes $P_{ij}$ compute $C_{ij} = A_{ij}B_j$
  - Time = $\Theta(n^2 / P)$

- Processes $P_{ij}$ reduce $C_{ij}$ on to $P_{i0}$, $0 \leq i \leq \sqrt{P}$
  - Time = $t_s \log \sqrt{P} + t_b n \log \sqrt{P} / \sqrt{P}$

- Total time = $\Theta(n^2 / P + t_s \log P + t_b n \log P / \sqrt{P})$
  - $P \leq n^2$
  - * More scalable than 1-dimensional decomposition
Matrix multiplication

In mathematics, matrix multiplication is the operation of multiplying a matrix with either a scalar or another matrix. This article gives an overview of the various ways to perform matrix multiplication.

**Ordinary matrix product**

This is the most often used and most important way to multiply matrices. It is defined between two matrices only if the number of columns of the first matrix is the same as the number of rows of the second matrix. If \( A \) is an \( m \text{-by-} n \) matrix and \( B \) is an \( n \text{-by-} p \) matrix, then their **product** is an \( m \text{-by-} p \) matrix denoted by \( AB \) (or sometimes \( A \cdot B \)). If \( C = AB \), and \( c_{ij} \) denotes the entry in \( C \) at position \((i,j)\), then

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj},
\]

for each pair \( i \) and \( j \) with \( 1 \leq i \leq m \) and \( 1 \leq j \leq p \). The algebraic system of "matrix units" summarises the abstract properties of this kind of multiplication.

**Calculating directly from the definition**

The picture to the left shows how to calculate the \((1,2)\) element and the \((3,3)\) element of \( AB \) if \( A \) is a \( 4 \times 2 \) matrix, and \( B \) is a \( 2 \times 3 \) matrix. Elements from each matrix are paired off in the direction of the arrows; each pair is multiplied and the products are added. The location of the resulting number in \( AB \) corresponds to the row and column that were considered.

\[
(AB)_{1,2} = \sum_{r=1}^{2} a_{r1} b_{r2} = a_{1,1}b_{1,2} + a_{1,2}b_{2,2}
\]

\[
(AB)_{3,3} = \sum_{r=1}^{2} a_{3,r} b_{r3} = a_{3,1}b_{1,3} + a_{3,2}b_{2,3}
\]

For example:

\[
\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 \times 3 + 0 \times 2 + 2 \times 1 & 1 \times 1 + 0 \times 1 + 2 \times 0 \\ -1 \times 3 + 3 \times 2 + 1 \times 1 & -1 \times 1 + 3 \times 1 + 1 \times 0 \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ 4 & 2 \end{bmatrix}
\]

**The coefficients-vectors method**

This matrix multiplication can also be considered from a slightly different viewpoint: it adds vectors together after being multiplied by different coefficients. If \( A \) and \( B \) are matrices given by:

\[
A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}
\]

The column-vectors of \( A \) and \( B \) can be multiplied in the usual way to produce the matrix product:

\[
AB = \begin{bmatrix} (a_{11}b_{11} + a_{12}b_{21}) & (a_{11}b_{12} + a_{12}b_{22}) \\ (a_{21}b_{11} + a_{22}b_{21}) & (a_{21}b_{12} + a_{22}b_{22}) \end{bmatrix}
\]
• 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 $(n^{\lg 7}) = O(n^{2.81})$ time. For sufficiently large $n$, therefore, it outperforms the naive ($n^3$) matrix-multiplication algorithm MATRIX-MULTIPLY from Section 26.1.
\[ C = AB \]

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

- \( r = ae + bf \)
- \( s = ag + bh \)
- \( t = ce + df \)
- \( u = cg + dh \)
- \( T(n) = 8T(n/2) + \Theta(n^2). \quad = O(n^3) \)
Strassen discovered a different recursive approach that requires only 7 recursive multiplications of $n/2 \times n/2$ matrices and $\Theta(n^2)$ scalar additions and subtractions, yielding the recurrence

$$T(n) = 7T(n/2) + \Theta(n^2)$$

$$= \Theta(n^{1.87})$$

$$= O(n^{2.81}).$$

(31.15)

Strassen's method has four steps:

1. Divide the input matrices $A$ and $B$ into $n/2 \times n/2$ submatrices, as in equation (31.9).

2. Using $\Theta(n^2)$ scalar additions and subtractions, compute 14 $n/2 \times n/2$ matrices $A_1, B_1, A_2, B_2, \ldots, A_7, B_7$.

