TU WIEN Informatics Parallel Computing

"Skriptum"
Lectures on Parallel Computing
SS2023

Jesper Larsson Träff
TU Wien, Faculty of Informatics
Institute of Computer Engineering
Research Group for Parallel Computing
Treitlstrasse 3, DG/191-4
1040 Wien

March–June 2020
March 2021
March 2022
March 2023

Version: 0.9 (March 17, 2023)

VORWORT

Dieses "Vor-Skriptum" ist als Lesehilfe für die Folien zu der Bachelor Vorlesung "Parallel Computing" gedacht. Wir versuchen auf die besonders wichtigen Punkte aufmerksam zu machen und die jeweiligen Vorlesungen zusammenzufassen. Ergänzende Textbücher, welche Material enthalten, die nicht in der Vorlesung besprochen werden, sind das Buch von Rauber and Rünger [54] sowie das Buch von Schmidt et al. [59]. Umgekehrt enthält die Vorlesung auch viel Material, welches nicht in diesen Büchern zu finden ist. Das Skriptum ist in Englisch verfasst.

Die mit $\star$ markierten Abschnitte sind nicht Teil des Stoffes für die Bachelorvorlesung.

CONTENTS

1

INTRODUCTION TO PARALLEL COMPUTING: ARCHITECTURES AND MODELS

1.1 FIRST BLOCK (1-2 LECTURES)

Parallel computers, meaning computers and computer systems with more than one processing element capable of executing a program, are everywhere. The number of processing elements, in modern terminology often called a *core* or *processor-core*, range from a few (embedded systems, mobile devices), to tens and hundreds (desktops, servers), to thousands, ten-thousands, and even millions in the largest High-Performance Computing (HPC) systems (see http://www.top500.org for some such systems). Every computer scientist has to be aware of this fact and know something about Parallel Computing.

Despite being an active area of research and also of commercial developments of actual parallel computer systems in the mid-80s to mid-90s of the last century, parallel computing was largely absent from main stream computer science during the 90s to early in the 2000 years. This has had and still has dire consequences. The area was largely missing from university curricula (e.g., parallel algorithms, programming and software development), leading to a lack of knowledgeable experts and professionals (and now to frequent rediscovery of already known results and techniques; it still makes much sense to read technical papers from the 80ties and 90ties).

1.1.1 *"Free lunch" and Moore's Law*

One reason for this was the "free lunch" phenomenon, also sometimes called *Moore's Law*, that the performance of sequential computers was observed (and projected) to increase exponentially, with a doubling rate of 18 to 24 months. This popular version of this "law" held from the 70s until the early- to mid-2000 years; but is not exactly what Gordon Moore actually speculated [50]). The exponential increase in performance made building and selling parallel computers commercially tough; many companies folded in the early 90s, and other companies changed their strategies (HPC was one niche where some companies could survive). On the other hand, "Moore's law" exerted an enormous pressure on processor manufacturers; also this had consequences

(leading, for instance, to many fantastic and fantastically useless HPC systems being built).

In the mid-2000 years the "free lunch" was largely over. The performance of sequential processors has not increased as dramatically since then, as has been documented by many (popular) studies (that may deserve a closer look)[1]. A way out to continue increasing nominal and possibly achieved performance is to employ *parallelism*.

1.1.2 *Performance of Processors*

For now, *nominal processor performance* is defined strictly processor-centrically as the maximum (best-case) number of operations (often: *FLoating point OPerations per Second*, *FLOPS*) that can be carried out per unit of time (second). The performance of a single processor core is calculated as the product of the clock frequency (number of "ticks" per second, usually measured in GHz) and the number of instructions that the processor can complete per clock cycle (FLOP's/cycle). The number of instructions per clock is determined by the processor architecture: number of pipelines, depth of pipelines, types of instructions (fused multiply-add, for instance), super-scalar capabilities, vectorization (*SIMD*) capabilities, etc.. [16]. Generally, it is an optimistic upper bound on the actual real-world performance of applications. Also, beware that the FLOPS abbreviation is ambiguous and quite unfortunate: sometimes the FLOP's are meant, sometimes the FLOP's/second.

Whether the nominal performance of a processor can be reached depends on at least two factors. First, whether the program/algorithm being executed contains operations in the right mix and with the right dependencies to allow full utilization of the components and features of the processor-core. For instance, a program solving a graph problem typically executes 0 FLOPS; it does not exploit any of the floating point capabilities of the processor (likely a major part). Second, the memory system must be able to supply the data needed to keep the processor busy. This is often a (even the) major reason for an observed, "poor" performance.

The ratio between processor performance and memory access time has not improved at the pace processor performance has improved (Moore's Law). The main idea to alleviate the gap has been the introduction of (larger and larger, hierarchically organized) caches [16]. Caches and the memory system will play an important role in later lectures.

Our current terminology is that a *processor* (CPU) consists of multiple *(processor-)cores*, also sometimes called *processing elements* (PE), or processing units (PU): The entities that can execute a program (and what is now called cores used to be called processors). A processor with a smaller number of cores (a handful, e.g., 2, 4, 8, 16, and 32) is called a *multi-core processor*, and

[1] see for instance https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

a processor with a large number of cores a *many-core processor*. The typical example of the latter is a graphics processing unit (GPU), which will play almost no role in these lectures. We will use only the term *multi-core* (where needed). The *nominal performance* of a multi-core processor is calculated by multiplying the nominal per-core performance with the number of cores.

Some recommended text-books to check up on computer systems and and computer architecture are [16, 52, 53] (possibly in later editions).

1.1.3 *Parallel vs. Distributed Computing*

The focus of Parallel Computing is to use parallel resources (processors, processor-cores) *efficiently* for solving given *computational (algorithmic) problems*. Towards this, Parallel Computing is concerned with *algorithms*, their *implementation* in suitable *programming languages* that realize more or less explicitly formulated *programming models* capturing the essentials for analyzing and reasoning aboutn programs, and the structure and capabilities of the *computer architecture*. We judge efficiency in all these respects, both theoretically and practically/experimentally. Parallel Computing is thus theoretical, practical and experimental/empirical Computer Science. Parallel Computing is thus much broader in scope than *parallel programming*, which will also be treated in these lectures (with OpenMP and MPI as examples).

Parallel Computing is intimately related to the disciplines of distributed and concurrent computing, and distinguishing is a matter of focus (what we are interested in). In these lectures we propose and use the following definitions.

Definition 1 (Parallel Computing) *The discipline of efficiently utilizing dedicated parallel resources for solving given computation problems.*

The focus of Parallel Computing is on *problem solving efficiency*, and a fundamental assumption is that the full computer system is at our disposal (dedicated). Interesting parallel computing problems are those that require significant interaction (communication, be it via memory reads/writes, or explicit communication over some interconnection network) between the parallel resources (cores), on systems that actually provide significant intercommunication and processing capabilities. Real, parallel computers are thus not thought of as spatially (widely) distributed [11].

Parallel Computing is related to and can benefit from results in *distributed* and *concurrent* computing, by which is meant the following (our definitions, others may disagree).

Definition 2 (Distributed Computing) *The discipline of making independent, non-dedicated resources available to cooperate toward solving specified problem complexes.*

The focus of distributed computing is on availability of resources that are not readily at hand, may be spatially widely distributed, may change dynamically, and may *fail*. In Parallel Computing, processor-cores *do not fail* (not in this lecture!). Specific, individual problems may be studied, or larger problem complexes. A central tenet in distributed computing is that there is no centralized control. Example: Acquiring resources from the cloud, subject to certain constraints and requirements, may be a distributed computing problem. Using the resources (as a (virtual) parallel machine) for solving efficiently the problem we are interested in (for instance, within given time constraints) is on the other hand a parallel computing problem.

Definition 3 (Concurrent Computing) *The discipline of managing and reasoning about interacting processes that may or may not progress simultaneously.*

The focus of concurrent computing is on *concurrency*, activities that may or may not happen at the same time, and therefore, on reasoning about and establishing correctness (in a broad sense) in such situations (e.g., process calculi [37, 48]). In contrast, parallel computing is specifically concerned with bounds on the performance that can also practically be achieved.

1.1.4 *Sample computational problems*

Some computational problems that will be considered throughout the lecture are:

- matrix-vector multiplication,
- matrix-matrix multiplication,
- merging of ordered sequences (of numbers, and objects),
- sorting numbers or objects from an ordered set (by merging, Quicksort, ...),
- computing sums and maxima over objects stored in arrays,

- performing general reductions with arbitrary, associative operators,
- prefix-sums over arrays,
- stencil-computations, and
- graph search problems.

1.1.5 *Models for Sequential and Parallel Computing*

For designing and analyzing parallel algorithms, a suitable model is needed. A good model is one which makes it possible to derive interesting algorithms and results, makes analysis tractable, and bears enough resemblance to actual machines and systems that the algorithms can be implemented and results be predictive (of, say, performance).

The latter is sometimes called a *bridging model* (we use the term in this fashion), and was originally introduced by Les Valiant [70] who proposed a specific model as bridge, the so-called *Bulk Synchronous Parallel* (*BSP*) model. A minimum requirement to a good bridging model is that if some algorithm A is shown to perform better than algorithm B in the model, then a (faithful) implementation of A should perform better than an (equally faithful) implementation of B on the real machine ("bridging"). Related to the bridging idea, is the (vague) notion of *performance portability*, which says that the good performance of a program can be preserved when going from one system to another. This is clearly a desirable property.

While there are various "bridging models" in sequential computing (the *RAM*, *Random Access Machine* being one, although not unproblematic, and with many restrictions), the situation is completely different for Parallel Computing. There are many different parallel computer architectures (multi-core CPU vs. GPU; distributed memory system vs. shared-memory system, etc.), at vastly different scales, and no model (so far) bridges them all to any useful extent. Also, model assumptions that are desirable for the design of algorithms do, to an even lesser extent than for sequential models, hold for parallel computer systems. Many such assumptions are related to the memory behavior. For instance the assumption of unit-time, uniform memory access of the RAM is already problematic for sequential computers, and even more so for large parallel systems with widely distributed memory.

1.1.6 *The PRAM Model*

One, extremely useful (but unrealistic) model of parallel computing is the *Parallel Random Access Machine* (*PRAM*), a natural generalization of the equally useful and pervasive, sequential *Random Access Machine* (*RAM*). Like the RAM, the PRAM assumes a large (as large as needed) memory where processors can read and write words in unit time. A PRAM has or employs a number

of processors that can be chosen as convenient (fixed, as a free parameter, or as a function of the input size). The processors all execute their own program, but do so in lock-step: strictly synchronized, all following the same clock and performing an instruction in each *time step*. This means that the machine is always in a well-defined *state* (the program counter of the processors, contents of the memory and the processor registers; state transitions happen instantaneously by the synchronous clock ticks), and reasoning with state invariants, as done with RAM algorithms, is a way to prove properties. A PRAM algorithm specifies what the processors are to do in each step.

With many processors operating in lock-step, it can potentially happen that more than one processor is accessing some memory word in the same time step. The PRAM model needs to define what happens in such cases. First, a memory word can in a step be either read or written; but not read by some processor(s) and written by another. For potentially *concurrent accesses* to a memory word in a step by two or more processors, there are three main variations of the PRAM that have been used in the literature:

- An *EREW* (Exclusive Read Exclusive Write) *PRAM* disallows accesses to the same memory word in the same step by more than one processor. It is the algorithm designers responsibility to make sure that simultaneous accesses do not happen.

- A *CREW* (Concurrent Read Exclusive Write) *PRAM* allows simultaneous (concurrent) reads to a word by more than one processor in a time step, but not simultaneous writes.

- A *CRCW* (Concurrent Read Concurrent Write) *PRAM* allows both simultaneous reads and writes to the same word in the same step. What happens when two or more processors write to a word in a step? In a *Common CRCW PRAM*, it must be ensured that the writing processors all write the same value. In an *Arbitrary CRCW PRAM*, either of the written values will survive in the memory word. A *Priority CRCW PRAM* has some priority associated with the processors, and the writing processor with the highest priority will successfully write its value to the memory word.

What happens in case the EREW/CREW/CRCW constraints are violated by our algoirthm is just a matter of model design: perhaps the machine breaks down, explodes, halts, delivers incorrect results, or some other outcome. The important requirement is that the algorithm designer has to make sure (prove!) that the constraints of the PRAM variant at hand are never violated when the algorithm is executed. Per definition, any algorithm that can be executed correctly on an EREW PRAM, can execute on any of the, in that sense, stronger models.

Since the PRAM is a purely theoretical construct (there has been several attempts to realize emulated PRAM's in real hardware, but so far none have

been entirely successful), convenient pseudo-code can be used liberally to express algorithms, as long as it is clear that the model assumptions are satisfied. In the pseudo-code in this lecture there is a construct for starting a set of processors, each being assigned an identity (some integer) to which it can refer. This is the `par`-construct that looks similar to a C-pseudo code `for`-loop but with freedom to express the range/set of processors to start. We will assume that starting a reasonably specified set of processors can be done even on an EREW PRAM in a constant number of operations, $O(1)$ per processor. If a PRAM would be realized in hardware, it would be the task of the run-time system and compiler to provide constructs for starting well-defined sets of processors with some well-defined (small) overhead.

Using the PRAM, we can already now give interesting algorithms for finding the maximum among n numbers, and for doing matrix-matrix multiplication of $n \times l$ and $l \times m$ matrices (into an $n \times m$ result matrix). PRAM-pseudo code is given below, and the results are summarized in the following theorems.

```
par (0<=i<n) b[i] = true;      // a[i] could be maximum
par (0<=i<n, 0<=j<n) {
  if (a[i]<a[j]) b[i] = false; // a[i] is not maximum
}
par (0<=i<n) if (b[i]) x = a[i];
```

Theorem 1 *The maximum of n numbers stored in an array can be found in $O(1)$ parallel time steps, using n^2 processors (and performing $O(n^2)$ operations) on a Common CRCW PRAM.*

In the program, the input is stored in the n-element array `a`, indexed C-style from 0 to $n-1$. The idea of this fastest possible algorithm is to do all the n^2 element pair comparisons in one parallel step (actually, the $n(n-1)$ comparisons with different element indices would suffice), and use the outcome to knock out the elements that cannot possibly be the maximum. This is done with the Boolean array `b`, which first records each of the n elements as a candidate for being a maximum, and then by the outcome of the comparisons marks elements that cannot be maximum by virtue of being smaller than some other element. The three `par`-constructs start n and n^2 processors, respectively, first for initializing the `b`-array, second for performing the comparisons in parallel, and finally writing out the maximum to the result variable `x`. Since in one step, several processors can discover that some element `a[i]` cannot be a maximum, concurrent writing to the same `b[i]` can happen. If and at which indices this happens is dependent on the input. When several processors write to a location `b[i]` or `x` in a step, they, however, write the same value, and therfore a Common CRCW PRAM would suffice. This is an interesting, maximally fast (there is nothing faster than constant time, and the constants here seem to be small) algorithm: The PRAM model is good for exposing the maximum amount of parallelism in a problem.

In order to avoid concurrent writing, a different algorithmic idea is needed: Instead of doing all pairwise comparisons in a step, only do up to $n/2$ comparisons between disjoint pairs of elements. The pseudo-code below implements this idea.

```
nn = n;
while (nn>1) {
  k = (nn>>1)+(nn&0x1); // ceil(nn/2) by bitwise operations
  par (0<=i<k) {
    if (i+k<nn) a[i] = max(a[i],a[i+k]);
  }
  nn = k;
}
```

Theorem 2 *The maximum of n numbers stored in an array can be found in $O(\log n)$ parallel time steps, using a maximum of $n/2$ processors (but performing only $O(n)$ operations) on a CREW PRAM.*

The algorithm in each iteration does comparisons between $\lfloor n/2 \rfloor$ pairs only, which reduces the number of possible maximum elements to $\lceil n/2 \rceil$. The comparison steps are iterated $\lceil \log_2 n \rceil$ times, after which a maximum element is left. This algorithm can be modified to run also on an EREW PRAM.

The last example turns the definition of matrix-matrix multiplication into parallel PRAM code. Since we do not (yet) know how to compute the sum of n elements (the n element products), this part of the definition is implemented as a sequential loop, but all nm sums are computed in parallel as specified by the outer `par`-construct.

```
par (0<=i<n, 0<=j<m) {
  C[i,j] = 0;
  for (k=0; k<l; k++) {
    C[i,j] += A[i,k]*B[k,j];
  }
}
```

Theorem 3 *Two $n \times l$ and $l \times m$ matrices can be multiplied into an $n \times m$ matrix C in $O(l)$ time steps and $O(lnm)$ operations on a CREW PRAM.*

The algorithm shown can be improved to also run on an EREW PRAM by using extra space for intermediate results. It can be made faster by employing a variant of the maximum finding algorithm to do the summations in parallel.

The complexity properties of the PRAM algorithms so far were stated in terms of the total number of parallel steps required (for the given input), the number of processors needed, the total number of operations carried out by all the processors during the course of execution, and the PRAM model assumed by the algorithm. The natural goal when studying the parallel complexity of

specific, given problems is to minimize these requirements on all counts: as few parallel steps, as few total operations, and as weak a PRAM model as possible. As the observations and theorems above show, some of these goals are contradictory and cannot be achieved simultaneously. A strong Common CRCW PRAM model made it possible to find the maximum of n numbers optimally fast (constant time), but at the additional cost of a large number of operations (Theorem 1). An algorithm for a weaker, possibly less expensive CREW PRAM using less operations was given; but use more time (parallel steps) (Theorem 2). We elaborate on these measures and trade-offs which will be a main theme in the following lectures.

The PRAM model has been productive in finding highly parallel, fast algorithms for many interesting problems, and also in establishing lower bounds on how fast and with how many resources (processors) problems can be solved [38]. Whether the algorithms studied so far are good or useful will be discussed in the next following parts.

Other theoretical models for parallel computing that we will meet but not use include comparator networks, systolic arrays, cellular automata, ...

More realistic models are harder to formalize and use, and will be taken up later are: Asynchronous, shared-memory machines with non-uniform memory access (NUMA), distributed memory systems with interconnect, etc.. The abbreviation NUMA stands for *Non-Uniform Memory Access*, in stark contrast to the *Uniform Memory Access* (UMA) assumption for the PRAM.

1.1.7 *Flynn's Taxonomy*

A different, frequently used, but less architecture oriented and rather crude characterization of parallel machines and systems (and even programs) is the so-called *Flynn's taxonomy* [25]. This taxonomy looks at the instruction and data stream(s) of the system. A *Single-Instruction, Single-Data* (*SISD*) system is a sequential computer. A *Single-Instruction, Multiple-Data* (*SIMD*) system, is one in which a single instruction can operate on a larger batch of data, like for instance a whole vector (array) of some size. Thus, *vector computers*, or processors with capabilities to operate on short vectors of a few words (most processors nowadays have such capabilities), are typical SIMD systems. A PRAM machine would be classified as *Multiple-Instruction, Multiple-Data* (*MIMD*), since each processor can execute an own instruction stream, each operating on its own stream of data. Finally, but not obviously, a *Multiple-Instruction, Single-Data (*MISD*)* system could be a deeply pipelined system where a single stream of data passes through several stages.

Flynn's taxonomy is sometimes used also to characterize *programming models* by which is meant the abstractions under which a program can be described (threads, processes, data access patterns, synchronization and communication mechanisms, etc.). A SIMD model for instance would be one in which there is

a single "logical" instruction stream (that might, as in a PRAM, be executed by many processors) that operates on some abstract "vectors" [12].

The characterization *Single-Program, Multiple-Data* (*SPMD*) is sometimes used to describe the situation where all processors in a parallel system execute the same program, but each processor may, at any time instant, be in a different part of the program, and operate thus on a different "data stream" than the other processors.

1.2 SECOND BLOCK (1-2 LECTURES)

The bar for Parallel Computing is high. We judge parallel algorithms and implementations by comparing against the *best possible* sequential algorithm for solving the given problem, and in cases where the best possible (lower bound) is not known, against the *best known* sequential algorithm. The reasoning is that we, by using the dedicated parallel resources at hand, want to improve over what we can already do with a sequential algorithm on our system. With our parallel machine, we want to solve problems faster and/or better on some account.

For now our parallel model and system we be left unspecified. Some number p of processor-cores interact to solve the problem at hand.

1.2.1 *Sequential and Parallel Time*

Parallel Computing is both a theoretical discipline and a practical/empirical/-experimental endeavor. As a theoretical discipline we are interested in the performance of algorithms in some models (RAM, PRAM, and more realistic settings), and typically look at the performance in the worst possible case (worst possible inputs) when the input size is sufficiently large. Let Seq and Par denote sequential and parallel algorithms for a problem we are interested in solving. Then by $T_{seq}(n)$ and $T_{par}(p,n)$ we denote the running times (in number of steps taken, for instance, depending on how the model accounts for time) of Seq and Par on worst-case inputs of size n with one, respectively, p processor-cores. The best possible and best known algorithms for solving a given problem are those with the best worst-case asymptotic complexities. As usual constants matter(!), but they will most often be ignored and hidden under $O, \Omega, \Theta, o, \omega$. Recall the definitions and rules for manipulating such expressions [21], and note that for parallel algorithms the worst-case time complexity is a function of two parameters, problem size n and number of processor-cores p. Saying that some $T_{par}(p,n)$ is in $O(f(p,n))$ then means that

$$\exists C > 0, \exists N, P > 0 : \forall n \geq N, p \geq P : 0 \leq T_{par}(p,n) \leq Cf(p,n)$$

and that some $T_{par}(p,n)$ is in $\Theta(f(p,n))$ that

$$\exists C_0, C_1 > 0 \exists N, P > 0 : \forall n \geq N, p \geq P : 0 \leq C_0 f(p,n) \leq T_{par}(p,n) \leq C_1 f(p,n) \quad .$$

Some typical sequential, best known/best possible worst-case complexities are [21]:

- $\Theta(\log n)$: Searching for an element in an ordered array of size n,
- $\Theta(n)$: Maximum finding in an unordered n element sequence, sum of the elements in an array, all prefix-sums over an array,
- $\Theta(n \log n)$: Comparison-based sorting of an n element array,
- $O(n^2)$: Matrix-vector multiplication, square matrix of order n,
- $O(n^3)$: Matrix-matrix multiplication, which is the best known to us in this lecture (but far from best known, see, e.g., [66]),
- $\Theta(n + m)$: Merging two ordered sequences of length n and m, graph search (DFS, BFS), and
- $O(n \log n + m)$: Dijkstra's Single-Source Shortest Problem algorithm on real, non-negative weight directed graphs with n vertices and m arcs with best known priority queue.

Regardless of how time per processor-core is accounted for, the time of the parallel algorithm Par when executed on p processor-cores is the time for the last processor-core to finish, assuming that all cores started at the same time (here, we make a lot of implicit assumptions, "same time" etc., that'll not be discussed further, but think about this). The rationale for this is twofold: Our problem is solved when the last processor has finished, and since our parallel system is dedicated, it has to be paid for until all processor-cores are again free for something else.

In parallel computing as a practical/empirical/experimental endeavor, Seq and Par denote concrete implementations of the algorithms, and $\mathrm{T}_{\mathrm{seq}}(n)$ and $\mathrm{T}_{\mathrm{par}}(p, n)$ measured running times for concrete, precisely specified inputs of size $O(n)$ on concrete and precisely specified systems. Designing measuring procedures and selecting inputs belong to empirical/experimental Computer Science, and is a highly non-trivial task; but one that will not be treated in great detail in these lectures. Suffice it to say that time is measured by starting the processor-cores simultaneously as far as this is possible, and accounting for time $\mathrm{T}_{\mathrm{par}}(p, n)$ by the last core to finish. Inputs may be either single, concrete inputs, or a whole larger set of inputs. Worst-case inputs may be difficult (impossible) to construct, and often also not interesting, so inputs are rather "typical" instances, "average-case" instances, randomly generated instances, inputs with particular structure, etc. (for recent criticism of and alternatives to worst-case analysis of algorithms, see [57]). The important thing for this lecture is that inputs, and generally, the whole experimental set-up is clearly described, so that claims and observations can be objectively verified (reproducibility).

