

MAX PLANCK INSTITUTE FOR DYNAMICS OF COMPLEX TECHNICAL SYSTEMS MAGDEBURG



COMPUTATIONAL METHODS IN SYSTEMS AND CONTROL THEORY

Scientific Computing II

Parallel Methods

Jens Saak and Martin Köhler

Summer Term 2019 OVGU Magdeburg

Computational Methods in Systems and Control Theory (CSC) Max Planck Institute for Dynamics of Complex Technical Systems



Preface



Why are you here?



Characterization

- A parallel computer is a
 - collection of processing elements
 - communicating and
 - cooperating

for the fast solution of a large problem.



Pseudo Parallelism, or Multitasking

modern operating systems simulate parallel execution by time slicing





Pseudo Parallelism, or Multitasking

Basic Types of Parallelism

Parallel Computing Basics

modern operating systems simulate parallel execution by time slicing



Distributed Memory

Computations executed on single unit with exclusive memory each





Pseudo Parallelism, or Multitasking

modern operating systems simulate parallel execution by time slicing



Computations executed on single unit with exclusive memory each

Shared memory

Several computational units share a common main memory.





race conditions,



race conditions,

execution order based accuracy issues,



race conditions,

execution order based accuracy issues,

deadlocks,



race conditions,

execution order based accuracy issues,

deadlocks,

data interdependence,



race conditions,

execution order based accuracy issues,

deadlocks,

data interdependence,

blocking problems on hardware level.



Introduction: Part I



1. Problem size exceeds desktop capabilities.



1. Problem size exceeds desktop capabilities.

2. Problem is inherently parallel (e.g. Monte-Carlo simulations).



- 1. Problem size exceeds desktop capabilities.
- 2. Problem is inherently parallel (e.g. Monte-Carlo simulations).
- 3. Modern architectures require parallel programming skills to be optimally exploited.





How many processing elements?



- How many processing elements?
- How complex are they?



- How many processing elements?
- How complex are they?
- How are they connected?



- How many processing elements?
- How complex are they?
- How are they connected?
- How is their cooperation coordinated?



- How many processing elements?
- How complex are they?
- How are they connected?
- How is their cooperation coordinated?
- What kind of problems can be solved?

The basic classification allowing answers to most of these questions is known as Flynn's taxonomy. It distinguishes four categories of parallel computers.





1. Single-Instruction, Single-Data (SISD)



- 1. Single-Instruction, Single-Data (SISD)
- 2. Multiple-Instruction, Single-Data (MISD)



- 1. Single-Instruction, Single-Data (SISD)
- 2. Multiple-Instruction, Single-Data (MISD)
- 3. Single-Instruction, Multiple-Data (SIMD)



- 1. Single-Instruction, Single-Data (SISD)
- 2. Multiple-Instruction, Single-Data (MISD)
- 3. Single-Instruction, Multiple-Data (SIMD)
- 4. Multiple-Instruction, Multiple-Data (MIMD)



a single processing element,



- a single processing element,
- executing a single instruction,
- on a single piece of data,
- in each step of the execution.



- a single processing element,
- executing a single instruction,
- on a single piece of data,
- in each step of the execution.

It is thus in fact the standard sequential computer implementing, e.g., the *von Neumann* model.

- a single processing element,
- executing a single instruction,
- on a single piece of data,
- in each step of the execution.

It is thus in fact the standard sequential computer implementing, e.g., the *von Neumann* model.

Examples
 desktop computers until Intel[®] Pentium[®] 4 era,
■ early netBooks on Intel [®] Atom [™] basis,
 pocket calculators,
 abacus,
embedded circuits



Flynn's Taxonomy of Parallel Architectures Single-Instruction, Single-Data (SISD)



Figure: Single-Instruction, Single-Data (SISD) machine model



In contrast to the SISD model in the MISD architecture we have

multiple processing elements,



In contrast to the SISD model in the MISD architecture we have

- multiple processing elements,
- executing a separate instruction each,
- all on the same single piece of data,
- in each step of the execution.



In contrast to the SISD model in the MISD architecture we have

- multiple processing elements,
- executing a separate instruction each,
- all on the same single piece of data,
- in each step of the execution.

The MISD model is usually not considered very useful in practice.



Flynn's Taxonomy of Parallel Architectures Multiple-Instruction, Single-Data (MISD)



Figure: Multiple-Instruction, Single-Data (MISD) machine model


Here the characterization is

- multiple processing elements,
- execute the same instruction,
- on a multiple pieces of data,
- in each step of the execution.

This model is thus the ideal model for all kinds of vector operations

 $c = a + \alpha b.$

Examples

- Graphics Processing Units,
- Vector Computers,
- SSE (Streaming SIMD Extension) registers of modern CPUs.



The attractiveness of the SIMD model for vector operations, i.e., linear algebra operations, comes at a cost.

Consider the simple conditional expression

if (b==0) c=a; else c=a/b;

The SIMD model requires the execution of both cases sequentially. First all processes for which the condition is true execute their assignment, then the other do the second assignment. Therefore, conditionals need to be avoided on SIMD architectures to guarantee maximum performance.



Flynn's Taxonomy of Parallel Architectures Single-Instruction, Multiple-Data (SIMD)



Figure: Single-Instruction, Multiple-Data (SIMD) machine model



MIMD allows

- multiple processing elements,
- to execute a different instruction,
- on a separate piece of data,
- at each instance of time.



MIMD allows

- multiple processing elements,
- to execute a different instruction,
- on a separate piece of data,
- at each instance of time.

Examples

- multicore and multi-processor desktop PCs,
- cluster systems.



MIMD computer systems can be further divided into three class regarding their memory configuration:

distributed memory

Every processing element has a certain exclusive portion of the entire memory available in the system. Data needs to be exchanged via an interconnection network.

shared memory

All processing units in the system can access all data in the main memory.

hybrid

Certain groups of processing elements share a part of the entire data and instruction storage.



Single Program, Multiple Data (SPMD)

SPMD is a programming model for MIMD systems. "In SPMD multiple autonomous processors simultaneously execute the same program at independent points."¹ This contrasts to the SIMD model where the execution points are not independent.

This is opposed to

Multiple Program, Multiple Data (MPMD)

A different programming model for MIMD systems, where multiple autonomous processing units execute different programs at the same time. Typically Master/Worker like management methods of parallel programs are associated with this class.

¹Wikipedia: http://en.wikipedia.org/wiki/SPMD



Flynn's Taxonomy of Parallel Architectures Multiple-Instruction, Multiple-Data (MIMD)



Figure: Multiple-Instruction, Multiple-Data (MIMD) machine model



Cloud	
 Network Storage 	
Local Storage	slow and
Tape	very slow
Hard Disk Drive (HDD)	
 Solid State Disk (SSD) 	
 Main Random Access Memory (RAM) 	medium
L3 Cache	
L2 Cache	C .
L1 Cache	fast
Registers	

Figure: Basic memory hierarchy on a single processor system.



Machine (6042MB)	
Socket P#0	
L3 (4096KB)	
L2 (256KB)	L2 (256KB)
L1 (32KB)	L1 (32KB)
Core P#0 PU P#0	Core P#2 PU P#1
Host: pc812 Indexes: physical Date: Mo 09 Jul 201	.2 13:37:17 CEST

Figure: A sample dual core Xeon[®] setup





Figure: A sample Core[™] 2 Quad setup



Memory Hierarchies in Parallel Computers Shared Memory: NUMA

Harbore (102428)
killelikein PAC (20428)
Enclari P40
(3(2698)
2 (2644) 12 (2644) 12 (2644) 12 (2644) 12 (2644) 12 (2644) 12 (2644)
L (1240)
Contral Contral Contral Contral Contral Contrals Contrals
NUMB NUMB NUMB NUMB NUMB NUMBE NUMBE
كالكالكالكالكا
NUMBer PG (DISCR)
Sector PE3
13 (2698)
2 (2644) 2 (2644) 2 (2644) 2 (2644) 2 (2644) 2 (2644) 2 (2644) 2 (2644)
L (200) L1 (200)
Com 740 Com 740 Com 740 Com 740 Com 7400 Com 7400 Com 7400 Com 7400 Com 7400
INNE NINE NINE NINE NINE NINE NINE
رگاهای از این
MUMAnute PA3 (20038)
Easter 742
c3 (2010)
12 (2664) 12 (2664) 12 (2664) 12 (2664) 12 (2664) 12 (2664) 12 (2664)
11 (104) 11 (204) 11 (204) 11 (204) 11 (204) 11 (204) 11 (204)
Com FAD
RUNZ RUNK RUNS RUNS RUNS RUNS RUNS
NUMBER PE3 (DINCE)
Societ P#3
L3 (2498)
12 (2668) 12 (2668) 12 (2668) 12 (2668) 12 (2668) 12 (2668) 12 (2668)
Li (100) Li (200)
Core PAG
N NO N NO N NO N NO N NO N NO
Paul ailtha

Figure: A four processor octa-core $\mathsf{Xeon}^{\textcircled{R}}$ system



Memory Hierarchies in Parallel Computers Shared Memory: NUMA + 2 GPUs

Subsection 10000 Subsection 100000 Subsection 100000 Subsection 100000 Subsection 100000 Subsection 1000000 Subsection 1000000000000000000000000000000000000	Machine (2008)
	NUMMINGS PED (1668)
Semicority (SIG) Comparing (Signal) Semicority (Signal) Comparing (Signal) Comparing (Signal) Comparing (Signal) Comparing (Signal) Comparing (Signal) Comparing (Signal) Comparing (Signal)	
Host advibed	Linken M1 D

Figure: A dual processor octa-core Xeon[®] system



Memory Hierarchies in Parallel Computers

General Memory Setting



Figure: Schematic of a general parallel system



Memory Hierarchies in Parallel Computers

General Memory Setting



Figure: Schematic of a general parallel system



Memory Hierarchies in Parallel Computers

General Memory Setting



Figure: Schematic of a general parallel system



The **Interconnect** in the last figure stands for any kind of Communication grid. This can be implemented either as

Iocal hardware interconnect,

or in the form of

network interconnect.

In classical supercomputers the first was mainly used, whereas in today's cluster based systems often the network solution is used in the one form or the other.



MyriNet

- shipping since 2005
- transfer rates up to 10Gbit/s
- Iost significance (2005 28.2% TOP500 down to 0.8% in 2011)

Infiniband

- transfer rates up to 300Gbit/s
- most relevant implementation driven by OpenFabrics Alliance², ~→ usable on Linux, BSD, Windows systems.
- features remote direct memory access (RDMA) ~→ reduced CPU overhead
- can also be used for TCP/IP communication

Omni-Path



- transfer rates up to 100Gbit/s
- introduced by $Intel^{\textcircled{R}}$ in 2015/2016
- rising significance
- Intel[®]'s approach to address cluster of > 10 000 nodes

²http://www.openfabrics.org/



nodes aligned on a string each being connected to at most two neighbors.



nodes aligned on a string each being connected to at most two neighbors.

2. Ring:

nodes are aligned in a ring each being connected to exactly two neighbors



nodes aligned on a string each being connected to at most two neighbors.

2. **Ring:**

nodes are aligned in a ring each being connected to exactly two neighbors

3. Complete graph:

every node is connected to all other nodes



nodes aligned on a string each being connected to at most two neighbors.

2. Ring:

nodes are aligned in a ring each being connected to exactly two neighbors

3. Complete graph:

every node is connected to all other nodes

4. Mesh and Torus:

Every node is connected to a number of neighbors (2-4 in 2d mesh, 4 in 2d torus).



nodes aligned on a string each being connected to at most two neighbors.

2. Ring:

nodes are aligned in a ring each being connected to exactly two neighbors

3. Complete graph:

every node is connected to all other nodes

4. Mesh and Torus:

Every node is connected to a number of neighbors (2-4 in 2d mesh, 4 in 2d torus).

5. k-dimensional cube / hypercube:

Recursive construction of a well connected network of 2^k nodes each connected to k neighbors. Line for two, square for four, cube fore eight.



nodes aligned on a string each being connected to at most two neighbors.

2. Ring:

nodes are aligned in a ring each being connected to exactly two neighbors

3. Complete graph:

every node is connected to all other nodes

4. Mesh and Torus:

Every node is connected to a number of neighbors (2-4 in 2d mesh, 4 in 2d torus).

5. k-dimensional cube / hypercube:

Recursive construction of a well connected network of 2^k nodes each connected to k neighbors. Line for two, square for four, cube fore eight.

6. **Tree:**

Nodes are arranged in groups, groups of groups and so forth until only one large group is left, which represents the root of the tree.



Performance Measures: Part



In general we call the time elapsed between issuing a command and receiving its results the runtime, or execution time of the corresponding process. Some authors also call it elapsed time, or wall clock time.



In general we call the time elapsed between issuing a command and receiving its results the runtime, or execution time of the corresponding process. Some authors also call it elapsed time, or wall clock time.

In the purely sequential case it is closely related to the so called CPU time of the process.



In general we call the time elapsed between issuing a command and receiving its results the runtime, or execution time of the corresponding process. Some authors also call it elapsed time, or wall clock time.

In the purely sequential case it is closely related to the so called CPU time of the process. There the main contributions are:

user CPU time: Time spent in execution of instructions of the process.



In general we call the time elapsed between issuing a command and receiving its results the runtime, or execution time of the corresponding process. Some authors also call it elapsed time, or wall clock time.

In the purely sequential case it is closely related to the so called CPU time of the process. There the main contributions are:

- **user CPU time:** Time spent in execution of instructions of the process.
- system CPU time: Time spent in execution of operating system routines called by the process.



In general we call the time elapsed between issuing a command and receiving its results the runtime, or execution time of the corresponding process. Some authors also call it elapsed time, or wall clock time.

In the purely sequential case it is closely related to the so called CPU time of the process. There the main contributions are:

- **user CPU time:** Time spent in execution of instructions of the process.
- **system CPU time:** Time spent in execution of operating system routines called by the process.
- waiting time: Time spent waiting for time slices, completion of I/O, memory fetches...



In general we call the time elapsed between issuing a command and receiving its results the runtime, or execution time of the corresponding process. Some authors also call it elapsed time, or wall clock time.

In the purely sequential case it is closely related to the so called CPU time of the process. There the main contributions are:

- **user CPU time:** Time spent in execution of instructions of the process.
- system CPU time: Time spent in execution of operating system routines called by the process.
- waiting time: Time spent waiting for time slices, completion of I/O, memory fetches...

That means the time we have to wait for a response of the program includes the waiting times besides the CPU time.



clock rate and cycle time

The clock rate of a processor tells us how often it can switch instructions per second. Closely related is the (clock) cycle time, i.e., the time elapsed between two subsequent clock ticks.



clock rate and cycle time

The clock rate of a processor tells us how often it can switch instructions per second. Closely related is the (clock) cycle time, i.e., the time elapsed between two subsequent clock ticks.

Example

A CPU with a clock rate of 3.5 GHz $=3.5\cdot10^9$ 1/s executes $3.5\cdot10^9$ clock ticks per second. The length of a clock cycle thus is

 $1/(3.5 \cdot 10^9)$ s = $1/3.5 \cdot 10^{-9}$ · s ≈ 0.29 ns



Different instructions require different times to get executed. This is represented by the so called cycles per instruction (CPI) of the corresponding instruction. An average CPI is connected to a process A via CPI(A).



Different instructions require different times to get executed. This is represented by the so called cycles per instruction (CPI) of the corresponding instruction. An average CPI is connected to a process A via CPI(A).

This number determines the total user CPU time together with the number of instructions and cycle time via

$$T_{U_CPU}(A) = n_{instr}(A) \cdot CPI(A) \cdot t_{cycle}$$


Different instructions require different times to get executed. This is represented by the so called cycles per instruction (CPI) of the corresponding instruction. An average CPI is connected to a process A via CPI(A).

This number determines the total user CPU time together with the number of instructions and cycle time via

$$T_{U_{-}CPU}(A) = n_{instr}(A) \cdot CPI(A) \cdot t_{cycle}$$

Clever choices of the instructions can influence the values of $n_{instr}(A)$ and CPI(A). \rightarrow compiler optimization.



A common performance measure of CPU manufacturers is the Million instructions per second (MIPS) rate.

It can be expressed as

$$MIPS(A) = \frac{n_{instr}(A)}{T_{U_{-}CPU}(A) \cdot 10^6} = \frac{r_{cycle}}{CPI(A) \cdot 10^6},$$

where r_{cycle} is the cycle rate of the CPU.



A common performance measure of CPU manufacturers is the Million instructions per second (MIPS) rate.

It can be expressed as

$$MIPS(A) = \frac{n_{instr}(A)}{T_{U_{-}CPU}(A) \cdot 10^6} = \frac{r_{cycle}}{CPI(A) \cdot 10^6},$$

where r_{cycle} is the cycle rate of the CPU.

This measure can be misleading in high performance computing, since higher instruction throughput does not necessarily mean shorter execution time.



More common for the comparison in scientific computing is the rate of floating point operations (FLOPS) executed. The MFLOPS rate of a program A can be expressed as

$$MFLOPS(A) = \frac{n_{FLOPS}(A)}{T_{U_{-}CPU}(A) \cdot 10^6} [1/s],$$

with $n_{FLOPS}(A)$ the total number of FLOPS issued by the program A.



More common for the comparison in scientific computing is the rate of floating point operations (FLOPS) executed. The MFLOPS rate of a program A can be expressed as

$$MFLOPS(A) = \frac{n_{FLOPS}(A)}{T_{U_{-}CPU}(A) \cdot 10^6} [1/s],$$

with $n_{FLOPS}(A)$ the total number of FLOPS issued by the program A.

Note that not all FLOPS (see also Chapter 4 winter term) take the same time to execute. Usually divisions and square roots are much slower. The MFLOPS rate, however, does not take this into account.

Time Measurement and Operation Counts CPU_Time versus Execution Time

Example (A simple MATLAB [®] test)						
	Input:					
	ct0=0; A=randn (1500);					
	tic ct0=cputime; pause(2) toc cputime-ct0					
	<pre>tic ct0=cputime; [Q,R]=qr(A); toc cputime-ct0</pre>					

CSC

Time Measurement and Operation Counts CPU_Time versus Execution Time

E	Example (A simple MATLAB test)								
	Input:		Output:						
	ct0=0; A=randn(1500):		Elapsed time is 2.000208 seconds.						
	tic		ans =						
	ct0=cputime; pause(2)		0.0300						
	toc cputime-ct0		Elapsed time is 0.733860 seconds.						
	tic		ans =						
	ct0=cputime;		21.6800						
	toc								
	Chartime_Cro		Executed on a 4×8 core Xeon [®] system.						

CSC



Obviously, in a parallel environment the CPU time can be much higher than the actual execution time elapsed between start and end of the process.

In any case, it can be much smaller, as well.



Obviously, in a parallel environment the CPU time can be much higher than the actual execution time elapsed between start and end of the process.

In any case, it can be much smaller, as well.

The first result is easily explained by the splitting of the execution time into user/system CPU time and waiting time. The process is mainly waiting for the sleep system call to return whilst basically accumulating no active CPU time.

Obviously, in a parallel environment the CPU time can be much higher than the actual execution time elapsed between start and end of the process.

In any case, it can be much smaller, as well.

The first result is easily explained by the splitting of the execution time into user/system CPU time and waiting time. The process is mainly waiting for the sleep system call to return whilst basically accumulating no active CPU time.

The second result is due to the fact that the activity is distributed to several cores. Each activity accumulates its own CPU time and these are summed up to the total CPU time of the process.



Definition (Parallel cost and cost-optimality)

The cost of a parallel program with data size n is defined as

$$C_p(n) = p * T_p(n).$$

Here $T_p(n)$ is the parallel runtime of the process, i.e., its execution time on p processors.

The parallel program is called cost-optimal if

$$C_p = T^*(n).$$

Here, $T^*(n)$ represents the execution time of the fastest sequential program solving the same problem.

In practice $T^*(n)$ is often approximated by $T_1(n)$.



The speedup of a parallel program

$$S_p(n)=\frac{T^*(n)}{T_p(n)},$$

is a measure for the acceleration, in terms of execution time, we can expect from a parallel program.

The speedup is strictly limited from above by p since otherwise the parallel program would motivate a faster sequential algorithm. See [RAUBER/RÜNGER '10] for details.

In practice often the speedup is computed with respect to the sequential version of the code, i.e.,

$$S_p(n) \approx rac{T_1(n)}{T_p(n)}.$$



Usually, the parallel execution of the work a program has to perform comes at the cost of certain management of subtasks. Their distribution, organization and interdependence leads to a fraction of the total execution, that has to be done extra.

Definition

The fraction of work that has to be performed by a sequential algorithm as well is described by the parallel efficiency of a program. It is computed as

$$E_p(n)=\frac{T^*(n)}{C_p(n)}=\frac{S_p(n)}{p}=\frac{T^*}{p\cdot T_p(n)}.$$

The parallel efficiency obviously is limited from above by $E_p(n) = 1$ representing the perfect speedup of p.



In many situations it is impossible to parallelize the entire program. Certain fractions remain that need to be performed sequentially. When a (constant) fraction f of the program needs to be executed sequentially, Amdahl's law describes the maximum attainable speedup.



In many situations it is impossible to parallelize the entire program. Certain fractions remain that need to be performed sequentially. When a (constant) fraction f of the program needs to be executed sequentially, Amdahl's law describes the maximum attainable speedup.

The total parallel runtime $T_p(n)$ then consists of

- $f \cdot T^*(n)$ the time for the sequential fraction and
- $(1-f)/p \cdot T^*(n)$ the time for the fully parallel part.



In many situations it is impossible to parallelize the entire program. Certain fractions remain that need to be performed sequentially. When a (constant) fraction f of the program needs to be executed sequentially, Amdahl's law describes the maximum attainable speedup.

The total parallel runtime $T_p(n)$ then consists of

- $f \cdot T^*(n)$ the time for the sequential fraction and
- $(1-f)/p \cdot T^*(n)$ the time for the fully parallel part.

The best attainable speedup can thus be expressed as

$$S_p(n) = rac{T^*(n)}{f \cdot T^*(n) + rac{1-f}{p}T^*(n)} = rac{1}{f + rac{1-f}{p}} \leq rac{1}{f}$$



Question

Is the parallel efficiency of a parallel program independent of the number of processors p used?

The question is answered by the concept of parallel scalability. Scientific computing and HPC distinguish two forms of scalability:



Question

Is the parallel efficiency of a parallel program independent of the number of processors p used?

The question is answered by the concept of parallel scalability. Scientific computing and HPC distinguish two forms of scalability:

strong scalability

captures the dependence of the parallel runtime on the number of processors for a fixed total problem size.



Question

Is the parallel efficiency of a parallel program independent of the number of processors p used?

The question is answered by the concept of parallel scalability. Scientific computing and HPC distinguish two forms of scalability:

strong scalability

captures the dependence of the parallel runtime on the number of processors for a fixed total problem size.

weak scalability

captures the dependence of the parallel runtime on the number of processors for a fixed problem size per processor.



Multicore and Multiprocessor Systems: Part I



Definition (Symmetric Multiprocessing (SMP))

The situation where two or more identical processing elements access a shared periphery (i.e., memory, I/O,...) is called *symmetric multiprocessing* or simply (SMP).

The most common examples are

Multiprocessor systems,

Multicore CPUs.





Figure: Schematic of a general parallel system





one physical memory resource,



- one physical memory resource,
- is shared among all processing units,



- one physical memory resource,
- is shared among all processing units,
- all having uniform access to it.



- one physical memory resource,
- is shared among all processing units,
- all having uniform access to it.

Especially that means that all memory locations can be requested by all processors at the same time scale, independent of which processor performs the request and which chip in the memory holds the location.



- one physical memory resource,
- is shared among all processing units,
- all having uniform access to it.

Especially that means that all memory locations can be requested by all processors at the same time scale, independent of which processor performs the request and which chip in the memory holds the location.

Local caches one the single processing units are allowed. That means classical multicore chips are an example of a UMA system.





one logical shared memory unit,



- one logical shared memory unit,
- gathered from two or more physical resources,



- one logical shared memory unit,
- gathered from two or more physical resources,
- each bound to (groups of) single processing units.



- one logical shared memory unit,
- gathered from two or more physical resources,
- each bound to (groups of) single processing units.

Due to the distributed nature of the memory, access times vary depending on whether the request goes to local or foreign memory.



