

PARALLEL COMPUTING AND ALGORITHMS

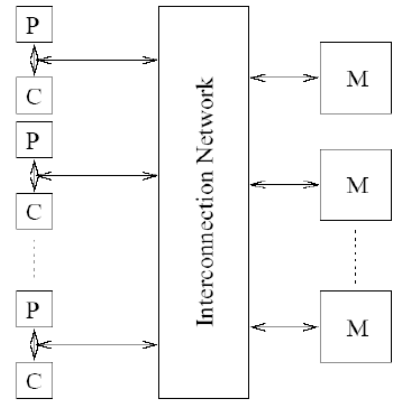
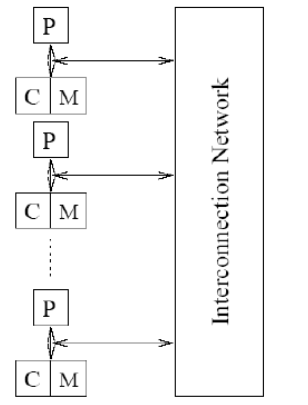
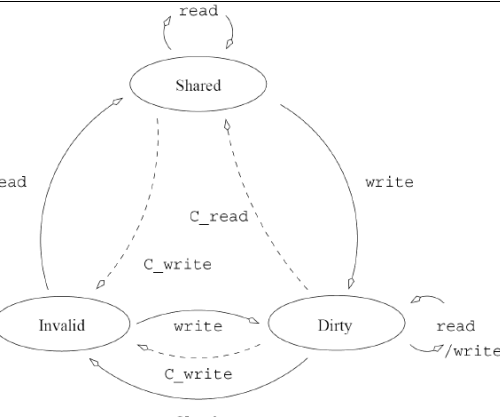
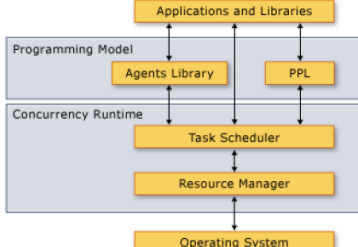
Motivation			
Concurrent computing	a form of computing in which programs are designed as collections of interacting computational processes <ul style="list-style-type: none"> sequentially on a single processor by interleaving the execution steps of each computational process in parallel by assigning each computational process to one of a set of processors that may be close or distributed across a network challenges: ensure the correct sequencing, coordinate access to shared resources tools: Threads, Mutex, Semaphores		
Cloud computing	Cloud computing is a type of Internet-based computing that provides shared computer processing resources and data to computers and other devices on demand.		
Parallel computing	a form of computation in which many calculations are carried out simultaneously tools: no Mutex nor Semaphores (we could, but we don't want to)		
	is the dominant paradigm in computer architecture nowadays, since power consumption became a concern challenges: more difficult to write, higher fault tolerance, larger amount of memory needed Pro: usually faster computation (n computers are not n times faster), Con: communication, synchronization		
	levels: bit-level: all bits of a word are computed in parallel instructions level: several instructions are computed parallel data parallelism: the same operations is computed on several data in parallel task parallelism: several tasks work together in parallel	examples: weather forecast DNA structures astronom. model	
FLOPS Floating operations /sec	MFLOP = Mega 10 ⁶	GFLOP = Giga 10 ⁹ -> normal computer	TFLOP=Tera 10 ¹² PFLOP=Peta 10 ¹⁵ -> good parallel computing
HPC	High Performance Computing		
hint	first use the right algorithms, than start programming parallel		
Architectures of parallel infrastructures			
Implicit Parallelism	processors have multiple functional units and execute multiple instructions in the same cycle by pipelining, superscalar execution, very long instruction word processors		
Explicite Parallelism	must specify concurrency and interaction between concurrent subtasks -> this is what we want try to minimize concurrency and synchronization		
Programming Models	Process based models multiple process private data, shared memory (synch)	<pre> graph TD Task[Task] --> Process[Process] Task --> Threads[Threads] </pre>	Lightweight processes and Threads one process with multiple threads all data is global (faster synch)
PRAM Parallel Random Access Machine	<ul style="list-style-type: none"> extension of the RAM multiple processors share clock, but exec different instruction global memory of unbounded size 	Handling memory access <ul style="list-style-type: none"> EREW (Exclusive-read, exclusive-write) CREW (Concurrent-read, exclusive-write) -> good ERCW (Exclusive-read, concurrent-write) CRCW (Concurrent-read, concurrent-write) -> very good 	
Concurrent write	Common: write only if all values are identical Arbitrary: write the data from a randomly selected processor Priority: follow a predetermined priority order Sum: Write the sum of all data items	(5,5) → 5 (5,7) → 5 or 7 (5 high priority, 7 low priority) → 5 (5,7) → 12	
Chunking Granularity	determining the amount of data to assign to each task (chunk or grain size) the size of a chunk, depends on algorithm and used hardware fine-grained parallelism: low arithmetic intensity, more communication, better splitting / load balance coarse-grained parallelism: high arithm. intensity, less communication, difficult to load balance efficiently		
Control Structure	SIMD (Single Instruction Multiple Data) , called SSE on Intel, e.g. vector operation a single control unit dispatches the same instruction to various processors that work on different data	MIMD (Multiple Instruction Multiple Data) simple MIMD: SPMD = Single Program Multiple Data each processor has its own control unit each processor can execute different instructions on different data items	
	Pro / Con	less HW, less memory needs regular structure, selectively turn off opera.	
	implementations	high performance workstation at low cost use existing software, processors can be added GPU (Graphic Processing Unit) DSP (Digital Signal Processors)	
Communication Models	Shared-Address-Space Platforms (Multiprocessors) part of the memory is accessible to all processors processors interact by modifying data objects	Message Passing Platforms (Multicomputers) own exclusive memory using sending messages	

Interconnection Network for HPC	Infiniband (very high throughput and very low latency, scalable, direct or switched interconnection) PCI Express 4 (high-speed serial bus standard, external cabling over Thunderbolt) NVIDIA NVLink (high-bandwidth, energy-efficient)		
Switching Hub		forwarding per MAC, non blocking spanning tree protocol: shortest path bridging	store-and-forward buffers until complete, error checking before forwarding cut-through forward immediately, buffer when port is busy, no error checking
	evaluation	<ul style="list-style-type: none"> • Diameter -> the distance between the farthest two nodes • Channel Bandwidth -> number of bits that can be communicated simultaneously over a link • Cross-Section Bandwidth -> the min number of wires one must cut to divide into two equal parts • Cost -> number of links/switches, length of wires 	
Message passing costs	t_s Startup time: spent at sending and receiving nodes t_h Per-hop time: number of hops (includes switch latencies, delays) t_w Per-word transfer time: includes overheads from message length m number of messages		cut-through cost: $t_{comm} = t_s + l * t_h + t_w * m$ $t_h \ll t_w \rightarrow$ viel kleiner als $\rightarrow t_{comm} = t_s + t_w * m$

Network Topologies

	Bus	Star	Crossbar	Multistage Netw	k-d Mesh	Hypercube	Tree-based
p inputs b outputs	<p>simple common bus good cost scalable</p>	<p>common node good perform scalable</p>	<p>$p * b$ switches</p>	<p>bus/crossbar mix</p>	<p>Linear Array 1D-Torus/Ring Mesh</p>	<p>3-d-hypercube $d = \log p$</p>	
diameter	1	2	1	$\log p$	1D: $p - 1$ 2D: $2(\sqrt{p} - 1)$	$\log p$	$2 \log \frac{p + 1}{2}$
links	$\frac{p(p - 1)}{2}$	$p - 1$	p^2		1D: $p - 1$ 2D: $2(p - \sqrt{p})$	$\frac{p \log p}{2}$	$p - 1$
bottle-neck	bus bandwidth limited nodes	central node	cost grows p^2 difficult to scale			con: different length	root
Examples	WLAN zone PCI bus	LANs with Bridge or Switch	non-block switch L1<->L2 - caches	Omega-Network		hypercube	Fat-Tree

Shared Memory Systems (SMS)