1.2.2 *Speed-up*

We measure the gain of the parallel algorithm Par over the (best known) sequential algorithm for inputs of size $O(n)$ by relating the two running times. This is the fundamental notion of *speed-up*:

Definition 4 (Absolute Speed-up) *The* absolute speed-up *of parallel algorithm Par over sequential algorithm Seq (solving the same problem) for input of size $O(n)$ on p processor-cores is the ratio of sequential to parallel running time, i.e.,*

$$S_p(n) = \frac{T_{seq}(n)}{T_{par}(p,n)} .$$

The notion of speed-up is meaningful in both theoretical (analyzed, in some model) and practical (measured running times for specific inputs) settings. Often speed-up is analyzed by keeping the problem size n fixed and varying the number of processor-cores p. Sometimes (scaled speed-up, see later) both input size n and number of processor-cores p are varied.

As an example, a parallel algorithm Par with $T_{par}(n,p) = O(n/p)$ would have an absolute speed-up of $O(p)$ for a best known sequential algorithm with $T_{seq}(n) = O(n)$. If $T_{par}(p,n) = O(n/\sqrt{p})$ the speed-up would be only $O(\sqrt{p})$.

A speed-up of $O(p)$ that is independent of n is said to be *linear*, and linear speed-up of p (by measurement, or by analysis of constants) is said to be *perfect*. Perfect speed-up is rare and hardly achievable (sometimes provably not, an example is given in the next lecture).

1.2.3 *'"Linear speed-up is best possible"*

Linear speed-up is the best that is possible. The argument for this is that a parallel algorithm running on p dedicated cores can be *simulated* on a single core in time no worse than $pT_{par}(p,n)$. If the speed-up is more than linear, the simulated execution would run faster than the best known sequential algorithm for our problem, which cannot be (or: in that case, an even better algorithm would have been constructed). A different version of the argument is: Use the parallel algorithm to construct an even faster sequential algorithm.

Despite this argument, *super-linear speed-up* is sometimes reported (mostly in a practical setting) [24, 32]. If the reasons for this are algorithmic, it can only be that the sequential and parallel algorithms are, on specific inputs, not doing the same amount of work (see below). Randomized algorithms, where more and different coin tosses are possibly done by the parallel algorithm, can exhibit super-linear speed-up. But also apparently deterministic algorithms, like search algorithms, can exhibit this behavior, if the way the search space

is divided depends on the number of processor-cores leading the parallel algorithm to complete the search more than proportionally faster than the sequential algorithm (illustrated a bit more concretely in the slides).

The argument that a linear speed-up is best possible, also tells us that for any parallel algorithm, it holds that $\mathrm{T}_{\mathrm{par}}(p,n) \geq \frac{\mathrm{T}_{\mathrm{seq}}(n)}{p}$. The best possible parallel algorithm Par for the problem solved by Seq cannot run faster than $\mathrm{T}_{\mathrm{seq}}(n)/p$.

For any parallel algorithm Par on input of size $O(n)$, there is of course a limit on the number of processor-cores that can be sensibly employed. For instance, putting in more processor-cores than there is actual work to be done makes no sense, and some processors would sit idle. More on this in Section 1.2.4. Speed-up claims are therefore (or should be) qualified with the range of processor-cores for which they apply.

1.2.4 *Cost and Work*

Our dedicated parallel system with p processor-cores running Par is kept occupied for $\mathrm{T}_{\mathrm{par}}(p,n)$ time, and this is what we have to "pay" for. The *cost* of a parallel algorithm is accordingly defined as $p\mathrm{T}_{\mathrm{par}}(p,n)$ (the "area").

We often use the term *work*. The work of a sequential algorithm Seq on input of size n is the number of operations (of some kind) carried out by the algorithm (so "work is time", sequentially). The work of a parallel algorithm Par on p processor-cores is the total work carried out by the p cores, excluding time and operations spent idling by some processors or by processors that are not assigned, that is anything that the cores might be doing that is not strictly related to the algorithm. With a formal model like the PRAM, this can be given a precise definition ("work is operations carried out by assigned processors"), in more realistic settings, we have to be careful (which idle times should count, which not). The work of parallel algorithm Par on input n is denoted $\mathrm{W}_{\mathrm{par}}(p,n)$.

Definition 5 (Cost-optimal Parallel Algorithm) *A parallel algorithm Par is* cost-optimal*, if its cost $p\mathrm{T}_{\mathrm{par}}(p,n)$ is $O(\mathrm{T}_{\mathrm{seq}}(n))$ for a best known sequential algorithm Seq.*

Definition 6 (Work-optimal Parallel Algorithm) *A parallel algorithm Par with work $\mathrm{W}_{\mathrm{par}}(p,n)$ is* work-optimal*, if $\mathrm{W}_{\mathrm{par}}(p,n)$ is $O(\mathrm{T}_{\mathrm{seq}}(n))$ for a best known sequential algorithm Seq.*

Almost per definition, cost-optimal algorithms have linear speed-up. Work-optimal algorithms that are not cost-optimal can have linear speed-up for a smaller range of processors. The argument is that the work of the parallel algorithm is in the ball-park of the work of the best known sequential algorithm, but too many processors are used, some of which idle for too long. To avoid this, construct a better, cost-optimal algorithm with the same amount of work that runs on fewer processor-cores (simulation again).

As an extreme example, a parallel algorithm that executes a sequential algorithm on one out of p processors, is a work-optimal parallel algorithm (all but one processor idle), but it is definitely not cost-optimal. It trivially becomes cost-optimal by running it on only one processor, but this is of course uninteresting.

Algorithms that are not cost-optimal do not have linear speed-up. The PRAM algorithm of Theorem 1 takes $O(1)$ time with $O(n^2)$ processors and therefore has cost $O(n^2)$, which is far from $O(n)$. To determine the speed-up of this algorithm, we first have to observe that the algorithm can be simulated with $p \leq n^2$ processors in $O(n^2/p)$ parallel time steps. The speed-up is $S_p(n) = O(n/(n^2/p)) = p/n$. The speed-up is *not* independent of n, and actually decreases with n: The larger the input, the lower the speed-up.

The point of distinguishing work and cost, is to separate the discovery of parallelism from an all too specific assignment of the work to the actually available processors. A good, parallel algorithm is work optimal, and fast, when given enough processors (not the case in the situation in the example); a next design step is then to carefully assign the work to only as many processors as allowed to make the algorithm cost optimal (and have linear speed-up). The PRAM abstraction supports this strategy well: Processors can be assigned freely (with the **par**-construct), and the analysis focus on the number of operations actually done by the processors (the work).

1.2.5 *Relative Speed-up and Scalability*

While the absolute speed-up measures how well a parallel algorithm can improve over its best known sequential counterpart, it does not measure whether the parallel algorithm by itself is able to exploit the p processors well. This notion of *scalability* is captured by the relative speed-up.

Definition 7 (Relative Speed-up) *The* relative speed-up *of a parallel algorithm Par is the ratio of the parallel running time with one processor-core to the parallel running time with p processor-cores, i.e.,*

$$\mathrm{SRel}_p(n) \quad = \quad \frac{\mathrm{T}_{\mathrm{par}}(1,n)}{\mathrm{T}_{\mathrm{par}}(p,n)} \quad .$$

Assume that an arbitrary number of processors is available. Any parallel algorithm has, for any input of size $O(n)$, a fastest running time that it can achieve, denoted by $T\infty(n) = T_{par}(p', n)$ for some p'. Per definition, $T_{par}(p, n) \geq T\infty(n)$ for any number of processors p, and it thus holds that $SRel_p(n) = \frac{T_{par}(1,n)}{T_{par}(p,n)} \leq \frac{T_{par}(1,n)}{T\infty(n)}$.

The ratio $\frac{T_{par}(1,n)}{T\infty(n)}$ is called the *parallelism* of the parallel algorithm. It is clearly both the largest speed-up that can be achieved, as well as the largest number of processors for which linear, relative-speed-up can be achieved. If some number of processors p' larger than the parallelism is chosen, the definition says that $SRel_p(n) < p'$, that is, less than linear speed-up.

It is important to distinguish clearly between absolute and relative speed-up. The relative speed-up compares a parallel algorithm or implementation against itself, and expresses to what extent the processors are exploited well (linear, relative speed-up). Absolute speed-up compares the parallel algorithm against a baseline, and expresses how well it improves over the baseline. An algorithm may have excellent relative-speed up, but poor absolute speed-up. Is such a good algorithm? In any case, reporting only relative speed-up is grossly misleading, and should never be done. An absolute baseline must be defined, and absolute running times also stated. There are plenty of examples also in the scientific literature of basing claims on relative speed-ups only. For more on such pitfalls and misrepresentations, see for instance https://blogs.fau.de/hager/archives/5299.

The absolute speed-up compares the running time of the parallel algorithm against the running time of a best known or possible sequential algorithm. For such an algorithm it holds that $T_{seq}(n) \leq T_{par}(1, n)$, and therefore

$$S_p(n) \quad \leq \quad SRel_p(n) \quad .$$

The absolute speed-up is at most as large as the relative speed-up.

1.2.6 *Overhead and Load Balance*

A parallel algorithm for a computational problem usually performs more work than a corresponding best known sequential algorithm. Summarily, such work is termed *overhead*; thus overhead is work incurred by the parallelization that does not have to be done by the sequential algorithm. Beware, that this definition tacitly assumes that sequential and parallel algorithms are similar and can be compared ("extra work"); this is not always the case, sometimes a parallel algorithm is totally different from the best known sequential algorithm. Overheads can be caused by several factors, e.g.,

- communication and coordination,
- synchronization, or

- algorithmic overheads: extra or redundant work compared to a sequential algorithm.

Overheads are more or less inevitable, but if they are on the order of (within the bounds of) the sequential work, $O(\mathrm{T}_{\mathrm{seq}}(n))$, the parallel algorithm can still be cost- and work-optimal, and thus have linear, although not perfect speed-up. Often overheads increase with the number of processors p, giving a limit on the number of processors that can be used and still have linear speed-up. If overheads are always larger than the sequential work, the parallel algorithm will never have linear speed-up.

The overheads caused by communication and synchronization between processor-cores are often significant, and in later lectures, we will introduce a simple model for accounting for communication operations. Between communication operations, the processor-cores operate independently (although they could interfere indirectly through the memory and cache system, which will be discussed also in a later lecture) on parts of the problem. The intervals between communication and synchronization operations is sometimes referred to as the *granularity* of the parallel algorithm. A parallel computation in which communication and synchronization occurs rarely is called *coarse grained*, if communication and synchronization occurs frequently, *fine grained*. These are relative (and vague) terms. Machine models that can actually support fine grained algorithms, are also called fine grained. The PRAM is an extreme example: The processors can (and often do) communicate (via the shared memory) in every step, and they are lock-step synchronized.

In a parallel algorithm, the processors may not perform the same amount of work, and/or have different amounts of overhead. If we for the moment let $\mathrm{T}_{\mathrm{par}}(i,n)$ denote the time taken by some processor $i, 0 \leq i < p$, the *load imbalance* is defined as $\max_{0 \leq i,j < p} |\mathrm{T}_{\mathrm{par}}(i,n) - \mathrm{T}_{\mathrm{par}}(j,n)|$. Too large load imbalance is another reason that a parallel algorithm may have a too small (or non-linear) speed-up.

Good load balance means that $\mathrm{T}_{\mathrm{par}}(i,n) \approx \mathrm{T}_{\mathrm{par}}(j,n)$ for all processors i, j, and achieving this is called *load balancing*. Load balancing is always an issue in designing a parallel algorithm, explicitly by the construction of the algorithm, or implicitly by taking steps later to ensure a good load balance. We distinguish between *static load-balancing*, where the amount of work to be done can be divided up front among the processors, and *dynamic load balancing*, where the processors have to communicate and exchange work during the execution of the parallel algorithm. Static load balancing can be further subdivided into *oblivious, static load-balancing* where the problem can be divided over the processors based on the input size and structure alone, but regardless of the actual input, and *adaptive, problem-dependent, static load-balancing* where the input itself is needed to divide the work, and some preprocessing may be required. Some aspects of the load balancing problem (work-stealing, loop scheduling) will be discussed later in these lectures, but load balancing per

se is a too large subfield of parallel computing to be treated in much detail in these lectures.

Problems and algorithms where the input and work can be statically distributed to the processors, and where no further explicit interaction is required are called either *embarrassingly parallel*, *trivially parallel*, or *pleasantly parallel*. This are the best (but uninteresting) cases of easily parallelizable problems with linear or even perfect speed-up (although the realization that the problem is such could be non-trivial and unpleasant).

1.2.7 *Amdahl's Law*

Gene Amdahl made a simple observation on how to speed up programs [4], which when applied to parallel computing gives severe bounds on the speed-up that a parallel algorithm can achieve.

Theorem 4 (Amdahl's Law) *Assume that the work performed by sequential algorithm* Seq *can be divided into a strictly sequential fraction* $s, 0 < s \leq 1$*, independent of* n*, that cannot be parallelized at all, and a fraction* $r = (1 - s)$ *that can be perfectly parallelized. The parallelized algorithm is* Par*. The maximum speed-up that can be achieved by* Par *over* Seq *is* $1/s$*.*

The proof is straightforward.
Since $\mathrm{T}_{\mathrm{par}}(p, n) = s\mathrm{T}_{\mathrm{seq}}(n) + \frac{(1-s)\mathrm{T}_{\mathrm{seq}}(n)}{p}$, we get

$$\begin{aligned} \mathrm{S}_p(n) &= \frac{\mathrm{T}_{\mathrm{seq}}(n)}{s\mathrm{T}_{\mathrm{seq}}(n) + \frac{(1-s)\mathrm{T}_{\mathrm{seq}}(n)}{p}} \\ &= \frac{1}{s + \frac{1-s}{p}} \\ &\to \frac{1}{s} \text{ for } p \to \infty \quad . \end{aligned}$$

Amdahl's Law is devastating. Even the smallest constant, sequential fraction of the algorithm to be parallelized will limit and eventually kill speed-up. A sequential fraction of 10%, or 1%, sounds reasonable and harmless, but limits the speed-up to 10, or 100, no matter what else is done, and no matter how many processors are invested.

A sequential algorithm which falls under Amdahl's Law therefore cannot be used as the basis of a good, parallel algorithm: The speed-up will be restricted. Amdahl's Law is therefore rather an analysis tool: If it turns out that there is a (large) fraction of the algorithm at hand that cannot be parallelized, we have to look for a better algorithm, which means coming up with a new idea. This is what makes parallel computing a creative activity.

Typical victims of Amdahl's Law are:

- Input/output: For linear work algorithms, reading the input and writing the output will take $\Omega(n)$, and thus be a constant fraction of $O(n)$.
- Sequential preprocessing: As above.
- Maintaining sequential data structures, in particular sequential initialization, could easily turn out to be a fraction of the total work.
- Hard-to-parallelize parts, that are done sequentially (might look innocent, just a small part): If such parts take a constant fraction of the total work, Amdahl's Law applies.
- Long chains of dependent operations, not necessarily at the same processor-core. This will be a theme of the next lecture.

When analyzing and benchmarking parallel algorithms, input/output is often disregarded when accounting for sequential and parallel time. The defensible reason for this is that we are interested in how the "core" algorithm performs (speeds up), under the assumption that the input has already been read and distributed. In these lectures, our algorithms are small parts (building blocks) of larger applications, and thus in this larger context would not need input/output: The data are already where they should be, and also results do not have to be specially output but should just stay for the next building block.

In a good parallel algorithm, not falling victim to Amdahl's Law, the sequential part $s(n)$ will not be a constant fraction of the total work, but depend on, and decrease with n. If such is the case, Amdahl's Law, does not apply. Instead, a good speed-up can be achieved with large enough inputs. Parallel computing is about solving large, work-intensive problems.

1.2.8 *Efficiency and Weak Scaling*

As observed, there is for any parallel algorithm on input of size $O(n)$ always a fastest possible time, $\mathrm{T}\infty(n)$, that the algorithm can achieve. Thus, the parallel running time of an algorithm with good, linear speed-up (up to the number of processor-cores determined by the parallelism), can be written as $\mathrm{T}_{\mathrm{par}}(p,n) = O(T(n)/p + t(n))$, that is as a parallelizable term $T(n)$ and a non-parallelizable term $t(n) = \mathrm{T}\infty(n)$. If speed-up is not linear, the parallel running time is instead something like $\mathrm{T}_{\mathrm{par}}(p,n) = O(T(n)/f(p) + t(n))$ with strictly $f(p) < p$ ($f(p)$ in $o(p)$).

If we compare against a sequential algorithm with $\mathrm{T}_{\mathrm{seq}}(n) = O(T(n))$ (and $O(T(n) + t(n)) = O(T(n))$), a parallel algorithm where $t(n)/T(n) \to 0$ as $n \to \infty$ is also good, and can have linear speed-up for large enough n. The speed-up is namely

$$\mathrm{S}_p(n) = \frac{\mathrm{T}_{\mathrm{seq}}(n)}{\mathrm{T}_{\mathrm{par}}(p,n)} = O(\frac{T(n)}{T(n)/p + t(n)}) = O(\frac{1}{1/p + t(n)/T(n)}) \to O(p)$$

as n increases. This is called *scaled speed-up*, and the faster $t(n)/T(n)$ converges, the faster the speed-up converges. Against Amdahl's Law, the sequential part $t(n)$ should be as small as possible, and increase more slowly with n than the parallelizable part $T(n)$.

A good way (which we use throughout these lectures) is to state the performance of a work-optimal parallel algorithm as $\mathrm{T}_{\mathrm{par}}(p,n) = O(T(n)/p + t(n,p))$ with the assumption that $t(n,p)$ is $O(T(n))$ for fixed p, and $\mathrm{T}_{\mathrm{seq}}(n) = O(T(n))$. That is, we allow also the non-parallelizable part to depend on p, thus $t(n,p)$ instead of just $t(n)$. Often, however, this is just $t(n)$, independent of p.

The *efficiency* of a parallel algorithm Par is measured by comparing Par against the best possible parallelization of Seq.

Definition 8 (Parallel Efficiency) *The efficiency* $\mathrm{E}_p(n)$ *of Par compared to Seq is defined as*

$$\mathrm{E}_p(n) = \frac{\mathrm{T}_{\mathrm{seq}}(n)}{p\mathrm{T}_{\mathrm{par}}(p,n)} = \frac{\mathrm{S}_p(n)}{p} \quad .$$

As worked out in the definition, the efficiency is also the achieved speed-up divided by p, and the sequential time divided by the cost of the parallel algorithm.

It holds that

- $\mathrm{E}_p(n) \leq 1$.
- If $\mathrm{E}_p(n) = e$ for some constant e, the speed-up is linear.
- Cost-optimal algorithms have constant efficiency.

If an algorithm does not have constant efficiency (and speed-up independent of n), we can aim to maintain some desired, constant e efficiency by instead increasing the problem size n with the number of processors p. This is the notion of *iso-efficiency*.

Definition 9 (Weak Scalability) *A parallel algorithm Par is said to be* weakly scaling *relative to sequential algorithm Seq if for a desired, constant efficiency e there is a slowly growing function* $f(p)$ *such that the efficiency is* $\mathrm{E}_p(n) = e$ *for* n *in* $\Omega(f(p))$. *The function* $f(p)$ *is called the* iso-efficiency function.

How slowly should $f(p)$ grow? A possible answer is by another definition of weak scaling.

Definition 10 (Weak Scalability (alternative)) *A parallel algorithm Par is said to be* weakly scaling *relative to sequential algorithm Seq if by keeping the average work per processor* $\mathrm{T}_{\mathrm{seq}}(n)/p$ *constant at w, the running time of the parallel algorithm* $\mathrm{T}_{\mathrm{par}}(p,n)$ *remains constant. The input size scaling function is* $g(p) = {\mathrm{T}_{\mathrm{seq}}}^{-1}(pw)$.

The iso-efficiency function $f(p)$, which tells how n should grow as a function of p to maintain constant efficiency, should not grow faster than the input size scaling function $g(p)$, which tells how much n can at most grow if the parallel time is to be kept constant: $f(p)$ should be $O(g(p))$. Note, however, that if the sequential running time is more than linear, keeping constant efficiency requires n to increase faster than allowed by constant work weak scaling. For such algorithms, constant work is maintained with decreasing efficiency.

1.2.9 *Scalability Analysis*

How well is a parallel algorithm or implementation now performing against a sequential counterpart for the problem that we are interested in? *Scalability analysis* examines this, theoretically and practically.

- Strong scaling analysis: Keep n constant. The algorithm is *strongly scalable* (up to some maximum number of processors, as expressed by the parallelism) if the parallel time decreases proportionally to p (linear speed-up).
- Weak scaling analysis: Keep the average work per processor constant by increasing n. The algorithm *weakly scalable* if the parallel running time remains constant.

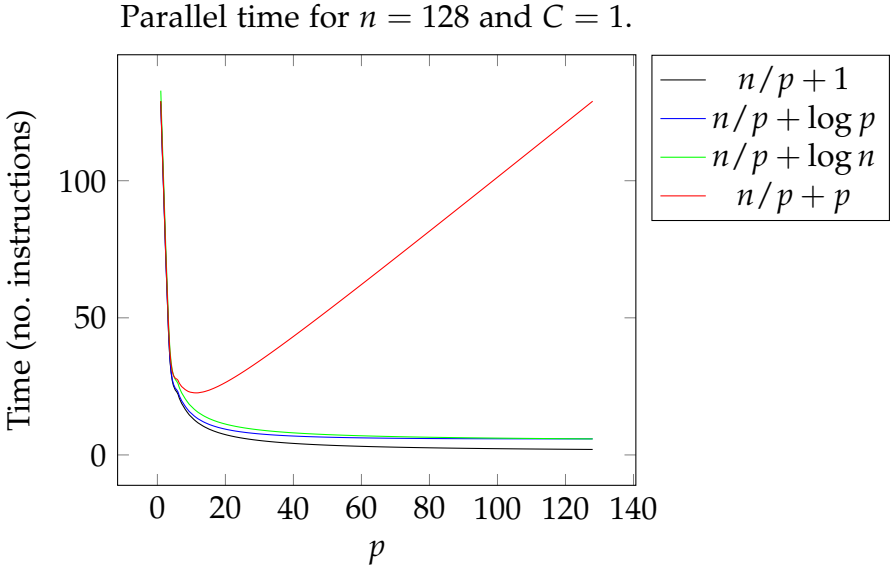
1.2.10 *Examples*

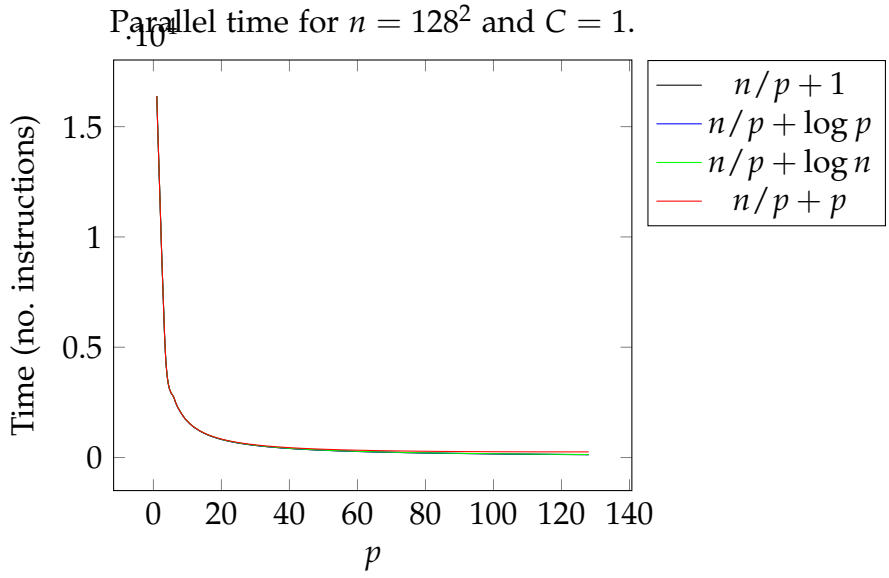
It is illustrative(!) to strengthen intuition to visualize parallel running time, (absolute) speed-up, efficiency, and iso-efficiency as functions of the number of processors put into solving a problem of size n (for different n). Let some such problems be given with best known sequential running times $O(n) \leq cn$, $O(n \log n) \leq c(n \log n)$, and $O(n^2) \leq cn^2$ as seen many times in these lectures, for some bounding constant $c, c > 0$ (the notation is sloppy: We mean that the constant of the dominating term hidden within the O is c).