- one logical shared memory unit,
- gathered from two or more physical resources,
- each bound to (groups of) single processing units.

Due to the distributed nature of the memory, access times vary depending on whether the request goes to local or foreign memory.

Examples are current multiprocessor systems with multicore processors per socket and a separate portion of the memory controlled by each socket. Also recent "cluster on a chip" design processors like AMDs bulldozer



Memory Hierarchy Non-Uniform Memory Access (NUMA)

-									_ = >			
Machine (1268)												
	Socket P#0 (16G8)					Socket P#1 (16G8)						
Ш	NUMANode P#0 (8192MB)					NURANode P#2 (8192MB)						
Ш	L3 (8192X8)					L3 (8152KB)						
Ш	L2 (2048K8)	L2 (2048KB)	L2 (2048KB)	L2 (2048KB)		L2 (2048K8)	L2 (2048KB)	L2 (2048KB)	L2 (2048K8)			
Ш	L1i (64KB)	L1i (64KB)	L1i (64KB)	L1i (64KB)		L1i (64KB)	L1i (64KB)	L1i (64KB)	L1i (64KB)			
Ш	L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)		L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)			
	Core P#0 Core P#1 PU P#0 PU P#1	Core P#2 Core P#3 PU P#2 PU P#3	Core P#4 Core P#5 PU P#4 PU P#5	Core P#6 Core P#7 PU P#6 PU P#7		Core P#0 Core P#1 PU P#16 PU P#17	Core P#2 PU P#18 PU P#19	Core P#4 Core P#5 PU P#20 PU P#21	Core P#6 Core P#7 PU P#22 PU P#23			
	NUMANode P#1 (3132M8)			1	NJMANode P#3 (3152HB)							
Ш	L3 (8192KB)					L3 (8192KB)						
Ш	L2 (2048KB)	L2 (2048KB)	L2 (2048KB)	L2 (2048KB)		L2 (2048KB)	L2 (2048KB)	L2 (2048KB)	L2 (2048KB)			
Ш	L1i (64K8)	L1/ (64KB)	L1(64K8)	L1I (64K8)		L1I (64K8)	L1) (64KB)	L1(64K8)	L11 (64K0)			
Ш	L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)		L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)	L1d (16KB) L1d (16KB)			
	Core P#0 Core P#1 PU P#8 PU P#9	Core P#2 PU P#10 Core P#3 PU P#11 PU P#11	Core P#4 Core P#5 PU P#12 PU P#13	Core P#6 Core P#7 PU P#14 PU P#15		Core P#0 Core P#1 PU P#24 PU P#25	Core P#2 Core P#3 PU P#26 PU P#27	Core P#4 Core P#5 PU P#28 PU P#29	Core P#6 Core P#7 PU P#30 PU P#31			
In Di	dexes: physical ite: Tue 09 Oct 2012 12:18:38 AM E	10T										

Figure: AMDs Bulldozer layout is a NUMA example.



Definition (cache coherence)

The problem of keeping multiple copies of a single piece of data in the local caches of the different processors that hold it consistent is called cache coherence problem.

Cache coherence protocols:

- guarantee a consistent view of the main memory at any time.
- Several protocols exist.
- Basic idea is to invalidate all other copies whenever one of them is updated.


Definition (Process)

A computer program in execution is called a process.

A process consists of:

- the programs machine code,
- the program data worked on,
- the current execution state, i.e., the context of the process, register and cache contents, ...

Each process has a separate address space in the main memory.

Execution time slices are assigned to the active processes by the operating system's (OS's) scheduler. A switch of processes requires exchanging the process context, i.e., a short execution delay.



Multiple processes may be used for the parallel execution of compute tasks.

On Unix/Linux systems the fork() system call can be used to generate child processes. Each child process is generated as a copy of the calling parent process. It receives an exact copy of the address space of the parent and a new unique process ID (PID).

Communication between parent and child processes can be implemented via sockets or files, which usually leads to large overhead for data exchange.



Definition (Thread)

In the thread model a process may consist of several execution sub-entities, i.e control flows, progressing at the same time. These are usually called threads, or lightweight processes.

All threads of a process share the same address space.



Two types of implementations exist:

user level threads:

- administration and scheduling in user space,
- threading library maps the threads into the parent process,
- quick task switches avoiding the OS.

kernel threads:

- administration and scheduling by OS kernel and scheduler,
- different threads of the same process may run on different processors,
- blocking of single threads does not block the entire process,
- thread switches require OS context switches.



Two types of implementations exist:

user level threads:

- administration and scheduling in user space,
- threading library maps the threads into the parent process,
- quick task switches avoiding the OS.

kernel threads:

- administration and scheduling by OS kernel and scheduler,
- different threads of the same process may run on different processors,
- blocking of single threads does not block the entire process,
- thread switches require OS context switches.

Here we concentrate on POSIX threads, or Pthreads. These are available on all major OSes. The actual implementations range from user space wrappers (pthreads-w32 mapping pthreads to windows threads) to lightweight process type implementations (e.g. Solaris 2).





Figure: N:1 mapping for OS incapable of kernel threads





Figure: 1:1 mapping of user threads to kernel threads





Figure: N:M mapping of user threads to kernel threads with library thread scheduler



1. Often the two notions parallel and concurrent execution are used as synonyms of each other. In fact concurrent is more general.



- 1. Often the two notions parallel and concurrent execution are used as synonyms of each other. In fact concurrent is more general.
- 2. The parallel execution of a set of tasks requires parallel hardware on which they can be executed simultaneously.



- 1. Often the two notions parallel and concurrent execution are used as synonyms of each other. In fact concurrent is more general.
- 2. The parallel execution of a set of tasks requires parallel hardware on which they can be executed simultaneously.
- 3. The concurrent execution only requires a quasi parallel environment that allows all tasks to be in progress at the same time.



- 1. Often the two notions parallel and concurrent execution are used as synonyms of each other. In fact concurrent is more general.
- 2. The parallel execution of a set of tasks requires parallel hardware on which they can be executed simultaneously.
- 3. The concurrent execution only requires a quasi parallel environment that allows all tasks to be in progress at the same time.
- 4. That means "parallel" execution defines a subset of "concurrent" execution.



When several threads/processes of a parallel program have read and write access to a common piece of data, access needs to be mutually exclusive. Failure to ensure this, leads to a **race condition**, where the final value depends on the sequence of uncontrollable/random events. Usually data corruption is then unavoidable.



When several threads/processes of a parallel program have read and write access to a common piece of data, access needs to be mutually exclusive. Failure to ensure this, leads to a **race condition**, where the final value depends on the sequence of uncontrollable/random events. Usually data corruption is then unavoidable.



When several threads/processes of a parallel program have read and write access to a common piece of data, access needs to be mutually exclusive. Failure to ensure this, leads to a **race condition**, where the final value depends on the sequence of uncontrollable/random events. Usually data corruption is then unavoidable.

Example

Thread 1	Thread 2	value
		0
read		0
increment		0
write		1
	read	1
	increment	1
	write	2



When several threads/processes of a parallel program have read and write access to a common piece of data, access needs to be mutually exclusive. Failure to ensure this, leads to a **race condition**, where the final value depends on the sequence of uncontrollable/random events. Usually data corruption is then unavoidable.

Example

	Thread 1	Thread 2	value]	Thread 1	Thread 2	value
			0				0
	read		0		read		0
ir	ncrement		0			read	0
	write		1		increment		0
		read	1		write		1
		increment	1			increment	1
		write	2			write	1



Definition (semaphore)

A semaphore is a simple flag (binary semaphore) or a counter (counting semaphore) indicating the availability of shared resources in a critical region.



Definition (semaphore)

A semaphore is a simple flag (binary semaphore) or a counter (counting semaphore) indicating the availability of shared resources in a critical region.

Definition (mutual exclusion variable (mutex))

The mutual exclusion variable, or shortly mutex variable, implements a simple locking mechanism regarding the critical region. Each process/thread checks the lock upon entry to the region. If it is open the process/thread enters and locks it behind. Thus, all other processes/threads are prevented from entering and the process in the critical region has exclusive access to the shared data. When exiting the region the lock is opened.



Definition (semaphore)

A semaphore is a simple flag (binary semaphore) or a counter (counting semaphore) indicating the availability of shared resources in a critical region.

Definition (mutual exclusion variable (mutex))

The mutual exclusion variable, or shortly mutex variable, implements a simple locking mechanism regarding the critical region. Each process/thread checks the lock upon entry to the region. If it is open the process/thread enters and locks it behind. Thus, all other processes/threads are prevented from entering and the process in the critical region has exclusive access to the shared data. When exiting the region the lock is opened.

Both the above definitions introduce the programming models. Actual implementations may be more or less complete. For example the <code>pthreads-implementation</code> lacks counting semaphores.



deadlock

A deadlock describes the unfortunate situation, when semaphores, or mutexes have not, or have inappropriately been applied such that no process/thread is able to enter the critical region anymore and the parallel program is unable to proceed.



Processes and Threads Dining Philosophers

Example (dining philosophers)



- Each philosopher alternatingly eats or thinks,
- to eat the left and right forks are both required,
- every fork can only be used by one philosopher at a time,
- forks must be put back after eating.

Figure: The dining philosophers problem

Image by Benjamin D. Esham / Wikimedia Commons [CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0)]



simple solution attempt

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

³http://en.wikipedia.org/wiki/Dining_philosophers_problem



simple solution attempt

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

All philosophers decide to eat at the same time \Rightarrow deadlock.

³http://en.wikipedia.org/wiki/Dining_philosophers_problem



simple solution attempt

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

All philosophers decide to eat at the same time \Rightarrow deadlock.

More sophisticated solutions avoiding the deadlocks have been found since [DIJKSTRA '65]. Three of them are also available on Wikipedia³.

³http://en.wikipedia.org/wiki/Dining_philosophers_problem



Multicore and Multiprocessor Systems: Part II



Common to all the following commands:

Compiling and linking needs to be performed with -pthread.

The pthread functions and related data types are made available in a C program using:

#include <pthread.h>



- thread unique identifier to distinguish from other threads,
- attr attributes for determining thread properties. NULL means default properties,
- start_routine pointer to the function to be started in the newly created thread,
- arg the argument of the above function.



- thread unique identifier to distinguish from other threads,
- attr attributes for determining thread properties. NULL means default properties,
- start_routine pointer to the function to be started in the newly created thread,
- arg the argument of the above function.

Note that only a single argument can be passed to the threads start function.



The argument of the start function is a void pointer. We can thus define:

```
struct point3d{ double x,y,z; };
struct norm_args{
   struct point3d *P;
   double norm;
};
struct norm_args args;
```

and upon thread creation pass

err=pthread_create(tid, NULL, norm, (void *) & args);

to a start function

```
void *norm(void *arg) {
   struct norm_args *args=(struct norm_args *) arg;
   struct point3d *P;
   P = args->P;
   args->norm = P->x * P->y * P->y * P->z * P->z;
   return NULL;
};
```



```
int main(int argc, char* argv[]){
   pthread_t tidl,tid2;
   struct point3d point;
   struct norm_args args;
   args.P = &point;
   point.x=10; point.y=10; point.z=0;
   pthread_create(&tidl, NULL, norm, &args);
   point.x=20; point.y=20; point.z=-50;
   pthread_create(&tid2, NULL, norm, &args);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);
}
```

Depending on the execution of thread tid1 the argument point may get overwritten before it has been fetched, the analogue holds for the norm argument inside the function.





they return from their start function,



- they return from their start function,
- they call pthread_exit() to cleanly exit,



- they return from their start function,
- they call pthread_exit() to cleanly exit,
- they are aborted by a call to pthread_cancel(),



- they return from their start function,
- they call pthread_exit() to cleanly exit,
- they are aborted by a call to pthread_cancel(),
- the process they are associated to is terminated by an exit() call.



int pthread_exit(void *retval);

- retval return value of the exiting thread to the calling thread,
- threads exit implicitly when their start function is exited,
- the return value may be evaluated from another thread of the same process via the pthread_join() function,
- after the last thread in a process exits the process terminates calling exit() with a zero return value. Only then shared resources are released automatically.


int pthread_join(pthread_t thread, void **retval);

- Waits for a thread to terminate and fetches its return value.
- thread the identifier of the thread to wait for,
- retval destination to copy the return value (if not NULL) to.



1. pthread_join(); we have seen this function above



- 1. pthread_join(); we have seen this function above
- 2. Mutex variable functions for handling mutexes as defined above



- 1. $pthread_join()$; we have seen this function above
- 2. Mutex variable functions for handling mutexes as defined above
- 3. Condition variable functions treat a condition variable that can be used to indicate a certain event in which the threads are interested. Condition variables may be used to implement semaphore like structures and triggers for special more complex situation that require the threads to act in a certain way.



- 1. $pthread_join()$; we have seen this function above
- 2. Mutex variable functions for handling mutexes as defined above
- 3. Condition variable functions treat a condition variable that can be used to indicate a certain event in which the threads are interested. Condition variables may be used to implement semaphore like structures and triggers for special more complex situation that require the threads to act in a certain way.
- 4. pthread_once() can be used to make sure that certain initializations are performed by one and only one thread when called by multiple ones.



Static/Macro initialization:



Static/Macro initialization:

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

mutex is the mutex variable to be initialized



Static/Macro initialization:

- mutex is the mutex variable to be initialized
- attr can be used to adapt the mutex properties, as for the pthreads NULL gives the default attributes,



Static/Macro initialization:

- mutex is the mutex variable to be initialized
- attr can be used to adapt the mutex properties, as for the pthreads NULL gives the default attributes,
- restrict⁴ is a C99-standard keyword limiting the pointer aliasing features and guiding compilers and aiding in the caching optimization.

⁴See also https://en.wikipedia.org/wiki/Restrict



Static/Macro initialization:

- mutex is the mutex variable to be initialized
- attr can be used to adapt the mutex properties, as for the pthreads NULL gives the default attributes,
- restrict⁴ is a C99-standard keyword limiting the pointer aliasing features and guiding compilers and aiding in the caching optimization.
- initialization may fail if the system has insufficient memory (error code ENOMEM) or other resources (EAGAIN)

⁴See also https://en.wikipedia.org/wiki/Restrict



If mutex is unlocked the function returns with the mutex in locked state,



- If mutex is unlocked the function returns with the mutex in locked state,
- If mutex is already locked the execution is blocked until the lock is released and it can proceed as above,



- If mutex is unlocked the function returns with the mutex in locked state,
- If mutex is already locked the execution is blocked until the lock is released and it can proceed as above,
- Four types of mutexes are defined:
 - PTHREAD_MUTEX_NORMAL
 - PTHREAD_MUTEX_ERRORCHECK
 - PTHREAD_MUTEX_RECURSIVE
 - PTHREAD_MUTEX_DEFAULT

All of them show different behavior when locked mutexes should again be locked by the same thread or a thread tries to unlock a previously unlocked mutex and similar unintended situations. This especially regards error handling and deadlock detection.



The function is equivalent to pthread_mutex_lock(), except that it returns immediately in any case.



- The function is equivalent to pthread_mutex_lock(), except that it returns immediately in any case.
- Success or failure are determined from the return value.



- The function is equivalent to pthread_mutex_lock(), except that it returns immediately in any case.
- Success or failure are determined from the return value.
- If the mutex type is PTHREAD_MUTEX_RECURSIVE the lock count is increased by one and the function returns success.



the function releases the lock



- the function releases the lock
- what exactly "release" means, depends on the properties of the mutex variable



- the function releases the lock
- what exactly "release" means, depends on the properties of the mutex variable
- e.g., for type PTHREAD_MUTEX_RECURSIVE mutexes it means that the counter is decreased by one and they become available once it reaches zero



- the function releases the lock
- what exactly "release" means, depends on the properties of the mutex variable
- e.g., for type PTHREAD_MUTEX_RECURSIVE mutexes it means that the counter is decreased by one and they become available once it reaches zero
- if the mutex becomes available, i.e., unlocked by the function call and there are blocked threads waiting for it, the threading policy decides which thread acquires mutex next.



destroys the mutex referenced by mutex



- destroys the mutex referenced by mutex
- the destroyed mutex then becomes uninitialized



- destroys the mutex referenced by mutex
- the destroyed mutex then becomes uninitialized
- pthread_mutex_init() can be used to initialize the same mutex variable
 again



- destroys the mutex referenced by mutex
- the destroyed mutex then becomes uninitialized
- pthread_mutex_init() can be used to initialize the same mutex variable
 again
- if mutex is locked or referenced, pthread_mutex_destroy() fails with error code EBUSY



Problem:

• Consider two mutex variables ma and mb, as well as two threads T1 and T2.



Problem:

- Consider two mutex variables ma and mb, as well as two threads T1 and T2.
- T1 locks ma first and then mb,



Problem:

- Consider two mutex variables ma and mb, as well as two threads T1 and T2.
- T1 locks ma first and then mb,
- T2 locks mb first and then ma,



Problem:

- Consider two mutex variables ma and mb, as well as two threads T1 and T2.
- T1 locks ma first and then mb,
- T2 locks mb first and then ma,
- If T1 is interrupted by the scheduler after locking ma, but before locking mb and in the meantime T2 succeeds in locking it, then the classical deadlock occurs.



Locking hierarchy solution:

The basic idea here is that all threads need to lock the critical mutexes in the same order. This can easily be guaranteed by hierarchically ordering the mutexes.



Locking hierarchy solution:

The basic idea here is that all threads need to lock the critical mutexes in the same order. This can easily be guaranteed by hierarchically ordering the mutexes.

Back off strategy solution:

When we want to keep the differing locking orders, we may use pthread_mutex_trylock() with a back off strategy.



Locking hierarchy solution:

The basic idea here is that all threads need to lock the critical mutexes in the same order. This can easily be guaranteed by hierarchically ordering the mutexes.

Back off strategy solution:

When we want to keep the differing locking orders, we may use pthread_mutex_trylock() with a back off strategy.

Locking is tried in the desired order,



Locking hierarchy solution:

The basic idea here is that all threads need to lock the critical mutexes in the same order. This can easily be guaranteed by hierarchically ordering the mutexes.

Back off strategy solution:

When we want to keep the differing locking orders, we may use pthread_mutex_trylock() with a back off strategy.

- Locking is tried in the desired order,
- when a trylock fails, the thread unlocks all previously locked mutexes (it backs off of the protected resources),



Locking hierarchy solution:

The basic idea here is that all threads need to lock the critical mutexes in the same order. This can easily be guaranteed by hierarchically ordering the mutexes.

Back off strategy solution:

When we want to keep the differing locking orders, we may use pthread_mutex_trylock() with a back off strategy.

- Locking is tried in the desired order,
- when a trylock fails, the thread unlocks all previously locked mutexes (it backs off of the protected resources),
- after the back off it starts over from the first one.



Static/Macro initialization:

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

cond the condition to be initialized



Static/Macro initialization:

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

- cond the condition to be initialized
- attr can be used to adapt the condition properties, as for the pthreads NULL gives the default attributes,



Static/Macro initialization:

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

- cond the condition to be initialized
- attr can be used to adapt the condition properties, as for the pthreads NULL gives the default attributes,
- restrict: see pthread_mutex_init()


Dynamic initialization:

Static/Macro initialization:

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

- cond the condition to be initialized
- attr can be used to adapt the condition properties, as for the pthreads NULL gives the default attributes,
- restrict: see pthread_mutex_init()
- every condition variable is associated to a mutex.



destroys the condition variable referenced by cond



- destroys the condition variable referenced by cond
- the destroyed condition then becomes uninitialized



- destroys the condition variable referenced by cond
- the destroyed condition then becomes uninitialized
- pthread_cond_init() can reinitialize the same condition variable



- destroys the condition variable referenced by cond
- the destroyed condition then becomes uninitialized
- pthread_cond_init() can reinitialize the same condition variable
- if cond is blocking threads when destroyed the standard does not specify the behavior of pthread_cond_destroy().



assumes that mutex was locked before by the calling thread,



- assumes that mutex was locked before by the calling thread,
- results in the thread getting blocked and at the same time (atomically) releasing mutex



- assumes that mutex was locked before by the calling thread,
- results in the thread getting blocked and at the same time (atomically) releasing mutex
- another thread may evaluate this to wake up the now blocked thread (see pthread_cond_signal())



- assumes that mutex was locked before by the calling thread,
- results in the thread getting blocked and at the same time (atomically) releasing mutex
- another thread may evaluate this to wake up the now blocked thread (see pthread_cond_signal())
- upon waking up the thread automatically tries to gain access to mutex again,



- assumes that mutex was locked before by the calling thread,
- results in the thread getting blocked and at the same time (atomically) releasing mutex
- another thread may evaluate this to wake up the now blocked thread (see pthread_cond_signal())
- upon waking up the thread automatically tries to gain access to mutex again,
- if it succeeds it should test the condition again to check whether another thread changed it in the meantime.



int pthread_cond_signal(pthread_cond_t *cond);

- if no thread is blocked on the condition variable cond there is no effect,
- otherwise, one of the waiting threads is woken up and proceeds as described above.

int pthread_cond_broadcast (pthread_cond_t *cond);

- wakes up all threads blocking on cond,
- all of them try to acquire the associated mutex,
- only one of them can succeed,
- the others get blocked on the mutex now.



equivalent to pthread_cond_wait() except that it only blocks for the period specified by abstime,



- equivalent to pthread_cond_wait() except that it only blocks for the period specified by abstime,
- if the thread did not get signaled or broadcast before abstime expires it returns with error code ETIMEDOUT.



Semaphores are not available in the POSIX Threads standard.



Semaphores are not available in the POSIX Threads standard.

However, they can be created using the existing mechanisms of mutexes and conditions.



Semaphores are not available in the POSIX Threads standard.

However, they can be created using the existing mechanisms of mutexes and conditions.

A counting semaphore should be a data type that acts like a counter with non-negative values and for which two operations are defined:

- 1. A signal operation increments the counter and wakes up a task blocked on the semaphore if one exists.
- 2. A wait operation simply decrements the counter if it is positive. If it was zero already the thread is blocking on the semaphore.



data structure for the semaphore:

```
typedef struct _sema_t{
    int count;
    pthread_mutex_t m;
    pthread_cond_t c;
} sema_t;
```

the initialization

```
void InitSema(sema_t *ps) {
    pthread_mutex_init(&ps->m,NULL);
    pthread_cond_init(&ps->c,NULL);
```

and the cleanup

```
void CleanupSema(void *arg) {
    pthread_mutex_unlock((pthread_mutex_t *) arg);
```



```
void ReleaseSema(sema t *ps) { // signal operation
  pthread mutex lock (&ps->m) ;
  pthread cleanup push (CleanupSema, &ps->m);
    ps->count++;
    pthread_cond_signal(&ps->c) ;
 pthread_cleanup_pop (1);
void AcquireSema(sema t *ps) { // wait operation
 pthread_mutex_lock(&ps->mutex);
 pthread cleanup push (CleanupSema, &ps->m);
    while (ps->count==0)
      pthread cond wait (&ps->c, &ps->m) ;
    ps->count--;
  pthread cleanup pop(1);
```



Example (Producer/Consumer queue buffer protection)

- A buffer of fixed size *n* is shared by
- a producer thread generating entries and storing them in the buffer if it is not full,
- a consumer thread removing entries from the same buffer for further processing unless it is empty.
- For the realization of the protected access two semaphores are required:
 - 1. Number of entries occupied (initialized by 0),
 - 2. Number of free entries (initialized by n).

The Mechanism works for an arbitrary number of producers and consumers. (Details will be worked out on exercise sheet 2.)



- 1. Master/Slave model:
 - A master thread is controlling the execution of the program,
 - the slave threads are executing the work.



- 1. Master/Slave model:
 - A master thread is controlling the execution of the program,
 - the slave threads are executing the work.

2. Client/Server model:

- Client threads produce requests,
- Server threads execute the corresponding work.



- 1. Master/Slave model:
 - A master thread is controlling the execution of the program,
 - the slave threads are executing the work.
- 2. Client/Server model:
 - Client threads produce requests,
 - Server threads execute the corresponding work.
- 3. Pipeline model:
 - Every thread (except for the first and last in line) produces output that serves as input for another thread,
 - after a startup phase (filling the pipeline) the parallel execution is achieved.



