Parallel Computing

Peter Bastian and Olaf Ippisch

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
Im Neuenheimer Feld 368
D-69120 Heidelberg
Telefon: 06221/54-8261
E-Mail: peter.bastian@iwr.uni-heidelberg.de Telefon: 06221/54-8252
E-Mail: olaf.ippisch@iwr.uni-heidelberg.de

November 9, 2009

Parallel Computing

This set of lectures gives a basic introduction to the subject.
At the end you should have acquired:

- A basic understanding of different parallel computer architectures.
- Know how to write programs using OpenMP.
- Know how to write programs using message passing.
- Know how to write parallel programs using the CUDA environment.
- Know how to evaluate the quality of a parallel algorithm and its implementation.

Parallel Computing is Ubiquitous

- Multi-Tasking
  - Several independent computations ("threads of control") can be run quasi-simultaneously on a single processor (time slicing).
  - Developed since 1960s to increase throughput.
  - Interactive applications require "to do many things in parallel".
  - All relevant coordination problems are already present (That is why we can "simulate" parallel programming on a single PC".
  - "Hyperthreading" does the simulation in hardware.
  - Most processors today offer execution in parallel ("multi-core").
- Distributed Computing
  - Computation is inherently distributed because the information is distributed.
  - Example: Running a world-wide company or a bank.
  - Issues are: Communication across platforms, portability and security.

Parallel Computing is Ubiquitous

- High-Performance Computing
  - HPC is the driving force behind the development of computers.
  - All techniques implemented in today's desktop machines have been developed in supercomputers many years ago.
  - Applications run on supercomputers are mostly numerical simulations.
  - Grand Challenges: Cosmology, protein folding, prediction of earth-quakes, climate and ocean flows, . . .
  - but also: nuclear weapon simulation
  - ASCI (Advanced Simulation and Computing) Program funding: $ 300 million in 2004.
  - Earth simulator (largest computer in the world from 2002 to 2004) cost about $ 500 million.

What is a Supercomputer?

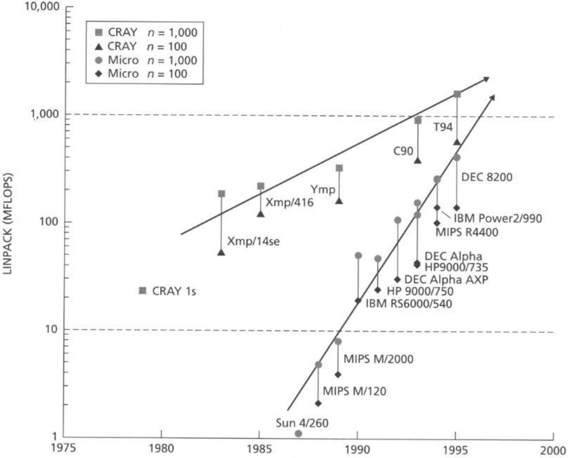
- A computer that costs more than $ 10 million

Computer	Year	$	MHz	MBytes	MFLOP/s
CDC 6600	1964	7M $	10	0.5	3.3
Cray 1A	1976	8M $	80	8	20
Cray X-MP/48	1986	15M $	118	64	220
C90	1996		250	2048	5000
ASCI Red	1997		220	$1.2{\cdot}10^6$	$2.4{\cdot}10^6$
Pentium 4	2002	1500	2400	1000	4800
Intel QX 9770	2008	1200	3200	>4000	10^6

- Speed is measured in floating point operations per second (FLOP/s).
- Current supercomputers are large collections of microprocessors
- Today's desktop PC is yesterdays supercomputer.
- www.top500.org compiles list of supercomputers every six months.

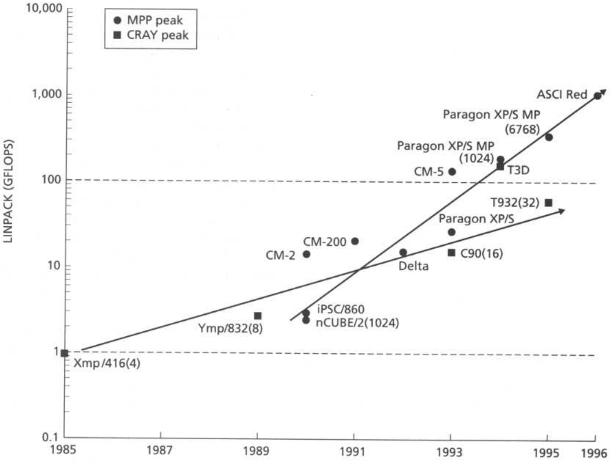
Development of Microprocessors

Microprocessors outperform conventional supercomputers in the mid 90s (from *Culler et al.*).

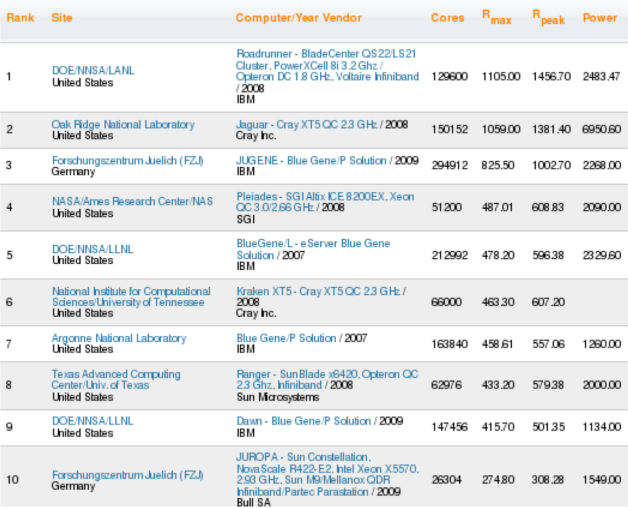


Development of Multiprocessors

Massively parallel machines outperform vector parallel machines in the early 90s (from *Culler et al.*).



TOP 500 November 2007



Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2008 IBM	129600	1105.00	1456.70	2483.47
2	Oak Ridge National Laboratory United States	Jaguar - Cray XT5 QC 2.3 GHz / 2008 Cray Inc.	150152	1059.00	1381.40	6950.60
3	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2009 IBM	294912	825.50	1002.70	2268.00
4	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz / 2008 SGI	51200	487.01	608.83	2090.00
5	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution / 2007 IBM	212992	478.20	596.38	2329.60
6	National Institute for Computational Sciences/University of Tennessee United States	Kraken XT5 - Cray XT5 QC 2.3 GHz / 2008 Cray Inc.	66000	463.30	607.20	
7	Argonne National Laboratory United States	Blue Gene/P Solution / 2007 IBM	163840	458.61	557.06	1260.00
8	Texas Advanced Computing Center/Univ. of Texas United States	Ranger - SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband / 2008 Sun Microsystems	62976	433.20	579.38	2000.00
9	DOE/NNSA/LLNL United States	Dawn - Blue Gene/P Solution / 2009 IBM	147456	415.70	501.35	1134.00
10	Forschungszentrum Juelich (FZJ) Germany	JUROPA - Sun Constellation, NovaScale R422-E2, Intel Xeon X5570, 2.93 GHz, Sun M9/Mellanox QDR Infiniband/Partec Parastation / 2009 Bull SA	26304	274.80	308.28	1549.00

3. BlueGene/P “JUGENE” at FZ Jülich: 294912 processors(cores), 825.5 TFLOP/s

Terascale Simulation Facility

BlueGene/L prototype at Lawrence Livermore National Laboratory outperforms Earth Simulator in late 2004: 65536 processors, 136 TFLOP/s,

Final version at LLNL in 2006: 212992 processors, 478 TFLOP/s

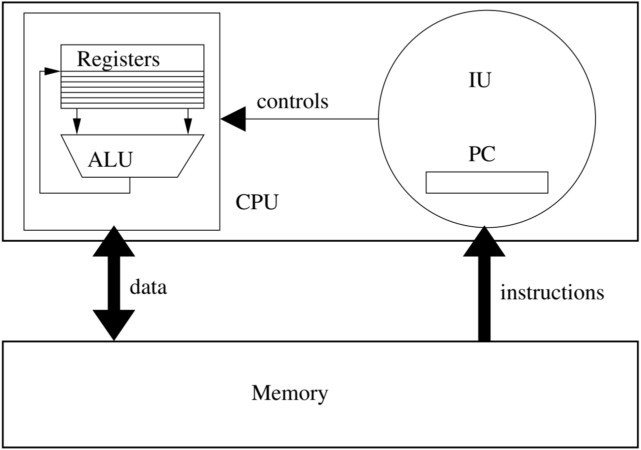
Efficient Algorithms are Important!

- Computation time for solving (certain) systems of linear equations on a computer with 1 GFLOP/s.

N	Gauss ($\frac{2}{3}N^3$)	Multigrid ($100N$)
1.000	0.66 s	10^{-4} s
10.000	660 s	10^{-3} s
100.000	7.6 d	10^{-2} s
$1 \cdot 10^6$	21 y	0.1 s
$1 \cdot 10^7$	21.000 y	1 s

- Parallelisation does not help an inefficient algorithm.
- We must parallelise algorithms with good sequential complexity.

Von Neumann Computer



IU: Instruction unit

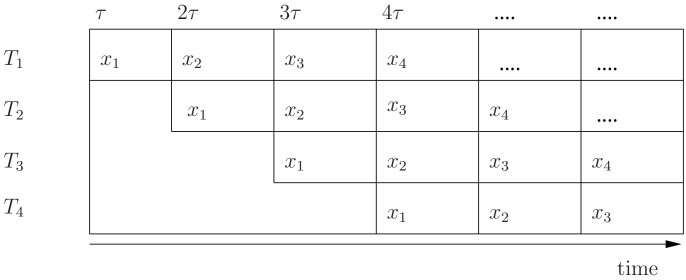
PC: Program counter

ALU: Arithmetic logic unit

CPU: Central processing unit

Single cycle architecture

Pipelining: General Principle



	τ	2τ	3τ	4τ	....	....
T_1	x_1	x_2	x_3	x_4	....	....
T_2		x_1	x_2	x_3	x_4	....
T_3			x_1	x_2	x_3	x_4
T_4				x_1	x_2	x_3

- Task T can be subdivided into m subtasks $T_1, \dots, T_m$.
- Every subtask can be processed in the **same** time τ.
- All subtasks are **independent**.
- Time for processing N tasks:
  $T_S(N) = Nm\tau$
  $T_P(N) = (m + N - 1)\tau$.
- Speedup
  $S(N) = \frac{T_S(N)}{T_P(N)} = m\frac{N}{m+N-1}$.

Arithmetic Pipelining

- Apply pipelining principle to floating point operations.
- Especially suited for "vector operations" like $s = x \cdot y$ or $x = y + z$ because of independence property.
- Hence the name "vector processor".
- Allows at most $m = 10 \dots 20$.
- Vector processors typically have a very high memory bandwith.
- This is achieved with *interleaved* memory, which is pipelining applied to the memory subsystem.

Instruction Pipelining

- Apply pipelining principle to the processing of machine instructions.
- Aim: Process one instruction per cycle.
- Typical subtasks are (m=5):
  - Instruction fetch.
  - Instruction decode.
  - Instruction execute.
  - Memory access.
  - Write back results to register file.
- Reduced instruction set computer (RISC): Use simple and homogeneous set of instructions to enable pipelining (e. g. load/store architecture).
- Conditional jumps pose problems and require some effort such as branch prediction units.
- Optimising compilers are also essential (instruction reordering, loop unrolling, etc.).

Superscalar Architecture

- Consider the statements

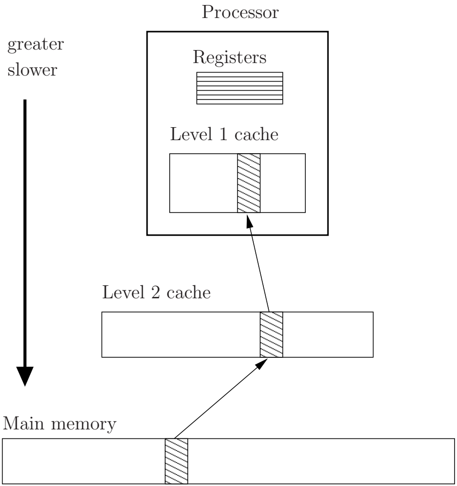
```
(1) a = b+c;
(2) d = e*f;
(3) g = a-d;
(4) h = i*j;
```

- Statements 1, 2 and 4 can be executed in parallel because they are independent.
- This requires
  - Ability to issue several instructions in one cycle.
  - Multiple functional units.
  - Out of order execution.
  - Speculative execution.
- A processor executing more than one instruction per cycle is called superscalar.
- Multiple issue is possible through:
  - A wide memory access reading two instructions.
  - Very long instruction words.
- Multiple functional units with out of order execution were implemented in the CDC 6600 in 1964.
- A degree of parallelism of 3. . . 5 can be achieved.

Caches I

While the processing power increases with parallisation, the memory bandwidth usually does not

- Reading a 64-bit word from DRAM memory can cost up to 50 cycles.
- Building fast memory is possible but too expensive per bit for large memories.
- Hierarchical cache: Check if data is in the level *l* cache, if not ask the next higher level.
- Repeat until main memory is asked.
- Data is transferred in *cache lines* of 32 . . . 128 bytes (4 to 16 words).

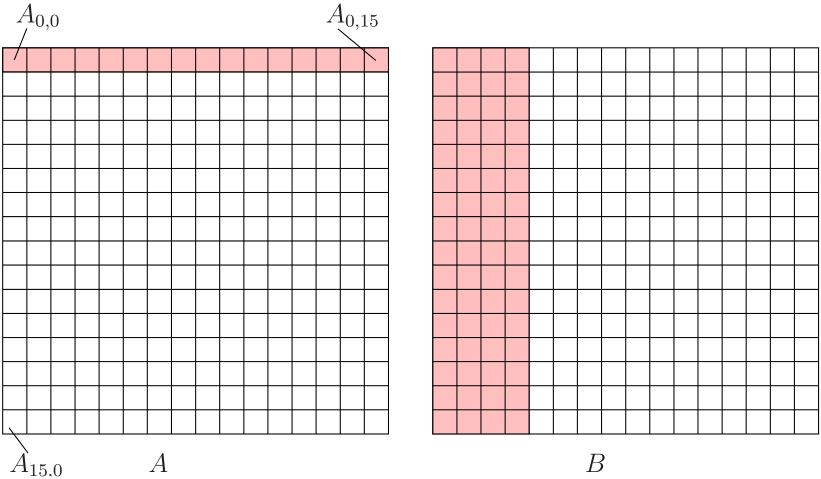


Caches II

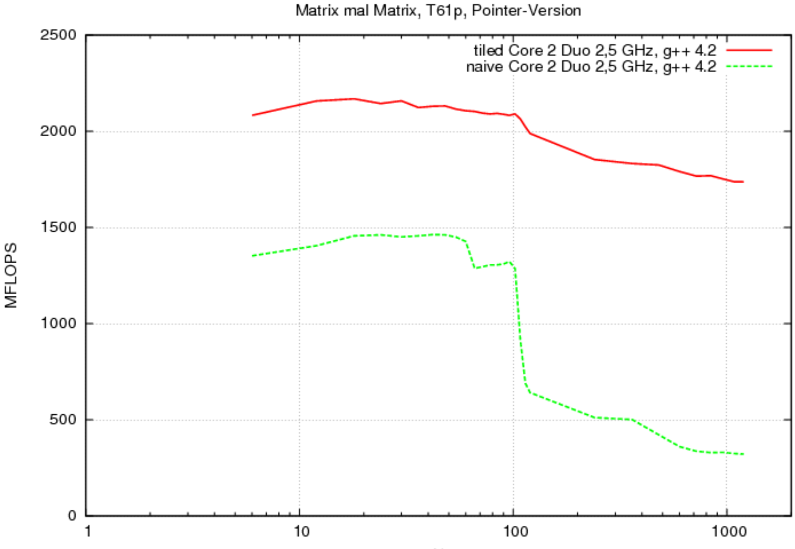
- Caches rely on *spatial and temporal locality*.
- There are four issues to be discussed in cache design:
    - *Placement*: Where does a block from main memory go in the cache? Direct mapped cache, associative cache.
    - *Identification*: How to find out if a block is already in cache?
    - *Replacement*: Which block is removed from a full cache?
    - *Write strategy*: How is write access handled? Write-through and write-back caches.
- Caches require to make code cache-aware. This is usually non-trivial and not done automatically.
- Caches can lead to a slow-down if data is accessed randomly and not reused.