We first assume that the linear $O(n)$ algorithm has been parallelized by algorithms running work-optimally in $O(n/p+1) \leq C(n/p+1)$, $O(n/p+$

$\log p) \leq C(n/p + \log p)$, $O(n/p + \log n) \leq C(n/p + \log n)$, and $O(n/p + p) \leq C(n/p + p)$, respectively, for some bounding constant $C, C > 0$: Also many examples of such algorithms have been (and will be) seen in these lectures.

We first assume that the bounding constants in sequential and parallel algorithms are roughly the same, "in the same ballpark", and normalize both constants to $c = C = 1$. We plot the parallel running time as functions of the number of processors p for $1 \leq p \leq 128$, and take $n = 128, 128^2$, respectively; these are really "small" problems for a linear time algorithm, $128^2 = 16K$ (and $128^3 = 2M$). The running times are shown in the following two plots.



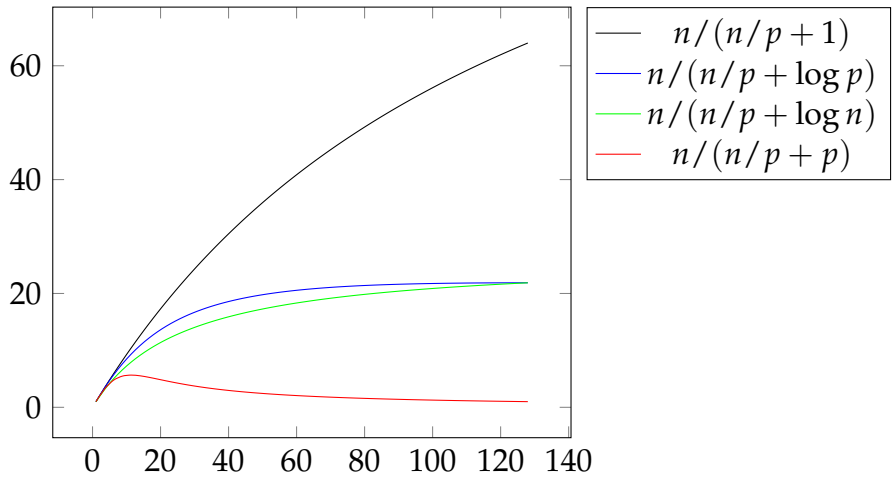


Running time plots do not very well distinguish the four different parallel algorithms; for the larger problem size, $n = 128^2$, there is virtually no difference to be seen. The shape of the curves for these linearly (perfect) scaling algorithms is hyperbolic (like $1/p$). Interesting is the parallel algorithm with

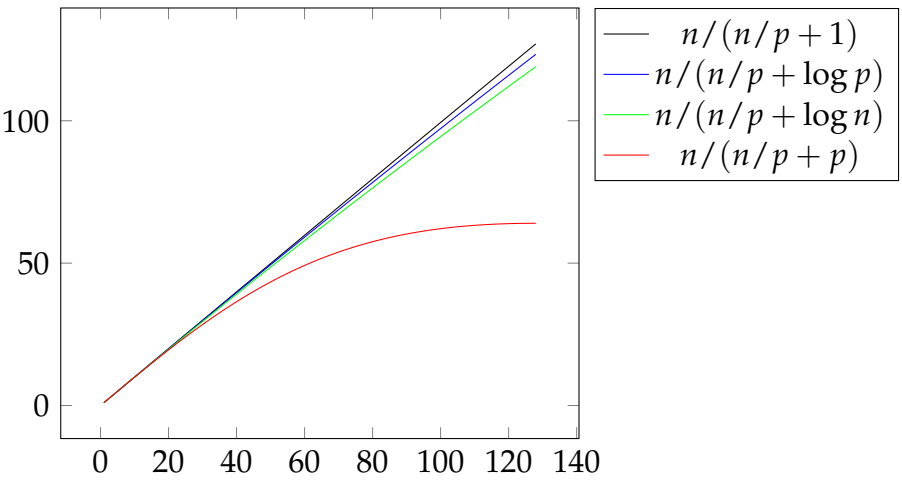
running time $O(n/p + p)$. For the small input with $n = 128$, running time decreases until about $p = 10$ processors, and then increases. Indeed the best possible running time of this algorithm is $T\infty(n) = \sqrt{n}$, and the parallelism is also $n/\sqrt{n} = \sqrt{n}$. This can be seen by minimizing $C(n/p + p)$ for p, which can be done by solving $Cn/p = Cp$ for p, giving $p = \sqrt{n}$.

Plotting instead the absolute (unit-less) speed-up against the linear (best known) $O(n)$ algorithm (with $c = C = 1$) can highlight the actually different behavior of the four parallel algorithms. We plot for three problem sizes $n = 128, 128^2, 128^3$.

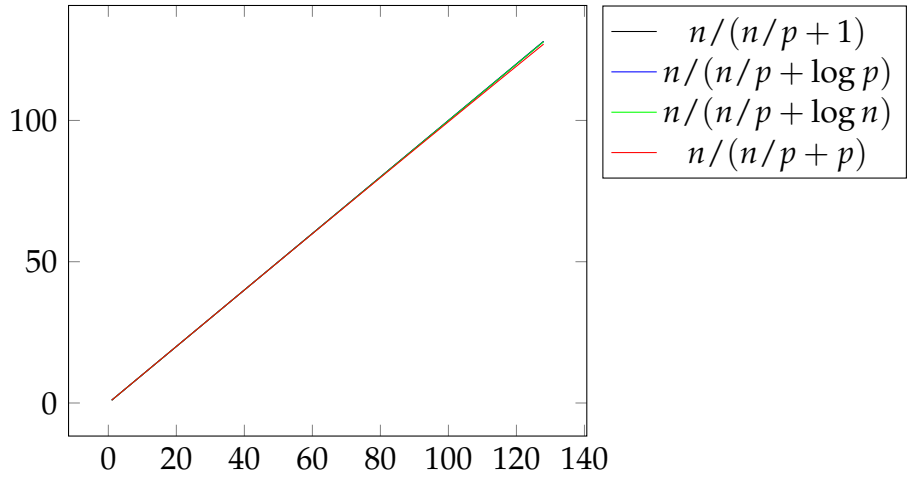
Speed-up for $n = 128$ and $c = C = 1$.



Speed-up for $n = 128^2$ and $c = C = 1$.



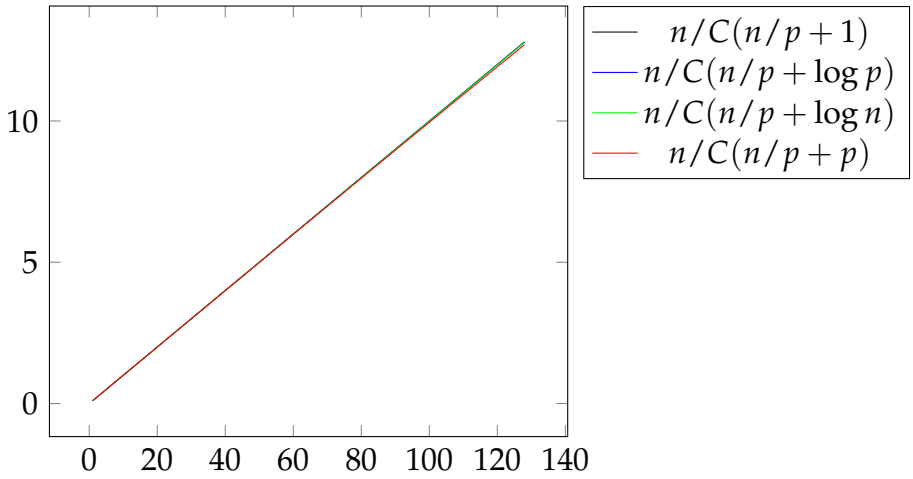
Speed-up for $n = 128^3$ and $c = C = 1$.



Speed-fup for the small problem size $n = 128$ is not impressive, and as we would like, except for the first parallel algorithm, but this changes drastically and impressively with as n grows. Indeed, for the "large" $n = 128^3$, all four parallel algorithms show perfect speed-up of almost 128 for $p = 128$.

If there is a difference in the bounding constants between sequential and parallel algorithms, say $c = 1$ and $C = 10$ which means that the parallel algorithm is a constant factor of 10 slower than the sequential one when executed with only one processr, speed-ups change proportionally:

Speed-up for $n = 128^3$ and $c = 1, C = 10$.

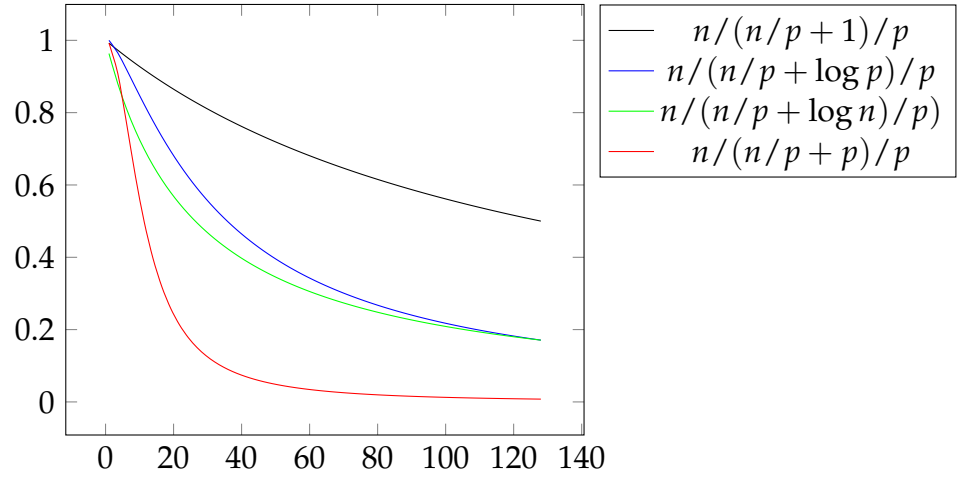


Here, only $1/C$th of the processors are doing productive work in comparison to the sequential algorithm. Constants *do* matter, and it is obviously important that sequential and parallel algorithms have leading constants in the same ballpark; otherwise a proportional part of the processors are somehow wasted.

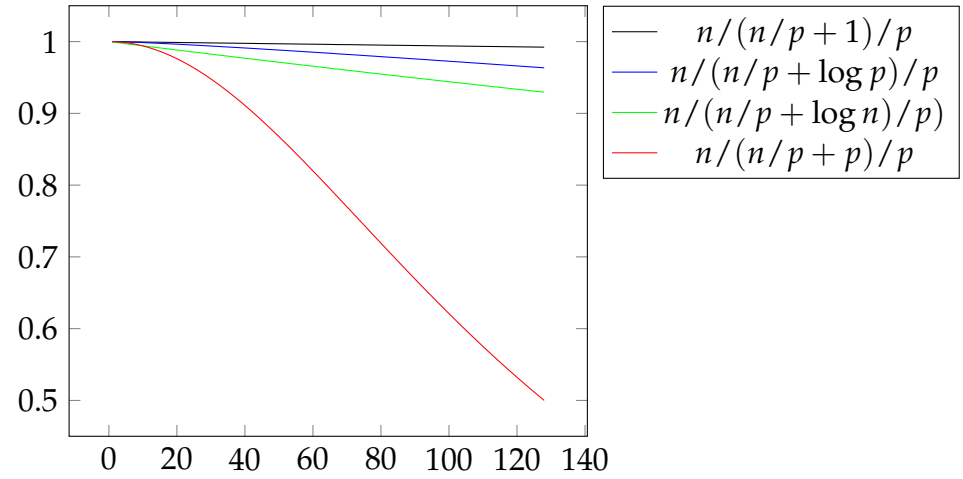
The parallel efficiency indicates how well the parallel algorithms behave in comparison to a best possible parallelization with running time cn/p. The

(unit-less) parallel efficiencies for the four parallel algorithms are plotted for $n = 128, 128^2, 128^3$.

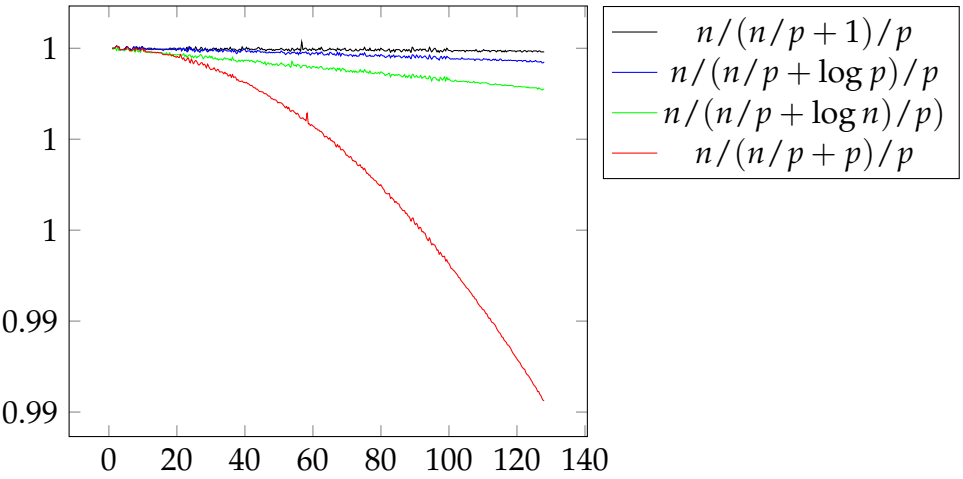
Parallel efficiency for $n = 128$ and $c = C = 1$.



Parallel efficiency for $n = 128^2$ and $c = C = 1$.



Parallel efficiency for $n = 128^3$ and $c = C = 1$.



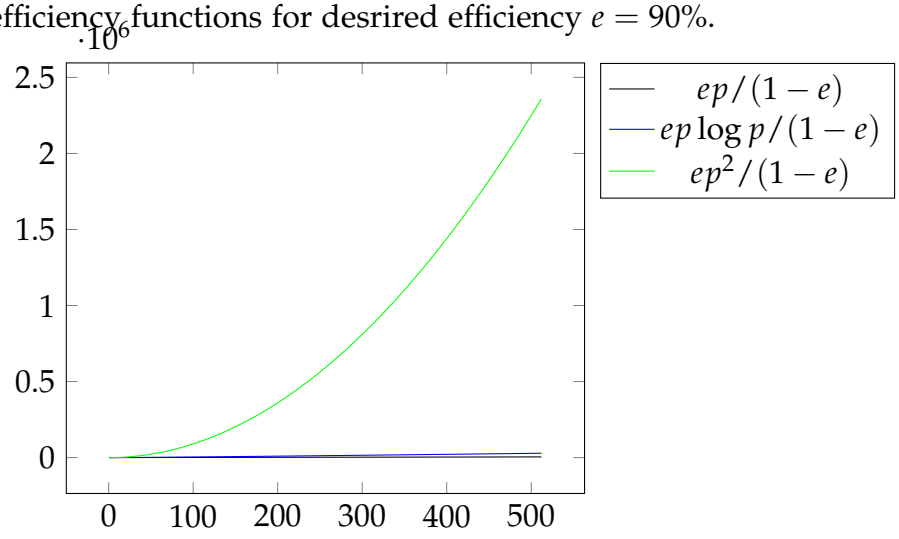
Indeed, for work-optimal parallelizations, the efficiency improves greatly with growing problem size n, and is already for $n = 128^3$ very close to 1 for all of the four parallelizations. The iso-efficiency functions more precisely tells how problem size must increase with p in order to maintain a given constant efficiency e. We calculate the iso-efficiency functions for the parallel algorithms as follows.

- For parallel running time $n/p + 1$ and desired efficiency e, we have $e = n/(p(n/p + 1)) = n/(n + p) \Leftrightarrow e(n + p) = n \Leftrightarrow n = ep/(1 - e)$.
- For parallel running time $n/p + \log p$ and desired efficiency e, we have $e = n/(p(n/p + \log p)) = n/(n + p \log p) \Leftrightarrow e(n + p \log p) = n \leftrightarrow n = ep \log p/(1 - e)$
- For parallel running time $n/p + p$ and desired efficiency e, we have $e = n/(p(/n/p + p)) = n/(n + p^2) \Leftrightarrow e(n + p^2) = n \Leftrightarrow n = ep^2/(1 - e)$

The case with parallel running time $n/p + \log n$ is more difficult. The efficiency calculation gives $e = n/(p(n/p + \log n)) = n/(n + p \log n)$ and therefore $n/\log n = ep/(1 - e)$, for which we do not know an analytical solution.

We plot the three analytical iso-efficiency functions below for $p, 1 \leq p \leq 512$ and $e = 90\%$.

Iso-efficiency functions for desrired efficiency $e = 90\%$.



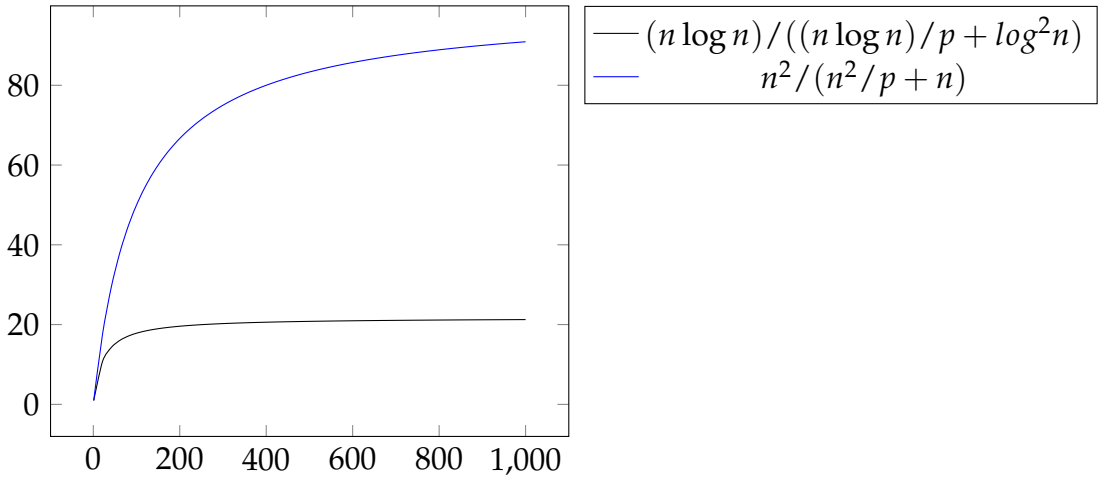
For the first two parallel algorithms, the iso-efficiency function is indeed "slowly growing", and according to one of the definitions of weak scalability, these algorithms are both strongly and weakly scaling. The last function, where the iso-efficiency function is in $O(p^2)$, it is a matter of taste whether this is still slowly growing. In the speed-up plots, we indeed let n grow exponentially $n = 128, 128^2, 128^3$, and the speed-up for the latter algorithms was excellent.

We now look at the non-linear time sequential algorithms. The $O(n \log n)$ algorithm could be a sorting algorithm (mergesort, say), could be paarallelized

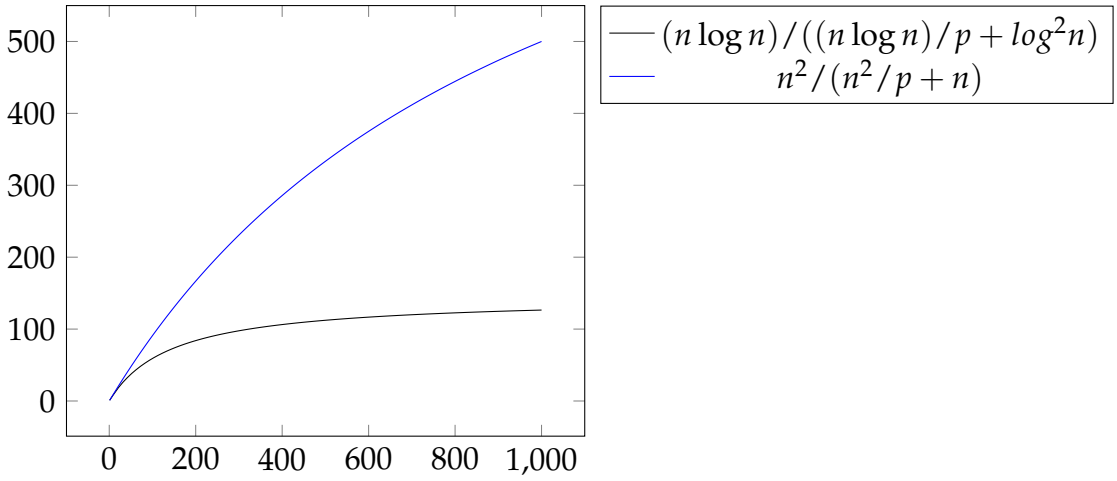
with running time $O((n \log n)/p + \log^2 n)$. The second algorithm is perhaps matrix-vector multiplication, which can easily be done work-optimally in parallel time $n^2/p + n$ (and easily also faster).

The corresponding speed-ups for $n = 100, 1\,000, 10\,000, 100\,000$ and $p, 1 \leq p \leq 1000$ are shown below.

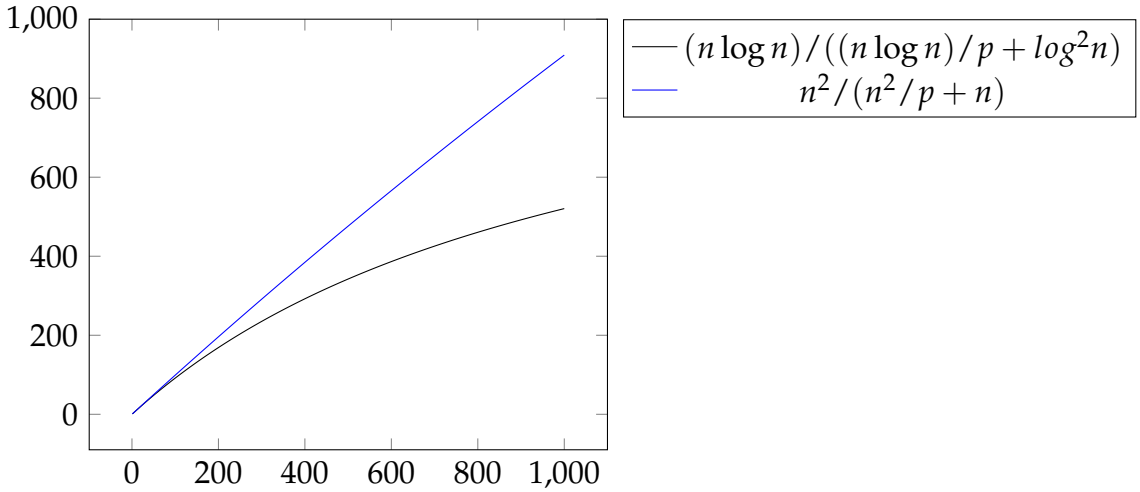
Speed-up for $n = 100$ and $c = C = 1$.



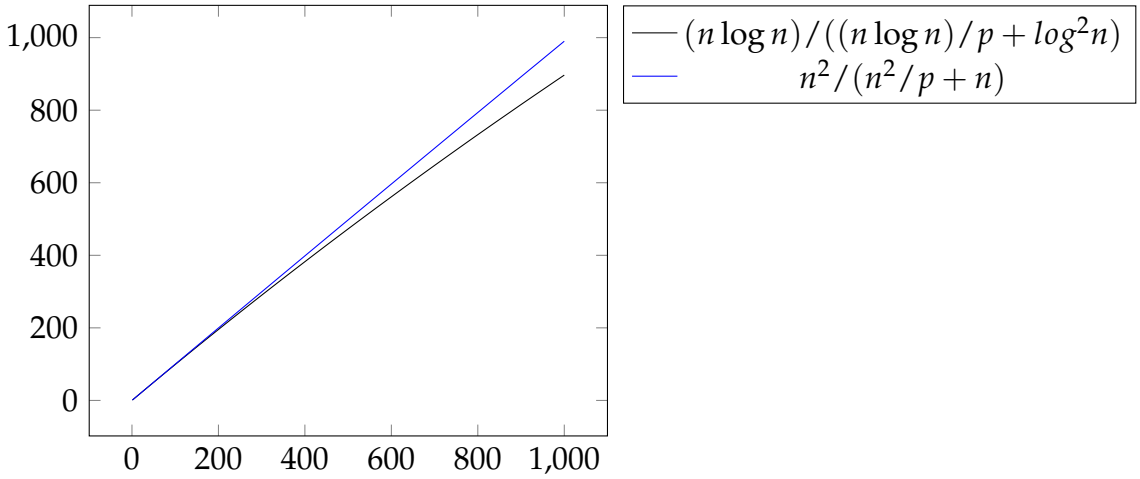
Speed-up for $n = 1\,000$ and $c = C = 1$.