- 1. Master/Slave model:
 - A master thread is controlling the execution of the program,
 - the slave threads are executing the work.
- 2. Client/Server model:
 - Client threads produce requests,
 - Server threads execute the corresponding work.
- 3. Pipeline model:
 - Every thread (except for the first and last in line) produces output that serves as input for another thread,
 - after a startup phase (filling the pipeline) the parallel execution is achieved.
- 4. Worker model:
 - equally privileged workers organize their workload,
 - an important variant is the task pool treated as detailed example next.



Idea:

Creation of a parallel threaded program that can dynamically schedule tasks on the available processors.



Task Pools Basic idea of the task pool

Idea:

Creation of a parallel threaded program that can dynamically schedule tasks on the available processors.

Key ingredients in the approach are:

- usage of a fixed number of threads
- organization of the pending tasks in a task pool,
- threads fetch the tasks from the pool and execute them leading to a dynamic assignment of the work load.



Task Pools Basic idea of the task pool

Idea:

Creation of a parallel threaded program that can dynamically schedule tasks on the available processors.

Key ingredients in the approach are:

- usage of a fixed number of threads
- organization of the pending tasks in a task pool,
- threads fetch the tasks from the pool and execute them leading to a dynamic assignment of the work load.

Main advantages

- automatic dynamic load balancing among the threads
- comparably small overhead for the administration of threads



data strucutre for one task:

```
typedef struct _work_t{
    void (*routine) (void*); //worker function to call
    void* arg;
    struct _work_t *next;
} work t :
```

data structure for the task pool:

```
typedef struct _tpool_t{
    int num_threads ; // number of threads
    int max_size, curr_size; // max./cur. number of tasks in pool
    pthread_t *threads; //array of threads
    work_t *head , *tail; // start/end of the task queue
    pthread_mutex_t lock; //access control for the task pool
    pthread_cond_t not_empty ; // tasks are available
    pthread_cond_t not_full ; // tasks may be added
} tpool_t ;
```



```
tpool t *tpool init(int num threads , int max size) {
  int i:
 tpool t *tpl;
  tpl=(tpool_t *) malloc (sizeof(tpool_t));
  tpl->num threads=num threads ;
  tpl->max size=max size ;
  tpl->cur size=0;
  tpl->head=tpl->tail=NULL;
 pthread_mutex_init(&tpl->lock, NULL);
  pthread cond init (&tpl->not empty, NULL);
  pthread_cond_init(&tpl->not_full, NULL);
  tpl->threads=(pthread t *) malloc(num threads *sizeof(pthread t));
  for(i=0; i<num threads; i++)</pre>
   pthread create(tpl->threads+i, NULL, tpool thread, (void *)tpl) ;
  return tpl;
```



```
void *tpool thread(void *vtpl) {
  tpool t *tpl=(tpool t *) vtpl;
 work t *wl ;
  for (;;) {
   pthread_mutex_lock(&tpl->lock);
    while (tpl->cur size==0)
      pthread_cond_wait(&tpl->not_empty , &tpl->lock);
    wl=tpl->head; tpl->cur size--;
    if(tpl->cur size==0)
     tpl->head=tpl->tail=NULL;
    else tpl->head = wl->next;
    if (tpl->cur_size==tpl->max_size-1) // pool full
      pthread_cond_signal(&tpl->not_full);
    pthread mutex unlock (&tpl->lock);
    (* (wl->routine)) (wl->arg);
    free(wl);
```



```
void tpool insert(tpool t *tpl, void(*f) (void*), void *arq){
  work t *wl ;
  pthread mutex lock(&tpl->lock);
  while(tpl->cur_size==tpl->max_size)
   pthread_cond_wait(&tpl->not_full, &tpl->lock);
  wl=(work t *) malloc(sizeof(work t));
  wl->routine=f; wl->arg=arg; wl->next=NULL ;
  if( tpl->cur size==0) {
    tpl->head=tpl->tail=wl;
   pthread cond_signal(&tpl->not_empty);
  else
    tpl->tail->next=wl; tpl->tail=wl;
  tpl->cur size++;
  pthread mutex unlock (&tpl->lock);
```



In contrast to Threads, different processes do not share their address space. Therefore, different ways to communicate in multiprocessing applications are necessary.

One possible way are shared memory objects. Unix-like operating systems provide at least one of:

- **old:** System V Release 4 (SVR4) Shared Memory⁵
- **new:** POSIX Shared Memory⁶.

Both techniques implement shared memory objects, like common memory, semaphores and message queues, which are accessible from different applications with different address spaces.

 ⁵System V Interface Definition, AT&T Unix System Laboratories, 1991
 ⁶IEEE Std 1003.1-2001Portable Operating System Interface System Interfaces



Common Memory Locations

- They are used to share data between applications.
- They are managed by the kernel and not by the application.
- **Each location is represented as a file in** /dev/shm/.
- They are handled like normal files.
- They are created using shm_open and mapped to the memory using mmap.
- Exist as long as no application deletes them.
- Even when the creating program exits they stay available,
- See manpage: man 7 shm_overview.



POSIX Semaphores

- Counting semaphores are available form different address spaces.
- They correspond to pthread_mutex_* in threaded applications.
- They are represented as a file in /dev/shm/sem.*.
- See manpage: man 7 sem_overview.

Message Queues

- They represent a generalized Signal concept which can transfer a small payload (2 to 4 KiB).
- They correspond to pthread_cond_* in threaded applications.
- They can be represented as file in /dev/mqueue.
- See manpage: man 7 mq_overview.



Multicore and Multiprocessor Systems: Part III



Mission

"The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer." ^a

"The Mission statement from http://www.openmp.org/about/about-us/



Mission

"The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer." ^a

"The Mission statement from http://www.openmp.org/about/about-us/

The OpenMP Architecture Review Board (ARB)

The ARB is a non-profit enterprise owning the OpenMP brand and responsible for overseeing, producing and approving the OpenMP standards.


Permanent Members of the ARB:

- AMD (Greg Stoner)
- ARM (Chris Adeniyi-Jones)
- Cray (Luiz DeRose)
- Fujitsu (Eiji Yamanaka)
- HP (Sujoy Saraswati)
- IBM (Kelvin Li)
- Intel (Xinmin Tian)
- Micron (Kirby Collins)
- NEC (Kazuhiro Kusano)
- NVIDIA (Jeff Larkin)
- Oracle Corporation (-TBD-)
- redhat (Torvald Riegel)
- Texas Instruments (Eric Stotzer)

⁷http://www.openmp.org/about/members/

(status: April 23, 2017)⁷



Open Multi-Processing (OpenMP) This is OpenMP: The API standard

History

```
Oct. 1997 OpenMP 1.0 for Fortran,
 Oct. 1998 OpenMP 1.0 for C/C++,
 Nov. 2000 OpenMP 2.0 for Fortran.
March 2002 OpenMP 2.0 for C/C++,
 May 2005 OpenMP 2.5 (first joint Fortran/C/C++ version),
 May 2008 OpenMP 3.0,
Sept. 2011 OpenMP 3.1,
 July 2013 OpenMP 4.0,
 Nov. 2015 OpenMP 4.5,
 Nov. 2018 OpenMP 5.0 (current standard).
```



Easy shared memory parallel adaption of existing sequential codes



Easy shared memory parallel adaption of existing sequential codes

Easy preservation of sequential implementations



- Easy shared memory parallel adaption of existing sequential codes
- Easy preservation of sequential implementations
- Easy porting to different platforms and compilers



- Easy shared memory parallel adaption of existing sequential codes
- Easy preservation of sequential implementations
- Easy porting to different platforms and compilers
- Parallel implementation of only fragments



- Easy shared memory parallel adaption of existing sequential codes
- Easy preservation of sequential implementations
- Easy porting to different platforms and compilers
- Parallel implementation of only fragments
- No extra runtime environment



- Easy shared memory parallel adaption of existing sequential codes
- Easy preservation of sequential implementations
- Easy porting to different platforms and compilers
- Parallel implementation of only fragments
- No extra runtime environment
- Easy to learn and apply





Most efficient use of shared memory systems



Most efficient use of shared memory systems

Automatic checking for (data dependencies,) data conflicts, race conditions, or deadlocks



Most efficient use of shared memory systems

Automatic checking for (data dependencies,) data conflicts, race conditions, or deadlocks

Automatic synchronization of input and output





Figure: Classification of the OpenMP extensions by tasks of the elements (Image Source: https://commons.wikimedia.org/wiki/File:OpenMP_language_extensions.svg).



1. Directives:

Basic control structures that initialize/end the parallel environments



1. Directives:

Basic control structures that initialize/end the parallel environments

2. Clauses:

Fine tuning parameters added to the directives.



1. Directives:

Basic control structures that initialize/end the parallel environments

2. Clauses:

Fine tuning parameters added to the directives.

3. Environment Variables:

Variables in the calling shell used to control the parallel environment without recompilation.



1. Directives:

Basic control structures that initialize/end the parallel environments

2. Clauses:

Fine tuning parameters added to the directives.

3. Environment Variables:

Variables in the calling shell used to control the parallel environment without recompilation.

4. Runtime Library Routines:

Runtime usable functions for the determination and modification of parameters of the parallel environment.



The #pragma directive was introduced in C89 as the universal method for extending the space of directives. It was further standardized in C99, where especially the token STDC was reserved for standard C extensions.

Example (standard C #pragma usage)

In part 1 of the Scientific Computing lecture we have seen the floating point environment for, e.g., checking the exception flags in IEEE arithmetic:

#include <fenv.h>
#pragma STDC FENV_ACCESS ON
/* starting here the compiler needs to assume we are accessing the
floating point status and mode registers*/



OpenMP is an extension in the sense of C89 and enabled by the

#pragma omp

preprocessor directive. It applies to the succeeding structural code block.



OpenMP is an extension in the sense of C89 and enabled by the

#pragma omp

preprocessor directive. It applies to the succeeding structural code block.

Compilers that do not know the omp pragma simply ignore it. The following switches enable OpenMP support for your code:

GNU GCC	-fopenmp
Intel ICC	-qopenmp
LLVM CLANG	-fopenmp
IBM XLC	-qsmp
PGI	-mp

Otherwise the omp pragmas are ignored and the sequential code version is compiled.



OpenMP is an extension in the sense of C89 and enabled by the

#pragma omp

preprocessor directive. It applies to the succeeding structural code block.

Compilers that do not know the omp pragma simply ignore it. The following switches enable OpenMP support for your code:

GNU GCC	-fopenmp
Intel ICC	-qopenmp
LLVM CLANG	-fopenmp
IBM XLC	-qsmp
PGI	-mp

Otherwise the omp pragmas are ignored and the sequential code version is compiled.

A list of compilers supporting OpenMP can be found at http://www.openmp.org/resources/openmp-compilers/



The parallel construct initializes a group of threads and starts parallel execution:

```
#pragma omp parallel [clause[[,]clause]...]
```

The clauses can be used to influence the behavior of the parallel execution. They will be explained later.

Available clauses for parallel:

- if(scalar expression)
- num_threads(integer expression)
- default(shared| none)
- private(list)
- firstprivate(list)
- shared(list)
- copyin(list)
- reduction(operation:list)



Example (A minimal OpenMP parallel "hello world" program)

```
#include <stdio.h>
int main(void)
{
    #pragma omp parallel
        printf("Hello,_world.\n");
    return 0;
}
```

The example automatically lets OpenMP tune the number of threads used to the number of available processors. Afterward the parallel execution environment is started and all threads execute the printf statement.



The loop construct specifies that the iterations of the loop should be distributed among the active threads.

```
#pragma omp for [clause[[,]clause]...]
for loops
```

The for-loop construct needs to be used inside a structured code block of parallel construct.

Available clauses for for:

- private(list)
- firstprivate(list)
- lastprivate(list)
- reduction(operator:list)
- schedule(kind[,chunk_size])
- collapse(n)
- ordered
- nowait



Since often the parallel environment is used to introduce a for-loop construction only, a shortcut parallel for exists for this special task

#pragma omp parallel for [clause[[,] clause]...]

With the exception of the nowait clause all clauses accepted by parallel and for can be used with parallel for with the identically same behaviors and restrictions.



Example (OpenMP parallel vector triad)

```
double triad(double *a, double *b, double *c, double *d, int length){
    int i, j;
    const int repeat=100;
    double start, end;

    get_walltime(&start);
    for (j=0; j<repeat; j++) {
        #pragma omp parallel for
        for (i=0; i<length; i++) {
            a[i]=b[i] + c[i] * d[i];
        } /*end of parallel section*/
    }
    get_walltime(&end);
    return repeat*length*2.0 / ((end-start) * 1.0e6); /* return MFLOPS */</pre>
```



Example (OpenMP parallel vector triad)

```
double triad(double *a, double *b, double *c, double *d, int length){
    int i, j;
    const int repeat=100;
    double start, end;

    get_walltime(&start);
    for (j=0; j<repeat; j++){
    #pragma omp parallel for
        for (i=0; i<length; i++){
            a[i]=b[i] + c[i] * d[i];
        } /*end of parallel section*/
    }
    get_walltime(&end);
    return repeat*length*2.0 / ((end-start) * 1.0e6); /* return MFLOPS */
}</pre>
```

Note that loop counters are protected automatically.



When different tasks are to be distributed among the encountering team of threads the sections construct can be used

```
#pragma omp sections [clause[[,] clause]...]
{
  [#pragma omp section]
  structured code block
  [#pragma omp section]
  structured code block
  ...
}
```

Available clauses for sections:

- private(list)
- firstprivate(list)
- lastprivate(list)
- reduction(operator:list)
- nowait



Analogous to the for construct, also sections can be used only inside a parallel construct. The parallel sections construct merges them for easier use

```
#pragma omp parallel sections [clause[[,] clause]...]
{
  [#pragma omp section]
   structured code block
  [#pragma omp section]
   structured code block
   ...
}
```

Available clauses are those available for parallel and sections with the exception of nowait, as in the case of for.



```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 50
int main (int argc, char *argv[]) {
  int i, nthrd, tid;
 float a[N], b[N], c[N], d[N];
 /* Some initializations */
 for (i=0; i<N; i++) {</pre>
   a[i] = i * 1.5;
   b[i] = i + 42.0;
   c[i] = d[i] = 0.0;
 /* Start 2 threads */
#pragma omp parallel shared(a,b,c,d,nthrd) private(i,tid) num_threads(2)
  tid = omp_get_thread_num();
  if (tid == 0) {
    nthrd = omp get num threads();
   printf("Number of threads = %d\n", nthrd);
 printf("Thread.%d.starting...\n",tid);
```



```
#pragma omp sections
#pragma omp section
        printf("Thread,%d,doing,section,l\n",tid);
        for (i=0; i<N; i++) {</pre>
          c[i] = a[i] + b[i];
        sleep(tid+2); /* Delay the thread for a few seconds */
      } /* End of first section */
#pragma omp section
        printf("Thread_%d_doing_section_2\n",tid);
        for (i=0; i<N; i++) {</pre>
          d[i] = a[i] * b[i];
        sleep(tid+2); /* Delay the thread for a few seconds */
      } /* End of second section */
    } /* end of sections */
   printf("Thread_%d_done.\n",tid);
  } /* end of omp parallel */
/* Print the results */
 printf("c:___");
```



```
for (i=0; i<N; i++) {
    printf("%.2f_", c[i]);
}
printf("\n\nd:__");
for (i=0; i<N; i++) {
    printf("%.2f_", d[i]);
}
printf("\n");
exit(0);</pre>
```



A construct that makes sure that a structured code block is executed by only one thread in a team of threads is given by the single directive.

#pragma omp single [clause[[,] clause]...]

Available clauses for the single construct are:

- private(list)
- firstprivate(list)
- lastprivate(list)
- nowait



Example (OpenMP 4.5 Example 1.11 — single1.c)

```
#include <stdio.h>
void work1() {}
void work2() {}
void main()
#pragma omp parallel
  #pragma omp single
      printf("Beginning.work1.\n");
  work1();
  #pragma omp single
      printf("Finishing_work1.\n");
  #pragma omp single nowait
      printf("Finished work1 and beginning work2.\n");
  work2();
```



The master construct specifies a structured block that is executed by the master thread of the team.

#pragma omp master

The following structured block is only executed by the master thread of the parallel team. There is no synchronization on entry or on exit with the other threads.



Example (OpenMP 4.5 Example 1.13 — master1.c)

```
void master example( float* x, float* xold, int n, float tol ) {
  int c = 0, i, toobig; loat error, y;
  #pragma omp parallel
    dof
       #pragma omp for private(i)
       for( i = 1; i < n-1; ++i ) { xold[i] = x[i]; }</pre>
       #pragma omp single
       toobig = 0;
       #pragma omp for private(i,y,error) reduction(+:toobig)
       for( i = 1; i < n-1; ++i ) {</pre>
         \mathbf{v} = \mathbf{x}[\mathbf{i}];
         x[i] = average( xold[i-1], x[i], xold[i+1] );
         error = \mathbf{v} - \mathbf{x}[\mathbf{i}];
         if( error > tol || error < -tol ) ++toobig;</pre>
       #pragma omp master
       { printf( "iteration %d, toobig=%d\n", ++c, toobig ); }
    while(toobig > 0);
```


The OpenMP task construct (introduced with OpenMP 3.0) allows to parallelize irregular algorithms. The task construct inserts a piece of work into a thread pool running in the background.

```
#pragma omp task [clause[[,] clause]...]
structured code block
```

Available clauses for task (not complete):

- if(scalar-expression)
- final(scalar-expression)
- default(shared | none)
- private(list)
- firstprivate(list)
- shared(list)
- priority(priority-value)



Using the taskwait directive:

#pragma omp taskwait

one can wait for the completion of all previously created tasks at any position inside a parallel region in order to synchronize the parallel execution.

Due to the fact that tasks are running in the background they are mostly emitted by a single thread or a sequential code block. Therefore, mostly the single and master directives are used.

Tasks have to be defined inside an OpenMP parallel region. The end of the parallel region, unless it is used with the nowait clause, is an implicit synchronization point and the program waits until all tasks created inside the parallel region are finished.



The support to describe data dependencies between tasks is one of the most beneficial features of the **OpenMP 4** standard. For scientific computing this means that algorithms relying on dependency-graphs can be parallelized without using other third-party code or libraries.

"Although we expect to see DAG-based models widely adopted, changes in other parts of the software ecosystem will inevitably affect the way that that model is implemented. The appearance of DAG scheduling constructs in the OpenMP 4.0 standard offers a particularly important example of this point. [...] However, the inclusion of DAG scheduling constructs in the OpenMP standard, along with the rapid implementation of support for them (with excellent multithreading performance) in the GNU compiler suite, throws open the doors to widespread adoption of this model in academic and commercial applications for shared memory." ^a

^aJack Dongarra et. al., Numerical Algorithms and Libraries at Exascale



The data dependencies are defined using the depend clause during the task creation:

#pragma omp task depend(direction:list) [depend(direction:list)] [clauses...]
structured code block

Each depend clause consists of a data-flow direction and a list of identifiers. Possible directions are:

- in The identifiers are **input** dependencies.
- out The identifiers are **output** dependencies.
- inout The identifiers are input and output dependencies.

The list of identifiers is a comma separated list of variables from which a pointer can be created.

Tasks with a common inout or output dependencies are executed in the order as they are created.



Example (OpenMP 4.5 Example 3.3.4 — task_dep4.c)

```
#include <stdio.h>
int main() {
    int x = 1;
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task shared(x) depend(out: x)
        x = 2;
        #pragma omp task shared(x) depend(in: x)
        printf("x_+_1_=_%d._", x+1);
        #pragma omp task shared(x) depend(in: x)
        printf("x_+_2_=_%d\n", x+2);
    }
    return 0;
}
```

Array elements, e.g. y[i] or A[i+ldA*j], are also valid identifiers in the depend clause. Intervals on arrays, like A[i:j], are also allowed.



Example (OpenMP 4.5 Example 3.3.5 — task_dep5.c)

```
// Assume BS divides N perfectly
void matmul depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N])
  int i, j, k, ii, jj, kk;
  for (i = 0; i < N; i + = BS) {
    for (j = 0; j < N; j+=BS) {
      for (k = 0; k < N; k+=BS) {
         #pragma omp task private(ii, jj, kk) firstprivate(i,j,k) \
           depend (in: A[i][k], B[k][j]) \
           depend ( inout: C[i][j] )
         for (ii = i; ii < i+BS; ii++ )</pre>
           for (jj = j; jj < j+BS; jj++ )</pre>
             for (\mathbf{kk} = \mathbf{k}; \mathbf{kk} < \mathbf{k}+\mathbf{BS}; \mathbf{kk}+\mathbf{k})
                C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
```



A synchronization construct that makes the threads wait until all threads in the team have reached this point and only then continues execution.

#pragma omp barrier

Note that all constructs that allow the nowait clause have an implicit barrier at their end. Still sometimes explicit synchronization is desirable.



The OpenMP clauses we have seen above can be divided into two classes

- 1. attribute clauses related to data sharing
- 2. clauses controlling data copying



The OpenMP clauses we have seen above can be divided into two classes

- 1. attribute clauses related to data sharing
- 2. clauses controlling data copying
 - clauses usually take a list of arguments
 - lists are comma separated and enclosed by ().
 - all list items must be visible to the clause



Data sharing attributes of a variable in a $\ensuremath{\texttt{parallel}}$ or $\ensuremath{\texttt{task}}$ construct can be one of

predetermined, e.g., loop counters in for or parallel for constructs are always private, const qualified variables are shared, more can be found in Section 2.9.1 of the OpenMP standard



Data sharing attributes of a variable in a $\ensuremath{\texttt{parallel}}$ or $\ensuremath{\texttt{task}}$ construct can be one of

- predetermined, e.g., loop counters in for or parallel for constructs are always private, const qualified variables are shared, more can be found in Section 2.9.1 of the OpenMP standard
- explicitly determined are those attributes where variables are referenced in a clause setting the attributes



Data sharing attributes of a variable in a $\ensuremath{\texttt{parallel}}$ or $\ensuremath{\texttt{task}}$ construct can be one of

- predetermined, e.g., loop counters in for or parallel for constructs are always private, const qualified variables are shared, more can be found in Section 2.9.1 of the OpenMP standard
- explicitly determined are those attributes where variables are referenced in a clause setting the attributes
- implicitly determined, are the attributes of variables referenced in a given construct but are neither predetermined nor explicitly specified



default (shared | none)

- determines the default attributes of variables in the context of a task or parallel construct.
- defaults to shared when not explicitly given in a parallel construct
- all other (except task) constructs inherit the default from the enclosing construct if no default clause is given explicitly.



default (shared | none)

- determines the default attributes of variables in the context of a task or parallel construct.
- defaults to shared when not explicitly given in a parallel construct
- all other (except task) constructs inherit the default from the enclosing construct if no default clause is given explicitly.

shared(list)

Sets the data sharing attributes of all variables in list to be of shared type. That means the variable is considered to be in the shared memory of the team of threads.



private(list)

Each variable of the list is declared to be a private copy of the thread and not accessible from other threads in the team. It can not be applied to variables that are part of other variables(*elements in arrays or members of a structure*).



private(list)

Each variable of the list is declared to be a private copy of the thread and not accessible from other threads in the team. It can not be applied to variables that are part of other variables(*elements in arrays or members of a structure*).

firstprivate(list)

As above but additionally the value of the item in the list is initialized from the corresponding original item when the construct is encountered. The clause has a few more restrictions found in the standard.



private(list)

Each variable of the list is declared to be a private copy of the thread and not accessible from other threads in the team. It can not be applied to variables that are part of other variables (elements in arrays or members of a structure).

firstprivate(list)

As above but additionally the value of the item in the list is initialized from the corresponding original item when the construct is encountered. The clause has a few more restrictions found in the standard.

lastprivate(list)

As private but causes the original item to be updated after the end of the region from the last iterate of the enclosed loop or the lexically last section in a sections region.



reduction(operator:list)

Accumulates all items of the list into a private copy according to the given operator and then combines it with the original instance.