<p>Platforms</p> <p>P = Processor C = Cache M = Memory</p>	<p>Uniform-memory-access (UMA)</p>  <p>local cache (fast) L1 u. L2 (Level 1)</p> <p>global/shared memory (access time to M are identical)</p> <p>Memory Controller (MCH) = Northbridge Interface Controller (ICH) = Southbridge</p> <p>e.g. Intel Front Side Bus Architecture</p>	<p>Non-Uniform-memory-access (NUMA)</p>  <p>local cache</p> <p>local memory (access also other M's)</p> <p>e.g. Intel Core i7 (Nehalem)</p>																								
<p>Caching</p> <p>Scenario</p> <p>s=shared d=dirty i=invalid w=write r=read f=flush</p>	<p>Faster memory access, Load complete block of memory and hope next access is in this block</p> <p>Cache coherence: ensure that cache is consistent to each other</p> <p>Update and Invalidate Protocol</p> <p>step 1 write-back: set an invalidate flag on other copies</p> <p>step 2 write-through: update other copies</p> <p>Example</p> <table border="1" data-bbox="320 842 919 999"> <thead> <tr> <th></th> <th>Start</th> <th>$P_0 w$</th> <th>$P_1 w$</th> <th>$P_2 r$</th> <th>$P_0 w$</th> </tr> </thead> <tbody> <tr> <td>P_0</td> <td>1(s)</td> <td>$\xrightarrow{w} 3(d)$</td> <td>$\rightarrow 3(i)$</td> <td>$\rightarrow 3(i)$</td> <td>$\xrightarrow{w} 5(d)$</td> </tr> <tr> <td>P_1</td> <td>1(s)</td> <td>$\rightarrow 1(i)$</td> <td>$\xrightarrow{w} 4(d)$</td> <td>$\rightarrow 4(s)$</td> <td>$\rightarrow 4(i)$</td> </tr> <tr> <td>P_2</td> <td>1(s)</td> <td>$\rightarrow 1(i)$</td> <td>$\rightarrow 1(i)$</td> <td>$\xrightarrow{r} f: 4(s)$</td> <td>$\rightarrow 4(i)$</td> </tr> </tbody> </table>		Start	$P_0 w$	$P_1 w$	$P_2 r$	$P_0 w$	P_0	1(s)	$\xrightarrow{w} 3(d)$	$\rightarrow 3(i)$	$\rightarrow 3(i)$	$\xrightarrow{w} 5(d)$	P_1	1(s)	$\rightarrow 1(i)$	$\xrightarrow{w} 4(d)$	$\rightarrow 4(s)$	$\rightarrow 4(i)$	P_2	1(s)	$\rightarrow 1(i)$	$\rightarrow 1(i)$	$\xrightarrow{r} f: 4(s)$	$\rightarrow 4(i)$	
	Start	$P_0 w$	$P_1 w$	$P_2 r$	$P_0 w$																					
P_0	1(s)	$\xrightarrow{w} 3(d)$	$\rightarrow 3(i)$	$\rightarrow 3(i)$	$\xrightarrow{w} 5(d)$																					
P_1	1(s)	$\rightarrow 1(i)$	$\xrightarrow{w} 4(d)$	$\rightarrow 4(s)$	$\rightarrow 4(i)$																					
P_2	1(s)	$\rightarrow 1(i)$	$\rightarrow 1(i)$	$\xrightarrow{r} f: 4(s)$	$\rightarrow 4(i)$																					
<p>VC++ Concurrency Runtime</p>	<p>Why: uniformity and predictability to applications that run simultaneously</p> <p>Pro: cooperative task scheduling (work-stealing algorithm), cooperative blocking</p> <p>Architecture: PPL (Parallel Patterns Library) - fine grained, Asynchronous Agents Library - coarse grained</p> <p>concurrency::parallel_for_each</p>																									
<p>vs</p>	<p>OpenMP (included in VC++, VS2015 supports 2.0) for parallel algorithms that are iterative efficient when degree of parallelism pre-determined and matches the available resources on the system for high-performance computing</p>	<p>Concurrency Runtime for less constrained computing environments dynamic scheduler that adapts to available resources and adjust degree of parallelism as workloads change easy for recursive problems</p>																								

Parallel programming with C++ (std:::)

thread	A thread is a single stream of control flow of a program. low-level, data exchange must be synchronized, is started automatically in constructor (in C++) uncaught exceptions in thread -> termination of entire program, static/global variables for each thread	
Executable obj	a) function object (functor) b) lambda expression c) pointer to a function	obj is copied as the arguments
e.g functor	<pre>#include <thread> void execObj(std::string text) { std::cout << text << std::endl; }</pre>	<pre>std::thread t1(execObj, "param"); t1.get_id(); // unique thread id t1.join(); // returns when finish</pre>
e.g. lambda e.g. Matrix multiplication	<pre>auto task = [param1,&param2,] { ... } for (row = 0; row < n; row++) // no synch needed for (column = 0; column < n; column++) c[row][column] = create_thread(dot_product(get_row(a, row), get_col(b, col)));</pre>	<pre>for (i=0; i<nThreads; i++) {thread(task)};</pre>
mutex mutual exclusion	a lockable object that is designed to signal when critical sections of code need exclusive access, preventing other threads with the same protection from executing concurrently and access the same memory locations.	
lock	a. isn't locked: lock mutex b. locked by other thread: wait until unlocked c. locked by this thread: deadlock, undefined behaviour	<pre>mutex mtx; mtx.lock();</pre>
unlock	releasing ownership over it	<pre>mtx.unlock();</pre>
condition_variable	Block the calling thread until notified to resume. It uses a unique_lock (over a mutex) to lock the thread	
declare		<pre>condition_variable readingAllowed;</pre>
wait	blocks until notified	<pre>unique_lock<mutex> lock(mtx); readingAllowed.wait(lock);</pre>
notify	wake up a blocking thread	<pre>readingAllowed.notify_one(); // or notify_all()</pre>
async and future	async: initiates a computation and returns (two modes: launch::async, launch::deferred) future: return type of async; get() -> blocks until the result is available	
pro	exception does not end in a crash, can be caught in .get()	
e.g.	<pre>auto fut1 = async(launch::async, &funct1, 35); // asynch, started immediately auto fut2 = async(launch::deferred, &funct1, 35); // deferred, started with get cout << fut2.get() << endl; // waiting cout << fut1.get() << endl; // waiting</pre>	
packaged_tasks	a packaged task wraps a callable element and allows its result to be retrieved asynchronously not started automatically, contains a stored task (e.g. function) and a shared state (e.g. int)	
e.g.	<pre>// create task for calling fibrec, argument of fibrec has to be defined later packaged_task<size_t(size_t)> task1(&fibrec); auto fut1 = task1.get_future(); // future for getting result // create task for calling fibrec, argument of fibrec is bound to 35 packaged_task<size_t(void)> task2(bind(&fibrec, 35)); auto fut2 = task2.get_future(); // future for getting result // call task1 in a parallel thread (move semantic) thread th(move(task1), 35); // hint for the compiler to 'move' instead of '=' task2(); // call task2 in this tread cout << fut1.get() << endl; // get result of task1 cout << fut2.get() << endl; // get result of task2 th.join(); // this tread waits on parallel thread th</pre>	
Synchronization primitives	atomic_xyz	all accesses are atomic (are not interrupted)
	atomic_flag	atomic bool but lock-free
	once_flag	used in call_once, makes sure that only one parallel threads executes the function
	recursive_mutex	allows a thread computing a recursive function to reenter a critical section
	lock_guard	locks a critical section; very simple usage; the only state is locked
	unique_lock	needs its unique mutex object; handles both states: locked and unlocked
assign operator	<pre>C& operator = (const C&c) { x=c.x; return *this; }</pre>	
move operator	<pre>C& operator = (C && c) { x=c.x; c.x=0; return *this; }</pre>	
serial vs. parallel for loop	serial for loop , no parallel algorithms in C++11/14 sequentially ordered steps e.g. reading n integers from a sequential file	parallel for loop , (since C++17) any order e.g. same task for each element of an array
Support	C++17 standards: most algorithms have overloads that accept execution policies (seq/par/par_unseq)	