Cache Use in Matrix Multiplication

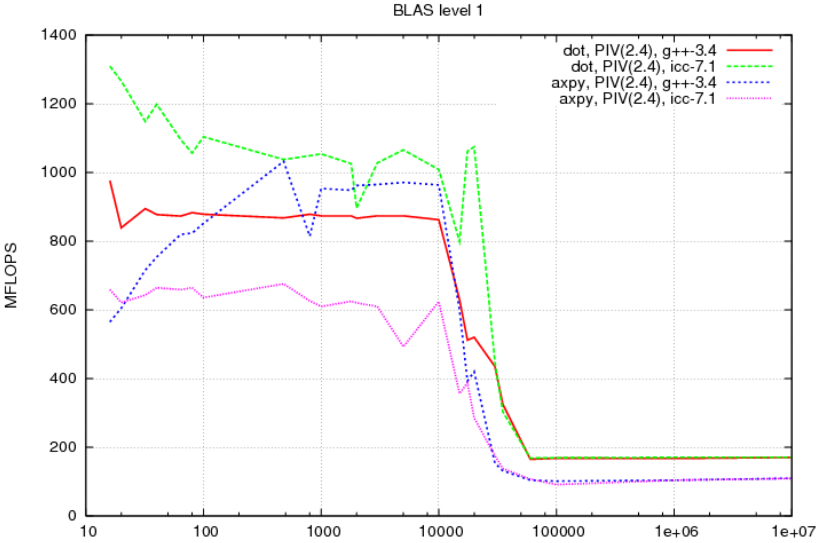
- Compute product of two matrices $C = AB$, i.e. $C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$
- Assume cache lines containing four numbers. C layout:



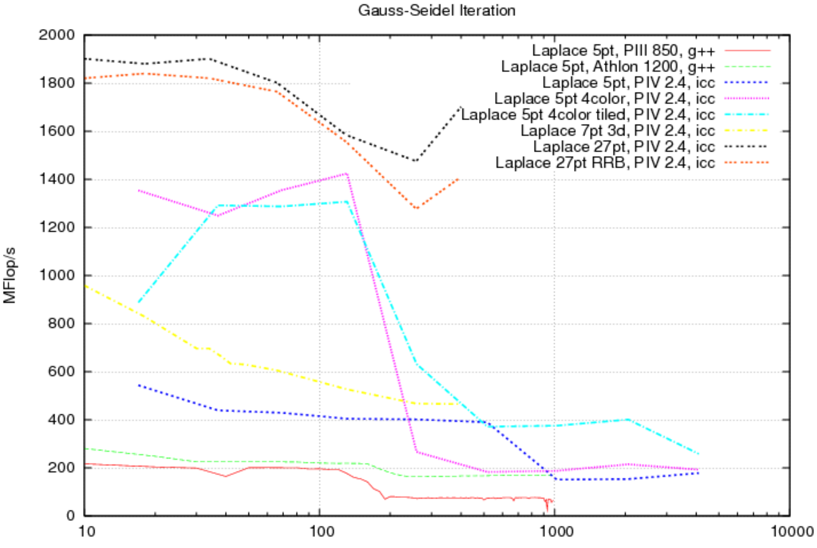
Matrix Multiplication Performance



BLAS1 Performance



Laplace Performance



Flynn's Classification (1972)

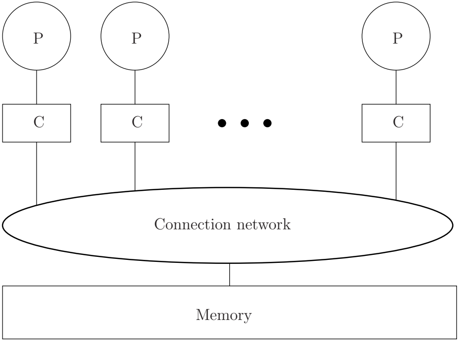
	Single data stream (One ALU)	Multiple data streams (Several ALUs)
Single instruction stream, (One IU)	**SISD**	**SIMD**
Multiple instruction streams (Several IUs)	—	**MIMD**

- **SIMD** machines allow the synchronous execution of one instruction on multiple ALUs. Important machines: ILLIAC IV, CM-2, MasPar. And now CUDA!
- **MIMD** is the leading concept since the early 90s. All current supercomputers are of this type.

Classification by Memory Access

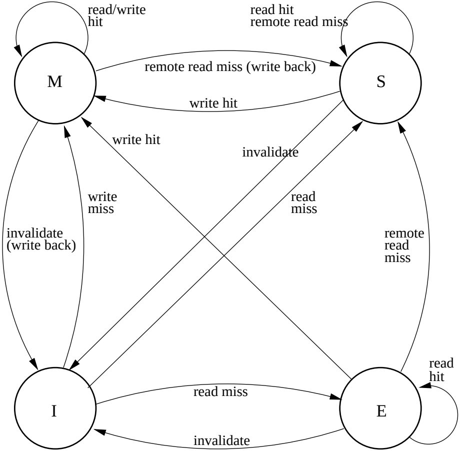
- Flynn's classification does not state *how* the individual components exchange data.
- There are only two basic concepts.
- **Communication via shared memory**. This means that all processors share a **global address space**. These machines are also called **multiprocessors**.
- **Communication via message exchange.** In this model every processor has its own **local address space**. These machines are also called **multicomputers**.

Uniform Memory Access Architecture



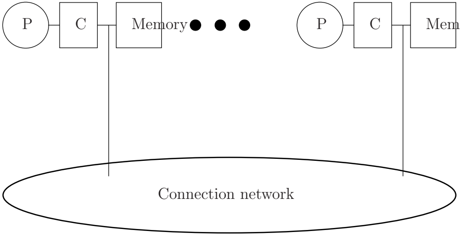
- UMA: Access to every memory location from every processor takes the same amount of time.
- This is achieved through a *dynamic network*.
- Simplest “network”: bus.
- Caches serve two reasons: Provide fast memory access (migration) and remove traffic from network (replication).
- *Cache coherence problem:* Suppose one memory block is in two or more caches and is written by a processor. What to do now?

Bus Snooping, MESI-Protocol



- Assume that network is a bus.
- All caches *listen* on the bus whether one of their blocks is affected.
- *MESI-Protocol*: Every block in a cache is in one of four states: *Modified, exclusive, shared, invalid*.
- Write-invalidate, write-back protocol.
- State transition diagram is given on the left.
- Used e.g. in the Pentium.

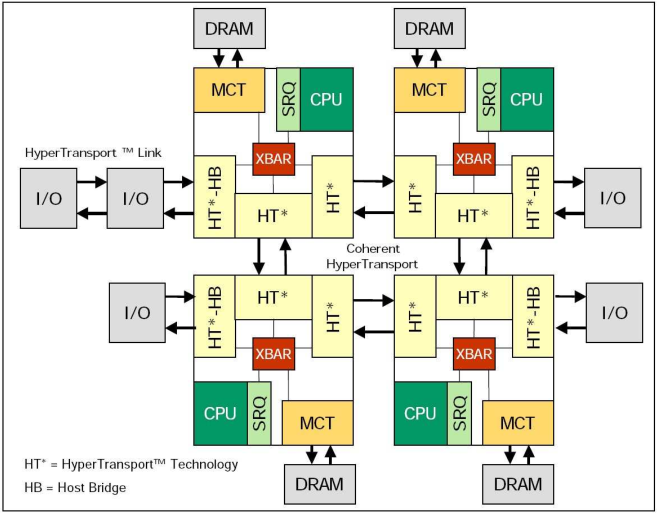
Nonuniform Memory Access Architecture



- Memories are associated with processors but address space is global.
- Access to local memory is fast.
- Access to remote memories is via the network and slow.
- Including caches there are at least three different access times.
- Solving the cache-coherence problem requires expensive hardware (ccNUMA).
- Machines up to 1024 processors have been built.

AMD Hammer Architecture

Generation 8, introduced 2001

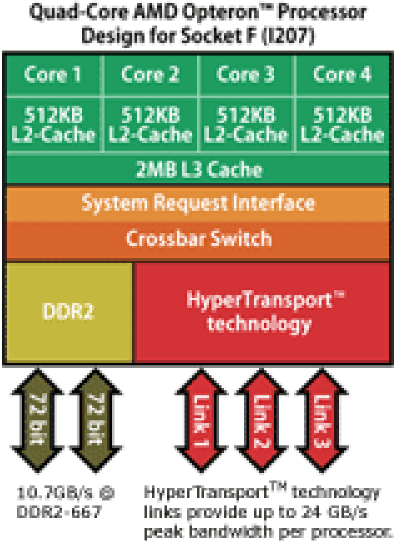


HT* = HyperTransport™ Technology

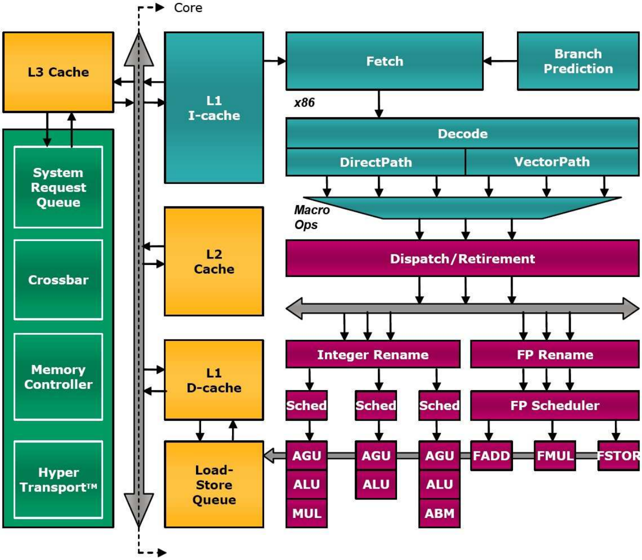
HB = Host Bridge

Barcelona

QuadCore, Generation 10h, 2007



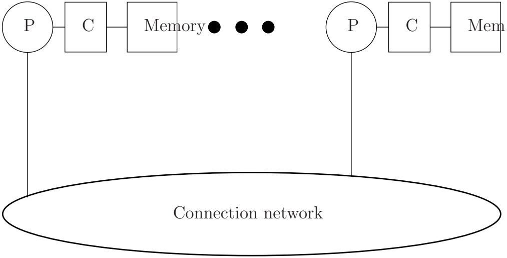
Barcelona Core



Barcelona Details

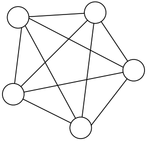
- L1 Cache
  - 64K instructions, 64K data.
  - Cache line 64 Bytes.
  - 2 way associative, write-allocate, writeback, least-recently-used.
  - MOESI (MESI extended by “owned”).
- L2 Cache
  - Victim cache: contains only blocks moved out of L1.
  - Size implementation dependent.
- L3 Cache
  - non-inclusive: data either in L1 *or* L3.
  - Victim Cache for L2 blocks.
  - Heuristics for determining blocks shared by some cores.
- Superscalarity
  - Instruction decode, Integer units, FP units, address generation 3-fold (3 OPS per cycle).
  - Out of order execution, branch prediction.
- Pipelining: 12 stages integer, 17 stages floating-point (Hammer).
- Integrated memory controller, 128 bit wide.
- HyperTransport: coherent access to remote memory.

Private Memory Architecture

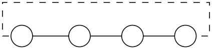


- Processors can only access their local memory.
- Processors, caches and main memory are standard components: Cheap, Moore's law can be fully utilised.
- Network can be anything from fast ethernet to specialised networks.
- Most scalable architecture. Current supercomputers already have more than 10^5 processors.

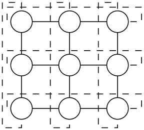
Network Topologies



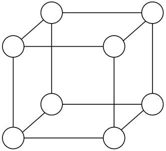
a) fully connected



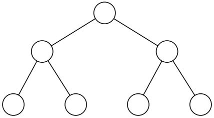
b) 1D-array, ring



c) 2D-array, 2D-torus



d) Hypercube, 3D-array

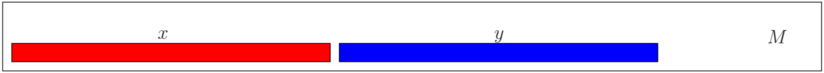


e) binary tree

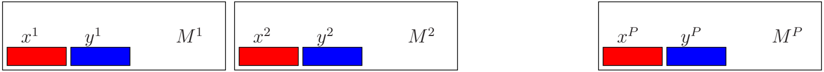
- There are many different types of topologies used for packet-switched networks.
- 1-,2-,3-D arrays and tori.
- Fully connected graph.
- Binary tree.
- Hypercube: HC of dimension $d \geq 0$ has 2^d nodes. Nodes x, y are connected if their bit-wise representations differs in one bit.
- k-dimensional arrays can be embedded in hypercubes.
- Topologies are useful in algorithms as well.

Comparison of Architectures by Example

- Given vectors $x, y \in \mathbb{R}^N$, compute scalar product $s = \sum_{i=0}^{N-1} x_i y_i$:
  1. Subdivide index set into P pieces.
  2. Compute $s_p = \sum_{i=pN/P}^{(p+1)N/P-1} x_i y_i$ in parallel.
  3. Compute $s = \sum_{i=0}^{P-1} s_i$. This is treated later.
- *Uniform memory access architecture*: Store vectors as in sequential program:



- *Nonuniform memory access architecture*: Distribute data to the local memories:



- *Message passing architecture*: Same as for NUMA!
- Distributing data structures is hard and not automatic in general.
- Parallelisation effort for NUMA and MP is almost the same.

What you should remember

- Modern microprocessors combine all the features of yesterdays supercomputers.
- Today parallel machines have arrived on the desktop.
- MIMD is the dominant design.
- There are UMA, NUMA and MP architectures.
- Only machines with local memory are scalable.
- Algorithms have to be designed carefully with respect to the memory hierarchy.

For Further Reading

ASC (former ASCI) program website.
http://www.sandia.gov/NNSA/ASC/

Achievements of Seymour Cray.
http://research.microsoft.com/users/gbell/craytalk/

TOP 500 Supercomputer Sites.
http://www.top500.org/

D. E. Culler, J. P. Singh and A. Gupta (1999).
Parallel Computer Architecture.
Morgan Kaufmann.

Communicating Sequential Processes

Sequential Program

Sequence of statements. Statements are processed one after another.

(Sequential) Process

A sequential program in execution. The state of a process consists of the values of all variables and the next statement to be executed.

Parallel Computation

A set of interacting sequential processes. Processes can be executed on a single processor (time slicing) or on a separate processor each.

Parallel Program

Specifies a parallel computation.

A Simple Parallel Language

```
parallel <program name> {
    const int P = 8;          // define a global constant
    int flag[P] = {1[P]};     // global array with initialization

    // The next line defines a process
    process <process name 1> [<copy arguments>]
    {
        // put (pseudo-) code here
    }
    ...
    process <process name n> [<copy arguments>]
    { . . . }
}
```

- First all global variables are initialized, then processes are started.
- Computation ends when all processes terminated.
- Processes share global address space (also called threads).