+	(0)		(0)	
*	(1)	^	(0)	
-	(0)	&&	(1)	
&	(~0)		(0)	
max	(Least number in reduction list item type)			
min	(Largest number in reduction list item type)			

Table: Operators for reduction with initialization values in ()



Example (OpenMP reduction minimal example)

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
int i, n;
float a[100], b[100], sum;
/* Some initializations */
n = 100;
for (i=0; i < n; i++)
 a[i] = b[i] = i * 1.0:
sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);
printf("____Sum_=_%f\n", sum);
```



The following are cited from OpenMP 3.1 API C/C++ Syntax Quick Reference Card:

"These clauses support the copying of data values from private or threadprivate variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team."

copyin(list)

"Copies the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the parallel region."

copyprivate(list)

"Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the parallel region."





OMP_SCHEDULE

Specifies the runtime schedule type. Available values are static, dynamic, guided, or auto together with an optional chunk size.



OMP_SCHEDULE

Specifies the runtime schedule type. Available values are static, dynamic, guided, or auto together with an optional chunk size.

OMP_NUM_THREADS

Must be set to a list of positive integers determining the numbers of threads at the corresponding nested level.



OMP_SCHEDULE

Specifies the runtime schedule type. Available values are static, dynamic, guided, or auto together with an optional chunk size.

OMP_NUM_THREADS

Must be set to a list of positive integers determining the numbers of threads at the corresponding nested level.

OMP_PROC_BIND

The value of this variable must be true or false. It determines whether threads may be moved between processors at runtime.



OMP_SCHEDULE

Specifies the runtime schedule type. Available values are static, dynamic, guided, or auto together with an optional chunk size.

OMP_NUM_THREADS

Must be set to a list of positive integers determining the numbers of threads at the corresponding nested level.

OMP_PROC_BIND

The value of this variable must be true or false. It determines whether threads may be moved between processors at runtime.

More environment variables can be found in Section 4 of the OpenMP standard.



We only treat thread and processor number related functions

void omp_set_num_threads(int num_threads)

Determines the number of threads in subsequent parallel regions that do not specify a num_threads clause.

int omp_get_num_threads(void)

Returns the number of threads in the current team.



int omp_get_max_threads(void)

Provides the maximum number of threads that could be used in a subsequent parallel construct.

int omp_get_thread_num(void)

Returns the thread ID of the current thread. IDs are integers from zero (the master thread) to the number of threads in the team minus one.

int omp_get_num_procs(void)

returns the number of processors available to the program.

More runtime library functions and detailed descriptions can be found in Section 3 of the OpenMP standard.



Example (Hello World revisited)

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
  int th id, nthreads;
#pragma omp parallel private(th id)
    th id = omp get thread num();
     printf("Hello World from thread %d\n", th_id);
     #pragma omp barrier
     if (th id == 0) {
       nthreads = omp get num threads();
       printf("There.are.%d.threads\n", nthreads);
  return EXIT SUCCESS;
```



Two important rules of thumb:

In case of nested loops it is usually best to apply the parallelization to the outermost possible loop.



Two important rules of thumb:

In case of nested loops it is usually best to apply the parallelization to the outermost possible loop.

It is in general a good idea to first optimize the sequential code and only then add parallelism to further increase the speed of execution.



Multicore and Multiprocessor Systems: Part IV



Example (OpenMP reduction minimal example)

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
int i, n;
float a[100], b[100], sum;
/* Some initializations */
n = 100;
for (i=0; i < n; i++)</pre>
 a[i] = b[i] = i * 1.0;
sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);
printf("____Sum_=_%f\n", sum);
```



The main properties of the reduction are

- accumulation of data via a binary operator (here +)
- intrinsically sequential operation causing a race condition in multi-thread based implementations (since every iteration step depends on the result of its predecessor.)



Tree Reduction Basic idea of tree reduction



Figure: Tree reduction basic idea.



Tree Reduction Basic idea of tree reduction



Figure: Tree reduction basic idea.

- ideally the number of elements is a power of 2
- best splitting of the actual data depends on the hardware used



Example (Another approach for the dot example)

Consider the setting as before $a, b \in \mathbb{R}^{100}$. Further we have four equal cores. How do we compute the accumulation in parallel?


Example (Another approach for the dot example)

Consider the setting as before $a, b \in \mathbb{R}^{100}$. Further we have four equal cores. How do we compute the accumulation in parallel? Basically 2 choices



Example (Another approach for the dot example)

Consider the setting as before $a, b \in \mathbb{R}^{100}$. Further we have four equal cores. How do we compute the accumulation in parallel? Basically 2 choices

1. Task pool approach: define a task pool and feed it with n/2 = 50 work packages accumulating 2 elements into 1. When these are done, schedule the next 25 and so on by further binary accumulation of 2 intermediate results per work package.



Example (Another approach for the dot example)

Consider the setting as before $a, b \in \mathbb{R}^{100}$. Further we have four equal cores. How do we compute the accumulation in parallel? Basically 2 choices

- 1. Task pool approach: define a task pool and feed it with n/2 = 50 work packages accumulating 2 elements into 1. When these are done, schedule the next 25 and so on by further binary accumulation of 2 intermediate results per work package.
- #Processors=#Threads approach: Divide the work by the number of threads, i.e. on our 4 cores each gets 25 subsequent indices to sum up. The reduction is then performed on the results of the threads.



Algorithm 1: Gaussian elimination — row-by-row-version

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition Output: A overwritten by L, U1 for k = 1 : n - 1 do 2 A(k + 1 : n, k) = A(k + 1 : n, b)/A(k, k);3 for i = k + 1 : n do 4 for j = k + 1 : n do 5 A(i, j) = A(i, j) - A(i, k)A(k, j);



Algorithm 1: Gaussian elimination — row-by-row-version Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition

Output: A overwritten by
$$L, U$$

1 for
$$k = 1 : n - 1$$
 do

$$A(k+1:n,k) = A(k+1:n,b)/A(k,k);$$

for $i = k+1:n$ do

or
$$i = k + 1 : n$$
 do
for $j = k + 1 : n$ do
 $| A(i,j) = A(i,j) - A(i,k)A(k,j)$

Observation:

- Innermost loop performs rank-1 update on the A(k + 1 : n, k + 1 : n) submatrix in the lower right,
- i.e. a BLAS level 2 operation.



Algorithm 2: Gaussian elimination — Outer product formulation

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition Output: $L, U \in \mathbb{R}^{n \times n}$ such that A = LU stored in A stored in A1 for k = 1 : n - 1 do 2 | rows= k + 1 : n; 3 | A(rows, k) = A(rows, k)/A(k, k); 4 | A(rows, rows) = A(rows, rows) - A(rows, k)A(k, rows);



Algorithm 2: Gaussian elimination — Outer product formulation

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition **Output**: $L, U \in \mathbb{R}^{n \times n}$ such that A = LU stored in A stored in A

- 1 for k = 1 : n 1 do
- 2 | rows= k + 1 : n;

$$A(rows, k) = A(rows, k)/A(k, k);$$

4 [A(rows,rows) = A(rows,rows) - A(rows,k)A(k,rows);

Idea of the blocked version

- Replace the rank-1 update by a rank-r update ,
- Thus replace the $O(n^2) / O(n^2)$ operation per data ratio the more desirable $O(n^3) / O(n^2)$ ratio,
- Therefore exploit the fast local caches of modern CPUs more optimally.

3



Algorithm 3: Gaussian elimination — Block outer product formulation

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition, *r* prescribed block size **Output**: A = LU with L, U stored in A 1 k = 1; 2 while k < n do 3 $\ell = \min(n, k + r - 1);$ Compute $A(k : \ell, k : \ell) = \tilde{L}\tilde{U}$ via Algorithm 7; 4 Solve $\tilde{L}Z = A(k : \ell, \ell + 1 : n)$ and store Z in A; 5 Solve $W\tilde{U} = A(\ell + 1 : n, k : \ell)$ and store W in A; 6 Perform the rank-r update: 7 $A(\ell + 1: n, \ell + 1: n) = A(\ell + 1: n, \ell + 1: n) - WZ;$ 8 $k = \ell + 1;$



Algorithm 3: Gaussian elimination — Block outer product formulation

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition, *r* prescribed block size **Output**: A = LU with L, U stored in A 1 k = 1: 2 while k < n do $\ell = \min(n, k + r - 1);$ 3 Compute $A(k : \ell, k : \ell) = \tilde{L}\tilde{U}$ via Algorithm 7; 4 Solve $\tilde{L}Z = A(k : \ell, \ell + 1 : n)$ and store Z in A; 5 Solve $W\tilde{U} = A(\ell + 1 : n, k : \ell)$ and store W in A; 6 7 Perform the rank-r update: $A(\ell + 1: n, \ell + 1: n) = A(\ell + 1: n, \ell + 1: n) - WZ;$ $k = \ell + 1$: 8

The block size r can be further exploited in the computation of W and Z and the rank-r update. It is used to optimize the data portions for the cache.























































We have basically two ways to implement naive parallel versions of the block outer product elimination in Algorithm 6.

Threaded BLAS available

- Compute line 4 with the sequential version of the LU
- Exploite the threaded BLAS for the block operations in lines 5–7



We have basically two ways to implement naive parallel versions of the block outer product elimination in Algorithm 6.

Threaded BLAS available

- Compute line 4 with the sequential version of the LU
- Exploite the threaded BLAS for the block operations in lines 5–7

Netlib BLAS

- Compute line 4 with the sequential version of the LU
- Employ OpenMP/PThreads to perform the BLAS calls for the block operations in lines 5–7 in parallel.



Both these approaches fall into the class of parallel codes described by the following paradigm.

Definition (Fork-Join Parallelism)

An algorithm that performs certain parts sequentially between others that are executed in parallel is *called fork-join-parallel*.



Figure: A sketch of the fork-join execution model.



Advantages

- Easy to achieve.
- Many threaded BLAS implementations available.
- Basically usable from any user code that requires linear system solves.

Disadvantages

- Very naive implementation.
- Sequential fraction limits the speedup (Amdahl's law).
- Therefore, only useful for small numbers of cores.



- A directed acyclic graph is a graph where
 - all edges have one distinct direction,
 - directions are such that no cycles are possible for any path in the graph.

Where is the connection to parallel mathematical algorithms?

• Consider every node in the graph a task in the computation.



- A directed acyclic graph is a graph where
 - all edges have one distinct direction,
 - directions are such that no cycles are possible for any path in the graph.

Where is the connection to parallel mathematical algorithms?

- Consider every node in the graph a task in the computation.
- Every task requires a certain number of previous tasks to have finished.



- A directed acyclic graph is a graph where
 - all edges have one distinct direction,
 - directions are such that no cycles are possible for any path in the graph.

Where is the connection to parallel mathematical algorithms?

- Consider every node in the graph a task in the computation.
- Every task requires a certain number of previous tasks to have finished.
- Also none of the previous tasks depend on the later ones.



- A directed acyclic graph is a graph where
 - all edges have one distinct direction,
 - directions are such that no cycles are possible for any path in the graph.

Where is the connection to parallel mathematical algorithms?

- Consider every node in the graph a task in the computation.
- Every task requires a certain number of previous tasks to have finished.
- Also none of the previous tasks depend on the later ones.
- Thus, the dependencies give us the directions and cycles can not appear by construction.









Figure: Dependency graph of Algorithm 6 for a 3×3 block subdivision.





Figure: The superiority of DAG scheduling of tasks over fork-join parallelism.



Multicore and Multiprocessor Systems: Part V



Algorithm 4: Conjugate Gradient Method

Input:
$$A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^{n}, x_{0} \in \mathbb{R}^{n}$$

Output: $x = A^{-1}b$
1 $p_{0} = r_{0} = b - Ax_{0}, \alpha_{0} = ||r_{0}||_{2}^{2}$;
2 for $m = 0, ..., n - 1$ do
3 if $\alpha_{m} \neq 0$ then
4 $\sum_{k=1}^{n} \sum_{m=1}^{n} \sum_{m=1}^{n}$



Algorithm 4: Conjugate Gradient Method

Input:
$$A \in \mathbb{R}^{n \times n}$$
, $b \in \mathbb{R}^{n}$, $x_{0} \in \mathbb{R}^{n}$
Output: $x = A^{-1}b$
1 $p_{0} = r_{0} = b - Ax_{0}$, $\alpha_{0} = ||r_{0}||_{2}^{2}$;
2 for $m = 0, \dots, n-1$ do
3 if $\alpha_{m} \neq 0$ then
4 $\sum_{k=1}^{n} \frac{v_{m} = Ap_{m}}{(v_{m}, p_{m})};$
 $\lambda_{m} = \frac{\alpha_{m}}{(v_{m}, p_{m})};$
 $x_{m+1} = x_{m} + \lambda_{m}p_{m};$
 $r_{m+1} = |r_{m+1}||_{2}^{2};$
 $p_{m+1} = r_{m+1} + \frac{\alpha_{m+1}}{\alpha_{m}}p_{m};$
else
 $\sum_{k=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \sum_{k=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \sum \frac{1}{k} \sum_{k=1}^{n} \frac{1}{k} \sum_{k=1}^{n} \frac{1}{k} \sum \frac{1}{k} \sum_{k=1}^{n} \frac{1}{k} \sum \frac{1}{k} \sum \frac{1$

CG uses

one matrix vector product (performing the main work),


Input:
$$A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^{n}, x_{0} \in \mathbb{R}^{n}$$

Output: $x = A^{-1}b$
1 $p_{0} = r_{0} = b - Ax_{0}, \alpha_{0} = ||r_{0}||_{2}^{2}$;
2 for $m = 0, \dots, n-1$ do
3 if $\alpha_{m} \neq 0$ then
 $\lambda_{m} = \frac{\alpha_{m}}{(v_{m}, p_{m})};$
 $x_{m+1} = x_{m} + \lambda_{m}p_{m};$
 $r_{m+1} = |r_{m+1}||_{2}^{2};$
 $p_{m+1} = r_{m+1} + \frac{\alpha_{m+1}}{\alpha_{m}}p_{m};$
else
 $\sum STOP;$

CG uses

one dot,



Input:
$$A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^{n}, x_{0} \in \mathbb{R}^{n}$$

Output: $x = A^{-1}b$
1 $p_{0} = r_{0} = b - Ax_{0}, \alpha_{0} = ||r_{0}||_{2}^{2}$;
2 for $m = 0, \dots, n-1$ do
3 if $\alpha_{m} \neq 0$ then
 $V_{m} = Ap_{m};$
 $\lambda_{m} = \frac{\alpha_{m}}{(v_{m}, p_{m})};$
 $x_{m+1} = x_{m} + \lambda_{m}p_{m};$
 $r_{m+1} = |r_{m-1}||_{2}^{2};$
 $p_{m+1} = r_{m+1} + \frac{\alpha_{m+1}}{\alpha_{m}}p_{m};$
else
 \downarrow STOP;

CG uses

two axpy,



Input:
$$A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^{n}, x_{0} \in \mathbb{R}^{n}$$

Output: $x = A^{-1}b$
1 $p_{0} = r_{0} = b - Ax_{0}, \alpha_{0} = ||r_{0}||_{2}^{2}$;
2 for $m = 0, \dots, n-1$ do
3 if $\alpha_{m} \neq 0$ then
4 $\lambda_{m} = \frac{\alpha_{m}}{(\nu_{m}, p_{m})};$
 $\lambda_{m} = \frac{\alpha_{m}}{(\nu_{m}, p_{m})};$
 $x_{m+1} = x_{m} + \lambda_{m}p_{m};$
 $r_{m+1} = ||r_{m+1}||_{2}^{2};$
 $p_{m+1} = r_{m+1} + \frac{\alpha_{m+1}}{\alpha_{m}}p_{m};$
else
 $\sum \text{STOP};$

CG uses

one nrm2,



Input:
$$A \in \mathbb{R}^{n \times n}$$
, $b \in \mathbb{R}^{n}$, $x_{0} \in \mathbb{R}^{n}$
Output: $x = A^{-1}b$
1 $p_{0} = r_{0} = b - Ax_{0}$, $\alpha_{0} = ||r_{0}||_{2}^{2}$;
2 for $m = 0, \dots, n-1$ do
3 if $\alpha_{m} \neq 0$ then
4 $\sum_{k=1}^{n} \sum_{m=1}^{k} \sum_{m=1}^{$

CG uses

and a nonstandard axpy operation with result in x.



The key ingredient in the CG method is the sparse matrix vector product (SpMVP).

We learned in part 1 of the lecture that sparse matrix operations are *bandwidth limited*, i.e., the crucial point is always the data transfer for matrix pattern and entries to the processing units.

On the other hand, the SpMVP is trivially parallel due to data parallelism. On multicore architectures the obvious questions are:

- What is the optimal number of threads to use?
- How should the data be distributed among the threads?

First question \rightsquigarrow treated in the exercises.



The second questions is investigated a lot in the literature. We will only sketch a small selection of approaches considering x = Ab for $x, b \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$ sparse with properties specified separately in the method descriptions.



The second questions is investigated a lot in the literature. We will only sketch a small selection of approaches considering x = Ab for $x, b \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$ sparse with properties specified separately in the method descriptions.

Naive row blocking. (e.g., using OpenMP parallel for)

If the matrix A is banded with moderate bandwidth and the number of entries per row is almost the same for all rows, simply grouping the rows in blocks of rows will likely do a good job.

The bandwidth limitations guarantee data locality on b.

Furthermore, the similar lengths of the sparse rows will automatically provide a proper load balancing.

This provides the easiest form of 1d-partitioning.



The simplest form of 2d-partitioning of the matrix A uses (blocks of) columns and (blocks of) rows at the same time. It is usually referred to as hypergraph partitioning since the choice fits the following definition.

Definition (Hypergraph)

A hypergraph is an ordered pair $(\mathcal{V}, \mathcal{E})$ of sets. It is a generalization of a graph that consists of vertices (in the set \mathcal{V}) and hyperedges in the set \mathcal{E} . In contrast to an edge in a graph a hyperedge can be an arbitrary subset of \mathcal{V} and not just a pair.



Sparse Linear Systems of Equations Sparse Matrix Vector Products

Example

Schematic representation of a hypergraph with seven vertices and four hyperedges.





The idea of hypergraph partitioning is to use the hyperedges to find the optimal partitioning of the vertices into k equal sets for optimal balancing of the workload and data communication.

The problem of finding the optimal partition is however np-hard. Therefore cheap heuristics are employed to approximate the optimal partition.

An interesting variant especially for symmetric patterns is the corner symmetric partitioning.



Figure: Corner symmetric partitioning of the arrowhead matrix with 2 partitions.





Figure: arrowhead matrix pattern and connectivity graph.

The central node 8 is called *vertex separator*. The identification of such a (group of) node(s) is the central question in the graph model based partitioning. Successive application of this idea leads to the nested dissection scheme.







Figure: arrowhead matrix pattern and connectivity graph.

The central node 8 is called *vertex separator*. The identification of such a (group of) node(s) is the central question in the graph model based partitioning. Successive application of this idea leads to the nested dissection scheme.



Recall:

A *preconditioner* is an invertible linear operator P that approximates the action of A^{-1} for a linear system Ax = b.

- Invertibility required to ensure proper preservation of solution,
- preconditioner need not be formed as a matrix, as long as its action on a vector can be provided as a function,
- main purpose of the preconditioner is the grouping of eigenvalues, ideally in a single cluster at +1.



Algorithm 5: Preconditioned Conjugate Gradient Method