Speed-up for $n = 10\,000$ and $c = C = 1$.



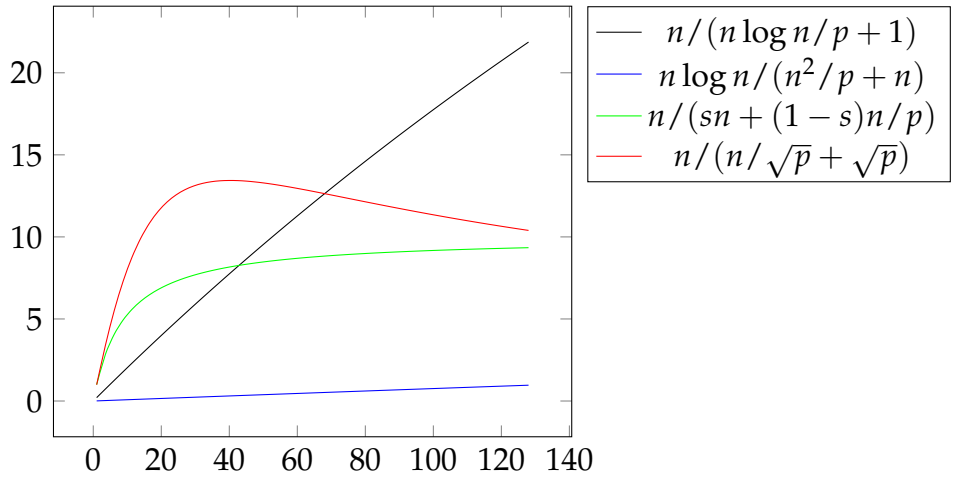
Speed-up for $n = 100\,000$ and $c = C = 1$.



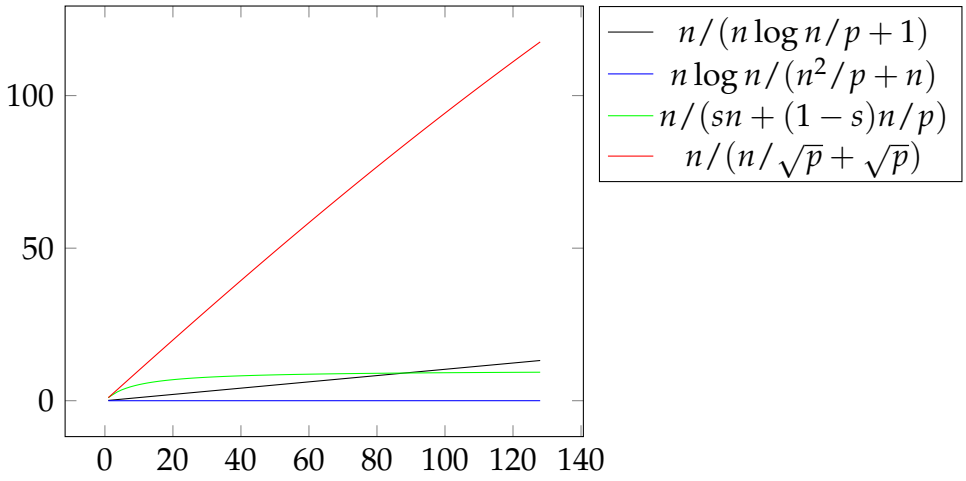
The parallelization of the low complexity algorithm with sequential running time $O(n \log n)$ does not scale as well as the other algorithm. For an $O(n^2)$ algorithm, and input of size $n = 100\,000$ is already large, and we did not plot for larger n here. However, both algorithms clearly approach a perfect speed-up with growing n.

Finally, we illustrate what happens with non-work optimal parallel algorithms. Assume we have parallel algorithms with running time $O(n \log n/p + 1)$ relative to a linear time sequential algorithm, an $O(n^2/p + n)$ parallel algorithm relative to an $O(n \log n)$ best possible sequential algorithm (the parallel algorithm could be a parallelized counting sort as will be seen later), and an Amdahl case where the parallel algorithm has a sequential fraction $s, 0 < s < 1$ and parallel running time $O(sn + (1 - s)n/p)$. Lastly, a parallel algorithm with a running time of $O(n/\sqrt{p} + \sqrt{p}) = O(n/(n\sqrt{p}/p + \sqrt{p})$ relative to an algorithm that solves an $O(n)$ problem.

Speed-up for $n = 128$ and $c = C = 1$ and sequential fraction $s = 0.1$.



Speed-up for $n = 128^2$ and $c = C = 1$ and sequential fraction $s = 0.1$.



The two plots illustrate the Amdahl case well: Speed-up is bounded by $1/s$ (here 10 for $s = 10\%$) independently of n. The first two algorithms have a diminishing speed-up with increasing n. These two algorithms have parallel work determined by the problem size which is asymptotically larger than the sequential work. For the last algorithm, the parallel work increases "slowly" by a factor of $\sqrt{p}$ with p, and therefore the speed-up of this algorithm does indeed improve with increasing problem size n, but is $o(p)$ and not linear.

1.3 THIRD BLOCK (1-2 LECTURES)

This lecture takes a closer look at the way (parallel) work may be structured. The important structures discussed are work expressed as dependent tasks, and work expressed as loops of independent iterations. The latter can be seen

as an example of a recurring expression of computations in pseudo-code and actual programs: a *pattern*. The last part of the slides gives some examples of parallel algorithmic design patterns for which good parallelizations are known. Parallel design patterns can provide (whether explicitly or implicitly) useful guidance for building applications and sometimes serve as concrete building blocks. The last part of the slides is additional material (for casual reading), and will not be covered in a lecture.

1.3.1 *Directed Acyclic task Graphs*

A *Directed Acyclic (task) Graph (DAG)*, $G = (V, E)$, consists of a set of *tasks*, $t_i \in V$, which are sequential computations that will not be analyzed further (sometimes also called *strands*). Tasks are connected by directed *dependency edges*, $(t_i, t_j) \in E$. An edge (t_i, t_j) means that task t_j is *directly dependent* on task t_i, and cannot be executed before task t_i has completed, for instance because the input data for task t_j are produced as output data by task t_i. In general, a task t_j is *dependent* on a task t_i if there is a directed path from t_i to t_j in G. If there is neither a directed path from t_i to t_j, nor a directed path from t_j to t_i in G, the two tasks t_i and t_j are said to be *independent*. Independent tasks could possibly be executed in parallel, if processor-cores are available for this: Neither task needs input from the other, nor produces output to the other. A task t_i may produce data to more than one other task, that is there may be several outgoing edges from t_i. Likewise, a task t_j may need immediate input from more than one task, that is there may be several incoming edges to t_j. Since G is acyclic, there is at least one task t_r in G with no incoming edges; such tasks are called *root* or *start* tasks. Likewise, there is at least one task t_f with no outgoing edges. Such tasks are called *final*.

Many computations can be pictured as task graphs. The first example in the lecture slides is an execution of recursive Quicksort where the tasks are the computations done in pivot selection and partitioning. In later lectures, we will see how tasks graphs suitable for parallel execution can be generated dynamically (OpenMP tasks; Cilk). Another, often encountered type of task DAG is the *fork-join* DAG: A sequence of fork-join tasks, each of which has a number of forked tasks that are all connected to the next fork-join task (see lecture slides). This is the standard structure of OpenMP programs.

For computations structured as task graphs, there is normally a single start task taking input of size $O(n)$, and a single, final tasks producing the results of the computation. In a dynamic setting, the task graph typically depends on the input, which will be emphasized by writing $G(n)$.

Each task t_i has an associated amount of work, $T(t_i)$ (that typically also depends on n). The total amount of work of a given task graph $G = (V, E)$ with k tasks $t_0, t_1, \ldots, t_{k-1}$ is denoted by $T_1(n) = \sum_{i=0}^{k-1} T(t_i)$. We will again compare against a best known sequential algorithm for the problem we are solving, so $T_1(n) \geq \mathrm{T}_{\mathrm{seq}}(n)$.

Doing a computation as specified by a task graph G sequentially, by a single processor-core, amounts to the following. Pick a task t with no incoming edges, and execute it. Remove all outgoing edges (t, t') from G. Continue this process until there are no more tasks in G. Since G is acyclic, there is a least one root task from which the execution can be started, which will result in at least one task now with no incoming edges, etc. (if not, G would not be acyclic). Sequential execution of a task graph therefore amounts to executing the tasks (nodes) in some *topological order*. Any DAG has a topological order (which can be determined sequentially in $O(k)$ time steps). A task in the sequential execution that has become eligible for execution by having no incoming edges is said to be *ready*. Since all tasks of G are executed, each task exactly once, and there is always at least one ready task after completion of a task, the time taken for the sequential execution is $O(T_1(n))$.

Imagine that several processor-cores are available. A parallel execution of a computation specified by a task graph G could proceed as follows. Pick a ready task. If there is a processor-core that is not busy executing, assign the task to this core. When a task is completed, remove all outgoing edges, possibly giving rise to further, ready tasks (but also possibly not, tasks may have many incoming edges). Such a process is called a *schedule*. The important property of a schedule is that dependencies are respected (a task is not executed before all incoming edges have been removed, that is dependencies resolved and data made available), and processors are respected (at no time, a core is assigned more than one task; but a times, cores may be unassigned and idle).

We are interested in the time taken to execute the work $T_1(n)$ with some schedule with p processors. This is given by the time for the last task to finish. We denote the execution time by a (for now not further specified) p processor schedule by $T_p(n)$, and are of course interested in finding fast schedules.

No matter how scheduling is done, the total amount of work $T_1(n)$ can never be completed faster than $T_1(n)/p$, the best possible parallelization. Also, no matter how scheduling is done, tasks that are dependent on each other must be executed in order. Consider a heaviest path (one with the most total work) from the start task to the final task, $(t_r, t_1, \ldots, t_f)$, and define $\mathrm{T}\infty(n) = T(t_r) + T(t_1) + \ldots + T(T_f)$ as the work of such a heaviest path. Clearly, for any schedule, $T_p(n) \geq \mathrm{T}\infty(n)$. These two observations are often summarized as follows:

- *Work Law*: $T_p(n) \geq T_1(n)/p \geq \mathrm{T}_{\mathrm{seq}}(n)/p$,
- *Depth Law*: $T_p(n) \geq \mathrm{T}\infty(n)$.

The work on a heaviest path in a task graph G is often also called the *span*, or the *depth* of the DAG. A heaviest path is commonly referred to as a *critical path* with *length* $\mathrm{T}\infty$.

The Work Law can sometimes be tightened by the following observation. With p processor-cores, assign one core permanently to the work on a critical

path. This leaves $p-1$ processor-cores to work on the remaining work, which in the best case (Work Law) can be sped up by a factor of $p-1$. That is, for any p processor schedule it holds that $T_p(n) \geq \frac{T_1(n)-\mathrm{T}\infty(n)}{p-1}$. Let us call this observation the *refined Work Law*,

As an example, consider a fork-join DAG with start and final tasks t_r and t_f, with $T(t_r) = 1$ and $T(t_f) = 1$. The start task forks to a heavier task t_1 with $T(t_1) = 4$, and, say, 27 light tasks with one unit of work. All forked tasks join at the final task. Thus, $T_1(n) = 33$, and $\mathrm{T}\infty(n) = 1 + 4 + 1 = 6$. With $p = 3$, the Work Law says that $T_p(n) \geq 33/3 = 11$, the Depth Law that $T_p(n) \geq 6$, and the refined Work Law that $T_p(n) \geq (33-6)/(3-1) = 27/2 = 13$ (rounding down). The (relative) speed-up with p processors is therefore at most $33/13$.

Using what we saw in the previous lecture, for any schedule it holds that the speed-up is bounded as follows:

$$\mathrm{S}_p(n) = \frac{\mathrm{T}_{\mathrm{seq}}(n)}{\mathrm{T}_{\mathrm{par}}(p,n)} \leq \frac{T_1(n)}{T_p(n)} \leq \frac{T_1(n)}{\mathrm{T}\infty(n)} \quad .$$

The *parallelism* $\frac{T_1(n)}{\mathrm{T}\infty(n)}$ is therefore an upper bound on the achievable speed-up, and also gives the largest number of processor-cores for which linear speed-up could be possible.

The *critical path analysis* (finding the longest chain of sequential work over all processors), the Depth Law, is an important tool to analyze the potential for parallelizing a computation when thinking of the computation as a task graph. If the Depth Law reveals that the critical path $\mathrm{T}\infty(n)$ is a constant fraction of $T_1(n)$, Amdahl's Law applies. As always, this is a sign that a better algorithm and a better DAG must be found.

We now consider a specific scheduling strategy, so-called *greedy scheduling*. A greedy scheduler assigns a ready task to an available processor as soon as possible (task ready or processor available), meaning that a processor-core is idle only in the case there is no ready task. Greedy schedules have a nice upper bound on the achieved running time, which is captured in the following theorem.

Theorem 5 (Two-optimality of greedy scheduling) *Let $T_p(n)$ be the execution time of a DAG $G(n)$ with any greedy schedule on p processors, and let $T_p^*(n)$ be the execution time with a best possible p processor schedule. It holds that*

$$\begin{aligned} T_p(n) &\leq \lfloor T_1(n)/p \rfloor + \mathrm{T}\infty(n) \\ &\leq 2T_p^*(n) \quad . \end{aligned}$$

The proof can be sketched as follows. Divide the work of the scheduler into discrete steps. A step is called complete if all processor-cores are busy on some tasks, and incomplete if some cores are idle (because there is no ready task in that step). Then, the number of complete steps is bounded by $\lfloor T_1(n)/p \rfloor$ (if there were more, more than the total work $T_1(n)$ would have been executed),

and the number of incomplete steps by $T\infty(n)$ (each incomplete step reduces the work on a critical path). The Work and the Depth Law hold for any p processor schedule, in particular for a best possible schedule, by which the last upper bound follows. The theorem therefore states that the execution time that can be achieved by a greedy schedule is bounded by two times what can be achieved by a best possible schedule, a guaranteed two-approximation!

Neither the definition of greedy schedules nor the theorem says how a greedy scheduler can or should be implemented. But if it can be shown by some means that some proposed scheduling algorithm is greedy, the greedy scheduling theorem says that the running time is within a factor two of best possible.

These lectures will briefly touch on *work-stealing* which is a decentralized, randomized, greedy scheduling strategy for certain kinds of DAGs (like the one shown for Quicksort: strict, spawn-join DAG's) [6].

Some parallel programming models make it possible to (implicitly) construct task graphs. We will see in a later lecture how to parallelize Quicksort and other algorithms with OpenMP tasks (formerly, we used Cilk [14], which is unfortunately being deprecated from the `gcc` compilers).

1.3.2 *Loops of Independent Iterations*

Computations are often expressed as loops, in algorithm pseudo-code and in real programs. Some computation is to be performed for the different values of the loop iteration variable in the range of this variable, here in increasing order of the loop variable:

```
for (i=0; i<n; i++) {
  c[i] = F(a[i]+b[i]);
}
```

In this loop, however, the iterations (different values of the iteration variable `i`) are *independent* of each other (provided the function `F` has no side-effects): No computation for iteration i is affected by any computation for iteration i' before i, $i' < i$; and no computation for a later iteration i'', $i'' > i$, could possibly affect the computation for iteration i. In this case, the loop could be trivially parallelized by dividing the iteration space into p roughly even-sized blocks of about n/p iterations, and let each block be executed by a chosen processor-core.

The assignment of blocks, more generally individual iterations, to processor-cores is called *loop scheduling*, and can be done either fully explicitly (as sometimes needed when parallelizing with MPI, see later lectures), or implicitly with the aid of suitable compiler and runtime system, by marking the loop (actually a bad name, since "loop" normally implies order) as consisting of independent iterations (another misnomer in this context, "iteration" implies sequential dependency) and therefore parallelizable. An example, which

we will see in much detail in later lectures is the following OpenMP style parallelization of a loop:

```
#pragma omp parallel for
for (i=0; i<n; i++) {
  c[i] = F(a[i]+b[i]);
}
```

With the PRAM model, independent loop-computations were handled by simply assigning a processor to each iteration with the **par**-construct:

```
par (i=0; i<n; i++) {
  c[i] = F(a[i]+b[i]);
}
```

1.3.3 *Independence of Program Fragments*

Independent iterations, in general, independent program fragments (which could be tasks as in Section 1.3.1) can be executed in parallel by different, available processor-cores. The independence of program fragments is therefore a sufficient condition for parallel execution.

Straight-forward conditions for independence of program fragments are the three *Bernstein conditions* [10]. Let P_i and P_j be two program fragments, with P_j following after P_i in the program flow. Each of P_i and P_j has a set of (potential) input variables I_i and a set of (potential) output variables O_i (these sets can be determined statically, but whether a potential output variable will actually be assigned is in general undecidable). The fragments P_i and P_j are *dependent* if either

1. $O_i \cap I_j \neq \emptyset$ (a *true dependency*, or *flow dependency*), or
2. $I_i \cap O_j \neq \emptyset$ (an anti-dependency), or
3. $O_i \cap O_j \neq \emptyset$ (an output dependency).

The conditions are obviously sufficient but not necessary: Either may hold, but input or output may not be read or written, or read or written in some order such that the outcome of the parallel execution is still correct.

Dependencies between the iterations of a loop are called *loop carried dependencies*, and there are three types, corresponding to the Bernstein conditions.

In a *loop carried flow dependency*, the outcome of an earlier iteration affects the computation of a later iteration:

```
for (i=k; i<n; i++) {
  a[i] = a[i]+a[i-k];
}
```

Such iterations can therefore not be done in parallel and expecting a correct outcome.

In a *loop carried anti-dependency*, the outcome of a later iteration would have affected an earlier iteration, if the two iterations were reversed or carried out simultaneously:

```
for (i=0; i<n-k; i++) {
  a[i] = a[i]+a[i+k];
}
```

Finally, in a *loop carried output dependency*, two iterations write to the same output variable(s). If executed simultaneously, the output would not be well-defined (unless, as in the Common CRCW PRAM, the same value is written):

```
for (i=0; i<n-k; i++) {
  a[0] = a[i];
}
```

This is our first example of a *race condition*, on which we will learn more in later lectures.

Some loop carried dependencies can be removed by appropriate program transformations. For instance, the loop carried anti-dependency can be eliminated by introducing an auxiliary array `b` into which the results from the computations on array `a` are written:

```
for (i=0; i<n-k; i++) {
  a[i] = a[i]+a[i+k];
}
```

$\longrightarrow$

```
for (i=0; i<n-k; i++) {
  b[i] = a[i]+a[i+k];
}
```

This must be followed by a loop (of independent iterations) to copy `b` back to `a`, or by swapping the two arrays, if possible by the surrounding program logic.

1.3.4 *Parallel Patterns*

The loop of independent iterations is an example of a recurring expression of computations that can, if independence is fulfilled, be parallelized (as we shall see in more detail). There are other such frequently occurring algorithm patterns that can potentially be used to build whole applications.

The lecture slides touches briefly of some such patterns, with names that are often used in the literature:

- Loop, SIMD, data parallel
- Barrier synchronization
- Stencil

- Domain-decomposition
- Reduction, map-reduce
- Work-pool, master-worker
- Pipeline
- Collective data exchange (communication) patterns

In the lectures, we will not go into these patterns, but it is a good idea to skim over the slides.

1.4 FOURTH BLOCK (1 LECTURE)

We look at two concrete problems, namely merging of two ordered sequences, and computing the prefix-sums of elements in an array. The aim is to derive good, parallel algorithms that can actually be implemented on real, parallel systems (both shared- and distributed memory). While the usefulness of the merging problem is obvious, the lecture also motivates why computing prefix-sums is such an important parallel computing problem. The lecture also states the so-called "Master Theorem", a useful tool that will immediately solve (most of) the recurrences of the lectures.

1.4.1 *Merging Ordered Sequences in Arrays*

The *merging problem* is the following: Given two ordered sequences stored arrays A and B with n and m elements, respectively, from some universe with a total order $\leq$, construct an ordered $n + m$ element array C containing exactly the elements from A and B.

The standard, straight-forward sequential algorithm for merging steps through the arrays A and B hand-in-hand and in each iteration writes out the smaller element to the C array (code on the slides). This algorithm unfortunately seems strictly sequential: The output at position i of C depends on the relative order of all the previous elements in A and B, and there is not much that can be done in parallel. The complexity of the standard algorithm is $T_{seq}(n) = O(n + m)$. A different idea is required if the problem is to be given a good parallel solution.

Recall that merging and sorting algorithms are called *stable* if the relative order of equal elements in the input is preserved. For the merging problem, this means that the relative order of equal elements in the inputs arrays A and B is preserved in the output, and elements in array A that are equal to an element of B occur before the B element in the output array C. Stability is often a useful or even desired property. Some merging and sorting algorithms are naturally stable (the standard, sequential merging algorithm, for instance), some are not.

For some of the merging algorithms in the following, it is convenient to assume that all elements are distinct. Distinctness can be assumed without loss of generality, by making the elements distinct: Instead of merging elements, we merge triples (x, F, i) where x is an element from either A or B, F marks whether the element comes from A or from B, and i is the index of the element, whether in A or in B. We use lexicographic order, defined by $(x, F, i) < (x', F', i')$ if either $x < x'$, or if $x = x'$ and $F \neq F'$ and $F = A$, or if $x = x'$, $F = F'$ and $i < i'$.

Using this order will also ensure stability of any merging or sorting algorithm. The cost is extra space and a more expensive comparison (which should not be neglected, try!). It is therefore most often better if the merging or sorting algorithm is stable by design, without resorting to the "make-distinct trick".

1.4.2 *Merging by Ranking*

A different approach to merging is the following. For each element $A[i]$ in A, find the position j in B such that $B[j] < A[i] < B[j+1]$ (here we assume element distinctness, and for convenience that $B[-1] = -\infty$ and $B[m] = \infty$). The position j is called the *rank* of $A[i]$ in B, denoted by $\text{rank}(A[i], B)$. By knowing the rank of element $A[i]$ in B, we also know the position of $A[i]$ in the output array C: It is $i + \text{rank}(A[i], B)$.

We can now merge the elements of A and B into C by computing the ranks for all elements in A and B in the other array. The rank of any element of A in B can be computed by binary search in $O(\log m)$ time steps. The sequential complexity of *merging by ranking* is therefore $O(n \log m + m \log n) = O((n+m) \log \max(n, m))$, far worse than the standard, sequential merging algorithm.

However, merging by ranking can be performed in parallel: Assign a processor to each element of A and of B, let it compute the rank of the element in the other array and write the element to its position in the output array C. With $n + m$ processors, the algorithm takes $O(\log \max(n, m))$ time steps, so it is fast, but it is clearly not work-optimal: The work is the same as the sequential merging by ranking algorithm, $O((m+n) \log \max(n, m))$. We note also that when ranking is done concurrently by many processors, concurrent read capabilities (as in the CREW PRAM) are required of our system.

To reduce the work, a new idea is needed. We want to design an algorithm using p processors. This idea is to rank only some of the elements, more precisely approximately n/p of them. The input array A is divided into disjoint, consecutive blocks of size roughly n/p (see pictures on the slides), and the first element of each A block is ranked in B. Now the A block can be merged with the consecutive part of B determined by the rank of the first element of the A block, and the first element of the next A block, using our best known sequential merging algorithm. These pairs of blocks can all be merged in parallel. We now have a work-optimal, parallel merging algorithm. There

are p processors, which together spend $O(p \log \max(n, m))$ work on ranking the p elements from A, and together spend $O(n + m)$ time for merging pairs of blocks. It should also be obvious that the algorithm is correct (given the distinctness assumption).

Unfortunately, we cannot give a good bound on the time (desired is $O((n + m)/p)$). Since we do not know the inputs, and the arrays A and B can be arbitrarily interleaved in C, it can happen that for one A block the first element has a rank in B close to 0, and the first element of the next A block a rank close to $m - 1$. Merging this pair would therefore take $O(n/p + m)$ sequential time steps, and there would be no speed-up over the sequential algorithm. This is a classical *load balancing issue*.