Example: Scalar Product with Two Processes

- We neglect input/output of the vectors.
- Local variables are private to each process.
- Decomposition of the computation is on the **for**-loop.

```
parallel two-process-scalar-product {
    const int $N$=8;                          // problem size
    double $x[N]$, $y[N]$, $s$=0;             // vectors, result
    process $\Pi_1$
    {
        int $i$; double $ss$=0;
        for ($i = 0$; $i < N/2$; $i$++) $ss$ += $x[i]$*$y[i]$;
        $s$=$s$+$ss$;                         // danger!
    }
    process $\Pi_2$
    {
        int $i$; double $ss$=0;
        for ($i = N/2$; $i < N$; $i$++) $ss$ += $x[i]$*$y[i]$;
        $s$=$s$+$ss$;                         // danger!
    }
}
```

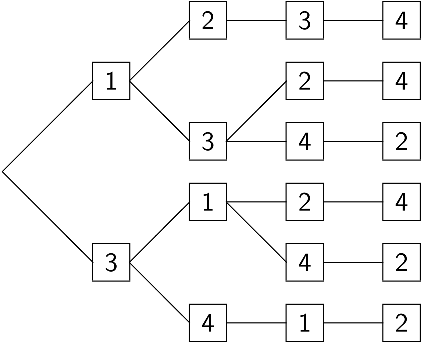
Critical Section

- Statement s=s+ss is not atomic:

	process Π_1		process Π_2
1	load s in R1	3	load s in R1
	load ss in R2		load ss in R2
	add R1, R2, store in R3		add R1, R2, store in R3
2	store R3 in s	4	store R3 in s

- The order of execution of statements of different processes relative to each other is not specified
- This results in an exponentially growing number of possible orders of execution.

Possible Execution Orders



Result of computation

$$s = ss_{\Pi_1} + ss_{\Pi_2}$$

$$s = ss_{\Pi_2}$$

$$s = ss_{\Pi_1}$$

$$s = ss_{\Pi_2}$$

$$s = ss_{\Pi_1}$$

$$s = ss_{\Pi_1} + ss_{\Pi_2}$$

Only some orders yield the correct result!

First Obervations

- **Work has to be distributed to processes**
- **Often Synchronisation of processes necessary**

Mutual Exclusion

- Additional *synchronisation* is needed to exclude possible execution orders that do not give the correct result.
- Critical sections have to be processed under *mutual exclusion*.
- Mutual exclusion requires:
  - At most one process enters a critical section.
  - No deadlocks.
  - No process waits for a free critical section.
  - If a process wants to enter, it will finally succeed.
- By $[s = s + ss]$ we denote that all statements between "[" and "]" are executed only by *one process at a time*. If two processes attempt to execute "[" at the same time, one of them is delayed.

Machine Instructions for Mutual Exclusion

- In practice mutual exclusion is implemented with special machine instructions to avoid problems with consistency and performance.
- *Atomic-swap*: Exchange contents of a register and a memory location *atomically*.
- *Test-and-set*: Check if contents of a memory location is 0, if yes, set it to 1 *atomically*.
- *Fetch-and-increment*: Load memory location to register and increment the contents of the memory location by one *atomically*.
- *Load-locked/store-conditional*: *Load-locked* loads a memory location to a register, *store-conditional* writes a register to a memory location *only if this memory location has not been written since the preceeding load-locked*. This requires interaction with the cache hardware.
- The first three operations consist of an atomic read-modify-write.
- The last one is more flexible and suited for load/store architectures.

Improved Spin Lock

- Idea: Use *atomic* − *swap* only if it has been found true:

```
parallel improved–spin–lock {
    const int $P = 8$;                 // number of processes
    int $lock$=0;                      // the lock variable
    process Π [int $p \in \{0, ..., P - 1\}$] {
        . . .
        while (1) {
            if ($lock$==0)
                if ($atomic - swap$(& $lock$,1)==0 )
                    break;
        }
        . . .                          // critical section
        $lock$ = 0;
        . . .
    }
}
```

- Getting P processors through the lock requires $O(P^2)$ time.

Parametrisation of Processes

- We want to write programs for a variable number of processes:

```
parallel many-process-scalar-product {
    const int $N$;                          // problem size
    const int $P$;                          // number of processors
    double $x[N]$, $y[N]$;                  // vectors
    double $s = 0$;                         // result
    process Π [int $p \in \{0, ..., P-1\}$]
    {
        int $i$; double $ss = 0$;
        for ($i = N * p/P$; $i < N * (p+1)/P$; $i$++)
            $ss$ += $x[i]$*$y[i]$;
        [$s = s + ss$];                     // sequential execution
    }
}
```

- *Single Program Multiple Data*: Every process has the same code but works on different data depending on p.

Scalar Product on NUMA Architecture

- Every process stores part of the vector as local variables.
- Indices are renumbered from 0 in each process.

```
**parallel** local-data-scalar-product {
    **const int** $P, N$;
    **double** $s = 0$;

    **process** $\Pi$ [ **int** $p \in \{0, \ldots, P-1\}$]
    {
        **double** $x[N/P+1]$, $y[N/P+1]$;
                                        // local part of the vectors
        **int** $i$; **double** $ss$=0;

        **for** $(i = 0, i < (p+1)*N/P - p*N/P; i{+}{+})$ $ss = ss + x[i]*y[i]$;
        $[s = s + ss;]$
    }
}
```

Parallelisation of the Sum

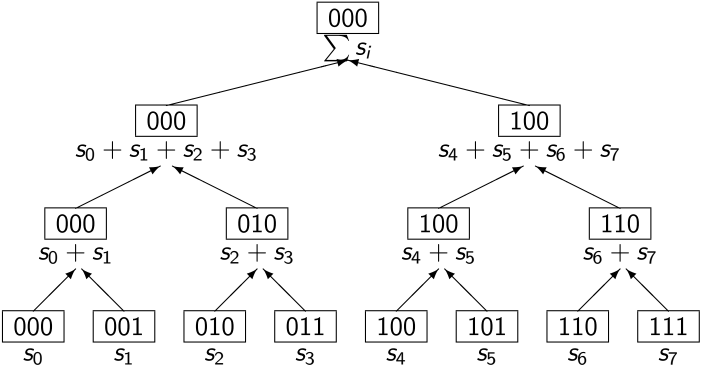
- Computation of the global sum of the local scalar products with $[s = s + ss]$ is not parallel.
- It can be done in parallel as follows ($P = 8$):

$$s = \underbrace{\underbrace{\underbrace{s_0 + s_1}_{s_{01}} + \underbrace{s_2 + s_3}_{s_{23}}}_{s_{0123}} + \underbrace{\underbrace{s_4 + s_5}_{s_{45}} + \underbrace{s_6 + s_7}_{s_{67}}}_{s_{4567}}}_{s}$$

- This reduces the execution time from $O(P)$ to $O(\log_2 P)$.

Tree Combine

Using a binary representation of process numbers, the communication structure forms a binary tree:



Implementation of Tree Combine

```
parallel parallel-sum-scalar-product {
    const int $N = 100$;                              // problem size
    const int $d = 4$, $P = 2^d$;                     // number of processes
    double $x[N]$, $y[N]$;                            // vectors
    double $s[P] = \{0[P]\}$;                         // results are global now
    int $flag[P] = \{0[P]\}$;                         // flags, must be initialized!
    process $\Pi$ [int $p \in \{0, ..., P-1\}$] {
        int $i$, $m$, $k$;
        for ($i = 0$; $i < d$; $i$++) {
            $m = 2^i$;                                // bit $i$ is 1
            if ($p\&m$) {$flag[m]$=1; break;}          // I am ready
            while (!$flag[p|m]$);                     // condition synchronisation
            $s[p] = s[p] + s[p|m]$;
        }
    }
}
```

Condition Synchronisation

- A process waits for a condition (boolean expression) to become true. The condition is made true by another process.
- Here some processes wait for the flags to become true.
- Correct initialization of the flag variables is important.
- We implemented this synchronisation using *busy wait*.
- This might not be a good idea in multiprocessing.
- The flags are also called *condition variables*.
- When condition variables are used repeatedly (e.g. when computing several sums in a row) the following rules should be obeyed:
  - A process that waits on a condition variable also resets it.
  - A condition variable may only be set to true again if it is sure that it has been reset before.

Barriers

- At a barrier a process stops until all processes have reached the barrier.
- This is necessary, if e.g. all processes need the result of a parallel computation to go on.
- Sometimes it is called global synchronisation.
- A barrier is typically executed repeatedly as in the following code fragment:

```
while (1) {
    compute something;
    Barrier();
}
```

Barrier for Two Processes

- First we consider two processes Π_i and Π_j only:

```
$\Pi_i$:                               $\Pi_j$:
while ($arrived[i]$) ;                 while ($arrived[j]$) ;
$arrived[i]=1$;                        $arrived[j]=1$;
while ($\neg arrived[j]$) ;            while ($\neg arrived[i]$) ;
$arrived[j]=0$;                        $arrived[i]=0$;
```

- *arrived*[i] that is true if process Π_i has arrived.
- Line 3 is the actual busy wait for the other process.
- A process immediately resets the flag it has waited for (lines 2,4).
- Lines 1,2 ensure that the flag has been reset before a process waits on it.

Barrier with Recursive Doubling II

```
parallel recursive-doubling-barrier {
    const int $d = 4$;  const int $P = 2^d$;
    int $arrived[d][P]$={$0[P \cdot d]$};                // flag variables
    process $\Pi$ [int $p \in \{0, ..., P-1\}$] {
        int $i$, $q$;
        while (1) {
            Computation;
            for ($i = 0$; $i < d$; $i$++) {              // all stages
                $q = p \oplus 2^i$;                      // flip bit $i$
                while ($arrived[i][p]$) ;
                $arrived[i][p]$=1;
                while ($\neg arrived[i][q]$) ;
                $arrived[i][q]$=0;
            }
        }
    }
}
```

What you should remember

- A parallel computation consists of a set of interacting sequential processes.
- Apart from the distribution of the work the synchronisation of the processes is also important
- Typical synchronisation mechanisms are: Mutual exclusion, condition synchronisation and barriers.

OpenMP

- OpenMP is a special implementation of multithreading
- current version 3.0 released in May 2008
- available for Fortran and C/C++
- works for different operating systems (e.g. Linux, Windows, Solaris)
- integrated in various compilers (e.g. Intel icc > 8.0, gcc > 4.2, Visual Studio $>=$ 2005, Sun Studio, ...)

Thread Model of OpenMP

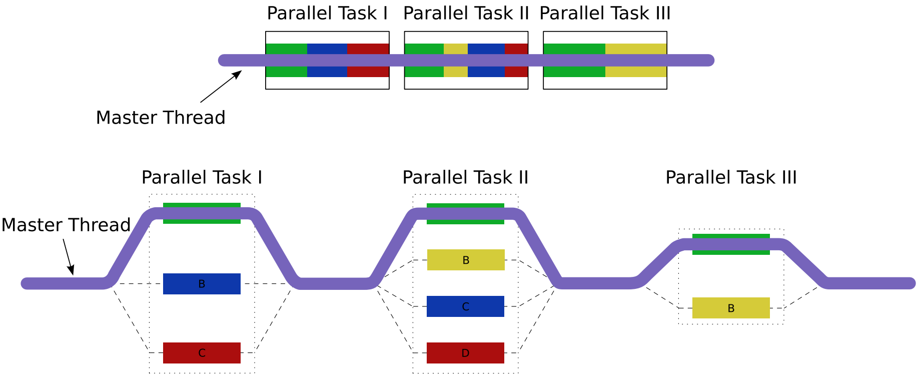


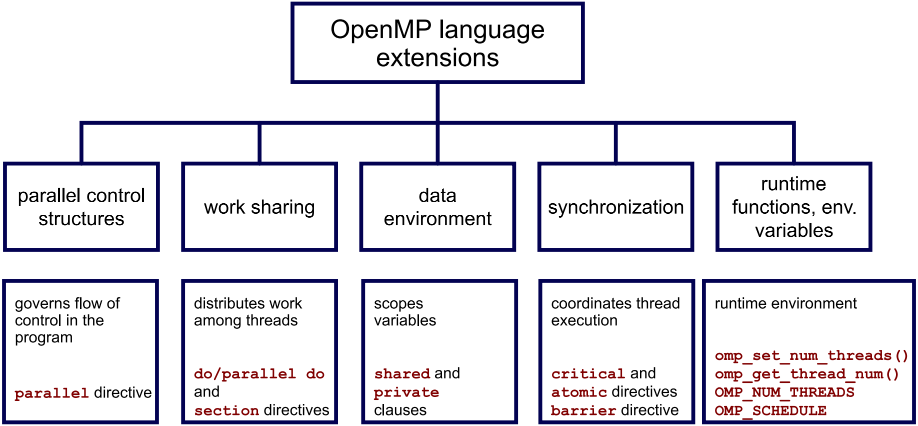
figure from Wikipedia: “OpenMP”

OpenMP directives

OpenMP directives

- are an extension of the language
- are created by writting special pre-processor commands into the source code, which are interpreted by a suitable compiler
- start with
  `#pragma omp`
  followed by a keyword and optionally some arguments

OpenMP Constructs



OpenMP language extensions

parallel control structures

work sharing

data environment

synchronization

runtime functions, env. variables

governs flow of control in the program

parallel directive

distributes work among threads

do/parallel do and **section** directives

scopes variables

shared and **private** clauses

coordinates thread execution

critical and **atomic** directives **barrier** directive

runtime environment

omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

figure from Wikipedia: “OpenMP”

Scalar Product with OpenMP

```
#ifdef _OPENMP
#include<omp.h>
#endif

double ScalarProduct(std::vector<double> a,
                     std::vector<double> b)
{
  const int N=a.size();
  int i;
  double sum = 0.0;
  #pragma omp parallel for
    for (i=0;i<N;++i)
      sum += a[i] * b[i];
  return(sum);
}
```

Thread Initialisation

parallel starts a block which is run in parallel

for the iterations are distributed among the threads (often combined with `parallel` to one line `\#pragma omp parallel for`.

sections inside a `sections` block there are several independent `section` blocks which can be executed independently.

if if an `if` is introduced in the thread initialisation command, the block is only executed in parallel if the condition after `if` is true.

Synchronisation

critical mutual exclusion: the block is only executed by one thread at a time.

atomic same as `critical` but tries to use hardware instructions

single the part inside the `single` block is only executed by one thread. The other threads are waiting at the end of the block.

master the part inside the `master` block is only executed by the master thread. It is skipped by all other threads.

barrier each thread waits till all threads have reached the barrier

nowait usually threads wait at the end of a block. If `nowait` is used they continue immediately

Accessibility of Variables

shared All threads are accessing the same (shared) data.

private Each thread has its own copy of the data.

default is used to specify what's the default behaviour for variables. Can be `shared`, `private` or `none`.

reduction If `reduction(`*`operator`*`,`*`variable`*`)` is specified, each thread uses a local copy of *`variable`* but all local copies are combined with the operator *`operator`* at the end. Possible operators are `+ * - / & ^ |`

Scheduling of Parallel For-Loops

The distribution of the total number of iterations to the individual threads can be influenced by the argument `schedule` of the pragma `for`. `schedule` is followed in brackets by one of five parameters and optionally after a comma the chunk size. The chunk size has to be an integer value known at compile time. The default schedule is implementation dependent.