```
Input: A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n, x_0 \in \mathbb{R}^n, A^{-1} \approx P \in \mathbb{R}^{n \times n}
    Output: x = A^{-1}b
1 r_0 = b - Ax_0, p_0 = z_0 = Pr_0, \alpha_0 = (r_0, p_0);
2 for m = 0 \cdot n - 1 do
          if \alpha_m \neq 0 then
3
                 v_m = Ap_m;
4
                \lambda_m = \frac{\alpha_m}{(v_m, p_m)_2};
5
                 x_{m+1} = x_m + \lambda_m p_m;
6
                 r_{m+1} = r_m - \lambda_m v_m;
7
                 z_{m+1} = Pr_{m+1};
8
                 \alpha_{m+1} = (r_{m+1}, \mathbf{z}_{m+1})_2;
9
                 p_{m+1} = \mathbf{z}_{m+1} + \frac{\alpha_{m+1}}{\alpha} p_m;
10
           else
11
                  STOP;
12
```



Let $D \in \mathbb{R}^{n \times n}$ be a diagonal matrix containing the diagonal of A. Then $P = D^{-1}$ is called Jacobi or diagonal preconditioner.

Properties

- $+\,$ embarrassingly parallel in computation and application,
- + storage requirement n double numbers,
- only useful for diagonally dominant systems.



The basic idea of SPAI is to find the best matrix P approximating A^{-1} , while maintaining the sparsity pattern of A.

$$\min_{\mathcal{P}(P)=\mathcal{P}(A)} \left\| AP - I \right\|_{F}^{2} = \min_{\mathcal{P}(P)=\mathcal{P}(A)}$$

$$\underbrace{\sum_{j=1}^{n} \|Ap_j - e_j\|_F^2}_{\underbrace{}}$$

n independent least squares problems



The basic idea of SPAI is to find the best matrix P approximating A^{-1} , while maintaining the sparsity pattern of A.

$$\min_{\mathcal{P}(P)=\mathcal{P}(A)} \|AP - I\|_F^2 = \min_{\mathcal{P}(P)=\mathcal{P}(A)}$$



n independent least squares problems

+ only SpMVP needed for the application,

+ *n* independent least squares problems allow two multicore approaches:

- rely on threaded BLAS when solving the least squares problems sequentially via dgeqrs() from LAPACK,
- use sequential BLAS with OpenMP for parallel solution of the least squares problems.
- efficient preconditioning requires additional fill-in, which leads to extra storage demands and increased computational complexity.











(a) elimination graph \mathcal{G}_1

(b) corresponding submatrix 1







(a) elimination graph \mathcal{G}_2

(b) corresponding submatrix 2







(a) elimination graph \mathcal{G}_3

(b) corresponding submatrix 3





(a) The filled graph $\mathcal{G}^+(A) = \mathcal{G}(F)$ (b) The final matrix $F = L + L^T$ with fill.

Figure: The filled graph and matrix of a Cholesky decomposition example.



Now



and thus, the forward elimination is purely sequential. Are we lost?

Consider the Cholesky factor:



Consider the Cholesky factor:



Consider the Cholesky factor:





Definition (column pattern)

The *j*-th column pattern \mathcal{P}_{*j} is the set of row indices of all non-diagonal nonzero entries in the *j*-th column.

Definition (Supernode)

A supernode is a set of contiguous column indices

$$\mathcal{I}(p) = \{p, p+1, \ldots, p+q-1\},$$

such that for all columns $i \in \mathcal{I}(p)$ we have

$$\mathcal{P}_{*i} = \mathcal{P}_{*(p+q-1)} \cup \{i+1, ; p+q-1\}$$



Supernodes, thus are special dense diagonal blocks that have the identically same pattern in each column below the diagonal block.



Supernodes, thus are special dense diagonal blocks that have the identically same pattern in each column below the diagonal block.

 Column modifications in forward substitution can be expressed in terms of supernodes rather than single diagonal entries.



- Supernodes, thus are special dense diagonal blocks that have the identically same pattern in each column below the diagonal block.
- Column modifications in forward substitution can be expressed in terms of supernodes rather than single diagonal entries.
- Inside the supernode block operations we can exploit parallelism.










































Sparse Linear Systems of Equations A Task Pool Approach to Parallel Triangular Solves

+ many elimination steps can be executed independently



 + many elimination steps can be executed independently
 + a simple task pool scheduling the independent tasks enables parallel execution and load balancing



- + many elimination steps can be executed independently
- $+\,$ a simple task pool scheduling the independent tasks enables parallel execution and load balancing
- elimination tree must be computed to enable proper scheduling and identification of independent tasks



- + many elimination steps can be executed independently
- + a simple task pool scheduling the independent tasks enables parallel execution and load balancing
- elimination tree must be computed to enable proper scheduling and identification of independent tasks

Remark

Note that elimination trees can be computed without computing the filled graph or the Cholesky factor first.



Dense Linear Algebra

 OpenBLAS based on the earlier GotoBLAS project OpenBLAS implements a complete set of optimized BLAS routines. On a machine with a single socket it is likely the fastest BLAS implementation one can get.⁸

⁸http://xianyi.github.io/OpenBLAS/ 9http://software.intel.com/en-us/intel-mkl ¹⁰http://icl.cs.utk.edu/plasma/software/



Dense Linear Algebra

- OpenBLAS based on the earlier GotoBLAS project OpenBLAS implements a complete set of optimized BLAS routines. On a machine with a single socket it is likely the fastest BLAS implementation one can get.⁸
- Intel[®] Math Kernel Library (MKL) is Intel[®]s optimized implementation of BLAS and LAPACK. It is the strongest opponent of OpenBLAS on single socket systems. On a system with several sockets no other BLAS library outperforms MKL.⁹

⁸http://xianyi.github.io/OpenBLAS/ ⁹http://software.intel.com/en-us/intel-mkl ¹⁰http://icl.cs.utk.edu/plasma/software/



Dense Linear Algebra

- 1. **OpenBLAS** based on the earlier GotoBLAS project OpenBLAS implements a complete set of optimized BLAS routines. On a machine with a single socket it is likely the fastest BLAS implementation one can get.⁸
- Intel[®] Math Kernel Library (MKL) is Intel[®]s optimized implementation of BLAS and LAPACK. It is the strongest opponent of OpenBLAS on single socket systems. On a system with several sockets no other BLAS library outperforms MKL.⁹
- 3. **PLASMA** The Parallel Linear Algebra Subroutines for Multicore Architectures employs DAG scheduling to increase performance of the linear algebra subsystem on multicore architectures.¹⁰

⁸http://xianyi.github.io/OpenBLAS/ ⁹http://software.intel.com/en-us/intel-mkl ¹⁰http://icl.cs.utk.edu/plasma/software/



1. **UMFPACK** comes as part of the SuiteSparse package of software libraries for sparse linear systems of equations. Uses thread parallel multifrontal techniques to solve linear systems of equations.¹¹

//www.boost.org/doc/libs/1_70_0/libs/numeric/ublas/doc/index.htm $^{13}\rm http://www.simunova.com/en/mtl4$

¹¹http://faculty.cse.tamu.edu/davis/suitesparse.html
¹²http:



- 1. **UMFPACK** comes as part of the SuiteSparse package of software libraries for sparse linear systems of equations. Uses thread parallel multifrontal techniques to solve linear systems of equations.¹¹
- Boost uBLAS "is a C++ template class library that provides BLAS level 1, 2, 3 functionality for dense, packed and sparse matrices."¹²

//www.boost.org/doc/libs/1_70_0/libs/numeric/ublas/doc/index.htm
¹³http://www.simunova.com/en/mtl4

¹¹http://faculty.cse.tamu.edu/davis/suitesparse.html
¹²http:



- 1. **UMFPACK** comes as part of the SuiteSparse package of software libraries for sparse linear systems of equations. Uses thread parallel multifrontal techniques to solve linear systems of equations.¹¹
- Boost uBLAS "is a C++ template class library that provides BLAS level 1, 2, 3 functionality for dense, packed and sparse matrices."¹²
- 3. MTL the Matrix Template Library provides an easy to use template based C++ interface to linear algebra operations. It relies on Boost for fast and efficient codes.¹³

//www.boost.org/doc/libs/1_70_0/libs/numeric/ublas/doc/index.htm
 ¹³http://www.simunova.com/en/mtl4

¹¹http://faculty.cse.tamu.edu/davis/suitesparse.html
¹²http:



- 1. **UMFPACK** comes as part of the SuiteSparse package of software libraries for sparse linear systems of equations. Uses thread parallel multifrontal techniques to solve linear systems of equations.¹¹
- Boost uBLAS "is a C++ template class library that provides BLAS level 1, 2, 3 functionality for dense, packed and sparse matrices."¹²
- 3. MTL the Matrix Template Library provides an easy to use template based C++ interface to linear algebra operations. It relies on Boost for fast and efficient codes.¹³
- 4. SuperLU_MT Supernode based multithreaded LU decomposition.¹⁴

//www.boost.org/doc/libs/1_70_0/libs/numeric/ublas/doc/index.htm
¹³http://www.simunova.com/en/mtl4

¹¹http://faculty.cse.tamu.edu/davis/suitesparse.html
¹²http:



PThreads and Scheduling/Memory Control

 nptl is the Native POSIX Linux Thread library that currently provides PThread support on most Linux platforms.¹⁵

¹⁵http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library ¹⁶http://code.google.com/p/likwid/ ¹⁷http://oss.sgi.com/projects/libnuma/



PThreads and Scheduling/Memory Control

- nptl is the Native POSIX Linux Thread library that currently provides PThread support on most Linux platforms.¹⁵
- likwid (Like I Knew What I Do) is a light weight library that supports software developers to design high performance scientific computing programs with little overhead.¹⁶

¹⁵http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library ¹⁶http://code.google.com/p/likwid/ ¹⁷http://oss.sgi.com/projects/libnuma/



PThreads and Scheduling/Memory Control

- nptl is the Native POSIX Linux Thread library that currently provides PThread support on most Linux platforms.¹⁵
- likwid (Like I Knew What I Do) is a light weight library that supports software developers to design high performance scientific computing programs with little overhead.¹⁶
- numactl referred to as libnuma by several Linux distributions, numactl is a small program/library that can be used to control placement of process memory in NUMA environments. The library version seems to be preferred by the Linux kernel policies.¹⁷

¹⁵http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library ¹⁶http://code.google.com/p/likwid/ ¹⁷http://oss.sgi.com/projects/libnuma/



GPU Computing and Accelerators: Part I





(b) Memory bandwidth

Figure: Throughput comparison of Multicore CPUs and CUDA enabled GPUs (taken from CUDA C Programming Guide)



Architecture	GFLOPS	GFLOPS/Watt	Utilization
Core i7-960	96	1.14	95%
Nvidia [®] GTX280	410	2.6	66%
Cell	200	5.0	88%
Nvidia [®] GTX480	940	5.4	70%
TI C66x DSP	74	7.4	57%

Table: Power efficieny comparison of Multicore CPUs and accelerator chips (taken from Conference Poster by F. Igual and M. Ali)





Figure: Schematic of a general parallel system





Figure: Memory configuration of a CUDA Device (taken from CUDA C Programming Guide)



Memory Hierarchy with Accelerators Field Programmable Gate Arrays (FPGAs)



Figure: Comparison of CPUs and FPGA execution models.



GPU Computing and Accelerators: Part II



CUDA is two things at the same time:

1. platform model

for the hardware implementation of general purpose graphics processing units made by the $N \text{VIDIA}^{\text{$\extsf{res}$}}$ Corporation.

2. programming model realizing the software implementation and scheduling of tasks of the parallel programs on the above hardware.



A *thread*, or more precisely *GPU-thread* is the smallest unit of data and instructions to be executed in a parallel CUDA program.



A *thread*, or more precisely *GPU-thread* is the smallest unit of data and instructions to be executed in a parallel CUDA program.

In contrast to CPU-threads a task switch between GPU-threads is usually almost for free due to the special CUDA architecture.



A *thread*, or more precisely *GPU-thread* is the smallest unit of data and instructions to be executed in a parallel CUDA program.

In contrast to CPU-threads a task switch between GPU-threads is usually almost for free due to the special CUDA architecture.

Definition (warp)

The CUDA hardware consists of streaming multi-processors that are executing several threads simultaneously. The GPU-threads are therefore grouped in so called *warps* of threads per multi-processor.



A *thread*, or more precisely *GPU-thread* is the smallest unit of data and instructions to be executed in a parallel CUDA program.

In contrast to CPU-threads a task switch between GPU-threads is usually almost for free due to the special CUDA architecture.

Definition (warp)

The CUDA hardware consists of streaming multi-processors that are executing several threads simultaneously. The GPU-threads are therefore grouped in so called *warps* of threads per multi-processor.

The number of threads in a warp may depend on the hardware. One finds mostly 32 threads per warp which in turn is the smallest number of tasks executed in SIMD style.



A *block* is a larger group of threads that can contain 64–512 threads.



A *block* is a larger group of threads that can contain 64–512 threads.

Ideally it contains a multiple of 32 threads so it can be split optimally into warps by the CUDA environment for scheduling.



A *block* is a larger group of threads that can contain 64–512 threads.

Ideally it contains a multiple of 32 threads so it can be split optimally into warps by the CUDA environment for scheduling.

Definition (grid)

The actual work to be performed by a program or algorithm is distributed to a one or two dimensional *grid* of blocks.



A *block* is a larger group of threads that can contain 64–512 threads.

Ideally it contains a multiple of 32 threads so it can be split optimally into warps by the CUDA environment for scheduling.

Definition (grid)

The actual work to be performed by a program or algorithm is distributed to a one or two dimensional *grid* of blocks.

The grid represents the largest freedom in design that the developer has.



Compute Unified Device Architecture (CUDA) Basic Definitions



Figure: Grids of Thread Blocks (taken from CUDA C programming guide)



The central notions to understand data management in a CUDA program are those of *host* and *device*. Here *host* refers to the computer that hosts the GPU. Especially the CPU and memory of the host are relevant. The *device* then is the GPU installed on the host system.


The central notions to understand data management in a CUDA program are those of *host* and *device*. Here *host* refers to the computer that hosts the GPU. Especially the CPU and memory of the host are relevant. The *device* then is the GPU installed on the host system.

In case multiple GPUs are installed on a single host system with multiple CPUs, each GPU is connected to a single CPU representing a single NUMA node of the host system.



The central notions to understand data management in a CUDA program are those of *host* and *device*. Here *host* refers to the computer that hosts the GPU. Especially the CPU and memory of the host are relevant. The *device* then is the GPU installed on the host system.

In case multiple GPUs are installed on a single host system with multiple CPUs, each GPU is connected to a single CPU representing a single NUMA node of the host system.

The host CPU controls the execution of the program. However host and device may execute their tasks asynchronously. When not specified differently data transfers between them serve as implicit synchronization points.



Definition (kernel)

The *kernel* is the core element of a CUDA parallel program. It represents the function that specifies the work a certain thread in a block on a grid has to execute.

We will see in the course of this Chapter how the thread executing the kernel knows which part of the global problem it has to perform.



We will next introduce the most basic elements of the CUDA C language extension. These consist of two important things.

- 1. **qualifiers** that apply to functions and specify where the function should be executed,
- 2. **launch size specifiers** that control the grid and block sizes that are used to run a kernel.

An extensive API, defining C-style functions and data types to be used in CUDA programs, together with a handful of libraries for several kinds of tasks (e.g., a BLAS implementation) complete the picture.



GPU Computing Applications											
Libraries and Middleware											
CUFFT CUBLAS CULA CURAND MAGMA CUSPARSE		Thrust NPP		VSIPL SVM OpenCurrent		PhysX OptiX iRay		cuDNN TensorRT	MATLAB Mathematic		
	Programming Languages										
- c c+		C++	Fortran		an	Java Python Wrappers		DirectCompute		Directives (e.g. OpenACC	
	CUDA-enabled NVIDIA GPUs										
	Pascal Architecture (compute capabilities 6.x) Maxwell Architecture (compute capabilities 5.x)		GeForce 1000 Series			Quadro P Series			Tesla P Seri		
			GeForce 900 Series			Quadro M Series			Tesla M Ser		
Kepler Architecture (compute capabilities 3.x)			GeForce 700 Series GeForce 600 Series			Quadro K Series			Tesla K Series		
Fermi Architecture (compute capabilities 2.x)			GeForce 500 Series GeForce 400 Series		Quadro Fermi Series						
		200	Entertainmer	nt		rofessional Graphics		Hide	enformance inputing		

Figure: The CUDA GPU computing applications framework (taken from CUDA C programming guide)



 __global__ This qualifier applies to a function and is used to indicate that it in fact represents a kernel.



- __global__ This qualifier applies to a function and is used to indicate that it in fact represents a kernel.
- __device__ The qualifier that specifies functions that should be run on the device, but are not kernels. It can be useful for subtasks called in a kernel. It also applies to variables determining them to reside on the device.



- __global__ This qualifier applies to a function and is used to indicate that it in fact represents a kernel.
- __device__ The qualifier that specifies functions that should be run on the device, but are not kernels. It can be useful for subtasks called in a kernel. It also applies to variables determining them to reside on the device.
- __host___ Being basically redundant this qualifier can be used to explicitly state that a function is to be executed on the host. It is therefore optional.



- __global__ This qualifier applies to a function and is used to indicate that it in fact represents a kernel.
- __device__ The qualifier that specifies functions that should be run on the device, but are not kernels. It can be useful for subtasks called in a kernel. It also applies to variables determining them to reside on the device.
- Lost___Being basically redundant this qualifier can be used to explicitly state that a function is to be executed on the host. It is therefore optional.
- _____shared__applies to a variable declaring that it should reside in the shared memory of a streaming multiprocessor



- __global__ This qualifier applies to a function and is used to indicate that it in fact represents a kernel.
- __device__ The qualifier that specifies functions that should be run on the device, but are not kernels. It can be useful for subtasks called in a kernel. It also applies to variables determining them to reside on the device.
- Lost___Being basically redundant this qualifier can be used to explicitly state that a function is to be executed on the host. It is therefore optional.
- _____shared__applies to a variable declaring that it should reside in the shared memory of a streaming multiprocessor
- __constant__applies to a variable specifying the residence in the constant memory.



- __global__ This qualifier applies to a function and is used to indicate that it in fact represents a kernel.
- __device__ The qualifier that specifies functions that should be run on the device, but are not kernels. It can be useful for subtasks called in a kernel. It also applies to variables determining them to reside on the device.
- __host___ Being basically redundant this qualifier can be used to explicitly state that a function is to be executed on the host. It is therefore optional.
- _____shared__applies to a variable declaring that it should reside in the shared memory of a streaming multiprocessor
- __constant__applies to a variable specifying the residence in the constant memory.

Note that __global__ and __device__ functions are not allowed to be recursive.



The basic launch size specification for a kernel takes the form

```
<<< grid , block size >>>
```

where grid specifies the block distribution and block size indicates the number of threads per block in the grid.

- <<<1, 1>>> launches 1 block with 1 thread
- \blacksquare <<<N, 1>>> launches N blocks with 1 thread each
- \blacksquare <<<1, N>>> launches 1 block with N threads
- \blacksquare <<<N, M>>> launches a 1d grid of N blocks running M threads each



Both the arguments can be two dimensional distributions. CUDA defines special tuple hiding types for these declarations. Using

```
dim3 grid(3,2)
dim3 threads(16,16)
```

one defines a 3×2 grid of blocks for running 256 threads arranged in a 16×16 local grid. These are then used in the launch specification as

<<< grid, threads>>>

Launch size specifications are simply appended to the kernel function name upon calling it.



The following examples are taken from the "CUDA by Example" book.

```
#include "../common/book.h"
__global__ void kernel( void ) { }
int main( void ) {
   kernel<<<1,1>>>();
   printf( "Hello, World!\n" );
   return 0;
}
```



Compute Unified Device Architecture (CUDA)

Introductory Examples

```
#include ".../common/book.h"
 _global__ void add( int a, int b, int *c ) {
    *c = a + b;
int main( void ) {
    int c;
    int *dev c;
    HANDLE ERROR ( cudaMalloc( (void **) & dev c, sizeof(int) ) );
    add<<<1,1>>>(2,7, dev c);
    HANDLE ERROR ( cudaMemcpv ( &c, dev c, sizeof(int),
                               cudaMemcpyDeviceToHost ) );
    printf( "2, +, 7, =, %d\n", c );
    HANDLE ERROR ( cudaFree ( dev c ) );
    return 0;
```



Compute Unified Device Architecture (CUDA)

Introductory Examples

```
#include ".../common/book.h"
device int addem( int a, int b ) {
    return a + b;
__global___ void add( int a, int b, int *c ) {
    \star c = addem(a, b);
int main( void ) {
    int c;
    int *dev c;
    HANDLE ERROR ( cudaMalloc( (void **) & dev c, sizeof(int) ) );
    add<<<1,1>>>(2,7, dev c);
    HANDLE ERROR ( cudaMemcpy ( &c, dev_c, sizeof(int),
                               cudaMemcpyDeviceToHost ) );
    printf( "2, +, 7, =, %d\n", c );
    HANDLE ERROR ( cudaFree ( dev c ) );
    return 0;
```



In order to be able to compile the previous examples, one needs to check a few prerequisites:

- NVIDIA[®] device drivers and hardware,
- NVIDIA[®] CUDA toolkit installation,
- compiler for the host code.



Basic information on CUDA in general can be found at http://www.nvidia.com/cuda.



Basic information on CUDA in general can be found at http://www.nvidia.com/cuda. The Toolkit and all the information on the included accelerated libraries and developer tools can be found at https://developer.nvidia.com/cuda-toolkit.



Basic information on CUDA in general can be found at http://www.nvidia.com/cuda. The Toolkit and all the information on the included accelerated libraries and developer tools can be found at https://developer.nvidia.com/cuda-toolkit.

Regarding the hardware, basically every NVIDIA[®] GPU released after the appearance of the GeForce 8800 GTX in 2006 is CUDA enabled. However, one needs to make sure that the OS version, the device driver and CUDA Toolkit version are fitting. Working combinations should be available in the toolkits documentation.



Basic information on CUDA in general can be found at http://www.nvidia.com/cuda. The Toolkit and all the information on the included accelerated libraries and developer tools can be found at https://developer.nvidia.com/cuda-toolkit.

Regarding the hardware, basically every NVIDIA[®] GPU released after the appearance of the GeForce 8800 GTX in 2006 is CUDA enabled. However, one needs to make sure that the OS version, the device driver and CUDA Toolkit version are fitting. Working combinations should be available in the toolkits documentation.

Regarding the compilers $\mathsf{NVIDIA}^{\textcircled{R}}$ recommends the following

- Microsoft Windows: Visual Studio
- **Linux:** Gnu Compiler Collection (GCC)
- MacOS: GCC as well via Apple's Xcode



We will in the following restrict ourselves to the Linux world again.

Consider our basic "Hello World!" example is stored in a text file called hello_world.cu. Using the nvcc compiler provided in the CUDA Toolkit we can compile it by

nvcc hello_world.cu

Since on Linux nvcc uses gcc to compile the host code this will also generate a binary called a.out. As for gcc we can specify the output filename, i.e. name of the resulting executable via

nvcc hello_world.cu -o hello_world

The file extension .cu is used to indicate that we have a C file with CUDA C extensions.



Among the further compiler options we meet many old friends:

- $-{\tt c}$ for generating object files of single .c or .cu files
- $-\mathbf{g}$ for generating debug information in the host code
- -pg the same for profiling information
 - $\ensuremath{\textsc{o}}$ for specifying the optimization level for the host code
 - -m specify 32 vs 64bit host architecture



Among the further compiler options we meet many old friends:

- $-{\tt c}$ for generating object files of single .c or .cu files
- $-g\,$ for generating debug information in the host code
- -pg the same for profiling information
 - $\ensuremath{\bigcirc}$ for specifying the optimization level for the host code
 - -m specify 32 vs 64bit host architecture

And we have a few more for the device code, e.g.

- $-\ensuremath{\mathsf{G}}$ generates debug information for the device code
- arch specifies the GPU architecture to be assumed, i.e. the compute capabilities of the device (e.g. -arch=sm_20)



GPU Computing and Accelerators: Part III



Feature Support	Compute Capability							
(Features differently supported)	1.0	1.1	1.2	1.3	2.×	3.0	3.5	
Atomic functions on 32-bit integer values in global memory	No		Yes					
atomicExch() on 32-bit floating point values in global memory	No			Yes				
Atomic functions on 32-bit integer values in shared memory	No		Yes					
atomicExch() on 32-bit floating point values in shared memory	No			Yes				
Atomic functions on 64-bit integer values in global memory		No		Yes				
Warp vote functions		No		Yes				
Double-precision floating-point numbers	No			Yes				
Atomic functions operating on 64-bit integer values in shared memory	No			Yes				
Atomic addition operating on 32-bit floating point values in global and shared	No			Yes				
memory								
ballot() (Warp Vote Functions)	No			Yes				
threadfence_system()	No			Yes				
syncthreads_count()	No			Yes				
syncthreads_and()	No			Yes				
syncthreads_or()	No			Yes				
Surface functions	No			Yes				
3D grid of thread blocks	No			Yes				
Funnel shift (see reference manual)	No						Yes	

Table: Compute Capabilities: Features by Compute Capability Version (from CUDA C Programming Guide version 5.0)

For the latest version see the CUDA C Programming Guide online.



	Compute Capability							
Technical Specifications	1.0 1.1	1.2	1.3	2.x	3.0	3.5		
Maximum dimensionality of grid of thread blocks	2 3							
Maximum x-dimension of a grid of thread blocks	$65535 = 2^{16} - 1$				$2^{31} - 1$			
Maximum y- or z-dimension of a grid of thread blocks	65535							
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	5	1024						
Maximum z-dimension of a block	64							
Maximum number of threads per block	512			1024				
Warp size	32							
Maximum number of resident blocks per multiprocessor			16					
Maximum number of resident warps per multiprocessor	24 32		48 64		j4			
Maximum number of resident threads per multiprocessor	768 1024		4	1536 2048)48		
Number of 32-bit registers per multiprocessor	8 K 16 K		32 K 64 K		K			
Maximum number of 32-bit registers per thread	128			63 255				
Maximum amount of shared memory per multiprocessor	16 KB			48 KB				
Number of shared memory banks	16			32				
Amount of local memory per thread	16 KB			512 KB				
Constant memory size	64 KB							
Cache working set per multiprocessor for constant memory	8 KB							
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB							
Maximum number of instructions per kernel	2 million			512 million				

Table: Compute Capabilities: Selected Technical Specifications (from CUDA C Programming Guide version 5.0)

For the latest version see the CUDA C Programming Guide online.



Compute capabilities 1.3

We have learned from Table 3 that double precision floating point numbers have been added in Version 1.3 of the CUDA compute capabilities. It additionally provides a fused multiply add operation merging multiplication and addition to be faster and more accurate, but non IEEE 754 compliant.

Compute Capabilities 2.0 and above

Compute capabilities 2.0 introduces IEEE 754 compliance for most parts of the standard as the default. The compiler switches <code>-ftz=false|true</code>, <code>-prec-div=true|false</code>, <code>-prec-sqrt=true|false</code> influence IEEE compliance of the computation. If the second option is used everywhere one switches to fast mode. The first options are the default though.



IEEE 754 Rounding Modes

- IEEE 754 defines four rounding modes
 - round to nearest,
 - round towards zero,
 - round towards $+\infty$,
 - round towards $-\infty$,

all of which are supported by CUDA. However in contrast to x86 CPUs where they can be dynamically switched, CUDA uses them statically.

Compiler intrinsics can be used to change the mode for individual operations, though.



Compute Unified Device Architecture (CUDA) CUDA and IEEE 754 Floating Point Computations

Main Differences to x86 CPUs

no dynamical control of rounding modes



Main Differences to x86 CPUs

- no dynamical control of rounding modes
- floating point exceptions not handled (especially all NaNs are silent)



Main Differences to x86 CPUs

- no dynamical control of rounding modes
- floating point exceptions not handled (especially all NaNs are silent)
- no status flags indicating the exceptions exist



Local versus Remote memory

Viewing from the host perspective, the device memory is remote memory that can only be accessed via the comparably slow system bus.

Looking at things from the device perspective the same hold for the hosts memory. Going even further, already the device memory may be considered slow from the view of the streaming multiprocessors. The local memory of the multiprocessors should be used to implement a user controlled cache.



Compute Unified Device Architecture (CUDA) Data Communication Issues

Thread Per-thread local memory Thread Block Per-block shared memory Grid 0 Block (1, 0) Block (2, 0) Block (0, 0) Block (1, 1) Block (2, 1) Block (0, 1) STREET IS Grid 1 Global memory Block (0, 0) Block (1, 0) Block (1, 1) Block (0, 1) Block (0, 2) Block (1, 2)

Figure: The CUDA memory hierarchy (taken from CUDA C programming guide)



Compute Unified Device Architecture (CUDA) Data Communication Issues

Consequences for CUDA Programs

• Keep data movements between device and host as little as possible



Consequences for CUDA Programs

- Keep data movements between device and host as little as possible
- If they are necessary, try to overlap communication and computations



Figure: Execution patterns for CUDA programs


Consequences for CUDA Programs

- Keep data movements between device and host as little as possible
- If they are necessary, try to overlap communication and computations
- Make use of multiprocessors local shared memory to cache buffer kernel operations and avoid frequent access to global device memory



Example

../Material/CUDAbyExample/chapter05/dot.cu
Note:

- automatic scaling of blocksPerGrid
- usage of local shared buffer cache
- synchronization in reduction block



qualifiers: __global__, __device__, __host__, __shared__, __constant__



- **qualifiers:** __global__, __device__, __host__, __shared__, __constant__
- launch size specifiers <<<grid, block size>>>



- **qualifiers:** __global__, __device__, __host__, __shared__, __constant__
- launch size specifiers <<<grid, block size>>>
- type dim3



- **qualifiers:** __global__, __device__, __host__, __shared__, __constant__
- launch size specifiers <<<grid, block size>>>
- type dim3
- predefined variables: threadIdx.x, blockIdx.x, blockDim.x, gridDim.x



- **qualifiers:** __global__, __device__, __host__, __shared__, __constant__
- launch size specifiers <<<grid, block size>>>
- type dim3
- predefined variables: threadIdx.x, blockIdx.x, blockDim.x, gridDim.x
- memory functions: cudaMalloc(), cudaFree(), cudaMemcpy()



- **qualifiers:** __global__, __device__, __host__, __shared__, __constant__
- launch size specifiers <<<grid, block size>>>
- type dim3
- predefined variables: threadIdx.x, blockIdx.x, blockDim.x, gridDim.x
- memory functions: cudaMalloc(), cudaFree(), cudaMemcpy()
- thread synchronization mechanism: __syncthreads();



- **qualifiers:** __global__, __device__, __host__, __shared__, __constant__
- launch size specifiers <<<grid, block size>>>
- ∎ type dim3
- predefined variables: threadIdx.x, blockIdx.x, blockDim.x, gridDim.x
- memory functions: cudaMalloc(), cudaFree(), cudaMemcpy()
- thread synchronization mechanism: __syncthreads();

Some have been introduced earlier. For the others and a few more we will go into some more detail now.



cudaError_t cudaFree (void* devPtr)

Frees the memory on the device that is refered to by devPtr.



cudaError_t cudaFree (void* devPtr)

Frees the memory on the device that is refered to by devPtr.

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

Allocate an amount corresponding to size of memory on the device and associate it to devPtr.



cudaError_t cudaFree (void* devPtr)

Frees the memory on the device that is refered to by devPtr.

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

Allocate an amount corresponding to size of memory on the device and associate it to devPtr.

Copy data between host and device. src and dst represent the source and destination memory locations. The direction of operation is specified by kind and can be either cudaMemcpyHostToDevice, or cudaMemcpyDeviceToHost. The count argument is used to specify the number of data items to be copied.



cudaError_t cudaGetDeviceCount (int* count)

Returns the number of compute-capable devices available in the system.



cudaError_t cudaGetDeviceCount (int* count)

Returns the number of compute-capable devices available in the system.

cudaError_t cudaChooseDevice (int* device, const cudaDeviceProp* prop)

Select compute-device which best matches criteria specified in prop. These can, e.g., be int major, int minor version numbers of the compute capabilities, or whether the chip is int integrated in the chipset or a plugged in device, but also simply the char name[256] of the device, and many more.



cudaError_t cudaGetDevice (int* device)

Returns which device is currently used by the program.



cudaError_t cudaGetDevice (int* device)

Returns which device is currently used by the program.

cudaError_t cudaSetDevice (int device)

Set device to be used for GPU executions



cudaError_t cudaGetDevice (int* device)

Returns which device is currently used by the program.

cudaError_t cudaSetDevice (int device)

Set device to be used for GPU executions

cudaError_t cudaDeviceSynchronize (void)

Wait for compute device to finish. If for the current device the synchronization flag cudaDeviceScheduleBlockingSync was set, the host thread will block until the device has finished its work.



const __cudart_builtin__ char* cudaGetErrorString (cudaError_t error)

Returns the message string for the error code given in error.



const ___cudart_builtin___ char* cudaGetErrorString (cudaError_t error)

Returns the message string for the error code given in error.