The lecture slides sketch two possible solution to this problem. The following theorem is claimed.

Theorem 6 *On a p processor system (where binary search can be performed), two ordered arrays A and B can be merged work-optimally in $O((n + m)/p + \log \max(n, m))$ time steps.*

1.4.3 *Merging by Co-ranking*

A different idea turns the parallel merging problem upside-down. The idea is to find for each position i in the output array C, the unique positions j and k in the input arrays A and B, such that by merging $A[0, \ldots, j - 1]$ and $B[0, \ldots, k - 1]$ we get exactly $C[0, \ldots, i - 1]$. The indices j and k are called the *co-ranks* for i, and the approach *merging by co-ranking* [63]. If a processor can determine the co-ranks for the first element of a block of $(n + m)/p$ elements of C and the co-ranks for the first element of the next block of C, the $(n + m)/p$ element block of C can be constructed by merging the blocks of A and B determined by the respective co-ranks.

By this approach, we can ensure that all of p processors have blocks of exactly the same size (plus/minus one element, if p does not divide $(n + m)$), and in that sense arrive at a perfectly load-balanced merging algorithm.

The observation of the following lemma tells how co-ranks can be computed.

Lemma 1 *For any index $i, 0 \leq i < n + m$, there are unique co-ranks j and k with $j + k = i$ such that*

1. *either $j = 0$, or $A[j - 1] \leq B[k]$, and*
2. *either $k = 0$, or $B[k - 1] \leq A[j]$.*

To find the co-ranks j and k for a given i, a binary-search like procedure can be applied, halving intervals in A and B until the conditions of the lemma are fulfilled. The code is shown in the lecture slides, and can (for parallel systems with a shared-memory) readily be implemented.

Theorem 7 *On a p processor system (where co-ranking can be performed), the merging problem can be solved work-optimally in $O(\frac{n+m}{p} + \log(n+m))$ time steps with p processor-cores. The algorithm is perfectly load balanced, and stable.*

Ranking and co-ranking are examples of static, problem-dependent load balancing: The blocks of the A and B arrays assigned to the processors all do have approximately the same total size, for the co-ranking approach exactly so, but where the blocks are located is determined by the input. The preprocessing needed for the load balancing step, after which the sequential block merging is done, takes $O(\log \max(n, m))$, which is not a constant fraction of the total work $O(n + m)$, so Amdahl's Law does not apply.

1.4.4 *Bitonic Merge⋆*

Bitonic merging is an example of an *oblivious merging* algorithm: The indices that are compared against each other depends only on n and m, the size of the input, and not the input itself. Bitonic merging does not require concurrent read capabilities of the system. Bitonic merging is an important example algorithm, and can in some situations have practical advantages over the merging algorithms in the previous sections. Bitonic merging, and bitonic merge sort was invented by Kenneth Batcher [9]. The lecture slides introduce *bitonic sequences*, and illustrate how bitonic merging works.

Bitonic merging and sorting can be analyzed using another model of parallel computation: *comparator networks*. Bitonic merge sort is not work-optimal, and it was a long standing open question of theoretical importance whether sorting networks of depth $O(\log n)$ and size $O(n)$ (number of comparators) exist [41, Section 5.3.4, Exercise 51]. The question was answered affirmatively in a famous paper by Ajtai, Komlós, and Szemerédi [3]. Another important result is "Cole's parallel merge sort", which shows that merging can be done in $O(\log n)$ time steps with n processors [19, 20]. Both results have very large constants hidden in the Os, and are in their original form not practically relevant.

1.4.5 *The Prefix-sums Problem*

Let an input array A of n elements from a set with an associative operator $\oplus$ be given. The ith *inclusive prefix-sum* for $0 \leq i < n$ is $B[i] = \bigoplus_{j=0}^{i} A[j]$, and the ith *exclusive prefix-sum* for $0 < i < n$ is $B[i] = \bigoplus_{j=0}^{i-1} A[j]$. The *prefix-sums problem* is to compute the (exclusive or inclusive) prefix-sums for all indices i. Computing all prefix-sums over an array is sometimes also called *scan* (mostly, but not always, denoting the inclusive prefix-sums, with *exscan* for the exclusive prefix-sums). Note that the ith inclusive prefix-sum can be computed from the ith exclusive prefix-sum by just adding with $\oplus$ the ith element. The

converse does not hold, unless an inverse of the $\oplus$ operation is given, and this is not always the case.

The prefix-sums problem is a generalization of the *reduction problem* which is to compute only the last, inclusive prefix-sum $B[n-1] = \bigoplus_{j=0}^{n-1} A[j]$.

Both problems are trivial to solve sequentially by a scan through the A array (thus the term), keeping a running sum in a register and writing it to $B[i]$. Improvements are possible by exploiting vector capabilities of the processor (make the compiler unroll the loop). The sequential complexity is $O(n)$ steps, and it is not possible to do better since $n-1$ sum computations are necessary.

Both reduction and prefix-sums can be seen as examples of parallel patterns (Section 1.3.4) or *collective operations*: Each of the p processors contributes some of the $n, n \geq p$ elements, and the processors together perform a reduction, or compute the prefix-sums with results stored at the processors (prefix-sums) or some selected root processor (reduction).

1.4.6 *Load Balancing with Prefix-sums*

The reduction operation is clearly useful. A frequently occurring book-keeping task in parallel computations is for the processor-cores to agree on some common value (could be a flag, telling whether the computation is done). This common value is computed by a parallel reduction. A *broadcast operation* may also be needed to distribute the outcome to the processors, or even better a combined reduce-broadcast which is commonly called an *allreduce operation*.

Applications of the prefix-sums problem are perhaps less obvious. Consider the following situation. Some expensive computation is to be done on some elements of a large array of n elements. It is not known a priori where these elements are, instead there is an associated marker array also of size n that for each index tells whether the associated element is to be worked on or not. All computations are independent of each other, thus there is potential for doing the work in parallel. We want to assign the element computations to p processors. The strategies for parallelizing loops that we have seen before (splitting the iteration range into p disjoint blocks, one for each of the p processor-cores) will not work well. Since it is not known which element indices are marked, it can easily happen that some blocks have many marked elements, while other blocks have no marked elements at all, and therefore little to do, apart from checking n/p indices and finding them unmarked. This is a typical load-balancing problem; the blocked merging by ranking algorithm had a similar problem. One processor may end up with all the work, and no speed-up is possible. Prefix-sums solves this load balancing problem, and this application is one of the most important applications of the prefix-sums problem in parallel computing and the reason why the problem is so important.

The solution is as follows. In some other array A of size n, put a 1 for each marked element, and a 0 for each non-marked element, which takes $O(n/p)$

parallel time steps (loop of independent iterations). Perform an exclusive prefix-sums computation on A into B. Now for each marked element, $B[i]$ is the number of marked elements up to (but not including) element i, and can therefore be used as index into another array storing only the marked elements consecutively. Assume that there are m marked elements (can be computed with a reduction operation over A, or from $B[n-1]$). Since these are now stored consecutively, the element array can be partitioned into p blocks of about m/p elements, on all of which the expensive computation has to be performed. All p processors now have about the same amount of non-trivial work to do, and much better load balance is achieved, especially if the element computations all take about the same time.

This pattern, often called *parallel array compaction*, occurs in many guises. One is parallelizing the sequential, linear-time partitioning step of the Quicksort algorithm. We do three mark-and-compact steps. First, the elements strictly smaller than the pivot are marked and compacted into an array part for the recursive call on the smaller elements. Second, the elements equal to the pivot (no recursive call needed) are compacted, and third, the elements strictly larger than the pivot are compacted into an array part for the larger elements. The total work is $O(n)$, although the constants are larger than the standard sequential partition implementations. How fast this is, depends on how fast the prefix-sums problem can be solved. The two Quicksort calls (on smaller and larger elements) are independent of each other, and can possibly be done in parallel (as will be discussed in later lectures).

If the partitioning steps is not parallelized, it will become a severe bottleneck for a parallel Quicksort implementation, consuming $O(n)$ time steps for the first Quicksort recursion level out of $O(n \log n)$ work in total, resulting in parallelism in the best case of only $O(\frac{n \log n}{n}) = O(\log n)$.

Another application of prefix-sums (scan) is given towards the end of the MPI lectures (sorting by counting, bucket sorting).

We now discuss three different solutions to the inclusive prefix-sums problem.

1.4.7 *Recursive Prefix-sums*

The first algorithm is a recursive, divide-and-conquer approach. Let A be an array of n elements for which to compute the inclusive prefix-sums into an array B. We reduce the problem to a prefix-sums problem of only $\lfloor n/2 \rfloor$ elements by computing into an array A' the sums of pairs of elements of A: $A'[i] = A[2i] \oplus A[2i+1]$, and recursively solve the prefix-sums problem on A' into an array B'. The prefix-sums B for the A array can be constructed from B': $B[2i] = B'[i-1] \oplus A[2i]$ and $B[2i+1] = B'[i]$ (with some care for the first, and the last element when n is odd). This can be implemented as shown below by a recursive function `Scan` that computes the prefix sums of the `n`-element array `A` into `A` itself.

```
void Scan(int n, int A[])
{
  if (n==1) return;

  int B[n/2];
  int i;

  for (i=0; i<n/2; i++) B[i] = A[2*i]+A[2*i+1];

  Scan(n/2,B);

  A[1] = B[0];
  for (i=1; i<n/2; i++) {
    A[2*i]   = B[i-1]+A[2*i];
    A[2*i+1] = B[i];
  }
  if (n%2==1) A[n-1] = B[n/2-1]+A[n-1];
}
```

It is easy to see by an inductive argument that the recursive algorithm (program) correctly computes the inclusive prefix-sums of A. If there is only one element in A ($n = 1$), $A[0]$ is indeed the prefix sum. Now assume that the function correctly computes the prefix-sums of an array B of $\lfloor n/2 \rfloor$ elements. For $i > 0$, the ith prefix-sum of A can be written as $\oplus_{j=0}^{i} A[i] = \oplus_{j=0}^{\lfloor i/2 \rfloor}(A[2j] \oplus A[2j+1])$ when i is odd, and $\oplus_{j=0}^{i} A[i] = \oplus_{j=0}^{\lfloor i/2 \rfloor}(A[2j] \oplus A[2j+1]) \oplus A[i]$ when i is even. By the initialization of B with $B[i] = A[2i] \oplus A[2i+1], 0 \leq i < \lfloor n/2 \rfloor$, it will then hold by the induction hypothesis that $B[i] = \oplus_{j=0}^{i}(A[2j] \oplus A[2j+1])$ after the recursive call, and then $\oplus_{j=0}^{i} A[i] = B[\lfloor i/2 \rfloor]$ when i is odd, and $\oplus_{j=0}^{i} A[i] = B[\lfloor i/2 \rfloor - 1] \oplus A[i]$ when i is even. This is what the program computes after the recursive call.

In each level of the recursion there is $O(n)$ work to be done for computing the pair-wise sums. Thus, the total work can be expressed by the following *recurrence relation*

$$\begin{aligned} W(n) &= W(n/2) + O(n) \\ W(1) &= 1 \end{aligned}$$

which can be solved by induction to give $W(n) = O(n)$. On each level of the recursion, the pairwise sums can be done in parallel (loop of independent iterations over the intermediate B' array of size $\lfloor n/2 \rfloor$) in $O(n/p)$ time steps. With p processors, this is $O(1)$, and the parallel time over all recursion levels is therefore expressed by

$$\begin{aligned} T(n) &= T(n/2) + O(1) \\ T(1) &= 1 \end{aligned}$$

which by induction gives $T(n) = O(\log n)$. The parallel running time with p processors is therefore in the best case $O(n/p + \log n)$.

To implement the algorithm with p processors, the pair-wise summing (loop) must be parallelized. The recursive call is done by all processors, but before the processors must wait for each other to have completed their part of the loop, for which a *barrier synchronization operation* is needed. Likewise, after the recursive call the processors must again wait for each other before they compute the results. Two barrier synchronizations are needed at each level for the recursion, for a total of $2\lfloor \log n \rfloor$.

Theorem 8 *The inclusive prefix-sums problem can be solved in parallel time* $O(n/p + \log n)$.

The recursive prefix-sums algorithm needs to allocate an intermediate array of size $\lfloor n/2 \rfloor$ elements at each recursive call (for a total of n elements). The pairwise summing has optimal spatial locality (see the next lecture) and can exploit the cache system well. It does about $2n$ summations with the $\oplus$ operations in the two parallel loops, about twice as many as the sequential algorithm.

1.4.8 *Solving Recurrences with the Master Theorem*

Recurrence relations, similar to the expression of work and time in the previous section, will occur often in these lectures, and many recursive algorithms give rise to this kind of very regular recurrence relations. Instead of doing an induction proof for each new recurrence, the solution to recurrences of this form can be summarized in a general theorem. This is often called the "Master Theorem" (for simple, regular divide-and-conquer recurrences), which exist in different versions. Here is one which covers the recurrences that will come up in these lectures:

Theorem 9 *Given a recurrence of the form*

$$T(n) \quad = \quad aT(n/b) + \Theta(n^d \log^e n)$$

for constants $a \geq 1$, $b > 1$, $d \geq 0$, $e \geq 0$, *and* $T(1)$ *some constant. The recurrence has the following closed-form solution*

1. $T(n) = \Theta(n^d \log^e n)$ *if* $a/b^d < 1$ *(equivalently* $b^d/a > 1$*),*
2. $T(n) = \Theta(n^d \log^{e+1} n)$ *if* $a/b^d = 1$ *(equivalently* $b^d/a = 1$*), and*
3. $T(n) = \Theta(n^{\log_b a})$ *if* $a/b^d > 1$ *(equivalently* $b^d/a < 1$*).*

When the recurrence relation models a recursive procedure, b is the shrinkage or reduction factor by which the subproblems get smaller, and a is the

proliferation or expansion factor, roughly the "number" (not necessarily integer) of subproblems to be solved at each recursion level. It is clear that the number of levels of the recursion is $\lceil \log_b n \rceil$. A proof analyzes such recursion trees, and can be found in any good algorithms' textbook, see for instance [1, 2, 21, 56], and also a recent paper by Kuszmaul and Leiserson [42]. A proof can be found in the appendix, and is much recommended to study.

APPLYING THE MASTER THEOREM TO PARALLEL PREFIX SUMS For the $W(n)$ recurrence above, $W(n) = W(n/2) + O(n)$, Case 1 applies (with $a = 1, b = 2, d = 1, e = 0$) which gives $W(n) = O(n)$.

For the $T(n)$ recurrence, $T(n/2) + O(1)$, Case 2 applies (with $a = 1, b = 2, d = 0, e = 0$) and gives $T(n) = O(\log n)$.

1.4.9 *Iterative Prefix-sums*

Theorem 8 can be achieved by a different looking, iterative algorithm. In fact, the iterative algorithm can be found from the recursive one by unfolding the recursions. An advantage of the iterative prefix-sums algorithm is that no intermediate array has to be allocated.

The algorithm has two phases, an up-phase, corresponding to the pair-wise sum computations before the recursive call, and a down-phase, corresponding to the sum computations on the return from the recursive call. Both up- and down-phases take $\lfloor \log n \rfloor$ iterations.

In the first up-phase iteration, sums of even-odd pairs are computed. In the next iteration, sums of pairs of every second elements are computed, in the third iteration, sum of pairs of every fourth elements, and so on. The down-phase reverses this pattern. The following code illustrates the algorithm.

```
int k, kk;
int i;

// up-phase
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  for (i=kk-1; i<n; i+=kk) A[i] = A[i-k]+A[i];
}
// down-phase
for (k=k>>1; k>1; k=kk) {
  kk = k>>1; // halve
  for (i=k-1; i<n-kk; i+=k) A[i+kk] = A[i]+A[i+kk];
}
```

The correctness of the up-down-phase inclusive prefix-sums algorithm (and implementation) can be proven by showing that certain invariant properties are maintained for each iteration and at the end imply the desired end result.

To formulate the invariants, let $a_i, 0 \leq i < n$ be the input sequence for which the inclusive prefix-sums are to be computed in $A[i]$, that is $A[i] = \oplus_{j=0}^{i} a_i$.

For the up-phase, the following invariant will hold before iteration $k, k = 0, 1, \ldots, \lfloor \log p \rfloor$: For each $i, i < n$ of the form $i = j2^k - 1$ for some $j > 0$, $A[i] = \oplus_{j=i+1-2^k}^{i} a_j$, that is, every 2^kth $A[i]$ will store the sum of the 2^k previous elements up to and including the ith element itself. This clearly holds before the first iteration ($k = 0$), since the input array is $A[i] = a_i = \sum_{j=i}^{i} a_j$. Assuming that the property holds before iteration $k, k > 0$, we have for that iteration which computes $A[i - 2^k] \oplus A[i]$ into $A[i]$ for elements $i = j2^{k+1}$ that $A[i] = (\oplus_{j=i-2^k+1-2^k}^{i-2^k} a_j) \oplus (\oplus_{j=i+1-2^k}^{i} a_j) = \oplus_{j=i+1-2^{k+1}}^{i} a_j$ for all i of the form $i = j2^{k+1} + 1$. Thus the invariant holds before the start of iteration $k + 1$. We can, by the way, observe that all $A[i]$ with $i = 2^k - 1$ for $k = 0, \ldots, \lfloor \log n \rfloor$ are "good" in the sense of correctly containing the ith prefix sum. The task of the down-phase is to make all other elements in A "good" as well. Also note here that the variables `k` and `kk` in the program are 2^k and 2^{k+1}, respectively, for the iteration count k in the proof.

The down-phase starts with the results computed in the A array by the up-phase. The invariant for the kth iteration for $k = \lfloor \log p \rfloor, \lfloor \log p \rfloor - 1, \ldots, 0$ is that each 2^kth element is "good", $A[i] = \oplus_{j=0}^{i} a_j$ for i of the form $i = j2^k - 1$. From the up-phase, this holds before the first iteration. In the iteration, the program computes $A[i] + A[i + 2^{k-1}]$ into $A[i + 2^{k-1}]$, so assuming the invariant to hold, we have that $A[i + 2^{k-1}] = (\oplus_{j=0}^{i} a_j) \oplus (\oplus_{j=i+2^{k-1}+1-2^{k-1}}^{i+2^{k-1}} a_j) = \oplus_{j=0}^{i+2^{k-1}} a_j$ by the "goodness" of $A[i]$ and the invariant from the up-phase for $A[i + 2^{k-1}]$. The iteration therefore makes $A[i + 2^{k-1}]$ "good", and $i + 2^{k-1}$ is of the form $j2^{k-1} - 1$ for the next iteration. After the last iteration when $k = 1$, this implies that $A[i] = \oplus_{j=0}^{i} a_i$ for all i, and thus the prefix-sums for all indices are correctly computed in the A array.

The algorithm achieves the bounds stated in Theorem 8. It also does about $2n$ summations with the $\oplus$ operations in the up- and down-phase parallel loops, about twice as many as the sequential algorithm. A drawback is that the pairs being summed are farther and farther apart $(1, 2, 4, \ldots)$, and thus the iterative algorithm has worse *spatial locality* than the recursive algorithm (more on spatial locality in the next lecture).

It is an important theoretical result that any sufficiently fast parallel prefix-sums algorithm has to do twice the number of $\oplus$ operations than sequentially required. Paraphrasing, something like the following result has been proved (using yet another model of parallel computation: the *arithmetical circuit*).

Theorem 10 *For computing the prefix-sums of an n-element input sequence, the following trade-off holds between* size *s (roughly number of* $\oplus$ *operations done by gates) and* depth *t (parallel time):* $s + t \geq 2n - 2$.

This was proved by Snir [65], a more intuitive proof can be found in [75].

The theorem says that for any fast (sub-linear) parallel prefix-sums algorithm, the speed-up (when counting the $\oplus$ operations) is at most about $p/2$. This is bad news for highly parallel algorithms running on a large number of processors which may use prefix-sums for array compaction and other important computations. The trade-off also tells us how many operations a best possible parallel prefix-sums algorithm is allowed to perform.

1.4.10 *Non Work-optimal, Faster Prefix-sums*

The two previous algorithms executed the loops summing pairs of elements $2\lfloor \log n \rfloor$ times. The next algorithm will reduce this to about $\lceil \log n \rceil$ loops, but the price is that it is no longer work-optimal. The algorithm has been discovered many times, and in this lecture we use the name Hillis-Steele after some such discoverers [34]. The algorithm computes the prefix-sums in-place in the array A.

In the Hillis-Steele algorithm, a $\oplus$ computation is done for (almost) all of the n array elements in each iteration. In the first iteration, for each element i, except the first, $A[i]$ is updated by summing with its adjacent element, $A[i] = A[i-1] \oplus A[i]$. In the next iterations, the update is $A[i] = A[i-2] \oplus A[i]$, in the third iteration $A[i] = A[i-4] \oplus A[i]$, and so on, in iteration k, $A[i] = A[i-2^k] \oplus A[i]$. Each iteration can be written as a loop, unfortunately of flow (forward) dependent iterations. The dependencies can easily be eliminated, by performing the updates into a result array B, and swapping A and B after the iteration. The following code snippet shows how.

```
int *a, *b, *t;
a = A; b = B;

k = 1;
while (k<n) {
  // update into B
  for (i=0; i<k; i++) b[i] = a[i];
  for (i=k; i<n; i++) b[i] = a[i-k]+a[i];
  k <<= 1; // double

  // swap
  t = a; a = b; b = t;
}
if (a!=A) for (i=0; i<n; i++) A[i] = B[i]; // copy back when necessary
```

It is easy to prove by invariants that the Hillis-Steele algorithm correctly computes all inclusive prefix-sums. Assuming that $a[i] = a_i$ for the input sequence a_i, one invariant is clearly that before iteration k it holds that $a[i] = \oplus_{\max(i-2^k+1,0)}^{i} a_i$ for each $i > 0$, which implies the claim when $2^k \geq n-1$. As in the iterative prefix-sums program, the variable `k` is 2^k for iteration count $k, k \geq 0$. The number of iterations is clearly $\lceil \log n \rceil$. The work of the algorithm

is $O(n \log n)$, and it is clearly not work-optimal. This is summarized in the theorem below.

Theorem 11 *The inclusive prefix-sums problem can be solved in parallel time* $O(\frac{n \log n}{p} + \log n)$.

1.4.11 *Blocking*

What is the use of a prefix-sums algorithm that is not work-optimal? In itself, for solving the problem on input of size n, it is not useful, as the larger the n, the smaller the speed-up compared to the sequential, best possible algorithm.

Algorithms that are not work-optimal can, however, be useful in context, as building blocks, where some of their advantages (like being fast) may pay off, but without being hurt by the extra work they perform. The situation is like this: If p processors have already been allocated, we may as well use them to reduce the parallel time. Here, there is no point in rescheduling the work to fewer processors (as when making a work-optimal algorithm cost-optimal), the processors are there anyway, and have to be paid for.

The general idea is, by use of work-optimal algorithms, to reduce the problem at hand to a (possibly different) problem that can be solved on p processors, and use this solution, again with work-optimal algorithms to compute a solution to the original problem. For the whole algorithm to be work-optimal, the problem reduction and computation of the final solution must be done by work-optimal algorithms, but for the middle step where a smaller problem is solved on the available p processors, there may be "room enough" that a faster, but not work-optimal algorithm can be employed.

When applied to the prefix-sums problem, the idea is sometimes called *blocking*. An n-element input array A is given, as are p processors to solve the problem. The input array is divided into p *blocks* of about n/p elements each, for each of the processors. Each processor performs a sequential reduction on its block of elements, and puts the results into an array B of p elements, one for each processor. Now, all prefix-sums of B are computed (by either of the parallel prefix-sums algorithm). After this, each processor i adds $B[i]$ to the first element of its A block, and computes the prefix-sums over its A block. This completes the computation of the prefix-sums of A.

The complexity of this blocked prefix-sums algorithm, using Hillis-Steele as building block is $O(n/p + (p \log p)/p + n/p) = O(n/p + \log p)$, since Hillis-Steele is applied to an array of p elements only. In contrast to Theorem 8, the non-parallelizable term is $\log p$, not $\log n$ (and with Hillis-Steele, the constant is 1, and not 2, as would have been the case with the recursive or iterative prefix-sums algorithm).

Theorem 12 *The inclusive prefix-sums problem can be solved in parallel time* $O(n/p + \log p)$.