OpenMP

OpenMP	A standard for directive based parallel programming , for FORTRAN and C++ support for concurrency, synchronization, data handling --> mutexes, condition variables, data scope, init	
Programming Model	directives are based on the #pragma compiler directives (e.g. #pragma omp directive [clause list]) execute serially until the parallel directive, which creates a group of threads (#pragma omp parallel []) the main thread that encounter the parallel directive becomes the master of this group of threads (id=0)	
Clause List	Conditional Parallelization - if -> check if threads need to be created, evaluated at runtime Degree of Concurrency - num_threads(integer expr) -> number of threads Data Handling - private (variable list) -> variables are local to each thread T firstprivate (variable list) -> local variable, but initialized before the parallel directive shared (variable list) -> variables are shared across all threads threadprivate (variable list) -> variable is private to a thread	
	Default Clause - allows to affect the data-sharing attribute of variables default(shared) -> each currently visible variable is shared (unless threadprivate or const) default(none) -> shared if (explicitly listed within construct threadprivate or const for loop)	
	Reduction Clause - specifies how a variable is combined into a single copy after the master exits reduction(operator: variable list) - operators: +,*,-,&, ,^,&&,	
e.g.	<pre>int main() { // serial segment const int npoints = 1000000; int sum = 0; srand(clock()); #pragma omp parallel default(none) reduction(+:sum) num_threads(8) { // omp -> for (i=0; i < 8; i++) pthread_create(..,internal_thread_name,..); // parallel segment #pragma omp for for (int i = 0; i < npoints; i++) { double rand_x = rand()/double(RAND_MAX); if (((rand_x-0.5)*(rand_x-0.5)) < 0.125) sum ++; } } // serial segment cout << setprecision(10) << 4*sum/double/npoints) << endl; }</pre>	
concurrent tasks	#pragma omp for [clause list] { }	parallel iterations on threads; clauses: private, firstprivate, lastprivate, reduction, schedule (static, dynamic, guided, runtime), nowait (no implicit barrier at loop end), ordered
	#pragma omp sections [clause list]{ #pragma omp section /* structured block 1 */ #pragma omp section /* structured block 2*/ }	non-iterative parallel task assignment code sections be divided among threads
	#pragma omp parallel shared(n) { }	execute code in parallel, creates a group of threads
synchronization	#pragma omp barrier	synchronize all threads in a team, wait until all
	#pragma omp single [clause list]	executed on a single thread (not necessarily on master)
	#pragma omp master	only master thread should execute a section
	#pragma omp critical [(name)]	this code is only executed on one thread at a time
	#pragma omp atomic	memory location will be automatically updated
	#pragma omp ordered	for loop should be executed like a sequential loop
	#pragma omp flush [(variable list)]	all threads have the same view for shared objects
merge directive	#pragma omp parallel for shared(n) { }	
nest directives	<pre>#pragma omp parallel for shared(a,b,c) num_threads(4) { for(int i = 0; i < 128; i++) #pragma omp parallel for shared(a,b,c) num_threads(4) {...} }</pre> it is not allowed to bind to the same parallel directive for 'for', 'section', and 'single' per default, each inner 'for' directive generates a logical team which is still executed by the same thread otherwise set OMP_NESTED to TRUE	

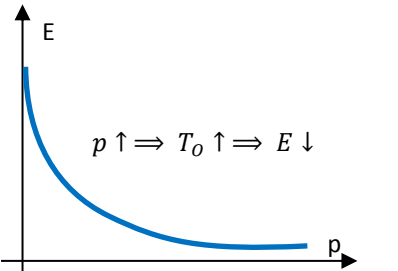
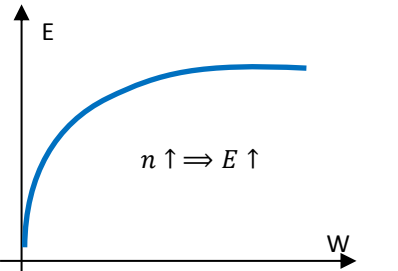
Performance Metrics for Parallel Systems

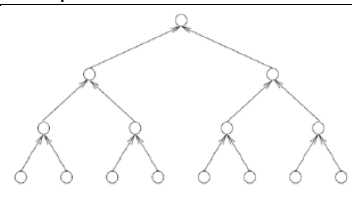
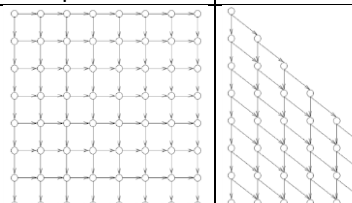
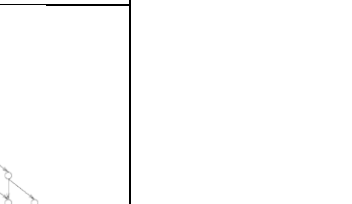
Analytical Modeling	Sequential Runtime: Evaluated by its runtime, identical on any serial platform	Parallel Runtime depends on: input size n, number of processors p, communication param
----------------------------	--	--

	Explanation	Formula	Example: adding
n	input size		n numbers
p	number of processors		on p processors
T_s	Serial runtime: time elapsed on a sequential computer	$W = \Theta(T_s)$	$T_s = \Theta(n)$
T_p	Parallel runtime: time elapsed from the start of first processor to the end of the last processor	$T_p = \frac{W + T_o(W, p)}{p}$	$T_p = \frac{n}{p} + 2 \log p$
T_o	Parallel Overhead: total time of all processors combined when non-useful	$T_o = p * T_p - T_s$ $= T_s * f * (p - 1)$	$T_o = p \log p$
S	Speedup: Ratio of the serial runtime of the best serial algorithm to the parallel algorithm lower bound: 0; upper bound: should be by p; superlinear due to caching and exploratory decomposition	$S = \frac{T_s}{T_p} = \frac{W}{T_p} = \frac{p W}{W + T_o(W, p)}$ 	$S = \frac{n}{\frac{n}{p} + 2 \log p}$
E	Efficiency: Speedup per processor	$E = \frac{S}{p} = \frac{T_s}{p * T_p} = \frac{1}{1 + \frac{T_o}{T_s}}$	$E = \frac{1}{1 + \frac{2p \log p}{n}}$
Cost	Cost: amount of total work	$Cost = p * T_p \geq W$	
Cost opt	a parallel system is cost optimal if cost of solving problem on a parallel computer is asymptotically identical to serial cost	$Cost = \Theta(W)$ $\rightarrow E = \Theta(1)$	$n = W$ $= \Omega(p \log p)$
K	efficiency coefficient	$K = \frac{E}{1 - E}$	
iso-E	What is the rate at which the problem size W must increase to p to keep the efficiency fixed . This rate determines the scalability of a system. The slower/smaller the better. (high scalable)	$W = f(p)$ $\rightarrow W = K * T_o(W, p)$ $W = \Omega(p)$	$W = K p \log p$ $f(p) = \Theta(p \log p)$

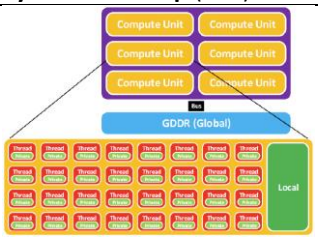
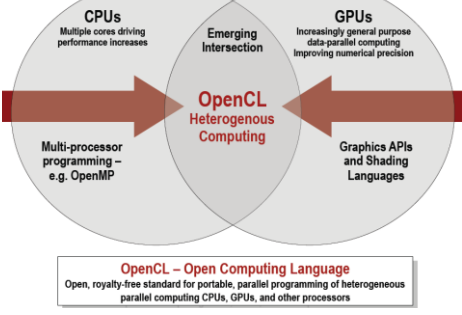
Scalability of Parallel Systems

<p>Amdahl's Law</p> <p>fixed workload improve runtime</p>	$W = \underbrace{f * W}_{W_{seq}} + \underbrace{(1 - f) * W}_{W_{par}} = \Theta(T_s)$ $f = \frac{W_{seq}}{W} \in [0, .1]$ $T_p = W_{seq} + \frac{W_{par}}{p} = f * W + \frac{(1 - f) * W}{p}$ $S = \frac{T_s}{T_p} = \frac{W}{f * W + \frac{(1 - f) * W}{p}} = \frac{p}{f(p - 1) + 1}$ $\lim_{p \rightarrow \infty} S = \frac{1}{f}$	<p>W: total work W_{seq}: sequential/serial work (non-parallelizable) W_{par}: parallel work f: fraction of sequential work</p> <p>Der sequentielle Anteil (f) begrenzt den Speedup bei vielen Prozessoren.</p>
<p>Gustafson's Law</p> <p>fixed runtime increase work & p</p>	$T_p = T_{seq} + T_{par}$ $\sigma = \frac{T_{seq}}{T_p} \in [0, 1]$ $T_p = \sigma * T_p + (1 - \sigma) * T_p$ $S' = \frac{T_p(p * W, 1)}{T_p(p * W, p)} = p - (p - 1) * \sigma$	<p>T_{seq}: time for the sequential work T_{par}: time for the parallel work S': scaled speedup</p> <p>based on data parallelism</p>
<p>Karp-Flatt</p> <p>includes overhead</p>	$T_p(W, p) = T_{seq}(W) + T_{over}(W, p) + \frac{T_{par}(W)}{p}$ $S = \frac{T_p(W, 1)}{T_p(W, p)} = \frac{1}{e + \frac{1 - e}{p}} \rightarrow e = \frac{\frac{1}{s} - \frac{1}{p}}{1 - \frac{1}{p}}$	<p>T_{over}: overhead time for each processor e: karp-flatt metric -> from relative speedup e ist kompatibel zu f</p>