- static chunks are distributed at the beginning of the loop to the different threads.
- dynamic each time a thread finishes its chunk it gets a new chunk.
- guided the chunk size is proportional to the number of unassigned iterations divided by the number threads (i.e. chunk size is getting smaller with time).
- runtime scheduling is controlled by the runtime variable `OMP_SCHEDULE`. It is illegal to specify a chunk size.
- auto scheduling is done by the compiler and/or runtime environment.

```
#define CHUNK_SIZE 10
#pragma omp parallel for schedule(dynamic,CHUNK_SIZE)
```

Better Scalar Product with OpenMP

```
#ifdef _OPENMP
#include<omp.h>
#endif

double ScalarProduct(std::vector<double> a,
                     std::vector<double> b)
{
  const int N=a.size();
  int i;
  double sum = 0.0, temp;
  #pragma omp parallel for shared(a,b,sum) private(temp)
    for (i=0;i<N;++i)
    {
      temp = a[i] * b[i];
      #pragma omp atomic
      sum += temp;
    }
  return(sum);
}
```

Improved Scalar Product with OpenMP

```
#ifdef _OPENMP
#include<omp.h>
#endif

double ScalarProduct(std::vector<double> a,
                     std::vector<double> b)
{
  const int N=a.size();
  int i;
  double sum = 0.0;
  #pragma omp parallel for shared(a,b) reduction(+:sum)
    for (i=0;i<N;++i)
      sum += a[i] * b[i];
  return(sum);
}
```

Parallel Execution of different Functions

```
#pragma omp parallel sections
{
  #pragma omp section
  {
    A();
    B();
  }
  #pragma omp section
  {
    C();
    D();
  }
  #pragma omp section
  {
    E();
    F();
  }
}
```

Special OpenMP Functions

There is a number of special functions which are defined in `omp.h`, e.g.

- `int omp_get_num_procs();`
  returns number of available processors
- `int omp_get_num_threads();`
  returns number of started threads
- `int omp_get_thread_num();`
  returns number of this thread
- `void omp_set_num_threads(int i);`
  set the number of threads to be used

```
#ifdef _OPENMP
#ifdef _OPENMP
#include<omp.h>
#endif
#include<iostream>
const int CHUNK_SIZE=3;

int main(void)
{
  int id;
  std::cout << "This␣computer␣has␣" << omp_get_num_procs() << "␣processors"
       << std::endl;
  std::cout << "Allowing␣two␣threads␣per␣processor" << std::endl;
  omp_set_num_threads(2*omp_get_num_procs());

  #pragma omp parallel default(shared) private(id)
  {
    #pragma omp for schedule(static,CHUNK_SIZE)
    for (int i = 0; i < 7; ++i)
    {
      id = omp_get_thread_num();
      std::cout << "Hello␣World␣from␣thread␣" << id << std::endl;
    }
    #pragma omp master
      std::cout << "There␣are␣" << omp_get_num_threads() << "␣threads" <<
           std::endl;
  }
  std::cout << "There␣are␣" << omp_get_num_threads() << "␣threads" <<
       std::endl;

  return 0;
}
```

Output

```
This computer has 2 processors
Allowing two threads per processor
Hello World from thread 1
Hello World from thread Hello World from thread 0
Hello World from thread 0
Hello World from thread 0
2Hello World from thread
1
Hello World from thread 1
There are 4 threads
There are 1 threads
```

Compiling and Environment Variables

OpenMP is activated with special compiler options. If they are not used, the #pragma statements are ignored and a sequential program is created. For icc the option is -openmp, for gcc it is -fopenmp

The environment variable OMP_NUM_THREADS specifies the maximal number of threads. The call of the function omp_set_num_threads in the program has precedence over the environment variable.

Example (with gcc and bash under linux):

```
gcc -O2 -fopenmp -o scalar_product scalarproduct.cc
export OMP_NUM_THREADS=3
./scalar_product
```

For Further Reading

- OpenMP Specification
  http://openmp.org/wp/openmp-specifications/
- OpenMP Tutorial in four parts
  http://www.embedded.com/design/multicore/201803777 (this is part four with links to part one to three)
- Very complete OpenMP Tutorial/Reference
  https://computing.llnl.gov/tutorials/openMP
- Intel Compiler (free for non-commercial use)
  http://www.intel.com/cd/software/products/asmo-na/eng/340679.htm

Basic Approach to Parallelisation

We want to have a look at three steps of the design of a parallel algorithm:

Decomposition of the problem into independent subtasks to identify maximal possible parallelism.

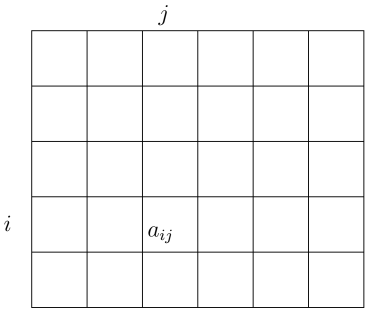
Control of Granularity to balance the expense for computation and communication.

Mapping of Processes to Processors to get an optimal adjustment of logical communication structure and hardware.

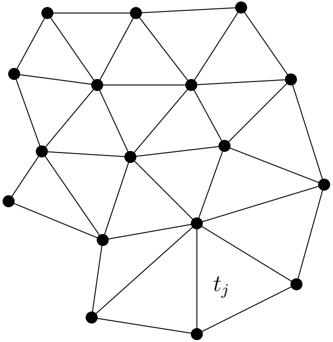
Data Decomposition

Algorithms are often tied to a special data structure. Certain operations have to be done for each data object.

Example: Matrix addition C = A + B, Triangulation



Matrix



Triangulation

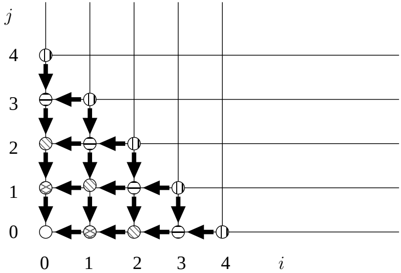
Data Interdependence

Often the operations for all data objects can't be done simultaneously.

Example: Gauß-Seidel/SOR-Iteration with lexicographic numbering.

Calculations are done on a grid, where the calculation at grid point (i,j) depends on the gridpoints $(i-1,j)$ and $(i,j-1)$. Grid point $(0,0)$ can be calculated without prerequisites. Only grid points on the diagonal $i+j = const$ can be calculated simultaneously.

Data interdependence complicates the data decomposition considerably.

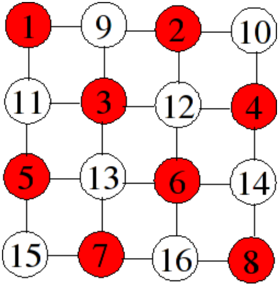


Increasing possible Parallelism

Sometimes the algorithm can be modified to allow for a higher data Independence.

With a different numbering of the unknowns the possible degree of parallelism for the Gauß-Seidel/SOR-iteration scheme can be increased:

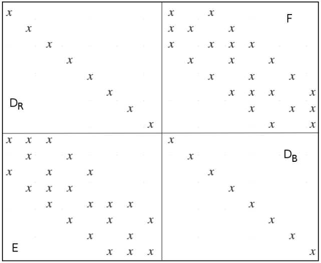
Every point in the domain gets a colour such that two neighbours never have the same colour. For structured grids two colours are enough (usually red and black are used). The unknowns of the same colour are numbered first, then the next colour . . . .



Red-Black-Ordering

The equations for all unknowns with the same colour are then independent of each other. For structured grids we get a matrix of the form

$$A = \begin{pmatrix} D_R & F \\ E & D_B \end{pmatrix}$$



However, while such a matrix transformation does not affect the convergence of solvers like Steepest-Descent or CG (as they only depend on the matrix condition) it can affect the convergence rate of relaxation methods.

Functional Decomposition

Functional decomposition can be done, if different operations have to be done on the same data set.

Example: Compiler

A compiler consists of

- lexical analysis
- parser
- code generation
- optimisation
- assembling

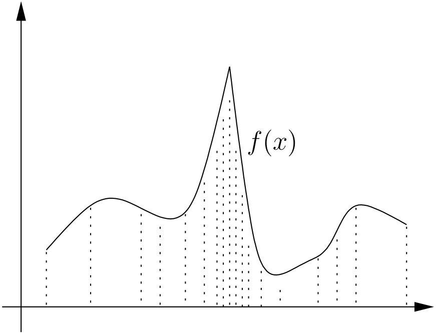
Each of these steps can be assigned to a separate process. The data can run through this steps in portions. This is also called “Macro pipelining”.

Irregular Problems

For some problems the decomposition cannot be determined a priory.

Example: Adaptive Quadrature of a function $f(x)$

The choice of the intervals depends on the function f and results from an evaluation of error predictors during the calculation.



Agglomeration and Granularity

The decomposition yields the maximal degree of parallelism, but it does not always make sense to really use this (e.g. one data object for each process) as the communication overhead can get too large.

Several subtasks are therefore assigned to each process and the communication necessary for each subtask is combined in as few messages as possible. This is called "agglomeration". This reduces the number of messages.

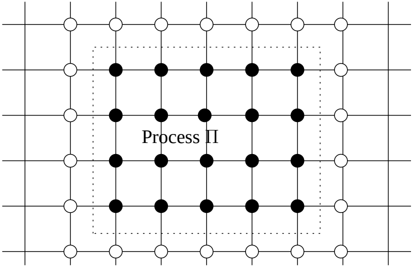
The *granularity* of a parallel algorithm is given by the ratio

$$\text{granularity} = \frac{\text{number of messages}}{\text{computation time}}$$

Agglomeration reduces the granularity.

Example: Gridbased Calculations

Each process is assigned a number of grid points. In *iterative* calculations usually the value at each node and it's neighbours is needed. If there is no interdependence all calculations can be done in parallel. A process with N grid points has to do $O(N)$ operations. With the partition it only needs to communicate $4\sqrt{N}$ boundary nodes. The ratio of communication to computation costs is therefore $O(N^{-1/2})$ and can be made arbitrarily small by increasing N. This is called *surface-to-volume-effect*.



Optimal Mapping of Processes to Processors

The set of all process Π forms a undirected graph $G_\Pi = (\Pi, K)$. Two processes are connected if they communicate.

The set of processors P together with the communication network (e.g. hypercube, field, ...) also forms a graph $G_P = (P, N)$.

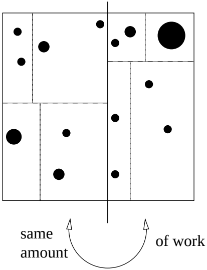
If we assume $|\Pi| = |P|$, we have to choose which process is executed on which processor. In general we want to find a mapping, such that processes which communicate are mapped to proximate processors. This optimisation problem is a variant of the *graph partitioning problem* and is unfortunately $\mathcal{NP}$-complete.

As the transmission time in cut-through networks of state-of-the-art parallel computers is nearly independent of the distance, the problem of optimal process to processor mapping has lost a bit of it's importance. If many processes have to communicate simultaneously (which is often the case!), a good positioning of processes is still relevant.

Load Balancing: Static Distribution

Bin Packing At beginning all processors are empty. Nodes are successively packed to the processor with the least work. This also works dynamically.

Recursive Bisection We make the additional assumption, that the nodes have a position in space. The domain is split into parts with an equal amount of work. This is repeated recursively on the subspaces.



Decomposition of Vectors

A vector $x \in \mathbb{R}^N$ is a ordered list of numbers where each index $i \in I = \{0, \dots, N-1\}$ is associated with a real number x_i.

Data decomposition is equivalent with a segmentation of the index set i in

$$I = \bigcup_{p \in P} I_p, \text{ with } p \neq q \Rightarrow I_p \cap I_q = \emptyset,$$

where P denotes the set of processes. For a good load balancing the subsets I_p, $p \in P$ should contain equal amounts of elements.

To get a coherent index set $\tilde{I}_p = \{0, \dots, |I_p| - 1\}$ we define the mappings

$$\begin{aligned} p: &\quad I \rightarrow P \text{ and} \\ \mu: &\quad I \rightarrow \mathbb{N} \end{aligned}$$

which reversibly associate each index $i \in I$ with a process $p(i) \in P$ and a local index $\mu(i) \in \tilde{I}_{p(i)}$: $\quad I \ni i \mapsto (p(i), \mu(i))$.

The inverse mapping $\mu^{-1}(p, i)$ provides a global index to each local index $i \in \tilde{I}_p$ and process $p \in P$ i.e. $p(\mu^{-1}(p, i)) = p$ and $\mu(\mu^{-1}(p, i)) = i$.

Common Decompositions: Cyclic

$$\begin{aligned} p(i) &= i \,\%\, P \\ \mu(i) &= i \div P \end{aligned}$$

$\div$ denotes an integer division and % the modulo function.

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	1	2	3	0	1	2	3	0	1	2	3	0
$\mu(i)$:	0	0	0	0	1	1	1	1	2	2	2	2	3

e.g. $I_1 = \{1, 5, 9\}$,
$\tilde{I}_1 = \{0, 1, 2\}$.

Common Decompositions: Blockwise

$$
p(i) = \begin{cases} i \div (B+1) & \text{if } i < R(B+1) \\ R + (i - R(B+1)) \div B & \text{else} \end{cases}
$$

$$
\mu(i) = \begin{cases} i \,\%(B+1) & \text{if } i < R(B+1) \\ (i - R(B+1)) \,\%\, B & \text{else} \end{cases}
$$

with $B = N \div P$ and $R = N \,\%\, P$

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	0	0	0	1	1	1	2	2	2	3	3	3
$\mu(i)$:	0	1	2	3	0	1	2	0	1	2	0	1	2

e.g. $I_1 = \{4, 5, 6\}$,
$\tilde{I}_1 = \{0, 1, 2\}$.

Decomposition of Matrices I

With a matrix $A \in \mathbb{R}^{N \times M}$ a real number a_{ij} is associated with each tupel $(i, j) \in I \times J$, with $I = \{0, \ldots, N-1\}$ and $J = \{0, \ldots, M-1\}$.
To be able to represent the decomposed data on each processor again as a matrix, we limit the decomposition to the one-dimensional index sets I and J.

We assume processes are organised as a two-dimensional field:

$$(p, q) \in \{0, \ldots, P-1\} \times \{0, \ldots, Q-1\}.$$

The index sets I, J are decomposed to

$$I = \bigcup_{p=0}^{P-1} I_p \text{ and } J = \bigcup_{q=0}^{Q-1} J_q$$

Decomposition of Matrices II

Each process (p, q) is responsible for the indices $I_p \times J_q$ and stores it's elements locally as $\mathbb{R}(\tilde{I}_p \times \tilde{J}_q)$-matrix.

Formally the decompositions of I and J are described as mappings p and μ plus q and ν:

$$\begin{aligned} I_p &= \{i \in I \mid p(i) = p\}, & \tilde{I}_p &= \{n \in \mathbb{N} \mid \exists i \in I : p(i) = p \wedge \mu(i) = n\} \\ J_q &= \{j \in J \mid q(j) = q\}, & \tilde{J}_q &= \{m \in \mathbb{N} \mid \exists j \in J : q(j) = q \wedge \nu(j) = m\} \end{aligned}$$

Decomposition of a 6 × 9 Matrix to 4 Processors I

$P = 1$, $Q = 4$ (columns), J: cyclic:

0	1	2	3	4	5	6	7	8	J
0	1	2	3	0	1	2	3	0	q
0	0	0	0	1	1	1	1	2	ν
🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	
🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	
🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	
🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	
🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	
🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	🟦 ...	🟩 ...	🟨 ...	🟥 ...	