The saving of a factor of 2 in the $\log p$ term does not sound like much. However, if pair-wise summing involves expensive communication (as is the case when the algorithm is used for distributed memory systems and implemented with MPI), such a factor can be worthwhile. There are more dramatic applications of the blocking technique in the literature. For instance, the fast, but not work-optimal Common CRCW PRAM maximum algorithm of Theorem 1 can be used to devise a work-optimal Common CRCW PRAM algorithm running in $O(\log \log n)$ time steps.

1.4.12 *Related problems*

In the prefix-sums and reduction problems, the elements were given in an array, and the array order determined the order of the application of the associative function $\oplus$. A natural, and it has turned out, extremely useful generalization of the prefix-sums problems is *list-ranking* problem. In the list-ranking problem, the elements on which to compute the prefix-sums are stored in an array, but the order in which to perform the $\oplus$ summations is determined by following an additional next element pointer (until the end of the list).

Although similar to the prefix-sums problem, the list-ranking problem turns out to be much more intricate and much more difficult to solve. For instance, although there are list-ranking algorithms similar to the Hillis-Steele algorithm, the simple blocking technique does not work here. It was a long standing problem to devise a fast, work-optimal, deterministic list-ranking algorithm.

The best deterministic result on an EREW PRAM is the $O(n/p + \log n)$ time algorithm of Anderson and Miller [5].

1.4.13 *A careful Application of Blocking⋆*

By a more careful application of blocking as described in Section 1.4.11, we can arrive at a parallel, reasonably fast, inclusive prefix-sums algorithm that achieves the optimal trade-off between "time" and "work" (measured as the number of $\oplus$ operations) captured in Theorem 10. The trick is to divide the input sequence of n elements into $p+1$ blocks (for p processors) of $n/(p+1)$ elements, intead of just p blocks as was done above. Assume now that $(p+1)$ divides n; this assumption can with a little care easily be lifted by dealing with some blocks of $\lceil n/(p+1) \rceil$ elements and some blocks with $\lfloor n/(p+1) \rfloor$ elements. The blocks are ordered, the first block contains the first $n/(p+1)$ input elements (block 0), the second block the next $n/(p+1)$ elements (block 1), and so on; the last block (block p) contains the last $n/(p+1)$ elements.

We measure the time t in the number of $\oplus$ that have to be carried out in sequence, and work (or size) s as the total number of $\oplus$ operations carried out by the p processors. The prefix-sums algorithm consists of three steps.

1. Compute for each of the first p blocks the inclusive prefix-sums for the $n/(p+1)$ elements in the block. This takes $t_1 = \frac{n}{p+1} - 1$ operations (time), and requires a total work of $s_1 = p\left(\frac{n}{p+1} - 1\right)$ operations.

2. Compute the inclusive prefix-sums for the sequence of the p sums of the first p blocks (this is for each block the prefix-sum for the last element). This takes time $t_2 = p - 1$ and work $s_2 = p - 1$ operations.

3. For the $p - 1$ blocks $1, 2, \ldots, p - 1$, excluding the first block 0, which is done (all prefix-sums computed by the first step), and the last block p which is special, add the prefix sum for the last block to the first $\frac{n}{p+1} - 1$ elements of the block. This results in the correct prefex-sums for all elements, since the prefix sum for the last element of each block is the prefix sum for the block that was computed in Step 2. This takes time $t_3 = \frac{n}{p+1} - 1$ and work $(p-1)\left(\frac{n}{p+1} - 1\right)$ operations. For the last block (block p), instead the prefix sum for block $p - 1$ is added to the first element of the block, and the inclusive prefix-sums for the $n/(p+1)$ elements of the block are computed. This takes the time $n/(p+1) = t_3 + 1$ operations, and another $n/(p+1)$ operations of work. The total work (number of operations) for the last step is therefore $s_3 = 1 + p\left(\frac{n}{p+1} - 1\right)$.

The total time for this algorithm is

$$\begin{aligned} t &= t_1 + t_2 + t_3 \\ &= \left(\frac{n}{p+1} - 1\right) + p - 1 + \left(\frac{n}{p+1} - 1\right) + 1 \\ &= 2\left(\frac{n}{p+1} - 1\right) + p \end{aligned}$$

The total work for this algorithm is

$$\begin{aligned} s &= s_1 + s_2 + s_3 \\ &= p\left(\frac{n}{p+1} - 1\right) + (p-1) + p\left(\frac{n}{p+1} - 1\right) + 1 \\ &= 2p\left(\frac{n}{p+1} - 1\right) + p \end{aligned}$$

The sume of work and time is

$$\begin{aligned} s + t &= 2p\left(\frac{n}{p+1} - 1\right) + p + 2\left(\frac{n}{p+1} - 1\right) + p \\ &= 2(p+1)\frac{n}{p+1} - 2(p+1) + 2p \\ &= 2n - 2 \end{aligned}$$

which is the best trade-off by Theorem 10. When carefully implemented, the algorithm could run in $O(n/p + p)$ time steps.

The same trick of dividing an input sequence into $p + 1$ blocks was used by Snir [64] to speed up binary search (in an ordered sequence) from $\log_2 n$ to $\log_{p+1} n$ comparison steps. It was also shown that this is best possible (note that this is constant if n is $O(p^k)$ for some constant $k \geq 1$).

1.5 EXERCISES

Exercises on general material. PRAM model.

1. An easy PRAM algorithm.
2. Non-commutative
3. Matrix-vector
4. Graph degrees with incidence matrix
5. Graph degrees with adjacency lists
6. Show that $a[i] = \oplus^{i}_{\max(i-2^k+1,0)} a_i$ is an invariant for the non work-optimal inclusive prefix-sums algorithm of Section 1.4.10.
7. Implement the optimal trade-off inclusive prefix-sums algorithm outlined in Section 1.4.13. The implementation should be entirely in-place, that is computation done on the input (and output) array with no extra arrays and only some constant number of additional variables (loop indices, running sums).
8. Give an algorithm for performing $p + 1$-ary (instead of binary) search in ordered arrays of n elements with $p \geq 1$ processors. Show that the running time of your algorithm is $O(\log_{p+1} n)$ (as claimed in Section 1.4.13).

2

SHARED-MEMORY PARALLEL SYSTEMS AND OPENMP

2.1 FIFTH BLOCK (1 LECTURE)

This lecture is an introduction to performance-relevant aspects of "real", parallel, shared-memory systems.

A naive, parallel shared-memory *system model* consists of a (fixed) number of processor-cores p connected to a large (but finite) shared memory. Every core can read/write every location in memory, but memory access is significantly more expensive than performing operations in the processor-core. Furthermore, memory accesses are not uniform, from each processors point of view some locations can be accessed (much) faster than other locations. Processors are not synchronized. All these assumptions are in stark contrast to those made for the idealized PRAM.

In a corresponding, shared-memory *programming model*, processes or threads (being executed by the processor-cores) can likewise access objects in a shared memory space. Processes or threads also have their own, private memory spaces that cannot be directly accessed by other processes or threads. There may be more processes or threads than processor-cores, these are scheduled to run by the operating (runtime) system (OS). Processes or threads are not synchronized, but the programming model defines means for synchronization and exchange of information via shared objects. In the next lectures, concrete shared-memory programming interfaces will be covered, namely `pthreads` and OpenMP. A programming model in which thread or processes can be executed by any of the processor-cores, chosen by the OS, is called *Symmetric MultiProcessing (SMP)* (here, we define SMP as a property of the programming model; there are other uses of the term, as in *Symmetric MultiProcessor where SMP is a hardware property* where SMP is rather an architectural property). This can have advantages, leaving it to the OS to exploit the processor-cores well, but can also have drawbacks (for instance related to the cache system, see below). In Parallel Computing, where our system is dedicated (Definition 1 again), we often program with only as many threads or processes as there are processor-cores (dedicated to us for exclusive use), and make sure that each thread or process is executed by one specific core. Ensuing this binding is sometimes called *pinning*, and will be discussed briefly in these lectures.

2.1.1 *On Caches and Locality*

The first difference between "real" shared-memory systems, and the naive model is the existence of caches. A (hardware) *cache* is a small, fast memory close to the processor-core that is used to store frequently used values, and thus to *amortize* the slow access times to the main memory. For instance, if a value that is read from memory can be reused 10 times, the effective main memory access time is one tenth of what it would have been if the value had to be read at every use. On the other hand, with no reuse, a cache might even introduce overhead in the memory access time. Note that *reuse* is an *algorithmic property*, and indeed, since many algorithms have locality of access properties (see next section), caches help immensely toward sustaining the illusion of fast, uniform memory access (the RAM model). However, some algorithms are truly "random access" and have no locality of accesses, and for such algorithms, caches do not help. Instead, the speed of the main memory accesses determines the performance of such algorithms. Examples are graph search problems (DFS, BFS) on very large graphs, where the access pattern is determined by the input graph, and the next graph node to be accessed would in most cases not be in the cache.

The ratio of the access times between data values fetched from memory and data in cache has increased over time (that is, improvements in memory performance has not kept up with improvements in processor performance). As a consequence, caches have grown larger (and typically take a substantial amount of space and transistors of the processor-chip), and the cache system has become more and more elaborate. Caches were part of the "free lunch", and the behavior of the cache system of a standard processor can normally not be changed. The ratio of accessing data in main memory and accessing data in the fastest cache (lowest level of the cache hierarchy) could easily be a factor of 10 or more.

2.1.2 *Cache System Recap*

The cache system of a standard processor does not work on the granularity of single values or words in memory, but on larger blocks of memory addresses. Also, caches map addresses (locations) of words in memory to addresses in the cache. The memory can be thought of as being segmented into small *blocks* (a typical block size could be 64 Bytes), and each block can be mapped to some cache line. A *cache line* thus stores a memory block, but also some additional meta information (bits and flags) needed by the cache system. The terms block and cache line are sometimes used interchangeably.

A cache in which each memory block is mapped to one, predetermined cache line is called *directly mapped*. The other extreme, a cache in which each memory block can be mapped to any cache line is called *fully associative*. A cache where each memory can be mapped to some predetermined, small set

of cache lines is called *set associative*. Modern processors have set associative caches with small k-set sizes for $k = 2, 4, 8, \ldots$, and are called *k-way set associative*. A directly mapped cache is a 1-way set associative cache. Direct cache mapping schemes can be easily implemented by a few integer division and modulo operations. Associative caches need additional search logic, and are more involved.

When a processor reads a word, the memory block to which the word belongs is calculated, and it is checked whether this block is in the cache. If so, the reference is a *cache hit*, and the word can be read fast from the cache. If not, the reference is a *cache miss*, and the block has to be read from slow memory into a corresponding cache line.

In an application, the cache *miss/hit rate* is the ratio of cache hits/misses over a longer sequence of memory references.

On a cache miss, a new block has to be read into a corresponding cache line. Since the cache is finite and much smaller than the main memory, it can easily happen that the cache or cache line is full, in which case there is a conflict and some cache line has to be *evicted*.

There are three types of cache misses. A *compulsory (cold)* cache miss happens when there are no address blocks in the cache, in which case every first reference to some block address will lead to a cache miss. A *capacity miss* happens when the cache (all cache lines) is full; it is inevitable that some line is evicted. Finally, a *conflict miss* happens when all cache lines in the set in which the block being read can fit are occupied. Thus, a conflict miss can happen, even when the cache as a whole is not full. Conflict misses can be particularly frequent for directly mapped caches, where it is normally easy (if the mapping function is known) to construct cases where every memory access will be a capacity miss (typically strided accesses with some bad stride). Conflict misses can happen only for directly mapped or set-associative caches. A fully associative cache would have only capacity misses. In a k-way set associative cache, either of the k cache lines can be evicted upon a conflict miss, and the choice which is called the *eviction or replacement policy*. Typically used replacement policies are *least recently used (LRU)* and *least frequently used (LFU)*, but such details may be difficult to find out.

On a write to a memory address, the workings of the cache system are a little more involved. If the block of the address written is already in the cache, it is (must be) overwritten (otherwise a subsequent read could deliver an outdated value). If it is not in the cache, either a cache line for that block is *allocated* (thus possibly resulting in a conflict miss), or the address is updated directly in memory. The former policy is called *write allocate*, the latter *write no-allocate*. On an update to a block already in a cache line, the value written may nevertheless be written to memory, which is called *write-through* cache. The other possibility, that the cache line is not written to memory, but kept until it is eventually evicted, is called *write back*.

The *granularity* of the cache system is at unit of memory blocks, and these hold several words (in todays processors, typically 64 Bytes which is 8 double floating point numbers). When an address is read into the cache, the whole memory block to which the address belongs is read. Thus, at the cost of one long read, a whole block of addresses will be in cache and some cache misses can be avoided. Such a cache system can benefit applications with two types of *locality of access*.

An application is said to have *temporal locality*, if the contents of a memory address is reused several time in brief succession (no or few other uses in between, so that eviction will not happen). An application is said to have *spatial locality* if addresses in the same block are also used (before the cache line is evicted). Again, we stress that access locality is a property of applications and algorithms, and only applications that have this property benefit from the cache system. It is a lucky incident that many applications have access locality, which is the reason why hardware caching is such a successful idea.

A good computer architecture textbook can provide additional detail on the cache system, some of which may be important for exploiting a given system efficiently, see for instance [16].

2.1.3 *Cache System and Performance: Matrix-matrix Multiplication*

Access locality matters; a standard, and highly illustrative example application is the matrix-matrix multiplication.

The matrix-matrix multiplication problem is to compute for an $n \times l$ input matrix A, and an $l \times m$ input matrix B, in an $n \times m$ output matrix C, all product-sums $C[i,j] = \sum_{k=0}^{l-1} A[i,k]B[k,j]$. The straight-forward (sequential) implementation takes three nested loops to do this.

```
for (i=0; i<n; i++) {
  for (j=0; j<m; j++) {
    C[i][j] = 0.0;
    for (k=0; k<l; k++) {
      C[i][j] += A[i][k]*B[k][j];
    }
  }
}
```

The work (sequential time) of this algorithm is clearly $O(nml)$, and $O(n^3)$ for square matrices. In Theorem 3, we observed that, in this implementation, two of the loops of independent iterations can be parallelized. A further observation is that the three loops can (essentially, only sometimes changes for the initialization of C). How well does this implementation perform (and compared to what)?

There are six $3! = 3\ 2\ 1 = 6$ permutations of the three loops. We ran them all on a few standard (Intel, AMD) processors, on medium large, square

matrices of order $n = 1,000$, with and without compiler optimizations (`gcc -O3`) and for both C `int` and `double` matrix elements. The results are surprising, and illustrative, and can be found on the slides (better: try at home). Briefly, we observed a factor of about $20 - 40$ between worst and best loop orders. The worst are the versions where the i loop is the innermost; best when the j loop is innermost.

The differences can be grossly explained by looking at the cache miss rate. Matrices in C are conventionally stored in row-major order. We assume that the cache is large enough to hold a single row of each of the three matrices, but no more. In that case, for the worst variants (i-loop innermost), each load of $A[i][k]$ and each write to $C[i][j]$ would result in a cache miss. For the best variants (j-loop innermost), $B[k][j]$ and $C[i][j]$ are both access in row-order (best possible spatial locality), so the miss rate is determined by the cache line size.

2.1.4 *Recursive, Divide-and-Conquer Matrix-Matrix Multiplication*

Other approaches to matrix-matrix multiplication solve the problem by doing the multiplications and additions not on individual elements, but instead on smaller submatrices that may fit better in the cache.A recursive formulation of such an approach splits the input matrices A and B roughly in half along both dimensions, recursively multiplies the submatrices, and compute the corresponding submatrices of C by adding the resulting submatrices.

Concretely, write the input matrices A and B as matrices of four submatrices.

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \text{ and } B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} \quad .$$

Then

$$C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix} \quad .$$

where the submatrix products $A_{00}B_{00}$ etc. are all computed recursively. Code is sketched on the lecture slides, and it is a good exercise to complete and implement. Dealing with matrices in C is cumbersome, care is needed when allocating (and freeing) space for intermediate submatrices. Submatrices are given implicitly by the start and end row and column indices of the original input and output matrices. For performance reasons, we usually look for a good cutoff value, that is the dimension of the matrix at which the recursive algorithm stops and where the remaining sub-problem (a matrix-matrix multiplication) is done by an iterative solution. The implementation shown in the slides performs similarly to the second best iterative implementation (but can be improved by more attention to cutoff and memory allocation).

The recursive formulation does 8 (recursive) matrix-matrix multiplications and 4 matrix additions. The total amount of work performed by the algorithm can be estimated by the following recurrence relation:

$$\begin{aligned} W(n) &= 8W(n/2) + O(n^2) \\ W(1) &= O(1) \quad . \end{aligned}$$

The recursion depth can be estimated by the following recurrence relation. Here, we are assuming that matrix addition is also done recursively, and has depth $O(\log n)$:

$$\begin{aligned} T(n) &= T(n/2) + O(\log n), \\ T(1) &= O(1) \quad . \end{aligned}$$

The recurrences are readily solved by the Master Theorem 9 which gives $W(n) = O(n^3)$ (Case 3 with $a = 8, b = 2, d = 2, e = 0$), and $T(n) = O(\log^2 n)$ (Case 2 with $a = 1, b = 1, d = 0, e = 1$). Thus, the work is of the same order as the straight-forward implementation, and the length of the critical path(s) if the computation is viewed as a task graph is $O(\log^2 n)$.

Volker Strassen brilliantly discovered that it is possible to do with only 7 matrix-matrix multiplications and 18 matrix additions [66] which gives rise to an algorithm with $W(n) = O(n^{2.81})$ (Master Theorem again).

2.1.5 *Blocked Matrix-Matrix Multiplication*

Instead of splitting the matrices recursively, the matrices can be split up front into submatrices of size $k' \times k''$ for some k', k'', and the matrix-matrix multiplication performed as the 3-loop iterative algorithm on these submatrices. This gives rise to an implementation with the same work, but now with 6 nested loops. If the submatrices are small enough to fit in cache, this implementation can perform better than the straight-forward implementation. The choice of best k', k'' depends on the size of the cache. Such an algorithm is called *cache-aware*, in contrast to a *cache-oblivious algorithm* which can have good or even optimal cache performance, regardless of the concrete size of the cache which does not have to be known by the algorithm [26, 27].

2.1.6 *Multi-core Caches*

The cache system in modern multi-core processor systems is structured in several dimensions. First, there is a hierarchy of caches of increasing size, L1, L2, L3 (perhaps more), with L1 the lowest level, closest to the processor-core, smallest, but fastest cache (typically 16KBytes), and L3 the *last level cache* (LLC), of typically several MBytes. The L1 cache is most often divided into a data cache and an instruction cache. The memory management system has another cache, the virtual page cache or *translation look-aside buffer* (TLB).

The L1, sometimes also the L2 caches are *private* to one processor-core (and therefore replicated), whereas from some level in the hierarchy, the caches are shared among more and more cores (example: the L2 cache might be shared among the cores on a single CPU "socket", the L3 among all cores in the parallel, multi-CPU "socket" system). Processors differ in the way the cache system is structured.

Caches in parallel multi-core systems pose new problems that do not manifest when a single processor-core works in isolation (doing for instance matrix-matrix multiplication), related to both semantics and performance.

The first is the *cache coherence problem* among private caches. Assume that a memory block is in the private L1 caches of two different cores. What should happen if one core updates an address in the cache line where the block is kept? If the cache line will *eventually* be updated in the other core's cache to reflect the change, the cache system is said to be *coherent*. If the cache line is *never* updated as a response to the update of the other core, the cache system is *non-coherent*. Updated as a response can mean that either the cache line is indeed modified with the new value, or that it is *invalidated* such that the next reference from the other core to the block in the cache line will result in a cache miss. Keeping caches coherent is a non-trivial task that requires a complex algorithm in the processor hardware, a *cache coherence protocol*. This protocol can affect performance by excessive *cache coherence traffic*. The cache coherence protocol cannot normally be influenced (or with difficulty, or only to some extent). Cache coherence is a strong property, that guarantees that the processor-cores have a consistent view of individual memory addresses. Let a be an address (location) in memory. A cache coherent system fulfills:

1. If core c writes to a at time t_1 and reads a at a later time $t_2, t_2 > t_1$, and there are no other writes (by c or any other core) to a between t_1 and t_2, then c reads the value written at t_1 (local consistency).

2. If core c_1 writes to a at time t_1 and another core c_2 reads a at a later time $t_2, t_2 > t_1$ and no other core writes to a between t_1 and t_2, then c_2 reads the value written by c_1 at t_1 (update transfer).

3. If core c_1 and core c_2 write to a at the same time, then either the value written by c_1 or the value written by c_2 is stored at a (write consistency, order).

The terms *eventually*, *later*, *at the same time* are modalities: something will happen. When something will happen is not said. Also note that the term *later* assume that the read and write *events* can be ordered relative to some (virtual) global time. It is possible to formulate the cache coherency axioms without any reference to such a virtual, global time.

Current, shared-memory multi-core systems are cache coherent, but there has been exceptions (often in the HPC area), and it is frequently debated

whether cache coherence is a reasonable expectation for many-core parallel systems with a very large number of cores [49].

The second problem is a phenomenon called *false sharing* which is caused by the granularity of the cache system. Recall that cache lines map consecutive blocks of addresses, say 8 double words. If some block is in the private caches of two or more cores, any update that one core performs to an address of that block will affect the other core's cache, either by an update or by an invalidation of the cache line. In particular, updates to two different addresses `&x` and `&y` in the block by the two cores, will create coherence traffic, even through `x` and `y` are not in any way related. This can degrade the expected performance significantly [68]. Some examples of false sharing are given throughout the lecture slides. Avoiding false sharing requires attention to allocation and use of variables, attempting to ensure that independent and frequently used and updated variables are on different cache lines. *Padding* is a wasteful such strategy that uses only one address per memory block (of the critical data structures).

2.1.7 *The Memory System*

The cache system is part of the *memory hierarchy* which, for our purposes, will mainly be the large *main memory*, beyond which are disks and other types of *external memory*. The characteristic of the memory hierarchy is that as memory up (from L1 to L2 to L3 caches to main memory, etc.) the hierarchy get larger and larger, the access times (and often also the granularity of access) also increase. Any textbook on computer architecture will give approximate ratios of access times, and details on granularity [16, 52].

A final, important part of the memory system, not mentioned so far, is the *write buffer* in which writes to the main memory are buffered and processed in the pace that the memory system can process updates. The write buffer, as long as it has capacity, makes writes to memory appear fast. Write buffers may be simple FIFO buffers but can also be sorted, and usually coalesce writes to the same address. The interaction with the cache system is highly non-trivial, but for single-core processors, write buffers like caches were part of the "free lunch" in that they transparently made (most) memory writes appear much faster than the actual memory access times. For multi-core processors, the existence of write buffers is no longer transparent, as will be explained below.

In a hierarchical memory system, memory access times are not uniform. The first time an address or block is accessed, access time depends on where in the hierarchy the address is located, and later accesses may be less expensive due to the cache system. Different addresses, residing in different parts of the hierarchy likewise have different access times. Modern memory systems are highly NUMA.

The memory system for multi-core parallel systems has additional structure, and additional restrictions. In a multi-core CPU, not every core has a

direct connection to the main memory, instead the cores share a small(er than the number of cores) number of *memory controllers*. The memory is banked along the memory controllers. The memory access times for a particular core depend on the "closeness" to the memory controller for the bank in which the accessed address is contained. Access times to different addresses are again non-uniform. The non-uniformity becomes even more prominent for parallel systems consisting of several multi-core CPUs. Access to memory that is controlled by a different CPU than the core issuing the access requires communication between the CPUs, and can take significantly longer than access to memory controlled by the CPU of the core.

Not taking the NUMA architecture and behavior of the memory system into account can become a serious performance issue. To some extent, NUMA effects can be alleviated by paying attention to the placement of data used by an application, and partly this is done automatically by the virtual memory system. An often used virtual memory page allocation policy is the so-called "first touch" policy, by which a virtual memory page will be put physically in the memory bank closest to the core that does the first access to the page. An application can attempt a good placement of virtual memory pages by "touching" pages (addresses) by the cores that will most heavily use the pages.

2.1.8 *Super-linear Speed-up caused by the Memory System*

Although super-linear (absolute) speed-up was claimed to be impossible, it can nevertheless happen and be observed on real, parallel systems. What is wrong with the argument presented in Section 1.2.3?