Big-O-Notation	$g(x) = O(f(x))$ oder $g(x) \in O(f(x))$ $g(x) = \Omega(x)$ $g(x) = \Theta(a(x)) = O(a(x)) \wedge \Omega(a(x))$	$O =$ Obergrenze $\Omega =$ Untergrenze $\Theta =$ Obergrenze und Untergrenze $x_0 =$ Startwert $c =$ Konstante $\forall_{x \geq x_0}: g(x) \leq c f(x) $
Bsp	$g(x) = 7x^2 + 5$ $f(x) = x^2$ $c = 8, ab \ x_0 = ?$	$O(\text{mergesort}) = n * \log n$ $\Omega(\text{any sort}) = n * \log n$ $\Theta(\text{mergesort}) = n * \log n$
General	Es ist schwierig über kleine Instanzgrößen, das Wachstum in grossen Instanzgrößen vorauszusagen. Für verschiedene Instanzgrößen, werden oft verschiedene Algorithmen angewendet.	
scalable parallel system	T_0 grows sublinearly with W $W \uparrow \Rightarrow E \uparrow$ $W \uparrow$ and $p \uparrow \Rightarrow E$ const	a scalable parallel system can always be made cost-optimal if the number of processing elements and the size of the computation are chosen appropriately
Efficiency	Fixed problem size W 	Fixed number of processing elements p 
Degree of Concurrency	maximum number of operations (tasks) that can be executed simultaneously at any time in a parallel algorithm.	e.g. Gaussian elimination $W = \Theta(n^3)$ $C(W) = \Theta(n^2) = \Theta(W^{\frac{2}{3}}) < \Theta(W)$ $W = \Omega(p^{\frac{3}{2}})$ at least, to use them all
Minimum Execution Times	$\frac{d}{dp} T_p = 0$, let p_0 be the value of p as determined by this equation $T_p^{min} = \begin{cases} T_p(p_0) = \frac{W + T_0(W, p)}{p}, & \text{if } p_0 \leq C(W) \\ T_p(C(W)) = \frac{W + T_0(W, p)}{C(W)}, & \text{else} \end{cases}$	e.g. adding n numbers $\frac{d}{dp} T_p = -\frac{n}{p^2} + \frac{2}{p} = 0$ $T_p^{min} = 2 \log n$
Minimum Cost-optimal execution time	$T_p^{cost_opt} = \Omega\left(W \frac{W}{f^{-1}(W)}\right)$	$p_0 = \frac{n}{2}$, $Cost = p_0 T_p^{min}$ not cost-optimal

DAG (directed, acyclic graph)	Example 1	Example 2	
			
Nodes	$N = 15$	$N = 64$	$N = 36$
Height	$n = \log_k(N + 1) = 4$	$n = \sqrt{64} = 8$	$n = \frac{\sqrt{1 + 8N} - 1}{2} = 8$
Degree of Concurrency	$C(W) = 2^{n-1} = 8$	$C(W) = n = 8$	$C(W) = n = 8$
max. speedup $p = \infty$	$S = \frac{T_s}{T_p} = \frac{N}{n} = \frac{2^n - 1}{n}$	$S = \frac{n^2}{2n - 1}$	$S = \frac{N}{n} = \frac{n + 1}{2}$
$p = \frac{C(W)}{2}$	$S = \frac{N}{\frac{n+1}{2}} = \frac{2^n - 1}{n + 1}$ $E = \frac{S}{p} = \frac{\frac{2^n - 1}{n + 1}}{\frac{n + 1}{2}} = \frac{2^n - 1}{2^{n-2}(n + 1)}$		

Heterogeneous Shared Memory System (HSMS)

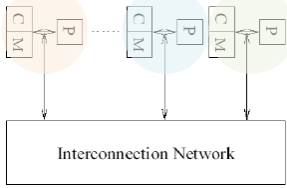
OpenCL	framework to run code on GPU
Performance Increase	<p>By Frequency (higher clock circuit rate): voltage reduction needed -> at its limit</p> <p>By number of cores: parallel coding needed</p> <p>By heterogeneity: handle different workload characteristics on different architectures</p>
Workloads	<p>control intensive (e.g. searching, sorting, parsing) -> best with CPUs</p> <p>data intensive (e.g. Image processing, simulation, modelling, data mining) -> best with GPUs</p> <p>compute intensive (e.g. iterative methods, numerical methods, financial modeling)</p>
HSA	Heterogeneous System Architect -> combine CPUs, GPUs, DSPs (Digital Signal Processing), FPGAs -> better performance and lower power consumption
Architecture	<p>SIMD (single instruction on multiple data) and Vector Processing (pipelining computation over long data)</p> <p>Hardware Multithreading (independent instruction streams (threads) are executed concurrently Simultaneous Multithreading (SMT) or Temporal Multithreading</p> <p>Hyperthreading = 4 physical cores results in 8 logical cores</p> <p>Multi-Core Architectures (both CPUs and GPUs)</p> <p>Systems-on-Chip (SoC) and the APU Accelerated Processing Unit (mix of CPU and GPU)</p>
GPU Architecture	 <p>GPUs tend to be heavily multithreaded, are design for process graphics</p> <p>Components</p> <p>Compute Unit (streaming multiprocessors, local memory: needs synch)</p> <p>Internal and external bus system</p> <p>Global memory: no synch</p>
Types	<p>C++ AMP (Accelerated Massive Parallelism) – open spec from Microsoft – based on DirectX 11, CPU fallback</p> <p>CUDA (Compute Unified Device Architecture) – from NVIDIA – supports C, C++ and Fortran, nvcc compiler</p> <p>OpenACC – from Cray/CAPS/NVIDIA/PGI – supports C, C++ and Fortran – pragma compiler directives</p> <p>OpenCL – from language C99 – 3 major code blocks – fastest – most complicated</p>
OpenCL	 <p>Block 1: Device program: kernels and subroutines operation executed by the work items (vector_add) C99 based syntax with vector operations</p> <p>Block 2: Host program: device and kernel preparation Platform and device handling creating contexts and command queues compiling OpenCL device programs</p> <p>Block 3: Host program: data and program enqueueing data allocation and management / filling in cmd queues / setting kernel arguments / running kernels / event handling</p>
Summary	portable and high-performance framework, for computationally intensive algorithms, use all ressources efficient parallel programming language, C99 with extensions for task and data parallelism, built-in functions defines hardware and numerical precision requirements open standard for heterogeneous parallel computing
e.g.	<pre> OCLData ocl = initOCL("./edges.cl", "edges"); processOCL(ocl, image, out2, hFilter, vFilter, fSize); // OpenCL kernel __kernel void edges(__read_only image2d_t source, __write_only image2d_t dest, __constant int* hFilter, __constant int* vFilter, int fSize, sampler_t sampler) { const int w = get_global_size(0); // number of global work items const int col = get_global_id(0); // global work item id ... uint4 pixel = read_imageui(source, sampler, coords); ... write_imageui(dest, coords, p); } </pre>