$P = 4, Q = 1$ (rows), I: blockwise:

0	0	0	🟥 ...	🟥 ...	🟥 ...	🟥 ...	🟥 ...	🟥 ...	🟥 ...	🟥 ...	🟥 ...
1	0	1	🟥 ...	🟥 ...	🟥 ...	🟥 ...	🟥 ...	🟥 ...	🟥 ...	🟥 ...	🟥 ...
2	1	0	🟦 ...	🟦 ...	🟦 ...	🟦 ...	🟦 ...	🟦 ...	🟦 ...	🟦 ...	🟦 ...
3	1	1	🟦 ...	🟦 ...	🟦 ...	🟦 ...	🟦 ...	🟦 ...	🟦 ...	🟦 ...	🟦 ...
4	2	0	🟩 ...	🟩 ...	🟩 ...	🟩 ...	🟩 ...	🟩 ...	🟩 ...	🟩 ...	🟩 ...
5	3	0	🟨 ...	🟨 ...	🟨 ...	🟨 ...	🟨 ...	🟨 ...	🟨 ...	🟨 ...	🟨 ...
I	p	μ									

Decomposition of a 6 × 9 Matrix to 4 Processors II

$P = 2$, $Q = 2$ (field), I: cyclic, J: blockwise:

			0	1	2	3	4	5	6	7	8	J
			0	0	0	0	0	1	1	1	1	q
			0	1	2	3	4	0	1	2	3	ν
0	0	0	🟦 …	🟦 …	🟦 …	🟦 …	🟦 …	🟩 …	🟩 …	🟩 …	🟩 …	
1	1	0	🟥 …	🟥 …	🟥 …	🟥 …	🟥 …	🟨 …	🟨 …	🟨 …	🟨 …	
2	0	1	🟦 …	🟦 …	🟦 …	🟦 …	🟦 …	🟩 …	🟩 …	🟩 …	🟩 …	
3	1	1	🟥 …	🟥 …	🟥 …	🟥 …	🟥 …	🟨 …	🟨 …	🟨 …	🟨 …	
4	0	2	🟦 …	🟦 …	🟦 …	🟦 …	🟦 …	🟩 …	🟩 …	🟩 …	🟩 …	
5	1	2	🟥 …	🟥 …	🟥 …	🟥 …	🟥 …	🟨 …	🟨 …	🟨 …	🟨 …	
I	p	μ										

Optimal Decomposition

There is no overall best solution for the decomposition of matrices and vectors!

- In general a good load balancing is achieved if the subsets of the matrix are more or less quadratic.
- A good coordination with the algorithm used is usually more important. For example cyclic decomposition is a good solution for LU-decomposition, but not for the solution of the resulting triangular systems.
- Furthermore linear algebra is rarely a goal in it's own, but used in a more general context, like the solution of partial differential equations. The decomposition is then often given by the discretisation and the algorithm has to be flexible enough to deal with it.

Matrix-Vector Multiplication (fully-occupied matrix)

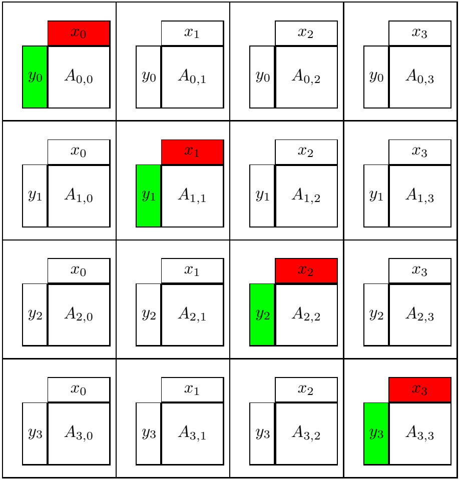
Aim: Calculate the product $y = Ax$ of a matrix $A \in \mathbb{R}^{N\times M}$ and a vector $x \in \mathbb{R}^M$.

Example: Matrix A distributed blockwise in a field, input vector x also blockwise as well as the result vector y.

The vector segment x_q is needed in each processor column.

Then each processor can calculate the local product $y_{p,q} = A_{p,q}x_q$.

Finally the complete result $y_p = \sum_q y_{p,q}$ is collected in the diagonal processor (p, p) with an all-to-one communication.



Matrix-Vector Multiplication: Parallel Runtime

Parallel runtime for a $N \times N$-matrix and $\sqrt{P} \times \sqrt{P}$ processors with a cut-through communication network:

$$
\begin{aligned}
T_P(N,P) &= \underbrace{\left(t_s + t_h + t_w \overbrace{\frac{N}{\sqrt{P}}}^{\text{vector}}\right) \operatorname{ld}\sqrt{P}}_{\substack{\text{Distribution of } x\\ \text{to column}}} + \underbrace{\left(\frac{N}{\sqrt{P}}\right)^2 2t_f}_{\substack{\text{local matrix-}\\ \text{vector-mult.}}} \\
&\quad + \underbrace{\left(t_s + t_h + t_w \frac{N}{\sqrt{P}}\right) \operatorname{ld}\sqrt{P}}_{\substack{\text{reduction}\\ (t_f \ll t_w)}} = \\
&= \operatorname{ld}\sqrt{P}(t_s + t_h)2 + \frac{N}{\sqrt{P}} \operatorname{ld}\sqrt{P} 2t_w + \frac{N^2}{P} 2t_f
\end{aligned}
$$

The contribution of the communication gets arbitrarily small if P and $N \rightarrow \infty$.

What you should remember

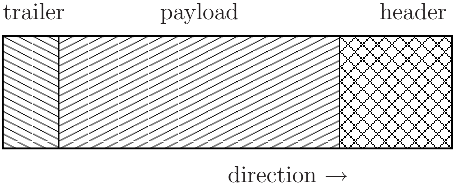
- The design of a parallel algorithm starts with decomposition of the task into parts which can be executed in parallel.
- Agglomerations is used to reduce the granularity and thus the communication overhead.
- Data decomposition is often important but can be done in different ways.
- There is no best solution for the decomposition of matrices and vectors. A good coordination with the algorithm is necessary.

Message Passing

- Developed in the 60s
- Aim: Better structuring of parallel programs (networks didn't exist yet)
- Idea: Data which is needed by other processors is send as messages over the network
- Various powerful solutions available. Differences in elegance.
  Examples:
  - PVM (Parallel Virtual Machine) developed since 1989
  - **MPI (Message Parsing Interface) developed since 1994**

Message Passing I

- Users view: Copy (contiguous) memory block from one address space to the other.
- Message is subdivided into individual packets.
- Network is packet-switched.
- A packet consists of an envelope and the data:



- Header: Destination, size and kind of data.
- Payload: Size ranges from some bytes to kilobytes.
- Trailer: E.g. checksum for error detection.

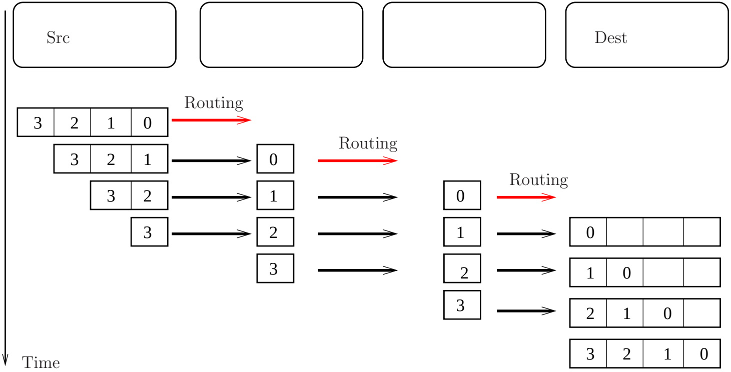
Message Passing II

- Communication protocol: Flow-control, error detection and correction.
- Time to send n bytes can be modelled as

$$t_{mess}(n) = t_s + n * t_b,$$

  t_s: latency, t_b: time per byte, t_b^{-1}: bandwidth.
- Latency is mostly software overhead, not hardware, and depends on the communication protocol.
- TCP/IP is an expensive wide-area network protocol with $t_s \approx 100\mu s$ (in Linux).
- Specialised networks with low-overhead protocols have $t_s \approx 3 \ldots 5\mu s$.

Cut-through Routing



- Pipelining on word level.
- Time to send n bytes: $t_{CT}(n, N, d) = t_s + t_h d + t_b n$.
- Time is essentially independent of distance since $t_h \ll t_s$.

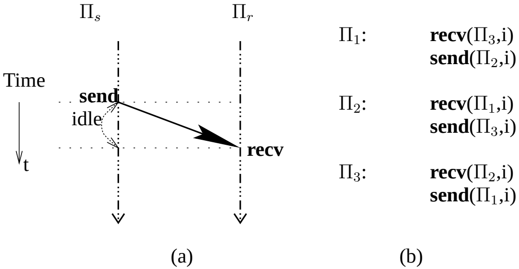
Send/Receive

The instructions for synchronous point-to-point communication are:

- **send**($dest - process$,$expr_1$,...,$expr_n$) Sends a message to the process $dest - process$ containing the expressions $expr_1$ to $expr_n$.
- **recv**($src - process$, var_1,..., var_n) Receives a message from the process $src - process$ and stores the results of the expressions in the variables var_1 to var_n. The variables have to be of maching type.

Blocking

Both **send** and **recv** are blocking, i.e. they are only finished, after the end of the communication. This synchronises the involved processes. Both sending and receiving process have to execute a matching pair of **send** and **recv** to avoid a deadlock.



Π_1: **recv**(Π_3,i)
send(Π_2,i)

Π_2: **recv**(Π_1,i)
send(Π_3,i)

Π_3: **recv**(Π_2,i)
send(Π_1,i)

Figure: (a) Synchronisation of two processes by a **send**/**recv** pair. (b) Example of a deadlock.

Guards

Blocking communication is not sufficient for all possible tasks.
Sometimes a process does not know which of several partner processes is ready for data exchange.
Possible Solution: Non-blocking functions for detection if a partner process is ready for data reception or sending:

- **int sprobe**($dest - process$)
- **int rprobe**($src - process$).

sprobe returns 1 (*true*), if the specified process is ready for a **recv**-operation, i.e. a **send**-command would *not* block the sender.
rprobe tests, if a **recv**-command would lead to a blockade. To avoid blocking of processes a communication instruction would be written as

- **if** (**sprobe**(Π_d)) **send**($\Pi_d, \ldots$);

The instructions **sprobe** and **rprobe** are also called *"guards*".

Implementation of **rprobe** and **sprobe**

Implementation of **rprobe** is easy. A **send**-instruction sends a message (or a part of it) to the receiver, where it is stored. The receiving process only needs to look up locally if a corresponding message (or part of it) has arrived.

Implementation of **sprobe** is more complicated, as usually the receiving process does not return any information to the sender. As one of the two instructions **sprobe**/**rprobe** is sufficient (at least in principle) only **rprobe** is used in practise.

Receive Any

- **recv_any**(*who*,var_1,. . . ,var_n)

  similar effect as **rprobe**

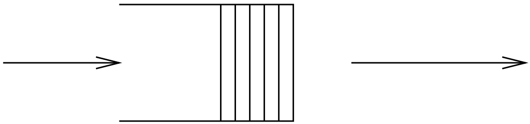
  allows reception of a message from any process

  sender is stored in the variable *who*

Asynchronous Communication

- **asend**$(dest - process, expr_1, \ldots, expr_n)$
- **arecv**$(src - process, var_1, \ldots, var_n)$

have the same semantic as **send**/**recv**, but are *non-blocking*.
In principle a process can execute any amount of **asend** instructions without delay.
The involved processes are *not* synchronised implicitly.
Beginning and end of the communication channel can be visualised as a waiting line, buffering all messages until they are accepted by the receiving process.



Check for Success

In practise buffer storage is limited. If the buffer is full, no further **asend** instruction can be executed. Therefore it is necessary to test if a former communication instruction has been completed.

asend/**arecv** are returning a unique identifier

- **msgid asend**(. . .)
- **msgid arecv**(. . .)

With this identifier the function

- **int** success(**msgid** *m*)

returns the state of the communication.

Mixing of Synchronous and Asynchronous Communication

We are going to allow mixing of the functions for asynchronous and synchronous communication. For example the combination of asynchronous sending and synchronous receiving can make sense.

Parallel Debugging

Debugging parallel programs is a complicate task. Possible tools are:

- Using `printf` or `std::cout`
- Writting to log files
- Using `gdb`
- Using specialised debuggers for parallel programs

Using `printf` or `std::cout`

```
std::cout << omp_get_thread_num() << ":␣a␣=␣" << a << std::endl;
```

- Output should be prefixed with the rank (number) of the process
- Output of different processes is mixed and written by the root process.
- The order of output from different processes must not be chronological
- If job terminates some output may never be written

Writing to log-files

```
template<class T>
void DebugOut(const std::string message, T i) const
{
    std::ostringstream buffer;
    buffer << "debugout" << myRank_;
    std::ofstream outfile(buffer.str().c_str(),std::ios::app);
    outfile << message << "␣" << i<< std::endl;
}
```

- Output from each processor goes to a separate file
- Output is complete for each processor (as file is always immediately closed afterwards)
- File is allways appended not erased ⇒ output of several runs can be mixed

Using gdb (with Message Passing)

Several instances of gdb can be started each attaching to one of the parallel processes using

```
gdb <program name> <PID>
```

To make sure that all processes are still at the beginning you can add an infinite loop

```
bool debugStop=true;
while (debugStop);
```

After gdb has attached the loop can be exited by using the gdb-command

```
set debugStop=true;
```

You may have to compile with

```
mpicxx -o <program name> -g -O0 <program source>
```

as the compiler may optimise the variable away if the option −O0 is absent.

Parallel Debuggers

- Parallel debuggers are providing a graphical user interface for the process describe before
- Commercial parallel debuggers are e.g. Totalview and DDT
- There is also a eclipse plugin available: PTP

The Message Passing Interface (MPI)

- Portable Library with functions for message exchange between processes
- Developed 1993-94 by a international board
- Available on nearly all computer platforms
- Free Implementations also for LINUX Clusters:**MPICH**[1] and **OpenMPI**[2] (former **LAM**)
- Properties of MPI:
    - library with C-, C++ and Fortran bindings (no language extension)
    - large variety of point-to-point communication functions
    - global communication
    - data conversion for heterogeneous systems
    - subset formation and topologies possible

MPI-1 consits of more than 125 functions, defined in the standard on 200 pages. We therefore only treat a small selection of it's functionality.