The argument that linear (perfect) speed-up is best possible assumes that the sequential and parallel system behave identically, in particular that memory accesses behave identically and take the same time in the two cases. Due to the memory hierarchy with large caches, exactly this may not be the case. Assume for simplicity an algorithm that can be parallelized well in the sense that the working set with p processors is $1/p$ of the working set on just one processor. As p grows, the smaller and smaller working set will fit in faster and faster caches in the memory hierarchy, effectively leading the memory accesses of the parallel algorithm to be much faster than for the sequential algorithm. The speed-up can exceed p by a factor equal to the ratio between effective, average sequential memory access time and effective, average parallel memory access time. As a consequence, super-linear speed-up of the form kp with $k > 1$ can indeed be possible and observed.

2.1.9 *Application Performance and the Memory Hierarchy*

The nominal performance of the CPU and processor-cores do not alone determine what the performance of some given application on a system will be. If the memory system is not able to supply data fast enough to the processor-

cores, the performance of the memory system (access times) will eventually determine the performance. What "fast enough" is, is determined by the application.

We say that an application is

- *memory-bound*, if the operations to be performed per unit read from or written to the memory take less time than reading/writing a unit from/to memory, and
- *compute-bound*, if the operations to be performed per unit read from or written to the memory take more time than reading/writing a unit from/to memory.

In a memory-bound application, the memory system and memory access times will determine the application's performance, and in a compute bound application the nominal processor performance will determine the application performance. Thus, the application determines whether to spend the money on a fast memory, or a fast processor.

This distinction is worked out quantitatively in the so-called *roofline performance model* [74]

2.1.10 *Memory Consistency*

While the memory hierarchy, cache system, and write buffer are all functionally transparent for a single core, this is no longer the case when multiple cores together are doing parallel computing.

When a program is executing sequentially, reads and writes to memory addresses (appear to) take place in the execution order of the program's instructions (a read instruction of an address written by an already executed write to that address, will return the value that was written). This is called the *program order* which is assumed to prove properties of the program by state invariants. When two programs are being executed concurrently by our asynchronous, parallel, multi-core system, it is (probably) a natural expectation that the outcome will be as if some *interleaving* of the two executions has taken place, that is that memory order follows program order. This is a particular kind of memory consistency which is called *sequential consistency* [43] which would allow us to prove properties of parallel programs much like we do for sequential programs. Only the possibility of different interleavings have to be considered.

Unfortunately, often due to the existence of per-core write buffers, modern multi-core systems are *not* sequentially consistent. This can best be seen by considering an example as given below. Two cores execute the respective pieces of code, the idea is to protect the code which is in the body of the `if`-statement such that at most one of the cores will be executing this body. The two flags `f0` and `f1` are in shared memory and can be read (and written) by both cores.

The question is whether we can prove this property ("at most one of the two cores can execute the `if`-body")?

```
// core 0                              // core 1
f0 = 0; // does not want to enter      f1 = 0; // does not want to enter

// ...                                 // ...

f0 = 1; // now wants to enter          f1 = 1; // now wants to enter
if (f1==0) {                           if (f0==0) {
  // has entered                         // has entered
}                                      }
```

We can argue by contradiction. Assume that one of the cores, say core 0, has entered the `if`-body. In that case, it has set its flag `f0` to 1, and read the other flag `f1` and found it to be 0. This means that core 1 cannot have reached the instruction where it sets its flag `f1` to 1, therefore is not in the `if`-body, and will also not be able to enter, since `f0` is still 1. Therefore, if one of the cores is in the `if`-body, the other cannot be (as is easily seen, it can of course happen that none of the cores enter), and the desired property holds. There is no interleaving of the two pieces of code that will lead to both cores being in the `if`-body, and the parallel program has the desired effect under sequential consistency.

The crucial observation is that the argument holds only under the assumption that reads and writes to memory happen in program order. If the memory system is not sequentially consistent, this might not be the case. For instance, with write buffers for the two cores, the following could happen. Both cores execute the initialization of the flags and the 0 values are written to memory. Now the cores proceed, execute their flag updates to 1, but these updates end up in the write buffers. Both cores execute the read of the flag in the `if`-expression, both return 0, and both enter the body, exactly what should not happen. What has happened is that the outcome of the write and the read instruction did not follow program order. This is a major problem: How can we reason about parallel programs running on such systems, how can we prove properties?

Answering these questions is way beyond this lecture. The programming interfaces that we will see in the next lectures (`pthreads`, OpenMP) will help us in that they give constructs to ensure and pose guarantees that at certain points in the execution, the memory is in a well-defined state (of the form: updates performed by one thread now visible to other threads). If used correctly, it will not be needed to pay attention to the exact behavior of the memory system. To do so, it is important that the hardware provides mechanisms to ensure that operations on memory (read and writes) have indeed been performed. Such mechanisms are operations to *flush* the write (and other) buffers, often called *memory fences*, and *atomic operations*.

Memory and cache behavior for parallel multi-core systems is painfully intricate. Being aware of the issues is essential for writing correct programs, and for getting the best possible performance. We summarize the two kinds of issues we have discussed:

- The *cache coherence problem*: What happens when different cores read-/write the same address?

- The *memory consistency problem*: What happens when different cores read/write different addresses?

2.2 SIXTH BLOCK (1-2 LECTURES)

`pthreads` is our first example of a concrete programming interface in the form of a library that implements a shared-memory programming model and is intended for running on parallel shared-memory systems. `pthreads` is an early example of a thread programming interface for C, still widely used, that has been used as a blueprint for many subsequent thread interfaces. Native threads in the C11 language follow essentially the `pthreads` interface. `pthreads` is standardized in POSIX (Portable Operating Systems Interface for uniX) as IEEE standard (IEEE POSIX 1003.1c).

From now on, the lectures will frequently use C as programming language, and the practical projects are to be implemented in C. The standard reference text is the book by Kernighan and Ritchie [40]. For good programming style in C, the book by Kernighan and Pike [39] is likewise valuable.

2.2.1 `pthreads` *Programming Model*

A *thread* is the *smallest unit of execution* that can be scheduled (and preempted) by the operating system (OS). In C and Unix/Linux, threads live inside *processes* and different threads share information that is global to the process. Threads in C are functions, and shared information is, for instance, global variables, static variables, file pointers, and the heap for dynamic memory allocation. Threads maintain their own stack and also the registers are private to a thread. It is also possible to allocate thread-local storage.

The main characteristics of the `pthreads` programming model are:

1. Fork-join parallelism. A thread can *spawn* any number of new threads (up to system limitations) and wait for completion of threads. Threads are referenced by *thread identifiers*.

2. Threads are symmetric *peers*, any threads can wait for completion of any other thread via the thread identifier.

3. Threads execute functions in the same program (SPMD model), but can be different for different threads (MIMD model). Initially only one main function thread is active.

4. Threads are scheduled by the operating system (OS) and may or may not run simultaneously on the different cores of the parallel system.

5. There is no implicit synchronization among threads, threads progress independently of each other.

6. Threads share global objects and information.

7. Coordination constructions for synchronization, and updates to shared objects are provided: mutexes, readers-and-writers locks, condition variables. All updates to shared information must be protected by coordination constructs, otherwise the program is illegal, and the outcome undefined.

`pthreads` does not come with a performance model (for analyzing the performance of `pthreads` programs), and does not come with (much of) a memory model either (for writing correct programs on hardware memory that is not sequentially consistent), except for requiring that updates to shared information is done via the coordination constructs of `pthreads`.

`pthreads` allows any number of threads to be spawned (subject to system limitations). Spawning more threads than the number of available cores in the parallel system at hand is called *oversubscription*. It is delegated to the operating system how and when threads are scheduled to run (even when there are fewer threads than cores), threads can also be preempted or suspend themselves, and this can to some extent be influenced by (non-standard) `pthreads` functionality that we will not go into in this lecture.

Oversubscription can have advantages (hiding latencies, giving freedom to the OS), but the *pragmatics* of Parallel Computing is mostly to have only as many threads as there are processor-cores, and assume that these threads all run simultaneously.

2.2.2 *pthreads in C*

`pthreads` is a library and the thread functionality can be used by linking the code against the `pthreads` library. C code using `pthreads` must include the function prototype header with the `#include <pthread.h>` preprocessor directive. All `pthreads` relevant functions and predefined objects are prefixed with `pthread_` which identifies the `pthreads` "name space". With `gcc`, code can be compiled using the `-pthread` option which enables linking against the library.

Most pthreads functions return an error code, and it is good practice to check the error code (which is often not done). The error code 0 means "success".

2.2.3 *Creating Threads*

When a C program with pthreads is started, the main() function is the only ("master") thread running. The master thread and any other thread can start new threads and wait for termination of any other thread. A thread is identified by an opaque object of type pthread_t which is set by the creation call and used to reference the now started thread. Thread identifiers can be compared for equality.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
void pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **retval);

pthread_t pthread_self(void);
int pthread_equal(pthread_t t1, pthread_t t2);
```

Code that is to run as a thread must be written as a C function with a single void* pointer argument. This pointer is used to point to a structure holding the actual, "real" arguments to the thread. The thread function will therefore often cast this void pointer to something more meaningful. The pointer to the function is given as an argument to the thread creation call together with a pointer to the actual arguments. Attributes will not be covered in this lecture, but can be used to control the way the thread is to run. C programming is brittle, it is easy to make mistakes with function and argument pointers, and such mistakes have grave consequences (memory corruption and program crashes).

When a thread function comes to the end, it should terminate itself by making the exit call which also takes a pointer that can point to information to be given back to the thread that intercepts the terminating thread. If return information is used, it must be allocated on the heap, definitely not on the stack where it will sooner or later disappear. Waiting for a thread to exit is done by the join call, which will update its void** pointer argument to point to the structure returned by the exiting thread. Thread identifiers can be exchanged freely between threads, and any thread can wait for any other thread to finish. In that sense, threads are "peers".

Slide Notes

The lecture slides gives a simple example how to create and terminate threads, and illustrates some things (C abuses) not to do.

The binding of threads to processor cores can be controlled by the following (non-standard) pthreads functions. A cpuset is a set data structure (bit vector)

representing a set of possible physical cores, numbered consecutively, and corresponding to the numbering of the cores on the shared-memory system, and should be manipulated through predefined macros.

```
int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize,
                           const cpu_set_t *cpuset);
int pthread_getaffinity_np(pthread_t thread, size_t cpusetsize,
                           cpu_set_t *cpuset);
```

2.2.4 *Loops of Independent Iterations in `pthreads`*

The patterns we have seen in the previous lectures can be implemented with `pthreads`. Loops of independent iterations, for instance, can be parallelized by assigning each thread a set (interval) of iterations. The tread function performs the iterations, taking the arguments for the loop from a suitable argument data structure.

Slide Notes

The lecture slides contains a few examples with different argument structures, and different division of work between starting and running threads.

2.2.5 *Race Conditions, Data Races*

In a thread model with shared memory (executed on a shared-memory multi-core system), it is possible for different threads to access and update shared variables. Since threads may execute concurrently, such updates may happen "at the same time". In such a situation the outcome is (for most systems, and we will assume this behavior) either the update by the one thread or the update by the other thread, and not something inbetween (also not: no update). But which thread succeeds with its update is undetermined. We say that the outcome of a concurrent update to a shared variable is *non-deterministic*, and such non-determinism may affect the final result of the whole program, often an undesirable situation. Since threads execute asynchronously (our thread model makes this assumption: no synchrony among threads, very much unlike the PRAM model), again the order of updates to shared objects is not defined, and either thread can be the "last" thread to perform an update (which, depending on the memory system behavior, may or may not become visible to the other threads in that order). Thread programs are inherently non-deterministic. In order to write correct programs that give a determinate, final output, we need to be able to deal with and restrict the non-determinism in updates and accesses to shared variables and objects.

Non-deterministic updates to shared objects and variables in a program that can lead to different, non-deterministic results of the program some of which are not correct, are commonly called *race conditions*. It is important

to keep in mind that asynchronous parallel programs are inherently non-deterministic (non-determinism is the price for the potential performance benefits of asynchronous parallelism), and that concurrent updates may not always lead to different, or wrong, final results.

Any thread programming model needs either means to reason about non-deterministic executions and updates to shared objects, or means to restrict and control non-determinism wherever it is crucial that updates are done in a certain, specific order, or both.

A particular kind of race condition is the *data race*. Technically, a *data race* is a situation where two or more threads access a shared object, and at least one of the accesses is an update (write). We note that it is undecidable to determine whether a program will have a data race, so automatically finding *all* race conditions (by a compiler) is algorithmically impossible.

Thread models like `pthreads`, and also OpenMP (and many others), forbid uncontrolled, concurrent updates to shared variables and objects, in particular forbid data races. Instead, they have constructs for threads to access and update shared objects. A way to look at such constructs is that they restrict the possible interleavings of asynchronous thread executions. We will see the main `pthreads` construct in the next section.

The following, standard example shows why data races can be harmful and lead to undesirable race conditions. The variable `a` is shared.

```
a = a+27;
```

With typical processors and instruction sets, this simple expression evaluation and assignment translates into three instructions (at least), namely (1) a load of `a` into a register, (2) an addition with a constant, and (3) a write to the location of `a`. The intention of the statement is that `a` is incremented by the constant 27. When several threads execute this code, it can easily happen that they all read the old value of `a`, perform the addition in their respective (private, non-shared) registers, and then race on the update to `a`: Instead of each thread incrementing by 27, only one increment will have happened. With many threads, many outputs are possible (increment by some multiple of 27), most of which are probably not that what was intended.

Data races are not always harmful. For instance, it might be unproblematic if all threads write the same value to the shared variable (as allowed with the Common CRCW PRAM, for instance). In the above example, it was harmful, and leading to very unintended outcome.

2.2.6 *Critical Sections, Mutual Exclusion, Locks*

`pthreads` programs (and OpenMP programs, see later) with data races are technically not correct, and programs with updates to shared variables by several threads that could happen concurrently are illegal. `pthreads` provides constructs to control access and updates to shared variables and shared objects.

The problem in the example above is not so much the individual data races on the shared variable a but rather the whole sequence of instructions involved in the update. When two threads at the same time comes to this little piece of code, what is required for the intended outcome is that either of the treads runs entirely before the other. We need to exclude exactly what happened above from the possible interleavings of the two thread executions.

A piece of code that should not be executed concurrently by several threads is commonly called a *critical section*. A thread running code in a critical section should exclude other threads from doing so, and since threads need to cooperate to ensure this, guaranteeing that a critical section is being executed by at most one thread is commonly called *mutual exclusion*. The *mutual exclusion problem* is to guarantee mutual exclusion, and it is not a trivial problem. It is not the purpose of these lectures to go into solutions (algorithms) for the mutual exclusion problem [33, 55]. Note that the code in a thread's critical section must not necessarily be the same for all threads. Rather, a critical section is a piece of a thread's code that should not be executed concurrently, in parallel with certain other pieces of code of other threads. The mutual exclusion problem is to ensure that this is the case.

A programming model mechanism that guarantees mutual exclusion is commonly called a *lock*. Locks provide mutual exclusion as follows. A thread that wants to enter a critical section tries to *acquire* the corresponding lock. If it succeeds, the thread is on its own in the critical section and do what it needs to do, typically read and write shared variables. When finished, it exits the critical section by *releasing* the lock. By now, other threads can enter the critical section by trying to acquire the lock. If a thread cannot acquire the lock, it cannot progress and is *blocked*. The lock acquire and release operations are often also called just *lock* and *unlock*.

Apart from guaranteeing mutual exclusion (at most one thread at a time can hold a given lock), the fundamental property of a lock is that it must be *deadlock free*. This means that if any number of threads (from one to many) are trying to acquire the lock, *eventually* one thread *must* succeed and get the lock. A perhaps desirable property is that any specific thread trying to acquire the lock will *eventually* acquire the lock, no matter which other threads are also trying to acquire the lock. A lock is said to be *starvation free* if it has this property that a thread is not *starved* forever. Locks are said to be *fair* if they provide stronger starvation freedom guarantees, for instance that a thread trying to acquire a lock "before" some other thread will also get the lock before.

In `pthreads` terminology, a lock is called a *mutex* (for mutual exclusion), and shared objects are only allowed to be updated by acquiring a mutex to do so. A mutex is identified by an opaque `pthread_mutex_t` type. Mutex'es must be initialized before use either statically (by assigning `PTHREAD_MUTEX_INITIALIZER`) or dynamically.

`pthreads` mutex'es guarantee mutual exclusion and are deadlock free, but *not* starvation free. In addition, they guarantee that all memory updates performed by a thread in the critical section before the release of the mutex will be visible to any other thread upon acquiring the lock. This is the `pthreads` memory model.

```
int pthread_mutex_init(pthread_mutex_t
                       *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
                          intpthread_mutex_init(pthread_mutex_t
                          *restrict mutex,
                          const pthread_mutexattr_t *restrict
                          attr);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

The data race on a in the example from above is properly avoided by protecting this critical section by mutex.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&lock);   // acquire lock
a = a+27; // now in critical section, alone
pthread_mutex_unlock(&lock); // release lock
```

Threads that try to execute the update concurrently will *serialize*: One thread after the other will be allowed to enter the critical section. It may even happen, if there is repeated competition for acquiring the lock, that some thread will never enter the critical section. If such happen, such a thread does not contribute any more to the computation and the possible speed-up is reduced accordingly. A lock where many treads are competing is said to be *contended*.

To allow threads to do something useful in case of contention, many lock models offer a *try-lock* operation. Try-lock tries to acquire the lock, and if the lock is not already held by some other thread, it immediately acquires the lock. If the lock is held by another thread, try-lock returns with a condition code. It is of course essential that try-lock acquires the lock, when possible, and does not return with a condition code. This would be useless, since acquiring the lock after checking the condition code could well fail because of some other thread having taken the lock inbetween.

Another means of alleviating lock serialization effects takes advantage of the situation that accesses and updates to shared objects are often asymmetric. In some (many) critical sections, shared variables are only read, while in other (fewer) also actual updates (writes) have to be performed. The threads that only need to read some shared object can do this concurrently, in parallel, while for the write, full mutual exclusion is needed and both other reading and writing threads must be excluded from the critical section. Readers-and-writers locks that are found in many thread programming models, provide this functionality. Readers-and-writers locks have a lock acquire operation for reading threads, and another lock acquire operation for writing threads. It is the programmer's responsibility to make sure that no updates (to shared variables) are performed in the critical sections when the lock is acquired for reading.

There are many ideas and algorithms for implementing locks (not treated in these lectures). An important pragmatic issue is how waiting for a lock is implemented, and how waiting (blocking) interacts with the operating system (OS). In a *spin lock*, the processor-core executing the blocked thread actively keeps testing (spinning) for the lock to become free. That is, the processor-core is kept busy for as long as the thread is blocked on the lock acquire operation. Acquiring the lock is fast for spin locks, and this implementation is typically advantageous when the critical sections are short and there is no thread oversubscription. With a *blocking lock*, the thread that is waiting for the lock to become free is suspended by the OS, and the processor-core that was executing the thread is free to do something else, for instance wake up and run another thread. Blocking locks may be advantageous when the shared-memory system is oversubscribed, and the lock waiting time can be spent for something else. In `pthreads`, spinning behavior can be requested explicitly by using spin locks. This (strange) `pthreads` design decision means that code has to be rewritten, if spin locks are desired.

```
int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

2.2.7 *Flexibility in Critical Sections with Condition Variables*

Since `pthreads` programs must be data race free, locks need to be used for transferring information between threads, for instance a value updated by a writing thread that is to be used by several reading threads. The following first solution is obviously wrong since it easily leads to a deadlock. A reading thread entering its critical section before the write will stay in the while-loop and keep the writer thread from setting the `written` flag.

```
// reader threads
pthread_rwlock_rdlock(lock);
while (!written);
a = b;
pthread_rwlock_unlock(lock);
```

```
// writer thread
pthread_rwlock_wrlock(lock);
b = ... ;
written = 1;
pthread_rwlock_unlock(lock);
```

The situation is quite common. A thread having entered its critical section cannot proceed before some condition is fulfilled that involves other threads to enter their critical section. A sometimes working solution is for the tread to leave the critical section and try again later, hoping for the condition to have been fulfilled. A more elegant solution is by so-called condition variables. A *condition variable* is an object associated with a mutex variable. A thread can *wait* on the condition variable, meaning that the thread is suspended and effectively out of the critical section (the lock is released), until some other thread performs a *signal* operation on the condition variable. When a waiting thread receives the signal and is woken up, the signalling thread will have left the critical section, such that mutual exclusion is always guaranteed with condition variables. More threads, for instance readers as in the example above, can wait on the same condition variable. A single signal operation will wake up either of the treads: `pthreads` provides no fairness guarantee, and no guarantee that a thread is not starved. To wake up all waiting threads, one after the other (mutual exclusion is always guaranteed) a *broadcast* is also provided. A signal operation on a condition variable where no thread is suspended is lost (unlike the case for the *semaphore*, another primitive synchronization mechanism, this one going back to Dijkstra in the early 60ties). The standard usage pattern for locks with condition variables is called a *monitor* [35], and some thread models and interfaces support monitors directly, `pthreads` indirectly via the condition variable mechanism.

```
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

A correct implementation for the situation in the example can now be given with condition variables.

```
pthread_cond_t data =
  PTHREAD_COND_INITIALIZER;

// reader threads
pthread_lock(lock);
while (!written)
  pthread_cond_wait(data);
a = b;
pthread_unlock(lock);
```

```
// writer thread
pthread_lock(lock);
b = ... ;
written = 1;
pthread_cond_broadcast(data);
pthread_unlock(lock);
```

Typically, the condition variable mechanism allows for so-called *spurious signals* or *spurious wakeups*, meaning a false or outdated signal being sent, so also in `pthreads`, and therefore the condition (here the flag `written`) is checked again upon being woken up.

2.2.8 *Versatile Locks from simpler Ones*

Standard constructions show that the more versatile readers-writers locks can be constructed from simple locks. Also different priority schemes (writer or readers preferred) can be implemented.

A thread barrier is a construct which makes it possible for a thread to define a point in the execution beyond cannot progress before a certain number of other threads have reached the barrier synchronization point. `pthreads` defines function interfaces for such barriers; the count is the number of threads required to reach the barrier point. Each barrier (there can be several) is identified by an opaque `pthread_barrier_t` object, which needs to be shared among the threads.

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
                         const pthread_barrierattr_t *restrict attr,
                         unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Barriers can also trivially be constructed from mutex'es with condition variables. Implementing efficient shared-memory barriers is non-trivial, however [47].

Slide Notes

An implementation of readers-writers locks in terms of standard locks with condition variables is shown in the lecture slides.

A final, common pattern is concurrent initialization, where one of the threads (the "first") should carry out come initialization code (function). This pattern can easily be implemented with mutexes, `pthreads` provides a shorthand.