```
cudaError_t cudaGetLastError ( void )
```

Returns the last error that has been produced by any of the runtime calls in the same host thread and resets it to cudaSuccess.



const __cudart_builtin__ char* cudaGetErrorString (cudaError_t error)

Returns the message string for the error code given in error.

cudaError_t cudaGetLastError (void)

Returns the last error that has been produced by any of the runtime calls in the same host thread and resets it to cudaSuccess.

```
cudaError_t cudaPeekAtLastError ( void )
```

As above but does not reset the error code.



cudaError_t cudaEventCreate (cudaEvent_t* event)

Creates, i.e., initializes the event object event.



The CUDA Application Programmers Interface: Events and Performance Measures

cudaError_t cudaEventCreate (cudaEvent_t* event)

Creates, i.e., initializes the event object event.

cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream = 0)

Record event. The record may take some time so before evaluation it is recommended to use cudaEventSynchronize() to make sure it has terminated.



The CUDA Application Programmers Interface: Events and Performance Measures

cudaError_t cudaEventCreate (cudaEvent_t* event)

Creates, i.e., initializes the event object event.

```
cudaError_t cudaEventRecord ( cudaEvent_t event, cudaStream_t stream = 0 )
```

Record event. The record may take some time so before evaluation it is recommended to use cudaEventSynchronize() to make sure it has terminated.

cudaError_t cudaEventSynchronize (cudaEvent_t event)

Wait until event has completed operations.



The CUDA Application Programmers Interface: Events and Performance Measures

cudaError_t cudaEventCreate (cudaEvent_t* event)

Creates, i.e., initializes the event object event.

```
cudaError_t cudaEventRecord ( cudaEvent_t event, cudaStream_t stream = 0 )
```

Record event. The record may take some time so before evaluation it is recommended to use cudaEventSynchronize() to make sure it has terminated.

cudaError_t cudaEventSynchronize (cudaEvent_t event)

Wait until event has completed operations.

Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds).



Compute Unified Device Architecture (CUDA)

The CUDA Application Programmers Interface: Events and Performance Measures

Example

A minimal performance measurement configuration:

```
cudaEvent_t start, stop;
cudaEventCreate(start);
cudaEventCreate(stop);
cudaEventRecord(start, 0);
```

```
// complete some tasks
```

```
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

```
float etime;
cudaEventElapsedTime( &etime, start, stop);
```



GPU Computing and Accelerators: Part IV

Compute Unified Device Architecture (CUDA)

Definition (Stream)

CSC

Streams are a mechanism that introduces an additional level of parallelism into the CUDA framework. While the basic setup, we have seen until here, is SIMD or more precisely SIMT, using streams one can have the GPU do different things at the same time. Streams are not as flexible and "general purpose" as tasks on the host CPU, though.

The basic power of streams is to have memory transfers and computational operations overlap in an **asynchronous** way. Note, however, that not all CUDA enabled devices support overlapping these operations. On top of that, not all CUDA enabled devices that do support the overlapping execution do so in the same way.



Asynchronous data transfers in CUDA are not only performed without synchronization to the actual computation, they are also intended to interact with the computation as little as possible. Especially, they should not interrupt the CPU from performing useful work in the program. They are therefore set up to use direct memory access (DMA) circumventing CPU interaction.

However, in order to do this, we need to use a special portion of host memory, that is guaranteed to stay in place during the operation. The default portion of host memory that we allocate using malloc() is paged memory. It can be anywhere in the virtual memory of the host and is allowed to move around, e.g., to get swapped to disk when more space is required.



Definition (page-locked memory)

Page-locked memory is a portion of memory that is guaranteed to keep its position in the virtual memory. It is not available for any kind of paging operations, such as swapping. Therefore, it is sometimes also called pinned memory.

Advantages of pinned memory:

- can be used for DMA safely
- transfer speeds can be up to $2 \times$ faster than to/from pageable memory

Disadvantages:

memory fragmentation increases and thus the usability deteriorates.



Over the years, NVIDIA[®] has changed the way things are implemented. This is not only regarding the API in the CUDA toolkit, but also the underlying device hardware. The very first CUDA enabled devices could not overlap transfers and executions at all. Then, some devices used separate engines for copy and kernel executions. Modern hardware usually has even two engines for performing transfers in direction to the host and to the device separately. Basically, we can classify the devices as follows:

Comp. Capab.	Properties
1.0	No overlap
1.1-	1 copy engine and 1 kernel execution engine
2.x-	1 kernel execution engine, 1 copy to host engine and 1
	copy to device engine
3.5-	eliminates the differences in asynchronous execution

Table: classification of CUDA enabled devices with respect to the ability of overlapping memory transfers and computations.



How can I know what my device can do?

The cudaDeviceProp structure can be used to find out whether a device supports overlapped operation and how many execution engines are available. The important members are

- int deviceOverlap indicating the availability of overlapped operations
- int asyncEngineCount storing the number of asynchronous execution engines available.

The important information, which type of asynchronous execution model is implemented in the hardware can thus be fetched with the cudaGetDeviceProperties() function.



We are following an NVIDIA $^{\textcircled{R}}$ developer's blog. 18 What we want to do is

- copy data to the device
- perform some task (kernel) on it
- get the result back to the host

Example

The the critical portion of the code would look like

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>> (d_a)
cudaMemcpy(a, d a, numBytes, cudaMemcpyDeviceToHost);
```

according to what we have learned until now. This is regarding the default execution stream.

//developer.nvidia.com/content/how-overlap-data-transfers-cuda-cc

¹⁸https:



Example (Creation and Destruction of Streams)

Consider we have the two variables

cudaStream_t stream1; cudaError_t result;

Then we can create a new stream using

result = cudaStreamCreate(&stream1);

and later get rid of it via

result = cudaStreamDestroy(stream1);



Example (Memory transfers)

Once we have acquired a new stream we have to tell the asynchronous copy routines to use it. The basic command <code>cudaMemcpyAsync()</code> takes the same arguments as <code>cudaMemcpy</code>. Only, it has an additional argument specifying the stream to use:

result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);

Example (Kernel Execution)

We need to use the extended launch size specification here:

```
<<< block distr., thread distr., dyn. mem. per block, associated stream >>>
```

The third argument can be used to allocate additional dynamic shared memory per block. We will use 0 here.

kernel<<<1, N, 0, stream1>>>(d_a);



The influence of the number of engines (especially for copying data) is best displayed in a simple example.

Consider we have a group of streams cooperating on <code>kernel()</code>. Think of a situation where splitting the problem data into chunks is necessary to fit the data into the device memory. We basically have two ways to implement the cooperation,

- $1. \ \mbox{loop}$ over the entire copy-work-copy block
- 2. loop over the work and copies separately

Note that the asynchronous copy acts different on the control flow than the cudaMemcpy(). While in the default stream, using cudaMemcpy(), we can rely on the fact that as soon as the command returns, all data has been transferred, in the case of cudaMemcpyAsync() it does not even guarantee that the copy operation has started at all. It will only have scheduled the operation in a first in first out (FIFO) list of pending operations on the corresponding asynchronous execution engine.



Example (Asynchronous Execution Version 1)

Looping over the entire block of copy-work-copy operations is described by the following code fragment


Example (Asynchronous Execution Version 2)

Looping over the single tasks in contrast looks like



C1060 Execution Time Lines

Sequential Version

Copy Engine	H2D - Stream 0		D2H - 0	
Kernel Engine		0		

Asynchronous Version I

Copy Engine	H2D - 1		D2H - I H2D - 2		D2H - 2 H2D - 3		D2H - 3 H2D - 4		D2H - 4
Kernel Engine		1.1		2		3		4	

Asynchronous Version 2





Figure: Execution time line on a device with a single copy engine.



C2050 Execution Time Lines

Sequential Version



Asynchronous Version I

H2D Engine	1.1	2	3	4		
Kernel Engine		1	2	3	4	
D2H Engine			1	2	3	4

Asynchronous Version 2





Figure: Execution time line on a device with separate copy engines for device to host (D2H) and host to device (H2D) operations.



Interoperability with Graphics

Using the same GPU for computations and graphical display of results is possible. See, e.g. CUDA by Example (Chapter 8), or CUDA C Programming Guide.



Interoperability with Graphics

Using the same GPU for computations and graphical display of results is possible. See, e.g. CUDA by Example (Chapter 8), or CUDA C Programming Guide.

Usage of Multiple GPUs

Usage of multiple GPUs in a single program requires the concepts of zero-copy host memory, and portable pinned memory. An introduction can be found in CUDA by Example (Chapter 11).



GPU Computing and Accelerators: Part V

CSC Open Computing Language (OpenCL)

Main Message

The abstraction for the programming and hardware models are very similar to the CUDA concepts. Mainly OpenCL delivers slightly more flexible implementations due to vendor independence and uses slightly different vocabulary for the single ingredients of the concept.

CUDA	OpenCL
thread	(Work) item
block	(Work) group
streaming multiprocessor	compute unit
(CUDA) processor	processing unit

Table: A short CUDA to OpenCL dictionary



Algorithm 6: Gaussian elimination — Block outer product formulation

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition, *r* prescribed block size **Output**: A = LU with L, U stored in A 1 k = 1: 2 while $k \leq n$ do $\ell = \min(n, k + r - 1);$ 3 Compute $A(k : \ell, k : \ell) = \tilde{L}\tilde{U}$ via Algorithm 7; 4 Solve $\tilde{L}Z = A(k : \ell, \ell + 1 : n)$ and store Z in A; 5 6 Solve $W\tilde{U} = A(\ell + 1 : n, k : \ell)$ and store W in A; 7 Perform the rank-r update: $A(\ell + 1: n, \ell + 1: n) = A(\ell + 1: n, \ell + 1: n) - WZ;$ 8 $k = \ell + 1;$



A











 $A(1: \ell, \ell + 1: n)$











W



























The central question for the hybrid CPU/GPU version of the algorithm now is where to execute the single steps of the algorithm compared to the DAG scheduled version.

Requirements

- Keep data transfers between host and device limited
- optimize usage of both host and device features
- assume that the entire matrix fits into the device memory.

The assumption on the matrix size may be loosened but will then lead to a completely different algorithm.







CPU	GPU	GPU
GPU	CPU	GPU
GPU	GPU	CPU



In each outer iteration step perform the leading $r \times r$ blocks LU decomposition



Algorithm 6: Conjugate Gradient Method