[1] http://www-unix.mcs.anl.gov/mpi/mpich

[2] http://www.open-mpi.org/

Simple Example in C (I)

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>  // provides MPI macros and functions

int main (int argc, char *argv[])
{
    int my_rank;
    int P;
    int dest;
    int source;
    int tag=50;
    char message[100];
    MPI_Status status;

    MPI_Init(&argc,&argv);  // begin of every MPI program

    MPI_Comm_size(MPI_COMM_WORLD,&P);  // number of
                                       // involved processes
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    // number of current process always between 0 and P-1
```

Simple Example in C (II)

```
    sprintf(message,"I␣am␣process␣%d␣of␣%d\n",my_rank,P);
    if (my_rank!=0)
    {
        dest = 0;
        MPI_Send(message,strlen(message)+1,MPI_CHAR,  // Send data
                 dest,tag,MPI_COMM_WORLD);            // (blocking)
    }
    else
    {
        puts(message);
        for (source=1; source<P; source++)
        {
            MPI_Recv(message,100,MPI_CHAR,source,tag,  // Receive data
                     MPI_COMM_WORLD,&status);          // (blocking)
            puts(message);
        }
    }

    MPI_Finalize();  // end of every MPI program

    return 0;
}
```

Simple Example in C++ (I)

```
#include <iostream>
#include <sstream>
#include <mpi.h>  // provides MPI macros and functions


int main (int argc, char *argv[])
{
    MPI::Init(argc,argv);  // begin of every MPI program

    int P = MPI::COMM_WORLD.Get_size(); // number of
                                        // involved processes
    int myRank = MPI::COMM_WORLD.Get_rank();
    int tag=50;

    // number of current process always between 0 and P-1
    if (myRank != 0)
    {
        std::ostringstream message;
        message << "I␣am␣process␣" << myRank << "␣of␣" << P << std::endl;
        int dest = 0;
        MPI::COMM_WORLD.Send(message.str().c_str(),      // Send data
            message.str().size()+1,MPI::CHAR,dest,tag);  // (blocking)
    }
```

Simple Example in C++ (II)

```
    else
    {
        std::cout << "I␣am␣process␣0␣of␣" << P << std::endl << std::endl;
        for (int source=1; source<P; ++source)
        {
            char message[100];
            MPI::COMM_WORLD.Recv(message,100,MPI_CHAR,  // Receive data
                 source,tag);                            // (blocking)
            std::cout << message << std::endl;
        }
    }

    MPI::Finalize();  // end of every MPI program

    return 0;
}
```

Compilation of Program

- Sample program is written in SPMD-Stile. This is not prescribed by the MPI Standard, but makes starting of program easier.
- Compiling, linking and execution is different for every implementation.
- Many implementations contain shell-scripts, which hide the location of the libraries. For MPICH the commands to compile the program and start 8 processes are

  ```
  mpicc -o hello hello.c
  mpirun -machinefile machines -np 8 hello
  ```

  In this case the names of the computers used are taken from the file `machines`.

  For C++ programs the command for compilation is

  ```
  mpicxx -o hello hello.c
  ```

Output of the Example Programs (with P=8)

```
I am process 0 of 8

I am process 1 of 8

I am process 2 of 8

I am process 3 of 8

I am process 4 of 8

I am process 5 of 8

I am process 6 of 8

I am process 7 of 8
```

Structure of MPI-Messages

MPI-Messages consist of the actual *data* and an *envelope* comprising:

1. number of the sender
2. number of the receiver
3. tag: an integer value to mark different messages between identical communication partners
4. communicator: subset of processes + communication context. Messages belonging to different contexts don't influence each other. Sender and receiver have to use the same communicator. The communicator `MPI_COMM_WORLD` is predefined by MPI and contains all started processes. In C++ communicators are objects over which the messages are sent. The communicator object `MPI::COMM_WORLD` is already predefined.

Initialising and Finishing

```
int MPI_Init(int *argc, char ***argv)

void Init(int& argc, char**& argv)
void Init()
```

Before the first MPI functions are used, `MPI_Init` / `MPI::Init` has to be called.

```
int MPI_Finalize(void)

void Finalize()
```

After the last MPI function call `MPI_Finalize` / `MPI::Finalize` must be executed to get a defined shutdown of all processes.

Communicator

All MPI communication functions contain an argument of type `MPI_Comm` (in C) or are methods of a communicator object. Such a *communicator* contains the following abstractions:

- *Process group:* builds a subset of processes which take part in a global communication. The predefined communicator `MPI_COMM_WORLD` contains all started processes.
- *Context:* Each communicator defines it's own communication context. Messages can only be received within the same context in which they are send. It's e.g. possible for a numerical library to define it's own communicator. The messages of the library are then completely separated from the messages of the main program.
- *Virtual Topologies:* A communicator represents a set of processes $\{0, \ldots, P-1\}$. This set can optionally be provided with an additional structure, e.g. a multi-dimensional field or a general graph.
- *Additional Attributes:* An application (e.g. a library) can associate any static data with a communicator. The communicator is then only a means to preserve this data until the next call of the library.

Communicators in C++

In C++ communicators are objects of classes derived from a base class `Comm`. The available derived classes are

`Intracomm` for the communication inside a group of processes. `MPI::COMM_WORLD` is an object of class `Intracomm` as all processes are included in `MPI::COMM_WORLD`. There exist two derived classes for the formation of topologies of processes

- `Cartcomm` can represent processes which are arranged on a Cartesian topology
- `Graphcomm` can represent processes which are arranged along arbitrary graphs

`Intercomm` for the communication between groups of processes

Determining Rank and Size

```
int MPI_Comm_size(MPI_Comm comm, int *size)

int Comm::Get_size() const
```

The number of processes in a communicator is determined by the function MPI_Comm_size / Comm::Get_size(). If the communicator is MPI_COMM_WORLD this is equal to the total number of started processes.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)

int Comm::Get_rank() const
```

Each process has a unique number inside a group represented by a communicator. This number can be determined by MPI_Comm_rank / Comm::Get_rank().

Blocking Communication

```
int MPI_Send(void *message, int count, MPI_Datatype dt,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *message, int count, MPI_Datatype dt,
             int src, int tag, MPI_Comm comm,
             MPI_Status *status);

 void Comm::Send(const void* buf, int count,
             const Datatype& datatype, int dest, int tag) const
 void Comm::Recv(void* buf, int count, const Datatype& datatype,
             int source, int tag, Status& status) const
 void Comm::Recv(void* buf, int count, const Datatype& datatype,
             int source, int tag) const
```

The first three arguments message, count, and dt, specify the actual data. message points to a contiguous memory block containing count elements of type dt. The specification of the data type makes data conversion by MPI possible. The arguments dest, tag and comm form the envelope of the message (the number of the sender/receiver is given implicitly by the invocation).

Data Conversion

MPI implementations for heterogeneous systems are able to do a automatic conversion of the data representation. The conversion method is left to the particular implementation (e.g. by XDR).

MPI provides the architecture independent data types:

MPI_CHAR, MPI_UNSIGNED_CHAR, MPI_BYTE
MPI_SHORT, MPI_INT, MPI_LONG, MPI_LONG_LONG_INT,
MPI_UNSIGNED, MPI_UNSIGNED_SHORT, MPI_UNSIGNED_LONG,
MPI_FLOAT, MPI_DOUBLE and MPI_LONG_DOUBLE.

IN C++ the datatypes are:

MPI::CHAR, MPI::UNSIGNED_CHAR, MPI::BYTE
MPI::SHORT, MPI::INT, MPI::LONG, MPI::LONG_LONG_INT,
MPI::UNSIGNED, MPI::UNSIGNED_SHORT, MPI::UNSIGNED_LONG,
MPI::FLOAT, MPI::DOUBLE and MPI::LONG_DOUBLE.

The MPI data type `MPI_BYTE` / `MPI::BYTE` is *never* converted.

Status

```c
typedef struct {
    int count;
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

In C MPI_Status is a struct containing information about the number of received objects, source rank, tag and an error status.

```cpp
int Status::Get_source() const
void Status::Set_source(int source)
int Status::Get_tag() const
void Status::Set_tag(int tag)
int Status::Get_error() const
void Status::Set_error(int error)
```

In C++ an object of class Status provides methods to access the same information.

Varieties of Send

- *buffered send* (`MPI_Bsend` / `Comm::Bsend`): If the receiver has not yet executed a corresponding **recv**-function, the message is buffered by the sender. If enough memory is available, a "buffered send" is always terminated immediately. In contrast to asynchronous communication the sending buffer `message` can be immediately reused.
- *synchronous send* (`MPI_Ssend` / `Comm::Ssend`): The termination of a synchronous send indicates, that the receiver has executed the **recv**-function and has started to read the data.
- *ready send* (`MPI_Rsend` / `Comm::Rsend`): A ready send can only be started if the receiver has already executed the corresponding **recv**. The call leads to an error else .

MPI_Send and MPI_Receive II

The MPI_Send-command either has the semantic of MPI_Bsend or MPI_Ssend, depending on the implementation. Therefore MPI_Send can, but don't have to block. The sending buffer can be reused immediately in any case.

The function MPI_Recv is in any case blocking, e.g. it is only terminated if message contains data. The argument status contains source, tag and error status of the received message.

MPI_ANY_SOURCE / MPI::ANY_SOURCE and MPI_ANY_TAG / MPI::ANY_TAG can be used for the arguments src and tag respectively. Thus MPI_Recv contains the functionality of **recv_any**.

Guard Function

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
               int *flag, MPI_Status *status);
bool Comm::Iprobe(int source, int tag, Status& status) const
bool Comm::Iprobe(int source, int tag) const
```

is a non-blocking guard function for the receiving of messages. `flag` is set to true `true` ($\neq 0$) if a message with matching source and tag can be received. The arguments `MPI_ANY_SOURCE` / `MPI::ANY_SOURCE` and `MPI_ANY_TAG` / `MPI::ANY_TAG` are also possible.

Non-blocking Communication

MPI provides the functions

```
int MPI_ISend(void *buf, int count, MPI_Datatype dt,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *req);
int MPI_IRecv(void *buf, int count, MPI_Datatype dt,
              int src, int tag, MPI_Comm comm,
              MPI_Request *req);

Request Comm::Isend(const void* buf, int count,
              const Datatype& datatype, int dest, int tag)
                  const
Request Comm::Irecv(void* buf, int count, const Datatype&
    datatype,
              int source, int tag) const
```

for non-blocking communication. They imitate the respective communication operations. With the `MPI_Request` / `MPI::Request`-objects the state of the communication job can be checked (corresponding to our **msgid**).

MPI_Request-Objects

The state of the communication can be checked with MPI_Request-objects returned by the communication functions using the function (among others)

```
int MPI_Test(MPI_Request *req,int *flag, MPI_Status *status);
```

flag is set to true ($\neq 0$) if the communication job designated by req has terminated. In this case status contains information about sender, receiver and error state.

```
bool Request::Test(Status& status)
bool Request::Test()
```

In C++ the Test method of the Request object returned by the communicator method returns true if the job initiated by the method call has terminated.
It is important to mind that the MPI_Request-object becomes invalid as soon as MPI_Test / Request::Test returns flag==true / true. It must not be used thereafter, as the MPI_Request-objects are managed by the MPI-implementation (so-called opaque objects).

Global Communication

MPI also offers functions for global communication where all processes of a communicator participate.

```
int MPI_Barrier(MPI_Comm comm);

void Intracomm::Barrier() const
```

blocks every process until all processes have arrived (e.g. until they have executed this function).

```
int MPI_Bcast(void *buf, int count, MPI_Datatype dt,
              int root, MPI_Comm comm);

void Intracomm::Bcast(void* buffer, int count,
              const Datatype& datatype, int root) const
```

distributes the message of process root to all other processes of the communicator.

Collection of Data

MPI offers various functions for the collection of data. For example:

```
int MPI_Reduce(void *sbuf, void *rbuf, int count,
        MPI_Datatype dt, MPI_Op op, int root, MPI_Comm comm);

void Intracomm::Reduce(const void* sendbuf, void* recvbuf, int
    count,
        const Datatype& datatype, const Op& op, int root) const
```

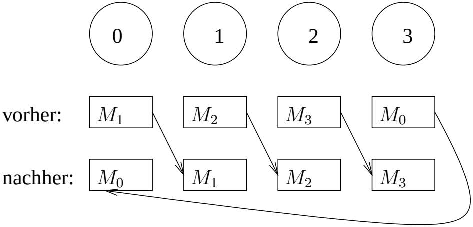
combines the data in the send buffer sbuf of all processes with the associative operation op. The result is available in the receive buffer rbuf of process root. As operations op e.g. MPI_SUM, MPI_MAX and MPI_MIN can be used to calculate the sum, maximum or minimum over all processes.

Remark

Global communications have to be called with maching arguments (e.g. root in MPI_Reduce) by all processes of a communicator.

Shifting along Ring: Creation of Deadlocks

Problem: Each process $p \in \{0, \dots, P-1\}$ has to transfer data to $(p+1)\%P$.



With *blocking* communication functions a deadlock is created using

$\dots$
send$(\Pi_{(p+1)\%P}, msg)$;
recv$(\Pi_{(p+P-1)\%P}, msg)$;
$\dots$

Deadlock Prevention

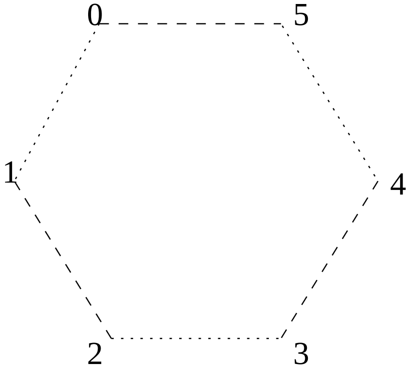
A solution with optimal degree of parallelism can be reached by colouring. Let $G = (V, E)$ be a graph with

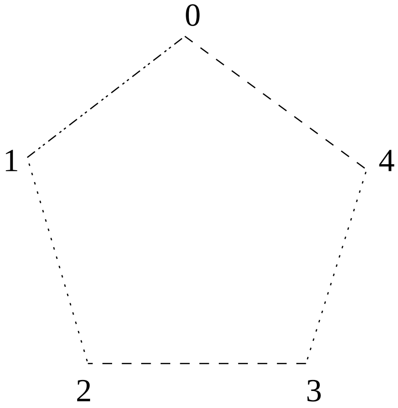
$$\begin{aligned} V &= \{0, \dots, P-1\} \\ E &= \{e = (p, q) | \text{process } p \text{ has to communicate with process } q\} \end{aligned}$$

Each edge then has to be given a colour so that the colours of all edges meeting at each node are unique. The colour assignment is given by the mapping $c\colon E \rightarrow \{0, \dots, C-1\}$, where C is the number of colours necessary. The communication can then be done in C steps, whereas only messages along edges of colour i are exchanged in step $0 \leq i < C$.

Number of Colours

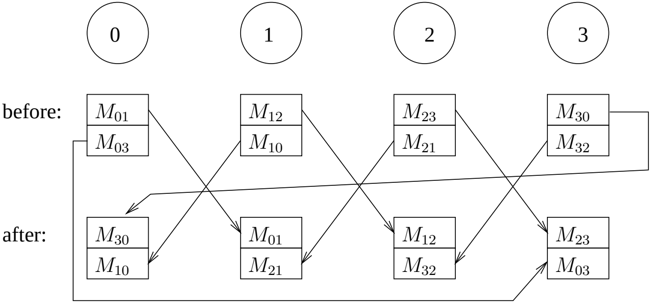
Two colours are needed for shifting along a ring if P is even, but three colours are needed if P is odd.





Exchange with all Neighbours

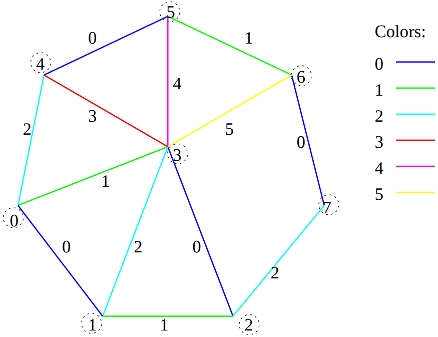
The algorithm can easily be modified to shift messages in both directions. Each Process exchanges then messages with both neighbours in the graph.



The number of colours is the same as in the case of simple shifting. Two messages are send along each edge. Deadlocks are prevented if the process with the smaller number sends first.

General Graphs

For general, undirected graphs the determination is not so simple and it can also be necessary to determine the colouring at runtime. Distributed algorithms are available to solve this problem.



Alternative: Timestamps

Creating Communicators based on Colours

A new communicator can be created with the function

```
int MPI_Comm_split(MPI_Comm comm, int colour,
                   int key, MPI_Comm *newcomm);

Intracomm Intracomm::Split(int colour, int key) const
```

`MPI_Comm_split` is a collective operation, executed by *all* processes of the communicator `comm`. All processes with the same value of the argument `colour` form a new communicator. The ordering (rank) within the new communicator is controlled by the argument `key`.

What you should remember

- Message passing is a general concept for data exchange. There are different realisations of this concept.
- Two different types of communication exist: Blocking (or synchronous) and Non-blocking (or asynchronous).
- MPI is a standardised message passing system. Different implementations exist implementing the MPI standard (e.g. **MPICH**, **OpenMPI**).
- C and C++ versions of the MPI functions exist
- Deadlocks in communication can occur with blocking communication functions and need to be averted by techniques like colouring or timestamps.