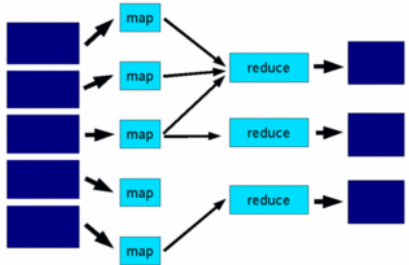
Decomposition and mapping techniques

<p>Parallel Algorithm Design</p>												
<p>Steps</p>	<ol style="list-style-type: none"> decompose in pieces of work (tasks) which can be performed concurrently fine-grained: large number of tasks -> better load balance coarse-grained: medium number of tasks map this tasks to processors (e.g. task 1,2 and 3 can run in parallel) usually small number of processes than tasks goal: reduction of communication overhead manage access to shared data process mapping (not important in our course) 											
<p>Granularity</p>	<p>number of tasks into which a problem is decomposed</p>											
<p>Degree of Concurrency</p>	<p>number of tasks that can be executed in parallel, may change during execution <i>maximal degree of concurrency = maximum number of tasks at any point during execution</i> <i>average degree of concurrency = $\frac{\text{total amount of work}}{\text{critical path length}}$</i></p>											
<p>Critical Path Length</p>	<p>the length of the longest path in a task dependency graph -> shortest time of execution</p>											
<p>Task interaction graph</p>	<p>the graph of tasks (nodes) and their interactions/data exchange (edges) is referred to as a task interaction graph</p>											
<p>Decomposition techniques</p>	<p>Recursive decomposition: divide-and-conquer strategy, with recursion (e.g. min of a list) Input/output/intermediate data partitioning: divide data in different part -> assign tasks to these partitions (e.g. matrix mult), each output can be computed as a function of the input Exploration Decomposition: create task during run-time (e.g. tile puzzle)</p>											
<p>Mapping</p>	<p>In general, the number of tasks is bigger than the processing elements -> mapping is needed Goal: minimize overhead (communication and idling) -> often contradicting</p> <table border="1" data-bbox="312 1608 1513 1870"> <tr> <td colspan="2" data-bbox="312 1608 911 1704"> <p>Static Mapping tasks are mapped to processes a-priori we should know the size of each task</p> </td> <td colspan="2" data-bbox="911 1608 1513 1704"> <p>Dynamic Mapping (Dynamic Load Balancing) tasks are mapped to processes at runtime</p> </td> </tr> <tr> <td data-bbox="312 1704 531 1870"> <p>data partitioning row/column/ block(grid)-wise cyclic, block-cyclic (e.g. LU Decomp)</p> </td> <td data-bbox="531 1704 798 1870"> <p>task graph partitioning opt.map = NPC binary tree (e.g. quicksort hyperc)</p> </td> <td data-bbox="798 1704 911 1870"> <p>hybrid</p> </td> <td data-bbox="911 1704 1235 1870"> <p>centralized (Master/Slave) master manages tasks slave execute tasks e.g. sort entries in each row of an nxn matrix</p> </td> <td data-bbox="1235 1704 1513 1870"> <p>distributed (Pipeline) process send/receive work from others, no bottleneck</p> </td> </tr> </table>			<p>Static Mapping tasks are mapped to processes a-priori we should know the size of each task</p>		<p>Dynamic Mapping (Dynamic Load Balancing) tasks are mapped to processes at runtime</p>		<p>data partitioning row/column/ block(grid)-wise cyclic, block-cyclic (e.g. LU Decomp)</p>	<p>task graph partitioning opt.map = NPC binary tree (e.g. quicksort hyperc)</p>	<p>hybrid</p>	<p>centralized (Master/Slave) master manages tasks slave execute tasks e.g. sort entries in each row of an nxn matrix</p>	<p>distributed (Pipeline) process send/receive work from others, no bottleneck</p>
<p>Static Mapping tasks are mapped to processes a-priori we should know the size of each task</p>		<p>Dynamic Mapping (Dynamic Load Balancing) tasks are mapped to processes at runtime</p>										
<p>data partitioning row/column/ block(grid)-wise cyclic, block-cyclic (e.g. LU Decomp)</p>	<p>task graph partitioning opt.map = NPC binary tree (e.g. quicksort hyperc)</p>	<p>hybrid</p>	<p>centralized (Master/Slave) master manages tasks slave execute tasks e.g. sort entries in each row of an nxn matrix</p>	<p>distributed (Pipeline) process send/receive work from others, no bottleneck</p>								
<p>Minimize Interaction Overhead</p>	<p>Maximize data locality -> reuse intermediate data Minimize volume of data exchange Minimize frequency of interactions</p> <p>Minimize contention and hot-spots Overlapping computations with interactions -> use non-blocking communication, multithreading</p>											

<p>LU Decomposition of linear equations</p> <p>A = square matrix L = lower triang. U = upper triang.</p>	<p style="text-align: center;">$A = LU$</p> $\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} * \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$ <p style="text-align: center;">$L_{1,1}, L_{2,2}, L_{3,3} = 1$</p> <div style="display: flex; justify-content: space-around;"> <div data-bbox="319 224 798 716"> </div> <div data-bbox="798 224 1133 716"> <p>given value overwritten value use computed value</p> </div> </div>	<p>Tasks:</p> <ol style="list-style-type: none"> 1. $A_{1,1} \rightarrow L_{1,1}U_{1,1}$ 2. $L_{2,1} = A_{2,1}U_{1,1}^{-1}$ 3. $L_{3,1} = A_{3,1}U_{1,1}^{-1}$ 4. $U_{1,2} = L_{1,1}^{-1}A_{1,2}$ 5. $U_{1,3} = L_{1,1}^{-1}A_{1,3}$ 6. $A_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$ 7. $A_{2,3} = A_{2,3} - L_{2,1}U_{1,3}$ 8. $A_{3,2} = A_{3,2} - L_{3,1}U_{1,2}$ 9. $A_{3,3} = A_{3,3} - L_{3,1}U_{1,3}$ 10. $A_{2,2} \rightarrow L_{2,2}U_{2,2}$ 11. $L_{3,2} = A_{3,2}U_{2,2}^{-1}$ 12. $U_{2,3} = L_{2,2}^{-1}A_{2,3}$ 13. $A_{3,3} = A_{3,3} - L_{3,2}U_{2,3}$ 14. $A_{3,3} \rightarrow L_{3,3}U_{3,3}$ 																																															
<p>Beispiel Diagonal Round Robin</p> <p style="text-align: center;">$p = 3$</p>	<p>Vorgehen</p> <table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>P_0</td><td>P_1</td><td>P_0</td></tr> <tr><td>P_2</td><td>P_1</td><td>P_0</td></tr> <tr><td>P_2</td><td>P_1</td><td>P_2</td></tr> </table>	P_0	P_1	P_0	P_2	P_1	P_0	P_2	P_1	P_2	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th></tr> </thead> <tbody> <tr><th>P_0</th><td>T_1</td><td>T_5</td><td>T_7</td><td>T_{12}</td><td></td><td></td><td></td><td></td></tr> <tr><th>P_1</th><td>T_4</td><td></td><td>T_6</td><td>T_8</td><td>T_{10}</td><td>T_{11}</td><td></td><td></td></tr> <tr><th>P_2</th><td></td><td>T_2</td><td>T_3</td><td>T_9</td><td></td><td></td><td>T_{13}</td><td>T_{14}</td></tr> </tbody> </table>		1	2	3	4	5	6	7	8	P_0	T_1	T_5	T_7	T_{12}					P_1	T_4		T_6	T_8	T_{10}	T_{11}			P_2		T_2	T_3	T_9			T_{13}	T_{14}	<p style="text-align: center;">$T_S = 14$ $T_P = 8$</p> $S = \frac{14}{8} = \frac{7}{4}$ $E = \frac{7}{4 * 3}$	<p style="text-align: center;">$Cost = T_P * p = 24$ $T_O = Cost - T_S = 10$ $T_P^{min} = 7$</p> $\Omega(p_0) = \frac{T_S}{T_P^{min}} = 2$
P_0	P_1	P_0																																															
P_2	P_1	P_0																																															
P_2	P_1	P_2																																															
	1	2	3	4	5	6	7	8																																									
P_0	T_1	T_5	T_7	T_{12}																																													
P_1	T_4		T_6	T_8	T_{10}	T_{11}																																											
P_2		T_2	T_3	T_9			T_{13}	T_{14}																																									

Distributed Memory Systems (DMS)



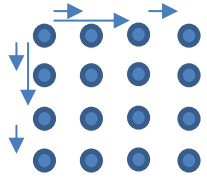

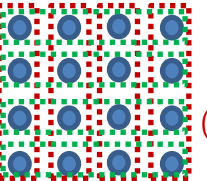
	shared memory systems	distributed memory systems
		for problems requiring vast amounts of data or computation
Task execution	tasks are carried out by threads	tasks are carried out by processes
Memory / Communication	each thread has private memory and access to shared-memory	processes have their own address space and can communicate to each other by different concepts: message passing, pipes, remotely shared-memory
Properties	cost of scaling the interconnect is very high large crossbar switches are very expensive	interconnects are inexpensive coarse-grained computations usually don't need a lot of shared memory
Examples		hypercube/toroidal mesh