```
Input: A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n, x_0 \in \mathbb{R}^n
    Output: x = A^{-1}b
1 p_0 = r_0 = b - Ax_0, \ \alpha_0 = ||r_0||_2^2;
<sup>2</sup> for m = 0, ..., n - 1 do
          if \alpha_m \neq 0 then
3
               v_m = Ap_m;
4
              \lambda_m = \frac{\alpha_m}{(v_m, p_m)};
5
                x_{m+1} = x_m + \lambda_m p_m;
6
                r_{m+1} = r_m - \lambda_m v_m;
7
               \alpha_{m+1} = \|r_{m+1}\|_2^2;
8
              p_{m+1} = r_{m+1} + \frac{\alpha_{m+1}}{\alpha_m} p_m;
9
          else
10
11
                  STOP:
```



There are mainly two observations we can draw from the algorithm.

- 1. The single steps need to be executed mainly sequentially
- 2. basically all operations are vector operations.

There is not much to distribute between host and device. To exploit the devices vector features all operations should be executed on the device. In case the matrix can not be stored in device memory completely it may be beneficial to use streams to split the operation into chunks that can be stored and operate on those streams in a round robin fashion.



Basic Idea

- Very similar to iterative linear solvers based on Krylov subspaces.
- Main ingredient is to use the basis of the subspace to project the eigenvalue problem to a much smaller space and solve it with dense methods there, i.e. A ∈ ℝ^{n×n} large and sparse U ∈ ℝ^{m×n}, m ≪ n orthogonal, then

$$\underbrace{UAU^{T}}_{m \times m} x = \lambda x$$

is an *m*-dimensional dense eigenproblem.

Here one can offload the solution of the small eigenvalue problem to the host, while the device keeps extending the basis further. The host can then decide whether the approximation is good enough, or the extension is required and the computation needs to continue.



- CUDA Math provides basically all math functions in math.h as device functions.
- **CUBLAS** the CUDA device based implementation of BLAS
- **CUFFT** CUDA based Fast Fourier Transforms, i.e., divide and conquer based computation of Fourier transforms of complex and real valued data sets.
- CURAND The CURAND library provides facilities that focus on the simple and efficient generation of high-quality pseudorandom and quasirandom numbers.
- CUSPARSE Vector-vector and matrix-vector operations where at least one participant is sparse.
- Thrust A C++ template library based on the Standard Template library (STL) for minimal effort implementation of parallel programs.
- **CUSOLVER** Solvers for Ax = b, or $x = \operatorname{argmin}_{z} ||Az b||$ and sequences thereof. (both sparse and dense)



Matrix Algebra on GPU and Multicore Architectures (MAGMA)¹⁹

"The MAGMA project aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current "Multicore+GPU" systems.

The MAGMA research is based on the idea that, to address the complex challenges of the emerging hybrid environments, optimal software solutions will themselves have to hybridize, combining the strengths of different algorithms within a single framework. Building on this idea, we aim to design linear algebra algorithms and frameworks for hybrid manycore and GPU systems that can enable applications to fully exploit the power that each of the hybrid components offers."

¹⁹http://icl.cs.utk.edu/magma/index.html



Formal Linear Algebra Methodology Environment (FLAME)²⁰

"The objective of the FLAME project is to transform the development of dense linear algebra libraries from an art reserved for experts to a science that can be understood by novice and expert alike. Rather than being only a library, the project encompasses a new notation for expressing algorithms, a methodology for systematic derivation of algorithms, Application Program Interfaces (APIs) for representing the algorithms in code, and tools for mechanical derivation, implementation and analysis of algorithms and implementations."

²⁰http://www.cs.utexas.edu/~flame/web/



"Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Get Started with Cusp today!"

²¹https://github.com/cusplibrary



"Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Get Started with Cusp today!"

Matrix formats:

- Coordinate (COO)
- Compressed Sparse Row (CSR)
- Diagonal (DIA)
- ELL (ELL)
- Hybrid (HYB)

²¹https://github.com/cusplibrary



"Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Get Started with Cusp today!"

More Features:

- Format conversion
- Dense Arrays
- File I/O (Matrix Market format)

²¹https://github.com/cusplibrary



"Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Get Started with Cusp today!"

Supported Iterative Solvers:

- Conjugate-Gradient (CG)
- Biconjugate Gradient (BiCG)
- Biconjugate Gradient Stabilized (BiCGstab)
- Generalized Minimum Residual (GMRES)
- Multi-mass Conjugate-Gradient (CG-M)
- Multi-mass Biconjugate Gradient stabilized (BiCGstab-M)

²¹https://github.com/cusplibrary



"Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Get Started with Cusp today!"

Preconditioners:

- Algebraic Multigrid (AMG) based on Smoothed Aggregation
- Approximate Inverse (AINV)
- Diagonal

²¹https://github.com/cusplibrary


CULA tools²²

"CULA is a set of GPU-accelerated linear algebra libraries utilizing the NVIDIA CUDA parallel computing architecture to dramatically improve the computation speed of sophisticated mathematics."

They have separate packages for sparse and dense operation. The libraries are however commercial.

Besides those, there are many scientific computing packages that support GPU operations in one way or the other. Also python has packages for both CUDA (pyCUDA) and OpenCL (pyOpenCL) and MATLAB supports (basically dense only) operation on CUDA devices.

²²http://www.culatools.com



"Ginkgo is a high-performance linear algebra library for manycore systems, with a focus on sparse solution of linear systems. It is implemented using modern C++ (you will need at least C++11 compliant compiler to build it), with GPU kernels implemented in CUDA."

²³https://github.com/ginkgo-project/ginkgo



"Ginkgo is a high-performance linear algebra library for manycore systems, with a focus on sparse solution of linear systems. It is implemented using modern C++ (you will need at least C++11 compliant compiler to build it), with GPU kernels implemented in CUDA."

Matrix formats:

- Coordinate (COO)
- Compressed Sparse Row (CSR)
- hybrid
- dense
- ELL-P
- SELL-P

²³https://github.com/ginkgo-project/ginkgo



"Ginkgo is a high-performance linear algebra library for manycore systems, with a focus on sparse solution of linear systems. It is implemented using modern C++ (you will need at least C++11 compliant compiler to build it), with GPU kernels implemented in CUDA."

More Features:

- Format conversion
- Dense Arrays
- File I/O (Matrix Market format)
- UFL Collection

²³ https://github.com/ginkgo-project/ginkgo



"Ginkgo is a high-performance linear algebra library for manycore systems, with a focus on sparse solution of linear systems. It is implemented using modern C++ (you will need at least C++11 compliant compiler to build it), with GPU kernels implemented in CUDA."

Supported Iterative Solvers:

- Conjugate-Gradient (CG)
- Biconjugate Gradient (BiCG)
- Biconjugate Gradient Stabilized (BiCGstab)
- Conjugate-Gradient squared (CGS)
- flexible CG (FCG)
- Generalized Minimum Residual (GMRES)

²³ https://github.com/ginkgo-project/ginkgo



"Ginkgo is a high-performance linear algebra library for manycore systems, with a focus on sparse solution of linear systems. It is implemented using modern C++ (you will need at least C++11 compliant compiler to build it), with GPU kernels implemented in CUDA."

Preconditioners:

Jacobi type

²³https://github.com/ginkgo-project/ginkgo



Distributed Memory Systems: Part I





Figure: Distributed memory computer schematic





Figure: Distributed memory computer schematic





Figure: Distributed memory computer schematic



1. **TOP500**²⁴:

List of the 500 fastest HPC machines in the world sorted by their maximal $LINPACK^{25}$ performance (in TFlops) achieved.

²⁴http://www.top500.org/ ²⁵http://www.netlib.org/benchmark/hpl/



1. **TOP500:**

List of the 500 fastest HPC machines in the world sorted by their maximal LINPACK performance (in TFlops) achieved.

2. Green500²⁴:

Taking into account the energy consumption the Green500 is basically a resorting of the TOP500 according to TFlops/Watt as the ranking measure.

²⁴http://www.green500.org/



1. **TOP500:**

List of the 500 fastest HPC machines in the world sorted by their maximal LINPACK performance (in TFlops) achieved.

2. Green500:

Taking into account the energy consumption the Green500 is basically a resorting of the TOP500 according to TFlops/Watt as the ranking measure.

3. (Green) Graph500²⁴:

Designed for data intensive computations it uses a graph algorithm based benchmark to rank the supercomputers with respect to GTEPS (10^9 Traversed edges per second). As for the TOP500 a resorting of the systems by an energy measure is provided, as the Green Graph 500 list²⁵.

²⁴http://www.graph500.org/

²⁵http://green.graph500.org/



The ten leading systems in the TOP500 list are currently (list of November 2018) of three different types representing the main streams pursued in increasing the performance of distributed HPC systems.

Mainly all HPC systems today consist of single hosts of one of the following three types. The performance boost is achieved by connecting ever increasing numbers of those hosts in large clusters.



1. Hybrid accelerator/CPU hosts,

Summit — IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM at DOE/SC/Oak Ridge National Laboratory United States Piz Daint — Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect, NVIDIA Tesla P100, Cray Inc. at Swiss National Supercomputing Centre (CSCS)

Switzerland



1. Hybrid accelerator/CPU hosts,

Summit — IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM at DOE/SC/Oak Ridge National Laboratory United States Piz Daint — Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect, NVIDIA Tesla P100, Cray Inc. at Swiss National Supercomputing Centre (CSCS)

Switzerland

2. Manycore and embedded hosts

Sunway TaihuLight — Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC

Sequoia — BlueGene/Q, Power BQC 16C 1.60 GHz at DOE/NNSA/LLNL United States



1. Hybrid accelerator/CPU hosts,

Summit — IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM at DOE/SC/Oak Ridge National Laboratory United States Piz Daint — Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect, NVIDIA Tesla P100, Cray Inc. at Swiss National Supercomputing Centre (CSCS)

Switzerland

2. Manycore and embedded hosts

Sunway TaihuLight — Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC

 ${\tt Sequoia-BlueGene/Q, Power BQC 16C 1.60 GHz at DOE/NNSA/LLNL}$

United States

3. Multicore CPU powered hosts,

SuperMUC-NG — ThinkSystem SD530, Xeon Platinum 8174 24C 3.1GHz, Intel

Omni-Path, Lenovo at Leibniz Rechenzentrum Germany



We have elaborately studied these hosts in the previous chapter.

Compared to a standard desktop (as treated there) in the cluster version the interconnect plays a more important role. Especially, Multi-GPU features may use GPUs on remote hosts (as compared to remote NUMA nodes) more efficiently due to the high speed interconnect.

Compared to CPU-only hosts, these systems usually benefit from the large number of cores generating high flop-rates at comparably low energy costs.



Manycore and embedded systems are designed to use low power processors to get a good flop per Watt ratio. They make up for the lower per core flop counts by using enormous numbers of cores.

BlueGene/Q

- Base chip IBM PowerPC 64Bit based, 16(+2) cores, 1.6GHz
- each core has a SIMD Quad-vector double precision FPU
- 16 user cores, 1 system assist core, 1 spare core
- cores connected to 32MB eDRAM L2Cache (half core speed) via crossbar switch
- crates of 512 chips arranged in 5d torus $(4 \times 4 \times 4 \times 4 \times 2)$
- chip-to-chip communication at 2Gbit/s using on-chip logic
- 2 crates per rack \rightsquigarrow 1024 compute nodes = 16,384 user cores
- interconnect added in 2 drawers with 8 PCIe slots (e.g. for Infiniband, or 10Gig Ethernet.)



Basically these clusters are a collection of standard processors. The actual multicore processors, however, are not necessarily of x86 or amd64 type, e.g. many employ IBM Power 9 processors.

Standard x86 or amd64 provide the obvious advantage of easy usability, since software developed for standard desktops can be ported easily. The SPARC and POWER processors overcome some of the x86 disadvantages (e.g. expensive task switches) and thus often provide increased performance due to reduced latency.



difference	name	meaning
	(symbol)	
	Kilobyte (kB)	10^3 Byte = 1000 Byte
2,40%	Kibibyte (KiB)	2^{10} Byte = 1024 Byte
	Megabyte (MB)	$10^{6} \text{ Byte} = 1000000 \text{ Byte}$
4,86%	Mebibyte (MiB)	2^{20} Byte = 1 048 576 Byte
	Gigabyte (GB)	$10^9 \text{ Byte} = 1000000000 \text{ Byte}$
7,37%	Gibibyte (GiB)	2^{30} Byte = 1 073 741 824 Byte
	Terabyte (TB)	10^{12} Byte = 1 000 000 000 000 Byte
9,95%	Tebibyte (TiB)	2^{40} Byte = 1 099 511 627 776 Byte
	Petabyte (PB)	$10^{15} \text{ Byte} = 1000000000000 \text{ Byte}$
12,6%	Pebibyte (PiB)	2^{50} Byte = 1 125 899 906 842 624 Byte
	Exabyte (EB)	$10^{18} \ {\rm Byte} = 1000000000000000000$ Byte
15,3%	Exbibyte (EiB)	$2^{60} {\rm Byte} = 1152921504606846976 {\rm Byte}$

Table: decimal and binary prefixes



The two standard prefixes in decimal and binary representations of memory sizes are given in Table 7. The decimal prefixes are also used for displaying numbers of floating point operations per second (flops) executed by a certain machine.

name	(location)	cores	LINPACK perfomance
			[TFlop/s]
Summit	(USA)	2 397 824	143 500.0
Sunway TaihuLight(China)		10 649 600	93 014.6
Sequoia	(USA)	1 572 864	16 324.8
Tianhe-2A	(China)	4 981 760	61 444.5

Table: Petascale systems available





Figure: Performance development of TOP500 HPC machines taken from TOP500 poster November 2014



ARCHITECTURES





CHIP TECHNOLOGY



Figure: Chip technologies of TOP500 HPC machines taken from TOP500 poster November 2014



INSTALLATION TYPE



Figure: Installation types of TOP500 HPC machines taken from TOP500 poster November 2014



ACCELERATORS/CO-PROCESSORS



Figure: Accelerators and Co-Processors employed in TOP500 HPC machines taken from TOP500 poster November 2014

Jens Saak



Distributed Memory Systems: Part II



Message passing

is the programming model commonly used for distributed memory systems, where each node has its own exclusive memory and we have an overall distributed address space. Exchange of data between the local memories of separate hosts is realized by sending messages between the hosts.

Usually, the communication is (network) socket based, although the basic principles can also be applied to multicore machines, e.g., by using shared memory blocks to implement the communication.



Communication operations in the Message Passing Interface (MPI) are belonging to two global classes categorized by their local (process on host) behavior.

Definition (blocking operation)

A communication operation is called blocking if the return of the process control to the calling process means that the operation has completed the entire transfer.

Definition (non-blocking operation)

In a non-blocking operation the process control is returned to the calling process as soon as the communication has been initiated. The communication may be ongoing while the calling process continues its program.



Communication Operations via Message Passing: Synchronous vs. Asynchronous

Looking at the same operations from a global perspective, i.e., not looking at the local message but the global communication, they determine the two classes of

Definition (synchronous communication)

The synchronous communication between a sending and a receiving process is implemented such that sending operations do not complete (i.e. return control to the calling process) before the receiving counterpart has at least started the execution.

Definition (asynchronous communication)

In asynchronous communication the sending and receiving process are not coordinated, i.e., the sender can execute its operation without the receiving counterpart waiting in its operation.



Communication Operations via Message Passing: Synchronous vs. Asynchronous

Example

- oral or telephone chats are synchronous communications, since all partners are engaged in the communication simultaneously.
- classic mail or electronic mail are asynchronous communication, where the sender never knows if, or when the message was actually received.



Communication between MPI processes can not only be classified via their influence on global or local process flow, but also with respect to the number of partners involved. MPI is distinguishing between

- point-to-point communication, where both ends are occupied by a single process, and
- collective communication where a single process sends out messages to multiple receiving processes, or collects messages from several sending processes.



Communication Operations via Message Passing: Point-to-Point Communication



Figure: Point-to-Point Communication



Communication Operations via Message Passing: Collective Communication



Figure: Broadcast Operation



Communication Operations via Message Passing: Collective Communication



Figure: Reduction Operation


Communication of Data

Communication Operations via Message Passing: Collective Communication



Figure: Scatter Operation



Communication of Data

Communication Operations via Message Passing: Collective Communication



Figure: Gather Operation



Assumptions

- All network links are bidirectional
- All-Port-Communication: each node can send out messages on all outgoing links simultaneously
- The same holds for receiving messages
- A messages consists of several bytes sent uninterruptedly



Assumptions

• The time for transmission of a message of *m* bytes size is

 $T(m)=t_s+mt_b,$

where t_s is a startup time and t_b is the time for sending a single byte.

The communication is such that the length of the path from source node to destination node in the corresponding network graph determines the number of time steps required.



Landau ⊖-notation

The $\Theta(g(x))$ notation describes a class of functions f for which roughly speaking we have that "f is growing essentially as fast as g." More precisely we have,

 $\Theta(g(x)) = \{f(x) \mid \exists c_1, c_2 > 0 \text{ and } x_0, \text{ such that} \\ \forall x \ge x_0 \quad c_1 |g(x)| \le f(x) \le c_2 |g(x)|\}$

This basically means $f \in \Theta(g)$ when $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(f)$.



Critical operations are the collective communication operations, since they produce a notable load on the entire range of links in the network. We will investigate the following in more detail:

- 1. broadcast
- 2. scatter
- 3. multi-broadcast (each node broadcasts)
- 4. total exchange (each node scatters)

In the following p specifies the number of nodes in the network.



Asymptotic Message runtimes in some Standard Network Topologies: Complete Graph



Figure: A complete graph network broadcast example

- all nodes connected, i.e., path length is one,
- by the assumptions all messages in all types of point to point and collective communication operations can be sent simultaneously,
- the operations can be performed in $\Theta(1)$.



Asymptotic Message runtimes in some Standard Network Topologies: Linear Array



Figure: A linear array network example

Single Broadcast

- The root node sends messages to its left and right neighbors starting with the most distant recipients,
- in all other steps each node forwards the message received from one neighbor in the previous step to its other neighbor.



Asymptotic Message runtimes in some Standard Network Topologies: Linear Array



Figure: A linear array network example

Single Broadcast

- The root node sends messages to its left and right neighbors starting with the most distant recipients,
- in all other steps each node forwards the message received from one neighbor in the previous step to its other neighbor.
- The minimal runtime is $\lfloor \frac{p}{2} \rfloor$ (root is the center node)
- The maximal runtime is p 1 (root is an end node)



Asymptotic Message runtimes in some Standard Network Topologies: Linear Array



Figure: A linear array network example

Single Broadcast

- The root node sends messages to its left and right neighbors starting with the most distant recipients,
- in all other steps each node forwards the message received from one neighbor in the previous step to its other neighbor.
- The minimal runtime is $\lfloor \frac{p}{2} \rfloor$ (root is the center node)
- The maximal runtime is p-1 (root is an end node)
- Thus the runtime class is $\Theta(p)$.



Asymptotic Message runtimes in some Standard Network Topologies: Linear Array

Multi Broadcast



Figure: A linear array network multi broadcast example



Asymptotic Message runtimes in some Standard Network Topologies: Linear Array

Multi Broadcast



Figure: A linear array network multi broadcast example



Asymptotic Message runtimes in some Standard Network Topologies: Linear Array

Multi Broadcast



Figure: A linear array network multi broadcast example



Asymptotic Message runtimes in some Standard Network Topologies: Linear Array

Scatter

The basic idea is that of the single broadcast, only the contents of the messages need to be treated more carefully. Therefore, the complexity is $\Theta(p)$ as well.

Total exchange

An upper bound to the runtime is given by p scatter operations, resulting in basically p^2 communication steps. In their book Rauber and Rünger present an algorithm that can do it in $\frac{p^2}{4}$. Anyway the complexity is $\Theta(p^2)$.



Communication Networks (revisited) Asymptotic Message runtimes in some Standard Network Topologies: Ring



Figure: A ring network example

The ring is a prototype for the linear array where the root node is always in the center. Thus, we get the same complexities as in the best case for the linear array. Note, however, that we need to cut the transmission at half way around the ring.



We consider the *d*-dimensional mesh with $\sqrt[d]{p}$ nodes per direction, such that we have *p* nodes in total as before. The diameter then is $d(\sqrt[d]{p}-1)$.







Figure: Two mesh network examples with diameter indications.



Single Broadcast

The single broadcast time is obviously proportional to the diameter of the network. This itself is proportional to the number of nodes in each direction. Therefore, the complexity class is $\Theta(\sqrt[d]{p})$.



Communication Networks (revisited) Asymptotic Message runtimes in some Standard Network Topologies: Mesh



Figure: The linear array embedded in a 2d mesh.

Scatter

- The picture shows clearly that the communication time is limited by that for the linear array from above.
- On the other hand, each node has d to 2d outgoing connections and p-1 messages need to be sent, i.e., $\lfloor \frac{p-d}{d} \rfloor$ is a lower limit.
- a scatter is possible in $\Theta(p)$.



Multi Broadcast

The multi broadcast time is observed similarly to be part of the complexity class $\Theta(\sqrt[d]{p})$.

Total Exchange

Rauber and Rünger in their book show a method that provides $\Theta(p^{\frac{d+1}{d}})$.



Distributed Memory Systems: Part III









Figure: The hypercube network in 4d



We denote the nodes in the *d*-dimensional hypercube by *d*-tuples of bits, i.e., we use $n_1, \ldots, n_p \in \{0, 1\}^d$. Let $a, b, c \in \{0, 1\}^d$ and a_i, b_i, c_i the *i*-th bit positions. We denote by \oplus the bitwise exclusive or operation, i.e.,

$$a_1 \ldots a_d \oplus b_1 \ldots b_d = c_1 \ldots c_d$$

with

$$c_i = \left\{ egin{array}{cl} 1 & ext{where } a_i
eq b_i, & ext{for } 1 \leq i \leq d. \ 0 & ext{otherwise} \end{array}
ight.$$



We denote the nodes in the *d*-dimensional hypercube by *d*-tuples of bits, i.e., we use $n_1, \ldots, n_p \in \{0, 1\}^d$. Let $a, b, c \in \{0, 1\}^d$ and a_i, b_i, c_i the *i*-th bit positions. We denote by \oplus the bitwise exclusive or operation, i.e.,

 $a_1 \ldots a_d \oplus b_1 \ldots b_d = c_1 \ldots c_d$

with

$$c_i = \left\{ egin{array}{cl} 1 & ext{where } a_i
eq b_i, & ext{for } 1 \leq i \leq d. \ 0 & ext{otherwise} \end{array}
ight.$$

Note that $\forall z \in \{0,1\}^d$ we have

$$00\ldots 0\oplus z=z,$$

and if $v, w \in \{0, 1\}^d$ differ in only a single bit, so do $v \oplus z$ and $w \oplus z$.



nodes are bit *d*-tuples,



- nodes are bit *d*-tuples,
- each node has d links to other nodes



- nodes are bit *d*-tuples,
- each node has d links to other nodes
- neighbors differ in a single bit position



- nodes are bit d-tuples,
- each node has d links to other nodes
- neighbors differ in a single bit position
- the diameter of the graph (i.e., the length of the longest path between two nodes) is d = log(p).



Communication Networks (revisited) Construction of Spanning Trees for Single Broadcasts

Definition (Spanning tree)

- A spanning tree of a graph is a tree that
 - picks one node of the graph as its root,
 - contains all other nodes as nodes or leaves once,
 - has only edges that represent valid links in the graph.



Construction Rules for root $00\dots 0$

1. all root connections coincide with the links in the graph.



- 1. all root connections coincide with the links in the graph.
- 2. children are generated by inverting a single bit right of the rightmost 1.



- 1. all root connections coincide with the links in the graph.
- 2. children are generated by inverting a single bit right of the rightmost 1.

The rules above imply



- 1. all root connections coincide with the links in the graph.
- 2. children are generated by inverting a single bit right of the rightmost 1.

The rules above imply

■ that all leave nodes end on a 1 bit,



- 1. all root connections coincide with the links in the graph.
- 2. children are generated by inverting a single bit right of the rightmost 1.

The rules above imply

- that all leave nodes end on a 1 bit,
- the depth of the tree is *d* + 1 since *d* bits are inverted on the path to the deepest leave 11...1.



Root nodes other than $00 \dots 0$

Spanning trees for other root nodes v are derived by replacing all nodes w by $w \oplus v$ in the entire tree for root 00...0.



Root nodes other than $00 \dots 0$

Spanning trees for other root nodes v are derived by replacing all nodes w by $w \oplus v$ in the entire tree for root 00...0.

Why is this the case? We noted above the properties of \oplus that

- \blacksquare 00 . . . 0 is the neutral element, and
- v, w differ in only a single bit $\Rightarrow v \oplus z$, $w \oplus z$ do so as well.

Thus, if (v, w) is a hypercube link, then $(v \oplus z, w \oplus z)$ is one as well.


Root nodes other than $00 \dots 0$

Spanning trees for other root nodes v are derived by replacing all nodes w by $w \oplus v$ in the entire tree for root 00...0.

Why is this the case? We noted above the properties of \oplus that

- \blacksquare 00 . . . 0 is the neutral element, and
- v, w differ in only a single bit $\Rightarrow v \oplus z$, $w \oplus z$ do so as well.

Thus, if (v, w) is a hypercube link, then $(v \oplus z, w \oplus z)$ is one as well.

Single Broadcast

The single broadcast can be implemented in $\Theta(\log p) = \Theta(d)$ successively descending through the spanning tree. It can also not be better than that since the diameter of the hypercube is d.



Scatter

A scatter operation needs to send out p-1 different messages along the *d* links of the root node. It can thus not be faster than $\lceil \frac{p-1}{d} \rceil$ time steps.

We will see in the following that this is the time also needed for a multi-broadcast. Since a single scatter can not be slower than that we immediately have that a scatter is $\Theta(\frac{p-1}{\log(p)}) = \Theta(\frac{p-1}{d})$.



Problem

The single broadcast spanning trees for the 2^d nodes in the *d*-dimensional hypercube are not disjoint in the sense that each link is only used by a single operation in each time step if the multi-broadcast is treated as 2^d isolated single broadcasts.

Observation

It is mandatory to construct spanning trees such that all sets of edges used in a single time step by the different single broadcasts are disjoint.



Definition

- The spanning tree for root node $t \in \{0,1\}^d$ is called T_t , and simply T_0 for t = 00...0.
- The set of edges active in time step *i* for T_t is called $A_i(t)$

Construction

The sets of active edges for root node $t \in \{0,1\}^d$ may be constructed such that for any two edges (x, y) and (x', y') in $A_i(0) x, y$ and x', y' do not differ in the same bit position and the sets for the other root nodes are derived as

$$A_i(t) = \{(x \oplus t, y \oplus t) \mid (x, y) \in A_i\} \qquad \forall 1 \le i \le m,$$

where m is the total number of time steps required.



Observation

The set A_i of active edges in the *i*-th step can have at most *d* entries, since we only have *d* bit positions available in the node labels.

Main Idea:

Construct the sets A_i such that $|A_i| = d$ for $1 \le i < m$ and $|A_m| \le d$.

What is *m*?

Since each of the $p = 2^d$ nodes in the tree has an incoming link, except the root, we have $2^d - 1$ edges in total that are distributed among the A_i , i.e.,

$$\left|\bigcup_{i=1}^m A_i\right| = 2^d - 1.$$



What is *m*?

Since each of the $p = 2^d$ nodes in the tree has an incoming link, except the root, we have $2^d - 1$ edges in total that are distributed among the A_i , i.e.,

$$\left|\bigcup_{i=1}^m A_i\right| = 2^d - 1.$$



What is m?

Since each of the $p = 2^d$ nodes in the tree has an incoming link, except the root, we have $2^d - 1$ edges in total that are distributed among the A_i , i.e.,

$$\left|\bigcup_{i=1}^m A_i\right| = 2^d - 1.$$

This immediately provides a first estimate for m:

$$m = \left\lceil \frac{2^d - 1}{d} \right\rceil$$



What is *m*?

Since each of the $p = 2^d$ nodes in the tree has an incoming link, except the root, we have $2^d - 1$ edges in total that are distributed among the A_i , i.e.,

$$\left|\bigcup_{i=1}^m A_i\right| = 2^d - 1.$$

This immediately provides a first estimate for m:

$$m = \left\lceil \frac{2^d - 1}{d} \right\rceil$$

Note that we can also not get better than that, since each node in the hypercube has to receive $2^d - 1$ messages from the other nodes across its *d* incoming links.



Definition

We collect some further notation:

•
$$N_k := \{t \in \{0,1\}^d \mid t \text{ has } k \text{ unit bits and } d-k \text{ zero bits.}\}$$

These sets have

$$n_k := |N_k| = \begin{pmatrix} d \\ k \end{pmatrix} = \frac{d!}{k!(d-k)!}$$

elements.

- The N_k are further subdivided into m_k equivalence classes R_{k1}, \ldots, R_{km_k} with respect to left rotation. They are ordered by rightmost concentration of the unit bits, i.e., R_{k1} is the class containing $(0^{d-k}1^k)$.
- The elements in the equivalence classes can be ordered by rightmost concentration of unit bits as well.
- n(t) is the global number of node t in this order.

• $m(t) = 1 + [n(t) - 1 \mod d]$ is t's local number of inside the equivalence class.



Let us denote the sets of destination nodes in A_i by E_i . Then we set:

$$\begin{split} & E_0 = \{00 \dots 0\} \\ & E_i = \{t \in \{0,1\}^d \mid (i-1)d + 1 \le n(t) \le id\} \\ & E_m = \{t \in \{0,1\}^d \mid (m-1)d + 1 \le n(t) \le 2^d - 1\} \end{split}$$

The set of active edges are then constructed by the rules: 1. connect $t \in E_i$ to start node t' with the m(t)th bit inverted, 2. if t = 11...1 and m(t) = d connect to t' = 101...1 instead.



By construction in each step the tree uses d edges and all sets $A_i(t)$ for the different t are disjoint. Thus, all 2^d single broadcasts can be performed simultaneously and the multi-broadcast can be done in $\Theta(\frac{p-1}{d})$.

Note that although the *d*-hypercube has only $\frac{d}{2} \cdot 2^d$ edges we can use $d \cdot 2^d$ links in the graph due to the assumption of bidirectional communication.



Distributed Memory Systems: Part IV



The Message Passing Interface is a standard for creation of parallel programs using the message passing programming model. It describes

- functionality,
- behavior,
- API syntax

of the required routines. It does, however, not prescribe any implementation details. It is, e.g., completely open by what means a message is transferred.



The MPI uses a specialized execution environment that spawns and administrates the instances of a process. Relevant functions for

- setup and destruction of the working environments context
- grouping processes
- actual message transmission

...

are collected in the mpi.h header file. We will see later for the case of the Open MPI^{26} implementation of the standard how we can compile and run a program using the MPI features.

²⁶http://www.open-mpi.org/



The most basic components of the MPI program are

#include <mpi.h>

to make the standard available. Then before we can use any message passing routines we need to initialize the execution context via

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

passing on the usual arguments of the main() function of our C program. After we have finished our MPI related work the execution context is destroyed using

int MPI_Finalize()

Processes may continue performing local work after the finalization, but with a very few exceptions none of the MPI function work anymore. It is mandatory to make sure all MPI operations have finished before calling MPI_Finalize().



Definition (Process group)

Processes in MPI may be clustered in so called process groups. These are ordered sets of instances of the program numbered from 0 to n - 1. The local numbers of the processes are called rank.

From the programmers view an MPI group is an object of type MPI_Group, which can be accessed via a handle. There exists one predefined group constant MPI_GROUP_EMPTY, denoting the empty group.

MPI process groups are useful to implement task parallel applications. MPI supports communication inside a group and point to point type communication between groups.



Generates the union of two existing groups by including all elements of the first group, followed by all elements of second group that are not in the first group.

- group1, group2 groups to include
- *newgroup handle of the group to create. This may be equal to the empty group MPI_GROUP_EMPTY.

The operation is not commutative but associative.





Produces a group at the intersection of two existing groups by including all elements of the first group that are also in the second group, ordered as in the first group.

- group1, group2 groups to intersect,
- *newgroup handle of the group to create. This may be equal to the empty group MPI_GROUP_EMPTY.

The operation is not commutative but associative.





Generates the new group from the difference of the existing groups by including all elements of the first group that are not in the second group, ordered as in the first group.

- group1, group2 groups to determine the difference from
- *newgroup handle of the group to create. This may be equal to the empty group MPI_GROUP_EMPTY.



Create a new group from an existing group by including a possibly reordered subset of the processes.

- group the existing group
- n number of ranks used in the new group
- ranks ordered list of members for the new group
- *newgroup handle of the group to create.



Create a new group from an existing group by excluding a possibly reordered subset of the processes.

- group the existing group
- n number of ranks used in the new group
- ranks ordered list of members to exclude from the new group
- *newgroup handle of the group to create.



int MPI_Group_size(MPI_Group group, int *size)

Determines the number of members of a group, returned in size.