For Further Reading

- *MPI: The different version of the Message-Passing Interface Standard*
  http://www.mpi-forum.org/docs/
- *MPICH-A Portable Implementation of MPI*
  http://www-unix.mcs.anl.gov/mpi/mpich
- *Open MPI: Open Source High Performance Computing*
  http://www.open-mpi.org/
- *Overview of available MPI Tutorials*
  http://www-unix.mcs.anl.gov/mpi/tutorial/

Introduction

- We want to solve a problem Π on a parallel computer.
- Example: “Solve system of linear equations $Ax = b$ with $A \in \mathbb{R}^{N\times N}$ and $x, b \in \mathbb{R}^{N}$”.
- Problem description does not include *how* it is solved.
- Π has a *problem size* parameter $N \in \mathbb{N}$.
- Π is solved on a parallel machine with P identical processors and a given communication network.
- Π is solved
  - on 1 processor with a *sequential* algorithm and
  - on P processors with a *parallel* algorithm.

How “good” is the parallel algorithm ?

Time Measurements

Depending on problem size N and processor number P we can define the following run-times:

- *Sequential run-time* $T_S(N)$: Time needed by a specified algorithm to solve Π for problem size N on one processor of the parallel machine.
- *Best sequential run-time* $T_{best}(N)$: Run-time of a *hypothetical* sequential algorithm that solves Π for any problem size N in the shortest possible time.
- *Parallel run-time* $T_P(N, P)$: Run-time of a given parallel algorithm to solve Π for problem size N on a parallel machine with P processors.

Speedup

Definition (Speedup)

Given the time measurements we can define the *speedup*

$$S(N, P) = \frac{T_{best}(N)}{T_P(N, P)}.$$

Remark.

The speedup is defined with respect to the best sequential algorithm! If it were defined via some $T_S(N)$, any speedup could be achieved by comparing against a slow sequential algorithm.

Speedup Bound

Theorem (Speedup bound)

The speedup $S(N,P)$ fullfills the inequality

$$S(N,P) \leq P.$$

Proof.

Suppose we have $S(N,P) > P$, then simulating the parallel algorithm on one processor would need time $PT_P(N,P)$ and

$$PT_P(N,P) = P\frac{T_{best}(N)}{S(N,P)} < T_{best}(N)$$

which is a contradiction to the definition of T_{best}. □

Superlinear Speedup

Remark.

The proof above assumes that simulation takes no additional time. There are inherently parallel algorithms where the sequential algorithm *must* simulate a parallel algorithm: Given a program for computing $f(n)$, find $n \in \{1, \ldots, P\}$ where its run-time is minimal.

Superlinear Speedup.

Some people measure speedups greater than P. This is most often because T_{best} must be replaced by some T_S in practise. The most common situation is that for increasing P and fixed N the local problems fit into cache and the sequential code is not cache-aware.

Efficiency

Definition (Efficiency)

The *efficiency* of a parallel algorithm is defined as

$$E(N,P) = \frac{S(N,P)}{P} = \frac{T_{best}(N)}{P T_P(N,P)}.$$

Theorem (Efficiency bound)

The speedup bound immediately gives

$$E(N,P) \leq 1.$$

Remark.

Interpretation: $E \cdot P$ processors are effectively working on the solution of the problem, $(1 - E) \cdot P$ are overhead.

Other Measures

Definition (Cost)

The *cost* of a parallel algorithm is defined as

$$C(N, P) = PT_P(N, P).$$

In contrast to the previous numbers it is not dimensionless.

Definition (Cost optimality)

An algorithm is called *cost optimal* if its cost is proportional to T_{best}. Then its efficiency $E(N, P) = T_{best}/C(N, P)$ is constant.

Definition (Degree of parallelism)

$\Gamma(N)$ is the maximum number of machine instructions that can be executed in parallel. Obviously $\Gamma(N) = O(T_P(N, 1))$.

Example: Scalar Product of Two Vectors

Run-time of best sequential algorithm is given by

$$T_S(N) = N2t_a,$$

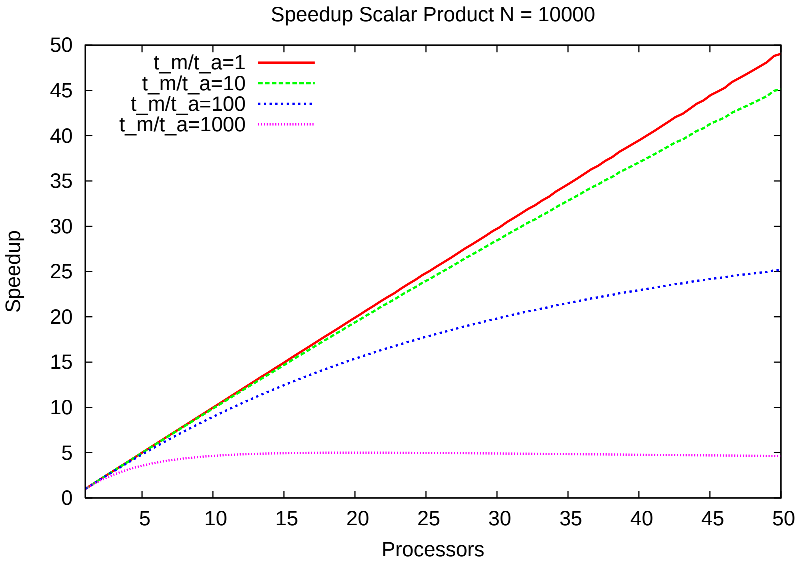
t_a: time for a floating point operation.
Run-time of parallel algorithm using tree combine:

$$T_P(N,P) = \left\lceil \frac{N}{P} \right\rceil 2t_a + \lceil \log_2 P \rceil (t_m + t_a),$$

t_m: time to send a number.
Speedup is

$$S(N,P) = \frac{N2t_a}{\left\lceil \frac{N}{P} \right\rceil 2t_a + \lceil \log_2 P \rceil (t_m + t_a)} = \frac{P}{\underbrace{\left\lceil \frac{N}{P} \right\rceil \frac{P}{N}}_{\text{load imbalance}} + \underbrace{\frac{P\lceil \log_2 P \rceil}{N} \frac{t_m+t_a}{2t_a}}_{\text{communication}}}.$$

Speedup Graph Scalar Product



Example: Gaussian Elimination

Run-time of best sequential algorithm is given by

$$T_S(N) = N^3 \frac{2}{3} t_a,$$

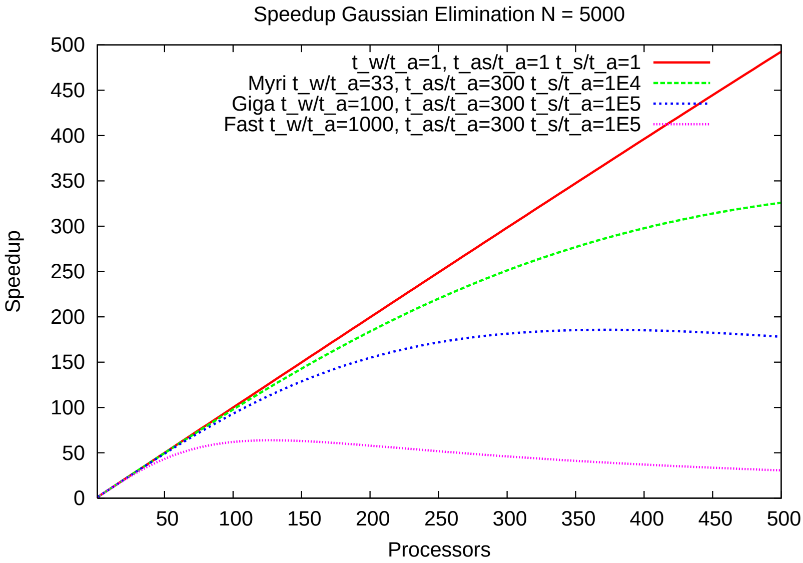
t_a: time for a floating point operation.
Run-time of parallel algorithm (row-wise cyclic, asynchronous):

$$\begin{aligned} T_P(N,P) &= (P-1)(t_s + Nt_m) + \sum_{m=N-1}^{1} \left(\left\lceil \frac{m}{P} \right\rceil m 2 t_a + t_{as} \right) \\ &\approx \frac{2}{3} \frac{N^3}{P} t_a + N t_{as} + P t_s + N P t_m, \end{aligned}$$

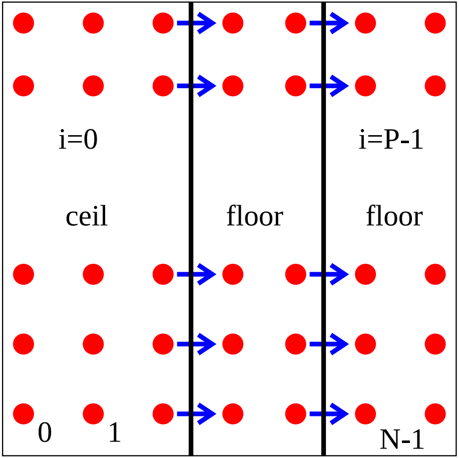
t_s: message latency, t_{as}: asynchronous latency.
Speedup is

$$S(N,P) = \frac{P}{1 + \frac{3}{2} \frac{P}{N^2} \left(P \frac{t_w}{t_a} + \frac{t_{as}}{t_a} + \frac{P}{N} \frac{t_s}{t_a} \right)}$$

Speedup Graph Gaussian Elimination



Example: A Grid Algorithm



- Sequential algorithm processes $N \times N$ grid points in lexicographic order from left to right, bottom to top. Time per grid point is t_{op}.
- Uses Pipelining.
- Granularity is fine.
- Process points sequentially until Π_{P-1} is busy.
- First $N \bmod P$ processors have $\lceil N/P \rceil$ points per row.
- Remaining $N - N \bmod P$ processors have $\lfloor N/P + 1 \rfloor$ points per row.

Grid Algorithm Speedup

Run-time of best sequential algorithm is given by

$$T_S(N) = N^2 t_{op},$$

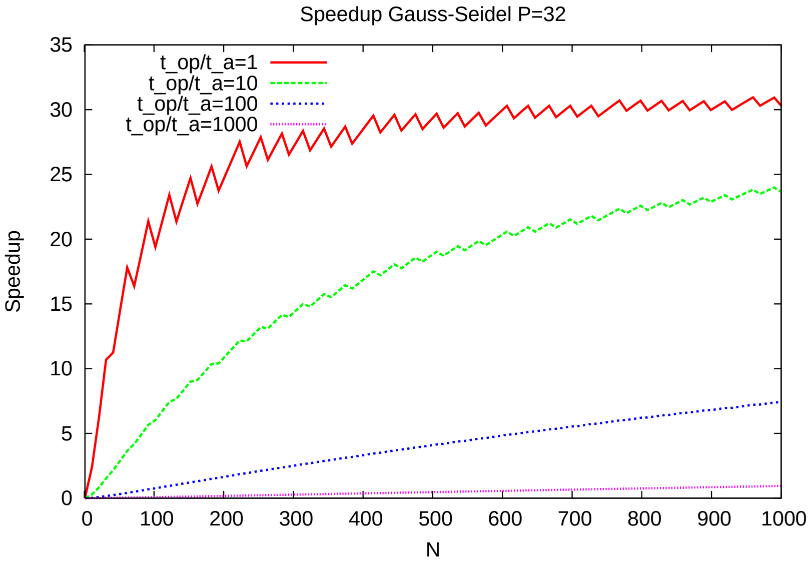
t_a: time for a floating point operation.
Run-time of parallel algorithm:

$$T_P(N,P) = \underbrace{\left(N - \left\lfloor \frac{N}{P} \right\rfloor\right) t_{op} + (P-2)t_{op}}_{\text{pipeline start-up}} + \underbrace{N\left(\left\lceil \frac{N}{P} \right\rceil t_{op} + t_m\right)}_{\text{last proc}}.$$

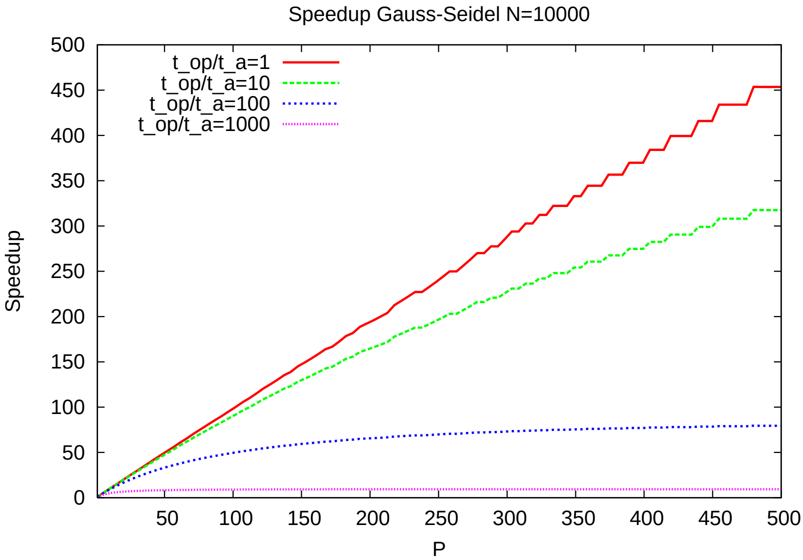
Speedup is

$$S(N,P) = \frac{P}{\underbrace{\frac{P}{N}\left\lceil \frac{N}{P} \right\rceil}_{\text{load imbalance}} + \underbrace{\frac{P(N - \lfloor N/P \rfloor)}{N^2}}_{\text{sequential part}} + \underbrace{\left(\frac{P(P-2)}{N^2} + \frac{P}{N}\right)\frac{t_m}{t_a}}_{\text{communication}}}$$

Speedup Graph Grid Algorithm Variable N



Speedup Graph Grid Algorithm Variable P



Reasons For Non-optimal Speedup

We can identify the following reasons for non-optimal speedup:

- *Load imbalance*: Not every processor has the same amount of work to do.
- *Sequential part*: Not all operations of the sequential algorithm can be processes in parallel.
- *Additional operations*: The optimal sequential algorithm cannot be parallelised directly and must be replaced by a slower but parallelisable algorithm.
- *Communication overhead*: Depends relative cost of computation and communication.

Scalability

- Scalability is the ability of a parallel algorithm to use an increasing number of processors: **How does $S(N, P)$ behave with P?**
- $S(N, P)$ has two arguments. **What about N?**
- We consider several different choices.

Fixed Size Scaling and Amdahl's Law

Fixed size scaling

Very simple: Choose N to be fixed. Equivalently, we can fix the sequential execution time $T_{\text{best}} = T_{\text{fix}}$.

Amdahl's Law (1967)

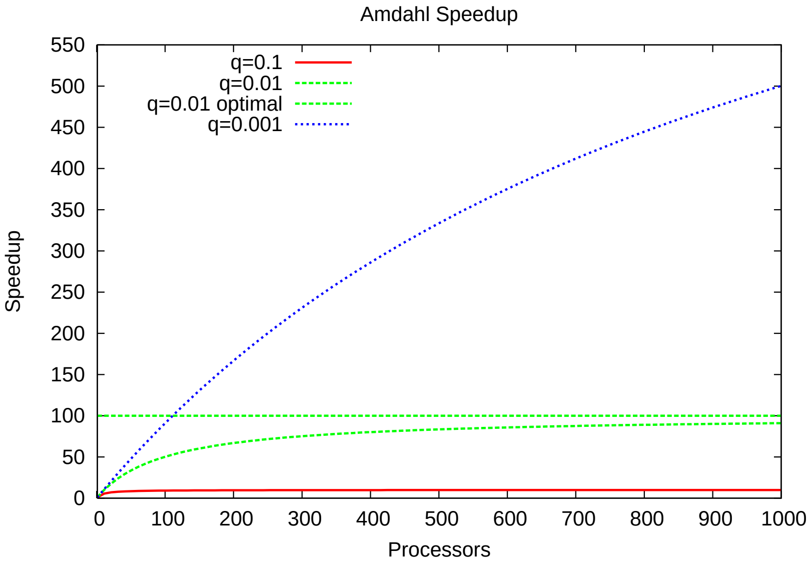
Assume a fixed sequential execution time. The part qT_{fix} with $0 \leq q \leq 1$ is assumed not to be parallelisable. The remaining part $(1-q)T_{\text{fix}}$ is assumed to be fully parallelisable. Then the speedup is given by

$$S_A(P) = \frac{T_{fix}}{qT_{fix} + (1-q)T_{fix}/P} = \frac{1}{q + \frac{1-q}{P}}.$$

Remark

$$S_A(P) \leq 1/q.$$

Speedup Graph Amdahl's Law



Other Scalings

The sequential part q is a function of N and usually decreases with increasing N. Therefore choose $N = N(P)$.