MapReduce Model	<p>introduced by Google in 2004 for large data set adopted for C++, C#, F#, Erlang, Java, Python, ... OpenSource Impl: Hadoop by Apache</p> <p>map: execute a function on all items of an input list reduce: take key-value and reduce to associated value</p> <p>e.g. compute the number of words for all available word lengths</p>	
MPI: Message Passing Interface	<p>Standardized and portable message-passing system for parallel computing architectures</p> <p>Language: in C and Fortran, adopted for Python and Java</p> <p>Implementations: MPICH, Open MPI, Microsoft MPI</p> <p>Cons: consider communication cost</p>	
Structures	<p>asynchronous paradigm: all concurrent tasks execute asynchronously</p> <p>loosely synchronous model: tasks are synchronize to perform interactions between these interaction these interations, tasks execute completely asynchronously</p> <p>SPMD model: most message-passing programs are written using Single Program Multiple Data model</p>	
Example: Greetings	<pre>int main(int argc, char* argv[]) { int numprocs, myid; MPI_Init(&argc, &argv); // initializes MPI environment, eval cmd line args MPI_Comm_size(MPI_COMM_WORLD, &numprocs); // determines the number of processes MPI_Comm_rank(MPI_COMM_WORLD, &myid); // determines the label (id) of calling process if (myid == 0) { const int bufLen = 100; char greeting[bufLen]; cout << "process " << myid << " of " << numprocs << " processes!" << endl; for (int i = 1; i < numprocs; i++) { // receives a message: receives a buffer with n-elements from a source // blocks until received, order of one sender is kept (nonovertaking) MPI_Recv(greeting, bufLen, MPI_CHAR, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); cout << greeting << endl; } } else { stringstream ss; ss << "process " << myid << " of " << numprocs << " processes!"; string greeting(ss.str()); // sends a message: send a buffer with n-elements to a destination // blocks until received (non-buffered) or fully copied to internal buffer(buffered) MPI_Send(greeting.c_str(), (int)greeting.size() + 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD); } MPI_Finalize(); // terminates MPI, clean-up MPI environment, should return MPI_SUCCESS return 0; }</pre>	<pre>\$(MSMPI_BIN)mpiexec.exe -n 10 "\$(TargetPath)"</pre>
I/O	<p>MPI_COMM_WORLD -> allows access for all processors to stdout and stderr, allows access for p₀ to stdin</p> <p>order of process is unpredictable -> I/O should be done with process 0</p>	

Buffering/ Blocking determined by implementation		Command	Pro	Cons
	Non-Buffered – blocking	Ssend	simple, safe	idling and deadlocks
	Buffered (on receiver) – blocking	Bsend	less idling	block during send
	Non-Buffered – non-blocking	Isend	non-blocking	ensure semantic
solve deadlocks with <code>MPI_Sendrecv</code> or <code>MPI_Sendrecv_replace</code> use <code>MPI_Test</code> to check if an operation has finished use <code>MPI_Wait</code> to wait until an operation has finished use <code>MPI_Status</code> variable to get information about <code>MPI_Recv</code> operation use <code>MPI_Get_count</code> to get the count of data items received				
Buffer (aus Sicht von MPI)	Externer Buffer: Buffer meines Programms Internen Buffer: Buffer von MPI			
Debug MPI	run with one or two processes, run all processes on a single computer, use assertions use synchronous instead of buffered communication, attach debugger to one of these processes			
Communicator MPI_Comm	Defines a set of processes that are allowed to communicate with each other (intra-communication) A process can belong to multiple communication domains Default Communicator <code>MPI_COMM_WORLD</code> includes all processes			
Processor Mappings	The programmer cannot explicitly specify how processes are mapped onto processors-> job of MPI library Supported Topologies: k-dim. Mesh, arbitrary graph, dist graph			
Partitioning Topologies	use <code>MPI_Comm_split</code> to split processes into certain subset use <code>MPI_Cart_create</code> to create new communicators from old ones (<code>MPI_Cart_coord</code> , <code>MPI_Cart_rank</code>) use <code>MPI_Cart_shift</code> to shift data in a cartesian topology use <code>MPI_Cart_sub</code> to form lower-dimensional grids			
Collective Communication Operations	Barrier synchronization		<code>MPI_Barrier(comm)</code>	
	Broadcast: one-to-all, all-to-all		<code>MPI_Bcast(buf, count, datatype, source, comm)</code>	
	Reduction: reduce, all-reduce		<code>MPI_Reduce(sendbuf, recvbuf, count, datatype, op, target, comm)</code> <code>MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)</code>	
	Prefix-sum		<code>MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)</code>	
	Personalized comm: gather and scatter		<code>MPI_Gather(sendbuf, sendcount, sendType, recvbuf, recvcount..)</code> <code>MPI_Allgather, MPI_Scatter, MPI_Alltoall</code>	
each of these operations is defined over a group corresponding to the communicator all processors in a communicator must call these operations				
Reduction Operations	Operation	Meaning		Datatypes
	<code>MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD</code>	Maximum, Minimum, Sum, Product		C integers and floats
	<code>MPI_LAND, MPI_LOR, MPI_LXOR</code>	Logical AND, Logical OR, Logical XOR		C integers
	<code>MPI_BAND, MPI_BOR, MPI_BXOR</code>	Bit-wise AND, Bit-wise OR, Bit-wise XOR		C integers and byte
	<code>MPI_MAXLOC, MPI_MINLOC</code>	max-min, min-min value-location		Data-pairs
MPI Version	MPI-1	1994	version 1.0	
	MPI-2	2000	+Threads -> use run openMP on a Node + I/O parallel access to files	
	MPI-3	2012	- C++ Bindings + Non-blocking collective operations + One-sided communication (RemoteMemoryAccess)	

Collective Communication

Data Exchange	Designing parallel algorithms on a distributed-memory system requires data exchange between processes . This exchange can significantly impact the efficiency because of interaction delays.				
efficient impl.	Improve performance, reduce development effort and cost, improve software quality				
Overhead	due to idling, contention (conflict), communication and excess computation (not performed by serial)				
Cost	depends on: communication model, network topology, data handling & routing (e.g. cut-through), protocols				
Message passing costs	Startup time t_s : time spent at sending and receiving nodes Per-hop time t_h : function of number of hops and includes factors like switch latencies, network delays, etc. Per-word transfer time t_w : overheads by the message length: bandwidth of links, error checking/correction. Communication cost $t_{comm} = t_s + l * t_h + m * t_w$ (size m , point-to-point messaging) Simplified cost model $t_{comm} = t_s + m * t_w$ (t_h is often very small)				
Overview of Global Communication	What data is spread?	Same data is sent to all nodes Broadcast		Personalized data is sent to different nodes Personalized	
	How is received data handled?	Aggregated (glued together)	Reduced (combined)	Aggregated Reduced	
	Who is sending to whom	One-to-all	One-to-All Broadcast		Scatter
		All-to-one	-	-	Gather All-to-One Reduction
	All-to-All	All-to-All Broadcast	All-Reduce	Total Exchange All-to-All Reduction	
Lowerbound	$p * \frac{\text{number of messages} * \text{avg. number of links}}{\text{number of links}}$		1. Find formula for the lower bound for an algorithm 2. Calculate it for the different comm. systems		

Process p $3 = p - 1$	Linear-Array / Rings	Two-dim. mesh Row, cols: \sqrt{p}	Hypercube $p = 2^d$ dim: $d = \log p$
One-to-All Broadcast All-to-One Reduction 0 M $\xrightarrow{\text{one-to-all broadcast}}$ 0 M 1 $\xrightarrow{\text{all-to-one reduction}}$ 1 M 2 $\xrightarrow{\text{all-to-one reduction}}$ 2 M 3 $\xrightarrow{\text{all-to-one reduction}}$ 3 M	Naïve $O(p - 1)$  Better (recursive doubling) $O(\log p)$ 	 $O(\log p)$ $T = (t_s + mt_w) * \log p$	
All-to-All Broadcast (All-Gather) All-to-All Reduction 0 M_0 $\xrightarrow{\text{all-to-all broadcast}}$ 0 $M_0..M_3$ 1 M_1 $\xrightarrow{\text{all-to-all broadcast}}$ 1 $M_0..M_3$ 2 M_2 $\xrightarrow{\text{all-to-all broadcast}}$ 2 $M_0..M_3$ 3 M_3 $\xrightarrow{\text{all-to-all broadcast}}$ 3 $M_0..M_3$	Efficient approach Send data in each step to the neighbour  $O(p - 1)$ $T = (p - 1)(t_s + mt_w)$	 1. Rows $(\sqrt{p} - 1)(t_s + mt_w)$ 2. Columns $(\sqrt{p} - 1)(t_s + mt_w\sqrt{p})$ $T = 2t_s(\sqrt{p} - 1) + mt_w(p - 1)$ $T = t_s \log p + mt_w(p - 1)$	
All-Reduce 0 M_0 $\xrightarrow{\text{all-reduce}}$ 0 M_a 1 M_1 $\xrightarrow{\text{all-reduce}}$ 1 M_a 2 M_2 $\xrightarrow{\text{all-reduce}}$ 2 M_a 3 M_3 $\xrightarrow{\text{all-reduce}}$ 3 M_a	inefficient approach all2one reduction + one2all broadcast. better approach all2all broadcast without incr size of m	$T_{hyper} = (t_s + mt_w) \log p$	
Prefix-Sum 0 M_0 $\xrightarrow{\text{prefix-sum}}$ 0 M_0 1 M_1 $\xrightarrow{\text{prefix-sum}}$ 1 $M_0 + M_1$ 2 M_2 $\xrightarrow{\text{prefix-sum}}$ 2 \dots 3 M_3 $\xrightarrow{\text{prefix-sum}}$ 3 $\sum_{i=0}^{p-1} M_i$	all2all broadcast but only with labels less or equals to k		
Scatter (One-to-All personalized communic.) Gather 0 $M_0 \dots M_3$ $\xrightarrow{\text{scatter}}$ 0 M_0 1 $\xrightarrow{\text{gather}}$ 1 M_1 2 $\xrightarrow{\text{gather}}$ 2 M_2 3 $\xrightarrow{\text{gather}}$ 3 M_3		$T = t_s \log p + mt_w(p - 1)$ $T = \Omega(mt_w(p - 1))$	
Total Exchange e.g. transpose a matrix (All-to-All Personalized Communication) 0 $M_{0,0} \dots M_{0,3}$ $\xrightarrow{\text{total exchange}}$ 0 $M_{0,0} \dots M_{3,0}$ 1 $M_{1,0} \dots M_{1,3}$ $\xrightarrow{\text{total exchange}}$ 1 $M_{0,1} \dots M_{3,1}$ 2 $M_{2,0} \dots M_{2,3}$ $\xrightarrow{\text{total exchange}}$ 2 $M_{0,2} \dots M_{3,2}$ 3 $M_{3,0} \dots M_{3,3}$ $\xrightarrow{\text{total exchange}}$ 3 $M_{0,3} \dots M_{3,3}$	$T_{ring} = \left(t_s + \frac{t_w mp}{2}\right) (p - 1)$	$T_{Mesh} = (2t_s + mpt_w)(\sqrt{p} - 1)$ $T_{naive hyper} = \left(t_s + \frac{t_w mp}{2}\right) \log p$ $T_{better hyper} = (t_s + t_w m)(p - 1)$	