3. Recursively compute the seven matrix products $P_i = A_iB_i$ for $i = 1, 2, \ldots, 7$.

4. Compute the desired submatrices $r, s, t, u$ of the result matrix $C$ by adding and/or subtracting various combinations of the $P_i$ matrices, using only $\Theta(n^2)$ scalar additions and subtractions.
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 (simplest 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).
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.
However, they are also concerned with:

- Concurrency and communication
- Need for speed (nee high performance), and
- Plethora and diversity of architecture
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
   1. The communication operations are balanced among tasks
   2. Each task communicates with only a small number of neighbours
   3. Tasks can perform their communications concurrently
   4. 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.
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
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 data base – 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:

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

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: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$
Output Data Decomposition: Example

A partitioning of output data does not result in a unique decomposition into tasks. For example, for the same problem as in previous foil, with identical output data distribution, we can derive the following two (other) decompositions:

<table>
<thead>
<tr>
<th>Decomposition I</th>
<th>Decomposition II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1: ( C_{1,1} = A_{1,1} B_{1,1} )</td>
<td>Task 1: ( C_{1,1} = A_{1,1} B_{1,1} )</td>
</tr>
<tr>
<td>Task 2: ( C_{1,1} = C_{1,1} + A_{1,2} B_{2,1} )</td>
<td>Task 2: ( C_{1,1} = C_{1,1} + A_{1,2} B_{2,1} )</td>
</tr>
<tr>
<td>Task 3: ( C_{1,2} = A_{1,1} B_{1,2} )</td>
<td>Task 3: ( C_{1,2} = A_{1,2} B_{2,1} )</td>
</tr>
<tr>
<td>Task 4: ( C_{1,2} = C_{1,2} + A_{1,2} B_{2,2} )</td>
<td>Task 4: ( C_{1,2} = C_{1,2} + A_{1,1} B_{1,2} )</td>
</tr>
<tr>
<td>Task 5: ( C_{2,1} = A_{2,1} B_{1,1} )</td>
<td>Task 5: ( C_{2,1} = A_{2,2} B_{2,1} )</td>
</tr>
<tr>
<td>Task 6: ( C_{2,1} = C_{2,1} + A_{2,2} B_{2,1} )</td>
<td>Task 6: ( C_{2,1} = C_{2,1} + A_{2,1} B_{1,1} )</td>
</tr>
<tr>
<td>Task 7: ( C_{2,2} = A_{2,1} B_{1,2} )</td>
<td>Task 7: ( C_{2,2} = A_{2,1} B_{1,2} )</td>
</tr>
<tr>
<td>Task 8: ( C_{2,2} = C_{2,2} + A_{2,2} B_{2,2} )</td>
<td>Task 8: ( C_{2,2} = C_{2,2} + A_{2,2} B_{2,2} )</td>
</tr>
</tbody>
</table>
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. modelling 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
Control is distributed, usually with some distribution of data structures.

Some processes may be dedicated to achieve better load balance.

Examples:

- Image processing: given a series of raw images, perform a series of transformation that yield a final enhanced image. Solve this in a functional decomposition (each process represents a different function in the problem) using data pipelining.

- Game playing: games feature an irregular search space. One possible move may lead to a rich set of possible subsequent moves to search.
  - Need an approach where work can be dynamically assigned to improve load balancing.
  - May need to assign multiple processes to work on a particularly promising lead.
Any problem that involve search (or computations) whose scope cannot be determined a priori, are candidates for control decomposition

- Calculations involving multiple levels of recursion (i.e. genetic algorithms, simulated annealing, artificial intelligence)
- Discrete phenomena in an otherwise regular medium (i.e. modelling localized storms within a weather model)
- Design-rule checking in micro-electronic circuits
- Simulation of complex systems
- Game playing, music composing, etc.
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-the-call is more important than worrying about under-worked processors (i.e. battlefield 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 $O(4-10)$ parallelism
  - Not concerned with portability
  - Platform has a parallelizing compiler

- OpenMP
  - Moderate $O(10)$ parallelism
  - Good quality implementation exists on the platform
  - Not scalable

- MPI
  - Scalability and portability are important
  - Needs some type of message passing platform
  - A substantive coding effort is required

- PVM
  - All MPI conditions plus fault tolerance
  - Still provides better functionality in some settings

- High Performance Fortran (HPF)
  - Like OpenMP, but new language constructs provide a data-parallel implicit programming model

- P-Threads
  - Not recommended
  - Difficult to correct and maintain programs
  - Not scalable to large number of processors

- High level libraries
  - POOMA and HPC++
  - Library is available and it addresses a specific problem