Gustafson Scaling

Choose $N = N(P)$ such that $T_P(N(P), P) = T_{\text{fix}}$. Motivation: Weather prediction.

Memory Scaling

Choose $N = N(P)$ such that memory requirements scale with the available memory: $M(N(P)) = M_0 P$. Useful for memory-bound applications, e.g. finite element methods.

Isoefficient Scaling

Choose $N = N(P)$ such that $E(N(P), P) = E_0$. This is not always possible! If such $N(P)$ exists the algorithm is called scalable.

Scalar Product: Fixed Sequential Execution Time

Remember the scalar product example:

$$\begin{aligned} T_S(N) &= N2t_a, \\ T_P(N,P) &= (N/P)2t_a + \log_2 P(t_m + t_a). \end{aligned}$$

Fixed size scaling $T_S(N,P) = T_{fix}$:

$$N2t_a = T_{fix} \Longrightarrow N_A = \frac{T_{fix}}{2t_a}.$$

This results in the speedup

$$S_A(P) = \frac{T_{fix}}{T_{fix}/P + \log_2 P(t_m + t_a)} = \frac{P}{1 + P\log_2 P\frac{t_m+t_a}{T_{fix}}}.$$

Scalar Product: Gustafson Scaling

Gustafson scaling $T_P(N(P), P) = T_{fix}$:

$$\frac{N}{P} 2t_a + \log_2 P(t_m + t_a) = T_{fix} \Longrightarrow N_G(P) = P\left(\frac{T_{fix} - \log_2 P(t_m + t_a)}{2t_a}\right).$$

There is an upper limit to P !
Assuming $T_{fix} \gg \log_2 P(t_m + t_a)$ we get $N_G(P) \approx P T_{fix}/(2t_a)$.

This results in the speedup

$$S_G(P) = \frac{N_G(P)2t_a}{N_G(P)2t_a/P + \log_2 P(t_m + t_a)} = \frac{P}{1 + \log P \frac{t_m + t_a}{T_{fix}}}.$$

Communication overhead is $O(\log P)$ instead of $O(P \log P)$.

Scalar Product: Memory Scaling

Gustafson scaling $M(N(P)) = M_0 P$:

$$M(N(P)) = wN = M_0 P \Longrightarrow N_M(P) = P(M_0/w).$$

There is no upper limit to P !

This results in the speedup

$$S_M(P) = \frac{N_M(P) 2t_a}{N_M(P) 2t_a / P + \log_2 P(t_m + t_a)} = \frac{P}{1 + \log P \frac{(t_m + t_a) w}{M_0 2 t_a}}.$$

Same as Gustafson scaling as long as $T_{fix} \gg \log_2 P(t_m + t_a)$.

Scalar Product: Isoefficient Scaling

Isoefficient scaling

$$E(N(P), P) = S(N(P), P)/P = E_0 \Rightarrow S(N(P), P) = E_0 P.$$

Inserting the speedup gives

$$S = \frac{P}{1 + \frac{P \log_2 P}{N} \frac{(t_m + t_a)}{2t_a}} \overset{!}{=} E_0 P \Longrightarrow N_I(P) = P \log_2 P \frac{E_0}{1 - E_0} \frac{t_m + t_a}{2t_a}.$$

The resulting speedup is $S(N_I(P), P) = PE_0$.

What you should remember

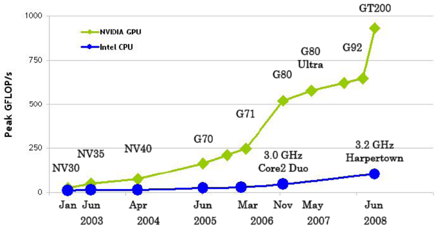
- Basic performance measures are speedup and efficiency.
- They are defined relative to the best sequential algorithm.
- Most parallel algorithms scale well if the problem size is increased with the number of processors.

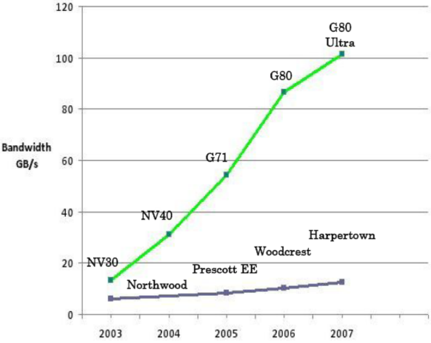
For Further Reading

Chapter 4 in V. Kumar, A. Grama, A. Gupta and G. Karypis (1994).
Introduction to Parallel Computing.
Benjamin/Cummings.

Motivation

- Development of graphics processors (GPU) is dramatic:





- GPUs are parallel processors!
- **GPGPU computing**: Use GPUs for general parallel computing.

GPU - CPU Comparison

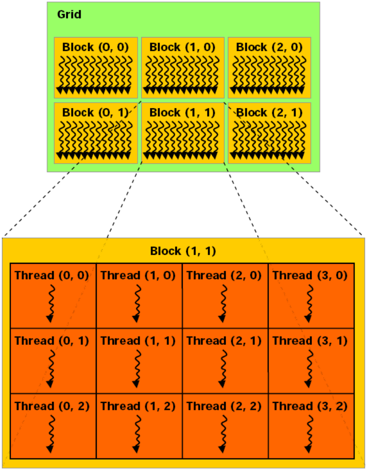
	Intel QX 9770	NVIDIA 9800 GTX
Available since	Q1/2008	Q1/2008
Cores	4	16 × 8
Transistors	820 Mio	754 Mio
Clock	3200 MHz	1688 MHz
Cache	4 × 6 MB	16 × 16 KB
Peak	102 GFlop/s	648 GFlop/s
Bandwith	128 GB/s	70.4 GB/s
Price	1200 $	150 $

Latest model GTX 280 has 30×8 cores and a peak of 1 TFlop/s.

CUDA

- Stands for **Compute Unified Device Architecture**
- Scalable hardware model with e.g. 4×8 processors in a notebook and 30×8 processors on a high end card.
- C/C++ programming environment with language extensions. Special compiler `nvcc`.
- The code to execute on the GPU can only be C.
- Run-time library and several application libraries (BLAS, FFT).
- Extensive set of examples.
- Coprocessor architecture:
  - Some code runs on GPU which then invokes code on the GPU.
  - Data must be copied explicitely between CPU and GPU memory (no direct access).

Programming Model at a Glance



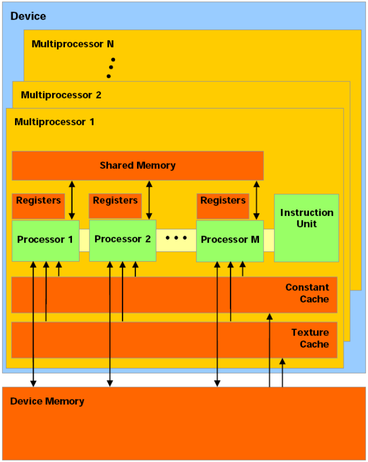
- Parallel threads cooperating via shared variables.
- Threads are organized into blocks of a “chosen” size.
- Blocks can be 1-, 2- or 3-dimensional.
- Blocks are organized in a grid of variable size.
- Grids can be 1- or 2-dimensional.
- # threads typically much larger than # cores (“hyperthreading”).
- Block size determined by HW/problem, grid size determined by problem size.
- No penalty for context switch.

Example of a Kernel

```
__global__ void scale_kernel (float *x, float a)
{
  int index = blockIdx.x*blockDim.x + threadIdx.x;
  x[index] *= a;
}
```

- `__global__` function type qualifier indicates that this function executes on the device and can only be called from host (“kernel”).
- Built-in variable `threadIdx` gives position of the thread within the block.
- Built-in variable `blockIdx` gives position of the block within the grid.
- Built-in variable `blockDim` gives the size of the block.
- Here, each thread is responsible to scale one element of a vector.
- The total number of threads needs to be ajusted to the size of the vector.

Hardware at a Glance



- A Multiprocessor (MP) consists of $M = 8$ “processors”.
- MP has one instruction unit and 8 ALUs. Threads executing different instructions are serialized!
- 8192 registers per MP, partitioned to threads at compile-time.
- 16 KB shared memory per MP organized in 16 banks.
- Up to 4GB global memory, latency 600 cycles, bandwith up to 80 GB/s .
- Constant and texture memory are cached and read-only.
- Most graphics card only provide single precision arithmetic.
- Arithmetic is not IEEE conforming.

Execution and Performance Issues

- Divergence: Full performance can only be achieved when all threads of a warp execute the same instruction.
- Threads are scheduled in *warps* of 32.
- Hyperthreading: A MP should execute more than 8 threads at a time (recommended block size is 64) to hide latencies.
- Shared memory access takes 2 cycles.
- Fastest instructions are 4 cycles (e.g. single precision multiply-add).
- Access to shared memory is only fast if each thread accesses a different bank, else access to banks is serialized.
- Access to global memory can be speed up through coalescing access to aligned locations. Requires use of special data types, e.g. `float4`.

CUDA Language Extensions

- Function type qualifiers
  - `__device__` on device, callable from device.
  - `__global__` on device, callable from host.
- Variable type qualifiers
  - `__device__` in global memory, lifetime of app.
  - `__constant__` in constant memory, lifetime of app.
  - `__shared__` in shared memory, lifetime of block.
- Directive for kernel invocation (see below).
- Built-in variables `__gridDim__`, `__blockIdx__`, `__blockDim__`, `__threadIdx__`, `__warpSize__` .

Hello CUDA I

```
// scalar product using CUDA
// compile with: nvcc hello.cu -o hello

// includes, system
#include<stdlib.h>
#include<stdio.h>

// kernel for the scale function to be executed on device
__global__ void scale_kernel (float *x, float a)
{
  int index = blockIdx.x*blockDim.x + threadIdx.x;
  x[index] *= a;
}

// wrapper executed on host that calls scale on device
// n must be a multiple of 32 !
void scale (int n, float *x, float a)
{
  // copy x to global memory on the device
  float *xd;
  cudaMalloc( (void**) &xd, n*sizeof(float) ); // allocate memory on device
  cudaMemcpy(xd,x,n*sizeof(float),cudaMemcpyHostToDevice); // copy x to device

  // determine block and grid size
  dim3 dimBlock(32);   // use BLOCKSIZE threads in one block
  dim3 dimGrid(n/32);  // n must be a multiple of BLOCKSIZE!

  // call function on the device
  scale_kernel<<<dimGrid,dimBlock>>>(xd,a);

  // wait for device to finish
  cudaThreadSynchronize();

  // read result
```

Hello CUDA II

```
  cudaMemcpy(x,xd,n*sizeof(float),cudaMemcpyDeviceToHost);

  // free memory on device
  cudaFree(xd);
}

int main( int argc, char** argv)
{
  const int N=1024;
  float sum=0.0;
  float x[N];
  for (int i=0; i<N; i++) x[i] = 1.0*i;
  scale(N,x,3.14);
  for (int i=0; i<N; i++) sum += (x[i]-3.14*i)*(x[i]-3.14*i);
  printf("%g\n",sum);
  return 0;
}
```

Scalar Product I

```
// scalar product using CUDA
// compile with: nvcc scalarproduct.cu -o scalarproduct -arch sm_11

// includes, system
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<sm_11_atomic_functions.h>

#define PROBLEMSIZE 1024
#define BLOCKSIZE 32

// integer in global device memory
__device__ int lock=0;

// kernel for the scalar product to be executed on device
__global__ void scalar_product_kernel (float *x, float *y, float *s)
{
  extern __shared__ float ss[]; // memory allocated per block in kernel launch
  int block = blockIdx.x;
  int tid = threadIdx.x;
  int index = block*BLOCKSIZE+tid;

  // one thread computes one index
  ss[tid] = x[index]*y[index];
  __syncthreads();

  // reduction for all threads in this block
  for (unsigned int d=1; d<BLOCKSIZE; d*=2)
    {
      if (tid%(2*d)==0) {
        ss[tid] += ss[tid+d];
      }
      __syncthreads();
```

Scalar Product II

```
  }

  // combine results of all blocks
  if (tid==0)
    {
      while (atomicExch(&lock,1)==1) ;
      *s += ss[0];
      atomicExch(&lock,0);
    }
}

// wrapper executed on host that uses scalar product on device
float scalar_product (int n, float *x, float *y)
{
  int size = n*sizeof(float);

  // allocate x in global memory on the device
  float *xd;
  cudaMalloc( (void**) &xd, size ); // allocate memory on device
  cudaMemcpy(xd,x,size,cudaMemcpyHostToDevice); // copy x to device
  if( cudaGetLastError() != cudaSuccess)
    {
      fprintf(stderr,"error␣in␣memcpy\n");
      exit(-1);
    }

  // allocate y in global memory on the device
  float *yd;
  cudaMalloc( (void**) &yd, size ); // allocate memory on device
  cudaMemcpy(yd,y,size,cudaMemcpyHostToDevice); // copy y to device
  if( cudaGetLastError() != cudaSuccess)
    {
      fprintf(stderr,"error␣in␣memcpy\n");
      exit(-1);
```

Scalar Product III

```
    }

  // allocate s (the result) in global memory on the device
  float *sd;
  cudaMalloc( (void**) &sd, sizeof(float) ); // allocate memory on device
  float s=0.0f;
  cudaMemcpy(sd,&s,sizeof(float),cudaMemcpyHostToDevice); // initialize sum on device
  if( cudaGetLastError() != cudaSuccess)
    {
      fprintf(stderr,"error␣in␣memcpy\n");
      exit(-1);
    }

  // determine block and grid size
  dim3 dimBlock(BLOCKSIZE);   // use BLOCKSIZE threads in one block
  dim3 dimGrid(n/BLOCKSIZE);  // n is a multiple of BLOCKSIZE

  // call function on the device
  scalar_product_kernel<<<dimGrid,dimBlock,BLOCKSIZE*sizeof(float)>>>(xd,yd,sd);

  // wait for device to finish
  cudaThreadSynchronize();
  if( cudaGetLastError() != cudaSuccess)
    {
      fprintf(stderr,"error␣in␣kernel␣execution\n");
      exit(-1);
    }

  // read result
  cudaMemcpy(&s,sd,sizeof(float),cudaMemcpyDeviceToHost);
  if( cudaGetLastError() != cudaSuccess)
    {
      fprintf(stderr,"error␣in␣memcpy\n");
      exit(-1);
```

Scalar Product IV

```
    }

  // free memory on device
  cudaFree(xd);
  cudaFree(yd);
  cudaFree(sd);

  // return result
  return s;
}

int main( int argc, char** argv)
{
  float x[PROBLEMSIZE], y[PROBLEMSIZE];
  float s;
  for (int i=0; i<PROBLEMSIZE; i++) x[i] = y[i] = sqrt(2.0f);
  s = scalar_product(PROBLEMSIZE,x,y);
  printf("result␣of␣scalar␣product␣is␣%f\n",s);
  return 0;
}
```

Remark: This is not the most efficient version. See the CUDA tutorial for a version that utilizes full memory bandwidth.