PARALLEL ALGORITHMS

Numerical Algorithms

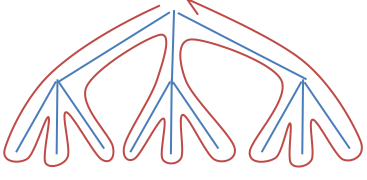
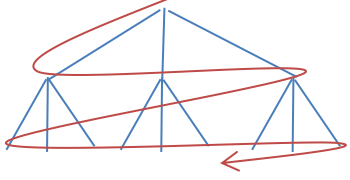
<p>Dense Matrix-Vector Multiplication</p>	<p>Gegeben: $A: n \times n - Matrix$ $x, y: n \times 1 - Vector$</p> <p>Gesucht: $y = Ax$</p>	$p \left\{ \begin{matrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{matrix} \right\} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}$
<p>Solution 1: Row-wise 1D Partitioning</p>	<p>1 row of matrix A per process (p=n)</p> <ol style="list-style-type: none"> all-to-all broadcast of x p_i computes $y_i = A_i * x$ <p>Using fewer than n processes (p<n) Each process owns $m = \frac{n}{p}$ rows of A and the corresponding elements of x same algorithm</p> <p>problem size: $W = \Theta(n^2)$</p>	$T_p = \Theta(n) + \Theta(n) = \Theta(n)$ $Cost = n * \Theta(n) = \Theta(n^2) \rightarrow cost\ optimal$ $T_p = t_s \log p + \frac{t_w n}{p} (p-1) + \frac{n^2}{p} = \Theta\left(\log p + n + \frac{n^2}{p}\right)$ $Cost = \Theta(p \log p + np + n^2) \rightarrow cost\ optimal\ for\ p = O(n)$ $T_0 = p \log p + np + n^2 - W = p \log p + np$ <p>Isoefficiency: $W = K T_0 = \frac{K p \log p}{W=p \log p} + \frac{K \sqrt{W} p}{W=p^2} \Rightarrow \max\ blue$ $\rightarrow f(p) = \Theta(p^2)$</p> <p>degree of concurrency: $C(W) = O(\sqrt{W}) = O(n)$ $p = O(n) \rightarrow n = \Omega(p) \rightarrow W = \Omega(p^2)$</p>
<p>Solution 2: 2D Partitioning</p>	<p>1 matrix element per process p_i owns $A_{i,j}$ last column own x_i</p> <ol style="list-style-type: none"> Align x along the diagonal Distribute x_i along columns n^2 parallel multiplications All-to-one reduction - rows 	$T_p \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$ $T_0 = p T_p - W = n^2 + t_s p \log p + t_w \sqrt{p} n \log p - n^2$ $p = O(f^{-1}(W)) \approx O\left(\frac{n^2}{\log^2 n}\right)$ <p>Resume: 2D faster for $p \leq n$, better isoefficiency and more scalable</p>
<p>Dense Matrix-Matrix Multiplication</p>	<p>Gegeben: $A, B: n \times n - Matrices$</p> <p>Gesucht: $C = AB$</p>	$\begin{pmatrix} 3 & 1 & 2 & 1 \\ 1 & 2 & 1 & 3 \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} \times \begin{pmatrix} \dots & \dots & 2 & 1 \\ \dots & \dots & 1 & 1 \\ \dots & \dots & 1 & 2 \\ \dots & \dots & 3 & 2 \end{pmatrix} = \begin{pmatrix} \dots & \dots & 12 & 10 \\ \dots & \dots & 14 & 11 \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$
<p>Solution 1: Block-Matrix Multiplication</p>	<p>divide into q blocks $\rightarrow (1 < q \leq n)$ $p = q^2$</p> <ol style="list-style-type: none"> all-to-all broadcast of Matrix A's blocks in each row all-to-all broadcast of matrix B's blocks in each column compute block $C_{i,j}$ <p>assumption: $W = n^3$</p>	$\begin{pmatrix} 3 & 1 & 2 & 1 \\ 1 & 2 & 1 & 3 \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} \dots & \dots & 2 & 1 \\ \dots & \dots & 1 & 1 \\ \dots & \dots & 1 & 2 \\ \dots & \dots & 3 & 2 \end{pmatrix} = \begin{pmatrix} 4 & 3 \\ 10 & 8 \end{pmatrix}$ $T_p = \frac{n^3}{q^2} + 2 \left(t_s \log q + \frac{t_w n^2}{q^2} (q-1) \right)$ <p>q block mult. 2 broadcasts</p> <p>cost-optimal for $p = O(n^2)$; isoefficiency $\Theta(p^{\frac{3}{2}})$ degree of concur. $C(W) = O(W^{\frac{2}{3}}) = O(n^2) = p \rightarrow W = \Omega(p^{\frac{3}{2}})$</p>
<p>Cannon's Matrix Multiplication</p>	<p>memory optimal each p computes one block and shifts $A_{i,k}$ in its row and $B_{k,j}$ in its columns</p>	$T_p = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w \frac{n^2}{\sqrt{p}}$ <p>isoefficiency $O(p^{\frac{3}{2}})$</p>
<p>DNS Matrix Multiplication</p>	<p>with intermediate data partitioning n^3 processes, each compute one scalar multiplication reduce vectors of n multiplication</p> <p>using fewer than n^3 processes assume $p = q^3$ for $q < n$ block partitioning: block size = $\frac{n}{q} \times \frac{n}{q}$ data partitioning: block size = $\left(\frac{n}{q}\right)^3$</p>	$T_p = O(1) + \Theta(\log n) = \Theta(\log n)$ <p>$\Theta(n^3 \log n)$, is not cost optimal</p> $T_p = \frac{n^3}{p} + t_s \log p + \frac{t_w n^2}{p^{\frac{2}{3}}} \log p$ <p>isoefficiency: $\Theta(p \log^3 p)$ cost optimal for $p = O\left(\frac{n^3}{\log^3 n}\right)$</p>
<p>Parallel Gauss Elimination</p>	<p>1D Row Partitioning</p> <p>2D Partit. with pipelining, $p = n^2$</p> <p>2D Partit. with pipelining, $p < n^2$</p>	$T_p = \frac{3}{2} n(n-1) + t_s n \log n + \frac{1}{2} t_w n(n-1) \log n$ <p>$O(n^2) \rightarrow is\ cost\ optimal$</p> <p>$O(n) \rightarrow is\ cost\ optimal$ more scalable than 1D</p> <p>$O\left(\frac{n^3}{p}\right) \rightarrow is\ cost\ optimal$ $n \gg p$</p>