```
int MPI_Group_rank(MPI_Group group, int *rank)
```

Find the rank (local number) of the current process in group.

Find out how different group1 and group2 are. The result is MPI_IDENT if they are the same, MPI_SIMILAR in case they only differ in the order of the processes and MPI_UNEQUAL otherwise.

Unused groups can be released by calling

int MPI_Group_free(MPI_Group *group)

On successful return group is set to MPI_GROUP_NULL



Definition (Communicators)

The participants in a communication operation in MPI are usually determined via so called communicators. MPI distinguishes two types of communicators

- intra-communicators for the collective communication inside a process group
- inter-communicators for the point-to-point like communication between two process groups.

If we are following the SPMD programming model and do not want to have task-parallelism in our code, we are usually fine with the predefined default communicator MPI_COMM_WORLD. When people simply speak of a communicator they usually refer to an intra-communicator. Communicators are objects of type MPI_Comm



int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

Create a new communicator for a subset of the processes.

- comm base communicator
- group process group the new communicator will be associated with. Must be a subgroup of the group associated to comm.
- *newcomm handle to the newly created communicator.



int MPI_Comm_size (MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)

are the communicator equivalents of the equally called group functions. For comm equal to MPI_COMM_WORLD the total number of processes and the global ranks are returned. Otherwise those of the associated group are given.

For the MPI_Comm_compare function the value MPI_IDENT here means that the underlying groups are in fact the same. MPI_CONGRUENT is returned if the groups are equal (including the order of the ranks) but not the same one group. If only the order differs the result is MPI_SIMILAR again and MPI_UNEQUAL otherwise.



Perform a blocking send operation.

- buf address of the sendbuffer
- count number of elements to send
- datatype type of send buffer elements
- dest the rank of the destination process inside comm
- tag a message identifier
- comm the communicator to use for the transmission



Performs a standard-mode blocking receive.

- buf address of the send buffer
- count number of elements to send
- datatype type of send buffer elements
- source the rank of the sending process inside comm
- tag a message identifier
- comm the communicator to use for the transmission
- status a status object containing information about the sender, the message tag, and possible errors. Also the length of the message received can be retrieved from it using the MPI_Get_count function. This can be set to the constant MPI_STATUS_IGNORE to save resources if not needed by the application.



Variants of these functions performing the send and receive in a single call or that are non-blocking, exist, for the details see the standard and the man pages of MPI_Sendrcv(), MPI_Isend(), MPI_Irecv().

For the non-blocking communication operations the function $MPI_Test()$ can be used to check whether a certain message has been transferred.



int MPI_Barrier(MPI_Comm comm)

Actually not performing a real communication this function makes sure that process flow stops until all processes in the group associated to comm have reached this point.

comm the communicator to use the barrier for



Broadcasts a message from one process to all other processes of the communicator.

- *buffer address of the send/receive buffer
- count number of elements to send
- datatype type of send buffer elements
- root the rank of the sending process
- comm the communicator to be use



int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Reduces values on all processes within a group associated to a communicator

- *sendbuf address of the send buffer
- *recvbuf address of the receive buffer (only relevant on root)
- count number of elements to send
- datatype type of buffer elements
- op the arithmetic operation to use in the reduce
- root the rank of the root/receiving process
- comm the communicator to be use



Distributes data from one process among all processes in the communicator

- *sendbuf address of the send buffer
- sendcount number of elements to send
- sendtype type of the send buffer elements
- *recvbuf address of the receive buffer
- recvcount number of elements to receive
- recvtype type of the receive buffer elements
- root the rank of the root/sending process
- comm the communicator to be use



Collects data from all processes on a single process.

- *sendbuf address of the send buffer
- sendcount number of elements to send
- sendtype type of the send buffer elements
- *recvbuf address of the receive buffer
- recvcount number of elements to receive
- recvtype type of the receive buffer elements
- root the rank of the root/receiving process
- comm the communicator to be use



Collects and redistributes data from all processes to all processes.

- *sendbuf address of the send buffer
- sendcount number of elements to send
- sendtype type of the send buffer elements
- *recvbuf address of the receive buffer
- recvcount number of elements to receive
- recvtype type of the receive buffer elements
- comm the communicator to be use



```
int MPI_Allreduce(void *sendbuf,
            void *recvbuf,
            int count,
            MPI_Datatype datatype,
            MPI_Op op,
            MPI_Comm comm)
```

Similar to the MPI_Reduce() function it combines values from all processes, but in addition it distributes the result back to all processes.

- *sendbuf address of the send buffer
- *recvbuf address of the receive buffer
- count number of elements to send
- datatype type of buffer elements
- op the arithmetic operation to use in the reduce
- comm the communicator to be use



The total exchange operation, i.e., every process sends to all other processes.

- *sendbuf address of the send buffer
- sendcount number of elements to send
- sendtype type of the send buffer elements
- *recvbuf address of the receive buffer
- recvcount number of elements to receive
- recvtype type of the receive buffer elements
- comm the communicator to be use


The obligatory "hello world!" program does no more than initializing the MPI context, printing the obligatory text from all instances and destroying the context again:

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char** argv){
    /* start MPI context*/
    MPI_Init(&argc, &argv);
    /*Do something*/
    printf("Hello_world\n");
    /* Stop MPI context*/
    MPI_Finalize();
    return 0;
}
```



In Open MPI^{27} a C wrapper compiler called <code>mpicc</code> is provided. Its sole purpose is to transparently

- add relevant compiler and linker flags to the user's compiler command line
- and then call the underlying compiler to perform the actual compilation.

Especially, we do not need to care where exactly the necessary MPI libraries are located and which additional flags are required. If we have specified additional parameters (e.g. for code optimization, or debugging), mpice passes them on to the underlying compiler.

E	xample
Т	hus, to compile the "hello world" code, we simply use:
r	mpicc hello_world.c -o hello_world -02

²⁷http://www.open-mpi.org/



The drawback of the MPI framework is that processes need to be started within a special runtime environment. In the case of Open MPI this is invoked using the mpirun tool:

>]

The tool takes a couple of options that allow to steer the number of processes spawned, including where they are spawned, control their working environment (path, working directory, environment variables, ...) and the redirection of standard input and output and many details more.



The most important options of mpirun for beginners are:

- -n <#> run this many copies, if unset Open MPI spawns one copy per processor (aliases are -c, --n, -np).
 - -H List of hosts (comma separate) to spawn the processes on (aliases -host, --host)
- -hostfile Provide a hostfile to use instead of the list above. (aliases and synonyms --hostfile, -machinefile, --machinefile)

Example

To run 1 copy of hello_world (from the local directory) each on the two hosts alpha, beta we may use

mpirun -np 2 -H alpha, beta ./hello_world



Distributed Memory Systems: Part V



For a 2d data field (like a matrix) there are basically 3 types of data distribution patterns:

- row/column blocks,
- row/column cyclic,
- checkerboard.

All of them have their advantages and disadvantages in different algorithms. We will treat them all in the case of the LU decomposition in the following.



Algorithm 7: Gaussian elimination — row-by-row-version

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition Output: A overwritten by L, U1 for k = 1 : n - 1 do 2 A(k + 1 : n, k) = A(k + 1 : n, b)/A(k, k);3 for i = k + 1 : n do 4 for j = k + 1 : n do 5 A(i, j) = A(i, j) - A(i, k)A(k, j);



Before diving into the details of data distribution we recall that after 4 steps of the row-by-row LU decomposition we have the following:

	<i>u</i> _{1,1}	<i>u</i> _{1,2}	<i>u</i> _{1,3}	<i>u</i> _{1,4}	<i>u</i> _{1,5}	<i>u</i> _{1,6}	<i>u</i> _{1,7}	u _{1,8}	U 1,9	<i>u</i> _{1,10}
	<i>I</i> _{2,1}	<i>u</i> _{2,2}	<i>u</i> _{2,3}	<i>u</i> _{2,4}	u _{2,5}	<i>u</i> _{2,6}	<i>u</i> _{2,7}	u _{2,8}	<i>u</i> 2,9	<i>u</i> _{2,10}
	<i>I</i> _{3,1}	I _{3,2}	<i>Ц</i> 3,3	<i>u</i> 3,4	<i>u</i> 3,5	<i>u</i> _{3,6}	<i>u</i> 3,7	U 3,8	<i>u</i> 3,9	<i>u</i> _{3,10}
	<i>I</i> 4,1	$I_{4,2}$	<i>I</i> 4,3	<i>u</i> 4,4	<i>u</i> 4,5	<i>u</i> 4,6	<i>U</i> 4,7	U 4,8	<i>U</i> 4,9	<i>u</i> 4,10
A ⁽⁴⁾ _	<i>I</i> _{5,1}	$I_{5,2}$	l _{5,3}	<i>I</i> 5,4	$a_{5,5}$	a 5,6	a _{5,7}	a 5,8	a 5,9	a _{5,10}
A	<i>I</i> _{6,1}	$I_{6,2}$	<i>I</i> _{6,3}	<i>I</i> _{6,4}	a 6,5	a 6,6	a _{6,7}	a _{6,8}	a 6,9	a _{6,10}
	<i>I</i> _{7,1}	I _{7,2}	I _{7,3}	I _{7,4}	a _{7,5}	a _{7,6}	a _{7,7}	a _{7,8}	a 7,9	a _{7,10}
	<i>I</i> _{8,1}	$I_{8,2}$	I _{8,3}	<i>I</i> 8,4	a 8,5	a 8,6	a _{8,7}	a 8,8	a 8,9	a _{8,10}
	<i>I</i> 9,1	<i>I</i> 9,2	<i>I</i> 9,3	<i>I</i> 9,4	a 9,5	a 9,6	a 9,7	a 9,8	a 9,9	a 9,10
	L/10,1	$I_{10,2}$	$I_{10,3}$	$I_{10,4}$	$a_{10,5}$	a _{10,6}	a _{10,7}	a _{10,8}	a _{10,9}	a10,10

Furthermore the blue and green parts will no longer be touched and the algorithm proceeds on the smaller lower right part A(5:10,5:10) only.



Basic Idea:

A

Group the rows/columns in blocks of $\lceil \frac{n}{p} \rceil$. Each processor then works on one of those blocks, performing all necessary operations that treat any rows/columns in the scope.

	F									
	<i>u</i> _{1,1}	<i>u</i> _{1,2}	<i>u</i> _{1,3}	<i>u</i> _{1,4}	^{<i>u</i>1,5} F	¹ ^{<i>u</i>1,6}	<i>u</i> _{1,7}	<i>u</i> _{1,8}	<i>u</i> _{1,9}	<i>u</i> _{1,10}
	<i>l</i> _{2,1}	u _{2,2}	И 2,3	<i>u</i> _{2,4}	<i>u</i> _{2,5}	u _{2,6}	И _{2,7}	И 2,8	U 2,9	u _{2,10}
	<i>I</i> _{3,1}	I _{3,2}	<i>и</i> _{3,3}	<i>u</i> _{3,4}	<i>u</i> _{3,5}	u _{3,6}	<i>u</i> _{3,7}	<i>и</i> _{3,8}	<i>и</i> _{3,9}	<i>u</i> _{3,10}
	<i>I</i> _{4,1}	I _{4,2}	I _{4,3}	<i>U</i> 4,4	<i>u</i> 4,5	- <i>u</i> 4,6	<i>u</i> 4,7	<i>u</i> 4,8	U 4,9	<i>u</i> 4,10
⁽⁴⁾ =	<i>I</i> _{5,1}	I _{5,2}	I _{5,3}	<i>I</i> _{5,4}	a _{5,5}	a _{5,6}	a _{5,7}	a 5,8	a 5,9	a _{5,10}
	<i>I</i> _{6,1}	<i>I</i> _{6,2}	<i>I</i> _{6,3}	<i>I</i> _{6,4}	a 6,5	a _{6,6}	a _{6,7}	a _{6,8}	a 6,9	a _{6,10}
	<i>I</i> _{7,1}	I _{7,2}	I _{7,3}	I _{7,4}	a _{7,5}	a _{7,6}	a _{7,7}	a _{7,8}	a _{7,9}	a _{7,10}
	<i>I</i> _{8,1}	I _{8,2}	1 _{8,3}	I _{8,4}	a 8,5	a 8,6	a _{8,7}	a 8,8	a 8,9	a _{8,10}
	1 9,1	<i>I</i> 9,2	<i>I</i> 9,3	<i>I</i> 9,4	<i>a</i> 9,5	5 ^{<i>a</i>9,6}	a 9,7	a 9,8	a 9,9	a _{9,10}
	/ _{10,1}	$I_{10,2}$	I _{10,3}	$I_{10,4}$	a _{10,5}	a _{10,6}	a _{10,7}	a _{10,8}	a _{10,9}	a _{10,10}



Processors P_1 and P_2 have no more work do do after step 4. \rightsquigarrow bad load balancing among the processors.

As a consequence we should not use the block distribution in cases when not the entire matrix is involved in all computations to make sure that all processors are equally well loaded. That means for parallel matrix-vector or matrix-matrix products it may serve well, but for the LU we need to find a data distribution that has a better distribution of the workload.

CSC Cyclic-row/-column Distribution

Basic Idea:

Instead of distributing blocks of rows/columns assign a single row/column to a process until all got one and then start over until all rows/columns are distributed.

	<i>u</i> _{1,1}	<i>u</i> _{1,2}	<i>u</i> _{1,3}	<i>u</i> _{1,4}	$u_{1,5} \mathbf{P_1}^{u_{1,6}}$	<i>u</i> _{1,7}	<i>u</i> _{1,8}	<i>u</i> 1,9	<i>u</i> _{1,10}
	<i>I</i> _{2,1}	и _{2,2}	и 2,3	<i>u</i> _{2,4}	$u_{2,5} P_2^{u_{2,6}}$	<i>u</i> _{2,7}	и 2,8	и 2,9	<i>u</i> _{2,10}
	<i>I</i> _{3,1}	<i>I</i> _{3,2}	и 3,3	<i>u</i> _{3,4}	^{U3,5} P3 ^{U3,6}	и 3,7	и 3,8	И 3,9	<i>u</i> _{3,10}
	<i>I</i> 4,1	<i>I</i> _{4,2}	<i>I</i> 4,3	<i>u</i> 4,4	<i>u</i> _{4,5} P ₄ <i>u</i> _{4,6}	<i>u</i> 4,7	u 4,8	U 4,9	<i>u</i> 4,10
⊿ ⁽⁴⁾ _	<i>I</i> _{5,1}	<i>I</i> _{5,2}	<i>I</i> _{5,3}	<i>I</i> _{5,4}	a _{5,5} P ₅ a _{5,6}	a _{5,7}	a _{5,8}	a _{5,9}	a _{5,10}
// _	<i>I</i> _{6,1}	<i>I</i> _{6,2}	<i>I</i> _{6,3}	<i>I</i> _{6,4}	a _{6,5} P1 a _{6,6}	a _{6,7}	a 6,8	a 6,9	a _{6,10}
	<i>I</i> _{7,1}	<i>I</i> _{7,2}	I _{7,3}	I _{7,4}	$a_{7,5} P_2^{a_{7,6}}$	a _{7,7}	a _{7,8}	a _{7,9}	a _{7,10}
	<i>I</i> _{8,1}	<i>I</i> _{8,2}	<i>I</i> 8,3	<i>I</i> _{8,4}	a _{8,5} P3 a _{8,6}	a _{8,7}	a 8,8	a 8,9	a _{8,10}
	<i>l</i> 9,1	<i>I</i> 9,2	<i>I</i> 9,3	<i>I</i> 9,4	^{a_{9,5}} P ₄ ^{a_{9,6}}	a 9,7	a 9,8	a 9,9	a _{9,10}
	[<i>I</i> _{10,1}	<i>I</i> _{10,2}	<i>I</i> _{10,3}	<i>I</i> _{10,4}	$a_{10,5} \mathbf{P_5}^{a_{10,6}}$	a _{10,7}	a _{10,8}	a _{10,9}	a _{10,10}

Sc Cyclic-row/-column Distribution

Obviously now the processors only start to become idle after n - p steps of the outermost loop, i.e. in $A^{(n-p)}$, which is reasonable for $p \ll n$. Still basically every processor is responsible for $\lceil \frac{n}{p} \rceil$ rows.

Sc Cyclic-row/-column Distribution

Obviously now the processors only start to become idle after n - p steps of the outermost loop, i.e. in $A^{(n-p)}$, which is reasonable for $p \ll n$. Still basically every processor is responsible for $\lceil \frac{n}{p} \rceil$ rows.

Pivoting

Since pivoting adds a considerable amount of extra communication effort, we do not neglect it here in contrast to earlier appearances. However, we restrict ourselves to the case of column pivoting. That means as the first step of the outer for loop we add the pivot selection and row swapping.



Algorithm 7: Gaussian elimination — row-by-row-version

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition Output: A overwritten by L, U1 for k = 1 : n - 1 do 2 $k_0 = \operatorname{argmax}_{i=k:n} |A(i, k)|;$ 3 Swap rows k and $k_0;$ 4 A(k+1:n,k) = A(k+1:n,b)/A(k,k);5 for i = k + 1 : n do 6 for j = k + 1 : n do 7 A(i,j) = A(i,j) - A(i,k)A(k,j);



1. Determination of the pivot element.

The column below the diagonal is owned by several processors in the distributed parallel case. That means each processor finds its local pivot element and afterward they are compared among all processors.



1. Determination of the pivot element.

The column below the diagonal is owned by several processors in the distributed parallel case. That means each processor finds its local pivot element and afterward they are compared among all processors.

2. Usage of the pivot element.

If we are lucky enough that the pivot row is owned by the same processor that owns the row containing the critical diagonal element, we are fine. We can perform a local row swap as in the sequential case. Otherwise the pivot row is exchanged with the process owning the "diagonal row".



1. Determination of the pivot element.

The column below the diagonal is owned by several processors in the distributed parallel case. That means each processor finds its local pivot element and afterward they are compared among all processors.

2. Usage of the pivot element.

If we are lucky enough that the pivot row is owned by the same processor that owns the row containing the critical diagonal element, we are fine. We can perform a local row swap as in the sequential case. Otherwise the pivot row is exchanged with the process owning the "diagonal row".

3. Distribution of the pivot row.

The pivot row is the key ingredient to the computation in the step. It is needed by all processors and thus needs to be broadcast to all active processors.



1. Determination of the pivot element.

The column below the diagonal is owned by several processors in the distributed parallel case. That means each processor finds its local pivot element and afterward they are compared among all processors.

2. Usage of the pivot element.

If we are lucky enough that the pivot row is owned by the same processor that owns the row containing the critical diagonal element, we are fine. We can perform a local row swap as in the sequential case. Otherwise the pivot row is exchanged with the process owning the "diagonal row".

3. Distribution of the pivot row.

The pivot row is the key ingredient to the computation in the step. It is needed by all processors and thus needs to be broadcast to all active processors.

4. Computation of the matrix element updates.

The update step can now be performed as in the sequential case. Only, each processor just works through the local rows it owns.



Basic Idea:

Distribution of the *d*-dimensional data array to a *d*-dimensional processor grid. Note that we can follow the blocked or cyclic variants just as in the case above.

$a_{1,1}$	$a_{1,2}$	a _{1,3}	$a_{1,4}$	a _{1,5}	$a_{1,6}$	$a_{1,7}$	a _{1,8}
P ₁₁		P	12	P	13	Р	14
$a_{2,1}$	a _{2,2}	a _{2,3}	a _{2,4}	a 2,5	a 2,6	a _{2,7}	a _{2,8}
$a_{3,1}$	a _{3,2}	a _{3,3}	a 3,4	a _{3,5}	a _{3,6}	a _{3,7}	a _{3,8}
P	21	P ₂₂		P ₂₃		P ₂₄	
$a_{4,1}$	a _{4,2}	a _{4,3}	$a_{4,4}$	a _{4,5}	$a_{4,6}$	a _{4,7}	a _{4,8}
$a_{5,1}$	a 5,2	a 5,3	a 5,4	a 5,5	a 5,6	a 5,7	a 5,8
P	31	P ₃₂		P	33	P ₃₄	
$a_{6,1}$	<i>a</i> _{6,2}	a _{6,3}	<i>a</i> _{6,4}	a _{6,5}	a _{6,6}	a _{6,7}	a _{6,8}
$a_{7,1}$	a 7,2	a 7,3	a 7,4	a 7,5	a 7,6	a _{7,7}	a _{7,8}
P ₄₁		P	42	P	43	Р	44
a _{8 1}	a 8.2	as 3	a 8 4	a _{8 5}	a _{8.6}	a _{8.7}	a _{8.8}

$a_{1,1}$	$a_{1,5}$	a _{1,2}	a _{1,6}	a _{1,3}	$a_{1,7}$	$a_{1,4}$	$a_{1,8}$
P ₁₁		P ₁₂		P ₁₃		P_{14}	
$a_{5,1}$	a 5,5	a 5,2	a 5,6	a 5,3	a 5,7	a 5,4	a 5,8
$a_{2,1}$	$a_{2,5}$	a _{2,2}	a _{2,6}	a _{2,3}	a _{2,7}	a _{2,4}	a _{2,8}
P ₂₁		P ₂₂		P ₂₃		P ₂₄	
$a_{6,1}$	$a_{6,5}$	a 6,2	a _{6,6}	a _{6,3}	a _{6,7}	$a_{6,4}$	a _{6,8}
a 3,1	a 3,5	a 3,2	a 3,6	a 3,3	a 3,7	a 3,4	a 3,8
а _{3,1} Р	а _{3,5} 31	а _{3,2} Р	а _{3,6} 32	^а з,з Р	а _{3,7} 33	а _{3,4} Р	а _{3,8} 34
а _{3,1} Р а _{7,1}	а _{3,5} 31 а _{7,5}	а _{3,2} Р а _{7,2}	а _{3,6} 32 а _{7,6}	а _{3,3} Р а _{7,3}	а _{3,7} 33 а _{7,7}	а _{3,4} Р а _{7,4}	а _{3,8} 34 а _{7,8}
a _{3,1} P a _{7,1} a _{4,1}	a _{3,5} 31 a _{7,5} a _{4,5}	a _{3,2} P a _{7,2} a _{4,2}	a _{3,6} 32 a _{7,6} a _{4,6}	а _{3,3} Р а _{7,3} а _{4,3}	a _{3,7} 33 a _{7,7} a _{4,7}	a _{3,4} P a _{7,4} a _{4,4}	a _{3,8} 34 a _{7,8} a _{4,8}
a _{3,1} P a _{7,1} a _{4,1} P	a _{3,5} 31 a _{7,5} a _{4,5} 41	a _{3,2} P a _{7,2} a _{4,2} P	a _{3,6} 32 a _{7,6} a _{4,6} 42	a _{3,3} P a _{7,3} a _{4,3} P	a _{3,7} 33 a _{7,7} a _{4,7} 43	a _{3,4} P a _{7,4} a _{4,4} P	a _{3,8} 34 a _{7,8} a _{4,8} 44

(a) blocked distribution

(b) cyclic distribution



Definition

Let $n = \prod_{i=1}^{d} n_i$ be the total problem size and n_i the degrees of freedom in the *i*-th direction. Also p, as before, the number of processors in total. We call $\mathbf{p} = (p_1, \ldots, p_d)$ a processor distribution if it holds

$$p \leq \prod_{i=1}^d p_i$$

On each processor we assume a local data distribution $\mathbf{b} = (b_1, \dots, b_d)$ with

$$n\leq\prod_{i=1}^d p_ib_i.$$

Ideally we want to have equality in both cases to achieve optimal load balancing.



The PBLAS project (details later) aims at providing a parallel distributed version of the BLAS library. In the previous Chapters we have investigated level 3 BLAS based block outer product versions of the LU decomposition.



domain decomposition

Similar to the splitting of the matrix into blocks on which smaller subproblems are solved, in domain decomposition²⁸ methods for boundary value problems the objective domain on which the problem is to be solved is subdivided into smaller parts. Then on each part a smaller independent boundary value problem is solved. The interaction between subdomains is only necessary if their intersection is non empty, i.e., they have a common "boundary", the interface. In each iteration step both processes rely on the result of the prior step and exchange the data on the interface to make it fit in a post-processing procedure.

- The interface is sometimes also called halo.
- The interface may be a single layer of unknowns, but can also be extended. One then speaks of overlapping domain decomposition methods.

²⁸http://www.ddm.org



Implementations of the MPI Standard

- Open MPI, Current bugfix release 3.1.4 implements MPI-3.1²⁹
- MPICH 3.3 (release of November 21, 2018) supports MPI-3.1³⁰
- MVAPICH: The current MVAPICH2 2.3.1 is based on MPICH v3.2.1³¹
- Intel[®] MPI Library: version 2019 update 4 implements MPI-3.1³²

²⁹http://www.openmpi.org ³⁰http://www.mpich.org/ ³¹http://mvapich.cse.ohio-state.edu/ ³²http://software.intel.com/en-us/intel-mpi-library



Scientific Software

- BLACS (Basic Linear Algebra Communication Subprograms) "is an ongoing investigation whose purpose is to create a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms." ³³
- ScaLAPACK a BLACS-based scalable distributed implementation of LAPACK (current version 2.0.2 of May 1, 2012)³⁴
- PBLAS (Parallel Basic Linear Algebra Subprograms) subproject of the above³⁵
- Boost starting with version 1.35 has a boost.MPI module providing a C++ friendly MPI framework. (current version 1.70.0 April 17, 2019)³⁶

35 http://www.netlib.org/scalapack/pblas_qref.html

³⁶http://www.boost.org/

³³http://www.netlib.org/blacs/

³⁴http://www.netlib.org/scalapack/



Scientific Software

- PETSC "is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations."³³
- SLEPC is the Scalable Library for Eigenvalue Problem Computations.³⁴
- PARPACK an extension to the ARPACK for eigenvalue computations using MPI and BLACS for parallel execution.³⁵

³³http://www.mcs.anl.gov/petsc/ ³⁴http://www.grycap.upv.es/slepc/ ³⁵http://www.caam.rice.edu/software/ARPACK/