Sorting Algorithms

<p>Overview</p>	<p>Most commonly used and well-studied kernels. Lower bound is $\Theta(n \log n)$. Verteilte Daten: jeder Prozess hat $\frac{n}{p}$ Daten. Parallel sortierte Sequence: aufwärts sortiert innerhalb eines Prozesses und sortiert nach prozessor id.</p>																																	
<p>Compare-Exch</p>	<p>process p1 and p2 exchange elem a and b. p1 keeps the min, p2 keeps the max. $T = t_s + t_w$</p>																																	
<p>Compare Split</p>	<p>Daten Austausch mit $\frac{n}{p}$ sortierten Daten. Merged alle und behält die zuständigen Daten. $T = t_s + t_w \frac{n}{p}$</p>																																	
<p>Sorting Network</p>	<p>network of comparators designed specifically for sorting (2 inputs, 2 outputs) (incr or decr)</p>																																	
<p>Parallel Odd-Even Transposition Sort</p>	$T_p = \underbrace{\frac{n}{p} \log \frac{n}{p}}_{\text{lokales sortieren}} + \underbrace{p}_{\text{Anzahl Phasen}} * \underbrace{\frac{n}{p}}_{\text{Comp.split}} = \frac{n}{p} \log \frac{n}{p} + n$ $Cost = p * T_p = n \log \frac{n}{p} + p * n$ $Speedup: S = \frac{T_s}{T_p} = O\left(\frac{n \log n}{\frac{n}{p} \log \frac{n}{p} + n}\right)$	<p>Unsorted</p> <table border="1"> <tr><td>3</td><td>2</td><td>3</td><td>8</td><td>5</td><td>6</td><td>4</td><td>1</td></tr> <tr><td>2</td><td>3</td><td>3</td><td>8</td><td>5</td><td>6</td><td>1</td><td>4</td></tr> <tr><td>2</td><td>3</td><td>3</td><td>5</td><td>8</td><td>1</td><td>6</td><td>4</td></tr> <tr><td>2</td><td>3</td><td>3</td><td>5</td><td>1</td><td>8</td><td>4</td><td>6</td></tr> </table> <p>Phase 1 (odd) Phase 2 (even) Phase 3 (odd) Phase 4 (even)</p> <p>$p = n \rightarrow Cost = \Theta(n^2)$ nicht kostenoptimal, da sortieren $n \log n$ verwenden sollte. $p = \log n \rightarrow Cost = \Theta(n \log n)$ kostenoptimal</p>	3	2	3	8	5	6	4	1	2	3	3	8	5	6	1	4	2	3	3	5	8	1	6	4	2	3	3	5	1	8	4	6
3	2	3	8	5	6	4	1																											
2	3	3	8	5	6	1	4																											
2	3	3	5	8	1	6	4																											
2	3	3	5	1	8	4	6																											
<p>Parallel Shellsort</p>	<ol style="list-style-type: none"> compare-split operation on process far aways odd-even transposition sort with $l \leq p$ $T_p = \underbrace{\Theta\left(\frac{n}{p} \log \frac{n}{p}\right)}_{\text{local sort}} + \underbrace{\Theta\left(\frac{n}{p} \log p\right)}_1 + \underbrace{\Theta\left(l \frac{n}{p}\right)}_2$																																	
<p>Bitonic Sort</p>	<p>a bitonic sequence has two tones (sequences): increasing and decreasing or vice versa (shift allowed)</p> <ol style="list-style-type: none"> build a bitonic sequence merge into a sorted sequence <p>Hypercube $T_p = \Theta(\log^2 n)$ Mesh $T_p = \underbrace{\Theta(\log^2 n)}_{\text{compare}} + \underbrace{\Theta(\sqrt{n})}_{\text{comm}}$ $\frac{n}{p}$ items on hypercube $T_p = \Theta\left(\frac{n}{p} \log \frac{n}{p}\right) + 2\Theta\left(\frac{n}{p} \log^2 p\right)$</p>																																	
<p>Quicksort</p>	<p>Simple, low overhead, optimal complexity $O(n \log n)$ recursive select one element as pivot, divide into 2 sequences (1 with smaller, 1 with bigger then pivot).</p>																																	
<p>PRAM Parallel Quicksort</p>	<p>CRCW (concurrent read, write) PRAM with concurrent writes resulting in an arbitrary write succeeding.</p>																																	
<p>SAS Quicksort</p>	<p>Shared Adress Space Quicksort recursive repeated for each process group and sub-array is assigned to a single process, in which case it proceeds to sort it locally global rearrangement: each p counts n greater/smaller than pivot, to know in which element to write</p> $T_p = \underbrace{\Theta\left(\frac{n}{p} \log \frac{n}{p}\right)}_{\text{local sort}} + \underbrace{\Theta\left(\frac{n}{p} \log p\right)}_{\text{array splits}} + \Theta(\log^2 p)$																																	
<p>Sequential Bucket Sort</p>	<p>assumption: the n-elements to be sorted are uniformly distributed over an interval [a,b]</p> <ol style="list-style-type: none"> divide the range [a,b] of input numbers into m equal sized intervals, called buckets each element is placed in its appropriate bucket (buckets have roughly identical number of elem) elements in the bucket are locally sorted <table border="1"> <tr> <td>$\sim \frac{n}{m}$ elements per bucket</td> <td>$\underbrace{O(n)}_{\text{placing}} + m * \underbrace{O\left(\frac{n}{m} \log \frac{n}{m}\right)}_{\text{local sort}}$ $T_p = n * \log \frac{n}{m}$</td> <td>Normal sort $m = 1 \rightarrow T_p = n * \log n$ Enum sort $m = n \rightarrow T_p = O(n)$</td> </tr> </table>		$\sim \frac{n}{m}$ elements per bucket	$\underbrace{O(n)}_{\text{placing}} + m * \underbrace{O\left(\frac{n}{m} \log \frac{n}{m}\right)}_{\text{local sort}}$ $T_p = n * \log \frac{n}{m}$	Normal sort $m = 1 \rightarrow T_p = n * \log n$ Enum sort $m = n \rightarrow T_p = O(n)$																													
$\sim \frac{n}{m}$ elements per bucket	$\underbrace{O(n)}_{\text{placing}} + m * \underbrace{O\left(\frac{n}{m} \log \frac{n}{m}\right)}_{\text{local sort}}$ $T_p = n * \log \frac{n}{m}$	Normal sort $m = 1 \rightarrow T_p = n * \log n$ Enum sort $m = n \rightarrow T_p = O(n)$																																
<p>Enumeration Sort</p>	<p>similar to bucket sort, but create for each number in the range a bucket, and put it in the right bucket</p>																																	

Parallel Bucket Sort	<p>each process is assigned a block of $\frac{n}{p}$ elements, number of buckets $m=p$, each process knows the range $[a,b]$</p> <ol style="list-style-type: none"> each process partitions its block of $\frac{n}{p}$ elements into p sub-blocks, one for each of the p buckets each process send $p - 1$ sub-blocks to the appropriate processe using a single all-to-all personalized communication each process sorts all the elements it receives by using an optimal sequential sorting algorithms $T_p = \underbrace{O\left(\frac{n}{p}\right)}_1 + \underbrace{O\left(\frac{n}{p^2} * p\right)}_2 + \underbrace{O\left(\frac{n}{p} \log \frac{n}{p}\right)}_3 = O\left(\frac{n}{p} \left(1 + \log \frac{n}{p}\right)\right) = O\left(\frac{n}{p} \log \frac{n}{p}\right)$
Sequential Sample Sort	<p>Similar to bucket sort without the unrealistic assumption of uniformly distributed elements a sample is selected from the n elements and choosing $m - 1$ elements (splitters) from the sorted sample. Splitter selection: each p: sort with quicksort, choose $p-1$ samples (equal divided); repeat with samples</p>
Parallel Sample S.	<p>$m = p$; share splitters with all-to-all broadcast;</p> $T_p = \underbrace{\Theta\left(\frac{n}{p} \log \frac{n}{p}\right)}_{\text{local sort}} + \underbrace{\Theta(p^2 \log p)}_{\text{sort sample}} + \underbrace{\Theta\left(p \log \frac{n}{p}\right)}_{\text{block partition}} + \underbrace{\Theta\left(\frac{n}{p}\right) + \Theta(p \log p)}_{\text{communication}}$

Graph Algorithms (kommt nicht)

DFS vs BFS	<p>DFS (Depth-First Search Algorithm)</p>  <p>Pro: small place $O(d * h)$ Cons: find sub-optimal solutions first Variation: set a maximum depth level</p>	<p>BFS (Best-First-Search Algorihtm)</p>  <p>Pro: Find best solutions first Cons: needs a lot of space $O(d^h)$</p>
-------------------	---	---

Code

- OpenMP
- Evt. OpenCL
- MPI
- Param order doesn't matter, but name should be clear
- 1/3 theory, 1/3 code, 1/3 run algorithm
- Cost-optimal / efficiency
- Exam is until parallel sorting
- Parallel graph search is a little bit to advance

120min, A hand written summary of 4 A4 page