```
int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));
```

2.2.9 *Locks in data structures*

Sequential data structures with their particular semantics and operations are often used in a parallel setting, and this can make a lot of sense. Threads might want to share a linked list, for instance used as the implementation of a set data structure with search, insert, and delete operations, or a stack, or a queue, or a hash map, etc., and use the data structure operations as the means for communication and synchronization. Likewise, hash maps, priority queues. As long as the data structure does not become a sequential bottleneck, by being too large, or by leading to thread serialization, shared (sequential) data structures can be helpful in formulating and implementing parallel algorithms.

The trivial way of making a(ny) sequential data structures useful in a parallel algorithm, is to use a single a global lock to protect all data structure operations. The already available, sequential implementation, perhaps complex and highly tuned, can be used right away, but the price is that all concurrent operations on the data structure will serialize, and can limit the possible speed-up of the algorithm. Thus, this solution is often not good enough. For data structures with read and write operation, like for instance the set which supports search (read) and insert/delete (write) operations, the more versatile readers-and-writers locks can alleviate some of the drawbacks. Read operations being perhaps frequent will have maximum possible concurrency, and only the write operations will be bottleneck operations.

When this is, for performance reasons or other, not acceptable, data structures and algorithms have to be rethought into more *concurrent data structures*. Some data structures, for instance linked lists, easily allow for implementations with more "fine-grained" locking or hands-over locking. The idea is to use a lock for each list element, and as the list is being traversed only acquire the lock for one or two of the elements currently visited. For longs lists, this make it possible for many threads to perform operations on different parts of the lists. But since a thread having acquired the locks on element at the front of the list will prevent any other threads from scanning through the list beyond this point, the improvement of this locking scheme is modest.

Developing data structures, even with the use of locks, that allow for a large amount of concurrent uses by many threads is non-trivial, and beyond the scope of these lectures. The point we make here is that locks can still be useful, but need to be used carefully (localized, short critical sections), and that in such cases a large number of locks will have to be used. There, the (space) efficiency of the lock implementations provided by `pthreads`, OpenMP and other thread models is important.

2.2.10 *Problems with Locks*

Locks and semaphores and similar constructs are concurrent computing constructs that were not designed for Parallel Computing with large numbers of threads. The typically (inherently) limited scalability is a reason to use them sparingly. Locks have other problems:

- Deadlocks can easily be programmed. For instance in a program with with two or more locks L_1 and L_2 (liked the linked list with hands-over locking), one thread may acquire the locks in the order L_1, L_2, and some other thread the locks in the order L_2, L_1. If the two threads execute roughly at the same time, they will both come to a point where they cannot proceed, because the lock the are trying to acquire is already taken by the other thread. This sounds trivial to avoid, but it is not. The deadlocking, two pieces of code may be in different parts of a large software package, perhaps not written by the same persons etc.. Each of the code pieces may in itself be correct, and tested in isolation, the deadlock situation will not show up. When the codes are run together, the program deadlocks. In that sense, locks are not a mechanism supporting modularity. A deadlock is always deadly, it proliferates and eventually the whole application cannot complete, because the deadlocked threads will not complete. In order to avoid deadlock when using multiple locks, locks have to be acquired in an agreed upon order. With multiple locks, the try-lock operation can often be useful.

- A special case of the situation above is the case where a thread having acquired lock L tries to acquire L again. This may deadlock; so-called *recursive locks* explicitly allow this (the number of unlock calls have to match the number of lock calls).

- Locks that protect long critical sections lead to possibly harmful serialization which can degrade performance (Amdahl's Law).

- Infinitely long critical sections, for instance by a thread crashing in the critical section, leads to deadlocks. Locks are not fault-tolerant.

- Since locks are often not fair, threads can be starved out, and actually not be contribution to the progress of the parallel algorithm.

- When threads have priorities (possible with `pthreads`, but not covered in this lecture) locks can lead to the effect that a lower prioritized thread prevents a thread with high priority from running, even when this would have been possible. The phenomenon is called *priority inversion*.

2.2.11 *Atomic Operations*

The problem with the `a = a+27;` example leading possibly to undesired final results was that the sequence of instructions in one threads' complex assignment operation (load, compute, store) could be interleaved with instructions executed by another thread. To prevent such interleavings, the assignment should be executed as an *atomic*, that is indivisible, unit. Mutual exclusion with locks is one way of guaranteeing atomic execution of the sequence of instructions.

Another way of ensuring atomic execution of compound operations is offered by hardware implemented *atomic instructions*. An atomic instruction caries out a complex (but relatively simple) compound operation as a unit that cannot be interfered with by other threads or processor-cores. One kind of atomic operation is for instance the fetch-and-add which can implement exactly the `a = a+27;` assignment with a single, indivisible instruction.

Atomic operations are offered by all modern multi-core processors and systems. Typical atomic instructions are, given as C like library calls:

1. `atomic_test_and_set(int *a)`: Return the contents of `a`, and set `*a` to 1 (true).

2. `atomic_fetch_and_increment(int *a)`: Return the contents of `a` and increment `*a`.

3. `atomic_fetch_and_add(int *a, int c)`: Return the contents of `a` and add `c` to `*a` (abbreviated FA, FAA).

4. `atomic_exchange(void *a, long b)`: Return the contents of `a` and update (exchange) `*a` to `b`.

5. `atomic_compare_and_swap(void *a, void *e, long u)`: Check whether the current contents of `a` equals the expected contents `*e` and if so, update `*a` to `u`, and return true; otherwise return false (abbreviated CAS, sometimes also, for instance in C, called compare-exchange).

Beyond this lecture: These atomic operations form a hierarchy (hence the numbering) on the power of what they can do [33]. All these operations are quite natural and helpful in many contexts. For instance, the `atomic_test_and_set` instruction is exactly what is needed to implement a lock.

Atomic operations are indeed instructions like any other processor instructions, meaning that they complete in some finite, bounded number of clock cycles, regardless of what other processor-cores might be doing (even executing and atomic operations). This essential property is called *wait-freeness*. This does not mean that atomic instruction are always fast, and mostly they are not. On the contrary, atomic operations are expensive, since they need to interact with the cache and memory system (write buffer), so like locks, they should be

used sparingly. But in contrast to locks, use of atomic operations cannot lead to deadlocks, and so a crashed (failed) thread will not affect the ability of the other threads to continue and make progress. Optimistically, we might assume that atomic operations are constant time $O(1)$ operations with relatively small constants, but bounded does not always mean constant.

Slide Notes

The trivial, but important prime finding example of the slides shows how an atomic counter can be useful for solving a load balancing problem.

```
int i = 2; // global counter
while (i<10000000) {
  j = i++;
  if (isPrime(j)) { ... // prime found, take action }
}
```

```
int i = 2; // global counter
while (i<10000000) {
  j = atomic_fetch_and_increment(&i);;
  if (isPrime(j)) { ... // prime found, take action }
}
```

In general, an operation on a data structure is said to be *wait-free*, if a thread executing the operation can always complete in a bounded amount of time, regardless of what the other threads are doing (including also performing the operation). An operation is said to be *lock-free*, if when several threads are performing the operation, some thread will be able to complete in a bounded amount of time. Wait-freeness is the non-blocking analogy of starvation-freeness, and *lock-freeness* the non-blocking analogy of deadlock-freeness. Like starvation freedom implies deadlock freedom, wait-freeness implies lock-freeness.

It can be shown that with sufficiently strong atomic operations (CAS), it is possible to give a wait-free implementation of any sequential data structure [33]. This is a theoretically strong result, but does not mean that wait- and lock-free data structures also perform well in practical contexts. We have seen that a wait-free counter can be useful, but other lock- and wait-free algorithms and data structures are beyond these lectures.

2.3 SEVENTH BLOCK (3 LECTURES)

OpenMP ("Open Multi Processing"), a standard for C and Fortran going back to around 1997, is our next example of a concrete programming interface that implements a shared-memory programming model and is intended for running on parallel shared-memory systems. Like `pthreads`, OpenMP is thread based, but offers much more and much stronger support for parallel computing. The main unit of parallelization in OpenMP was the loop of

independent iterations, see Section 1.3.2. From around OpenMP 3.0, support for task parallelism was introduced, see Section 1.3.1. This lecture (and the next ones) gives an introduction parallel programming with OpenMP, and covers the main features and constructs needed in Parallel Computing. There is more to OpenMP than we will cover here, though (in particular, thread teams will be silently circumvented, and also the recent support for "accelerators" like GPUs will not be treated). Some recommended or revealing books for OpenMP programmers are [18, 72, 46].

OpenMP is maintained and developed further by an *Architecture Review Board* (ARB) which includes academic institutions and industry in various roles. The OpenMP specification and additional information is freely available via `www.openmp.org`, including very helpful cheat-sheets, see for instance `https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-0519-web.pdf`.

2.3.1 *The OpenMP Programming Model*

Like `pthreads`, OpenMP is a fork-join thread model but threads are less explicit than in `pthreads` (no object identifying the thread). A *master thread* can fork (activate) a consecutively numbered set of working threads that include the master thread itself. The threads together share in executing work specified by a *work sharing construct* (e.g., loop of independent iterations, task graph). Upon completion, threads join, leaving the master thread to fork again a set of threads. An OpenMP program is a single program, all forked threads execute the same code (SPMD).

Main characteristics of the OpenMP programming model are:

1. Parallelism is (mostly) implicit through work sharing. All threads execute the same program (SPMD).
2. Fork-join parallelism: Master thread implicitly spawns threads through parallel region construct, threads join at the end of parallel region.
3. Each thread in a parallel region have a unique integer thread id, and threads are consecutively numbered from 0 to the number of threads minus one in the region.
4. The number of threads can exceed number of the processors/cores. Threads intended to be executed in parallel by available cores/processors.
5. Constructs for sharing work across threads. Work is expressed as loops of independent iterations and task graphs.
6. Threads can share variables; shared variables are shared among all threads. Threads can have private variables
7. Unprotected, parallel updates of shared variables lead to data races, and are erroneous.

8. Synchronization constructs for preventing race conditions.

9. Memory is in a consistent state after synchronization operations.

As for `pthreads`, OpenMP does not come with any performance model, and gives no guarantees or prescriptions for the behavior and performance of compiler and runtime system.

2.3.2 *OpenMP in C*

OpenMP requires compiler, library and runtime system system support, and must therefore be compiled with an OpenMP-capable compiler and linked against library and runtime system. Most C compilers are OpenMP-capable, for instance, OpenMP programs can be compiled with `gcc` by giving the `-fopenmp` option. C code using OpenMP must include the function prototype header with the `#include <omp.h>` preprocessor directive. All OpenMP relevant functions and predefined objects are prefixed with `omp_` which identifies the OpenMP "name space". Special OpenMP environment variables are prefixed with `OMP_`. OpenMP is not a language extension per se, but requires extensive compiler (and runtime) support for parsing and translating the `#pragma omp`-directives. OpenMP programs are C programs, but constructs like `for`-loops and compound statements are given their OpenMP meaning by `#pragma omp` compiler directives.

For the concrete the explanations in the following sections, we use `<...>` as meta-language designation for statements and non-empty lists of names, `[...]` to denote zero or more (optionally comma-separated) repetitions of some pragma element (clause), and `|` for exclusive choice.

2.3.3 *Fork-join Parallelism with the Parallel Region*

Threads are forked (spawned, generated, activated; many almost synonyms with some semantic differences) by the master thread reaching an OpenMP *parallel region construct* which is a structured C statement (simple statement or compound statement in curly brackets `{...}`) designated by the `omp parallel` pragma. In the parallel region, a defined number of threads will be active, all executing the structured statement (SPMD style). Once started, the number of threads in the parallel region cannot be changed. The threads can, by suitable library function calls, look up their thread id and the number of threads executing in the parallel region. The thread id is a C integer between 0 and one minus the number of threads in the region, that is thread id's are consecutive. Threads coming to the end of their execution of the code for the parallel region *join* with the other threads by performing a barrier synchronization, leaving only the master thread active after the parallel region. The barrier synchronization operation is implicit (not written out explicitly) with the end

of a parallel region, and it is essential for the OpenMP fork-join model that this cannot be changed. The thread id of the master thread is 0.

```
#pragma omp parallel [clauses]
<structured statement>
```

The number of threads in a region can be controlled either by the runtime environment, by a library call, or by the `num_threads()` clause for the `omp parallel` pragma. The latter take priority over the library call, which takes priority over the environment setting. When controlled by the environment, either a default number of threads is used, mostly equal to the number of processor-cores (or number of hardware supported threads; the CPU may support *hardware multi-threading*) of the system where the program is running, or as determined by the environment variable `OMP_NUM_THREADS`. The `OMP_NUM_THREADS` variable can be set to a number of threads larger than the number of processor-cores, that is, it is possible to run OpenMP programs with *oversubscription*. This is often useful for debugging, but rarely for performance.

An OpenMP program consists of a sequence of parallel regions and can be depicted as a fork-join task graph. The work of an OpenMP program executed with p threads is the total amount of work done by the threads over all parallel regions. The parallel execution time with p threads is the sum of the times taken by the slowest thread in each of the regions. A good OpenMP program has work proportional to the work of a best known sequential program for the given problem, and has a small number of regions in each of which the work is well balanced over the threads executing in the regions. In particular, the number of regions correspond to the part of the parallel OpenMP program that has strictly not been parallelized: The regions have to be activated one after the other.

2.3.4 *OpenMP Library Calls*

By suitable OpenMP library calls a thread can look up it's non-negative integer id, determine the number of threads in a parallel region, get the maximum number of threads allowed by the environment, and set the number of threads for a parallel region.

```
int omp_get_thread_num(void);
int omp_get_num_threads(void);
int omp_get_max_threads(void);
void omp_set_num_threads(int num_threads);
```

These OpenMP library calls are all *thread safe*, that is can be called concurrently, in parallel, without any risk of interference.

For measuring the time taken by the execution of a (sequence of) parallel regions, OpenMP provides standardized access to a (stable, high precision) timer.

```
double omp_get_wtime(void);
double omp_get_wtick(void);
```

The library function `omp_get_wtime()` returns the *wall clock time* in seconds since some point in the past. To report the time in milliseconds or microseconds of some OpenMP code, read the time before and after the piece of code, multiply the difference by 1000.0 or 1.000.000, respectively. The `omp_get_wtick()` call returns the resolution (in seconds) of the timer.

2.3.5 *Sharing variables*

Per default, all variable declared before a parallel region are shared by the threads in the region. Variables declared in the structured statement (block) of the parallel region are *private* (local) to each thread, that is a local copy for each thread will be created by the OpenMP compiler.

Sharing of variables can be controlled by sharing clauses to the `omp parallel` pragma directive.

```
private(<comma separated list of variables>)
firstprivate(<comma separated list of variables>)
shared(<comma separated list of variables>)
default(shared|none)
```

A list of variables declared by the master thread (before the parallel region) that will per default be shared in the parallel region can be made private which means that the compiler will generate a local copy for each thread. Variables declared private by the `private()` clause are *not* initialized. The `firstprivate()` clause additionally initializes each local copy to the value the variable had before the parallel region. Often, this is the desired, and perhaps implicitly assumed behavior. Note, that this can be expensive if the variable denotes a large, statically (compiler) allocated array as in `int a[1000];`. In contrast, for pointers the value of the pointer is copied, and not that to which it points. There are many possibilities for making non-sharing mistakes with OpenMP!

It is good practice (many say) to explicitly not share any variables declared by the master thread before a parallel region by using the `default(none)` clause, and explicitly then list the variables to be shared with the `shared()` clause. Such discipline forces one to think about which variables need to be shared and which not.

Shared variables can be read concurrently by the threads in the parallel region, but an OpenMP program in which it can happen that a thread updates a shared variable concurrently with other threads reading the shared variable is *incorrect*. This situation is a *data race*, and correct OpenMP programs must not have data races. OpenMP provides different means to avoid data races and still be able to exchange information between treads via shared variables.

2.3.6 *Work sharing: Master and Single*

The simplest work sharing OpenMP constructs designate work that is *not* to be shared among the threads, but rather executed by only one thread.

```
#pragma omp master
<structured statement>
```

Here, the work of the structured statement is done by the master thread alone (the thread with `omp_get_thread_num()==0`). The other threads will skip the structured statement code and just continue execution. There is *no* barrier synchronization implied following the master thread code. Also, the code of the master thread is *not* executed under mutual exclusion. That is, the master thread must not update shared variables that can potentially be read or updated concurrently by the other threads.

```
#pragma omp single [clauses]
<structured statement>
```

Here, the work of the structured statement is done by either one of the parallel, running threads, but it is not determined which of the threads; the OpenMP runtime system (or compiler) makes the decision. A parallel region can of course have several `single` statement blocks and each of the blocks may be executed by a different thread. The code executed by the chosen, single thread is, like for the `master` construct, not executed under mutual exclusion, so updates to shared variables possibly read by other threads are illegal. In contrast to the `master` construct, the `single` construct has an implied barrier at the end of the structured statement. A thread reaching this point, regardless of whether it was the thread executing the single designated statement or one of the other threads, cannot proceed until all threads have reached this point. This for instance implies that the number of encountered `single` statement blocks must be the same for all threads, so one must be careful with branches and loops in parallel regions.

The implied barrier at the end of the `single` block can be eliminated with the `nowait` clause. This can sometimes lead to better performance: A barrier can be expensive, and an OpenMP program should have no more barriers than absolutely necessary. On the other hand, a `nowait` clause can as easily make a correct program incorrect by introducing race conditions (data races). The `single` construct allows to make variables `private()` or `firstprivate()`; the `master` construct does not.

In the following example, the master thread reads input for a computation that is manually spread over the executing threads by the `for`-loop over `i`. Since there is no implied synchronization between the threads after the master has completed, and explicit OpenMP barrier (see next section) has been introduced, after which all threads can safely work on the input in the array `a`. The result in array `b` is written by some single thread, and in order to ensure that all

threads have completed their work before the array is written, again an explicit barrier is needed. The implicit barrier of this `single` construct is not needed (there is always a barrier at the end of the parallel region, so the barrier here is simply redundant) and is thererfore eliminated by a `nowait` clause.

```
#pragma omp parallel
{
  int i; // private i for each thread
...
#pragma omp master
  readdata(a,n);

#pragma omp barrier
  // compute
  for (i=0; i<n; i++) {
    b[i] = ...; // per thread computation from a into b
  }

#pragma omp barrier
#pragma omp single nowait
  writedata(b,n);
}
```

If the explicit barriers were omitted, correctness cannot be guaranteed: there are possible race conditions on both `a` and `b` arrays.

Code for single and master threads should be kept short, unless the other threads have sufficient other work to do, such that overall, all threads in the parallel region perform more or less the same amount of work.

2.3.7 *The explicit Barrier*

An explicit barrier, a point in the code of a parallel region beyond which no thread shall continue before all other threads have reached this point can be designated with the `barrier` construct, as we saw in the previous section.

```
#pragma omp barrier
```

An explicit barrier is sometimes necessary, for instance after a `master` construct, or in situations where threads read values computed by other threads. Here, a barrier (explicit or implicit) can be necessary to ensure that the other threads have indeed completed the computation of the required values.

2.3.8 *Work sharing: Sections*

The work to be done by some (part of an) algorithm can sometimes be expressed as some finite (small) set of independent pieces that can potentially be executed in parallel by a set of available threads. In OpenMP such work

can be identified and the independent pieces designated as such. This work structure is called `sections` with each independent piece forming a `section`.

```
#pragma omp sections [clauses]
<section block>
```

Each independent section of code (structured statement block) is marked as such.

```
#pragma omp section [clauses]
<structured statement>
```

A block of sections also ends with an implicit barrier synchronization point: No thread can continue beyond the sections code before all sections have been completed. This implicit barrier can be circumvented with the `nowait` clause. Before the block of sections, the sharing of variables can be restricted to either `private()` or `firstprivate()`.

In a parallel region with sections, the individual sections are assigned to the threads according to some schedule chosen by the OpenMP runtime system. Ideally, each thread will execute a section, and the threads will all run in parallel. If there are more sections than threads in the parallel region, some thread(s) will by necessity execute more than one section. Good OpenMP code will aim to make the amount of work in the sections balanced, in particular avoid having (too) few, very large sections that could lead to harmful load imbalance by many threads sitting idle at the barrier.

2.3.9 *Work sharing: Loops of Independent Iterations*

More substantial work is very often expressed as loops (of independent iterations). This was and is still the basic, fundamental premise of OpenMP. As we have seen, loops of independent iterations provide ample opportunity for keeping threads (processor-cores) busy by assigning consecutive blocks of loop iterations to threads. The assignment of particular blocks of iterations to threads is called OpenMP *loop scheduling*. Loop scheduling must at least fulfill that each iteration is executed exactly once by some thread, as the sequential semantics of the loop requires.

```
#pragma omp for [clauses] for (<canonical form loop range>)
<loop body>
```

In order that threads can independently (perhaps with aid from data structures in the OpenMP runtime system) schedule blocks of consecutive iterations, the loop range must confirm to certain rules. The most important such rule is that all threads in the parallel region will be able to determine the *same* loop range. Thus, in a standard C for-loop

```
for (i=start; i<end; i+=inc) {
  <loop body>
```

```
}
```

all threads must compute the same values for the start and end iteration, and must use the same increment (`i`, `start`, etc. are of course arbitrary variable names and expressions). These values must *not* change in any way during the execution of the loop. Also, loop ranges must be finite and determined, that is the for loop must *not* be a camouflaged, open-ended while loop. Such a range can easily be split into blocks of iterations by the compiler.

Finally, OpenMP poses restrictions on the form of the loop upper bound condition, which must be of the form `i<n`, `i<=n`, `i>n`, `i>=n`, or `i!=n` only (`i` is an arbitrary variable, and `n` an arbitrary expression). Also, the increments must take either of the forms `i++`, `i+=inc`, or `i=i+inc` (similar for decrements). Loops fulfilling such restrictions are said to be in *canonical form*.

There is a composite, shorthand directive for a parallel region with one parallel loop, which is one of the most frequent directives in OpenMP programs.

```
#pragma omp parallel for [clauses]
for (<canonical form loop range>)
<loop body>
```

Inherited from the `omp parallel` construct, the composite loop directive does not allow the `nowait` clause.

In order for the OpenMP program to be correct, loop iterations, regardless of the order in which they are executed by the threads, must not cause data races, that is concurrent reads and writes to shared variables: The loop iterations must be independent, and have neither forward, anti- or output dependencies.

A simple, sufficient rule for independence of loops is the following. The loop does array updates only, each iteration updates at most one array element, and no iteration refers to an element updated by another iteration.

Some loop carried dependencies in simple, array only loops can be eliminated by transforming the loops. A loop like

```
for (i=k; i<n; i++) a[i] = a[i]+a[i+k];
```

where, sequentially, `a[`i`]` is updated in iteration i with the (sequentially) not yet updated, and therefore "old" value `a[`$i+k$`]`, can, by introducing an array `aa` of the "old" values of array `a` before the loop, equivalently be written as

```
for (i=k; i<n; i++) aa[i] = a[i]+a[i+k];
// swap tmp = a; a = aa; aa = tmp;
```

The transformed loop is now a loop of independent iterations (also according to the simple rules for independent loops), and can therefore readily be parallelized with `#pragma omp parallel for`.

2.3.10 *Loop Scheduling*

Loop scheduling denotes the assignment of loop iterations to threads: How exactly is the work expressed by the loop of independent iterations shared across the threads of the parallel region?

For loop scheduling, the loop range (number of iterations) is divided into not necessarily same-sized *chunks* of consecutive iterations. Like the iterations, chunks are numbered consecutively such that chunks can be referred to by their number. The chunk numbering is for reference only, and not something that has to be computed or maintained explicitly by the OpenMP runtime system.

OpenMP provides three basic types of loop schedules.

In a *static* schedule, all chunks have (almost) the same size, and chunks are assigned in a round-robin fashion to the threads. For a loop range of n iterations, with chunksize c, and p threads, there are $k = \lceil n/c \rceil$ chunks, and the iterations of chunk $i, 0 \leq i < \lceil n/c \rceil$ are executed (one after another) by thread $i \bmod p$. That is, thread 0 executes the iterations of chunk 0, thread 1 the iterations of chunk 1, thread 2 the iterations of chunk 2, and so on. If there are more than p chunks, again thread 0 executes the iterations of chunk p, thread 1 the iterations of chunk $p + 1$, and so on, until all iterations of all chunks have been executed. If the loop range has been divided into at least p chunks, all threads can be kept busy, but not necessarily all of the time: That depends on the exact number of chunks, and on the time that each iteration takes which may be different for different iteration indices. For instance, if the work per iteration in chunks 0, p, $2p$ etc. is small (a condition on the loop iteration index fails for these chunks, so nothing to do except going through the iterations for the chunks), thread 0 might be able to finish much faster than the other threads.

Also in a *dynamic* schedule all chunks have the same size c, but the chunks are not assigned to the threads in any predetermined, static fashion. Chunks are executed by the threads in increasing chunk number, but each threads *dynamically* grabs the next not yet assigned chunk as soon as it has finished its execution of the iterations of its previous chunk. With a dynamic schedule, the above situation will not happen. As soon as thread 0 finishes (fast) with chunk 0 it will grab the next unassigned chunk, and thus help with finishing the loop iterations faster than the static schedule could possibly do.

Like in a dynamic schedule, a *guided* schedule assigns chunks to threads dynamically as the threads become available, but unlike both static and dynamic schedules, the chunk size is no longer fixed. Instead, when a thread has finished executing an earlier, smaller numbered chunk, it grabs a chunk for the next iteration that has not been executed and is also not in a chunk grabbed by another thread. The size in number of iterations for the chunk is equal to the number of remaining, not yet executed or assigned iterations divided by p, the number of threads.

The advantage of the static schedules is that computation of chunk numbers and assignment to threads can be done very fast and efficiently, essentially by each thread deciding for itself which chunks and executions it will have to execute (again, due to the restrictions on parallelizable loops to the canonical form). Thus, static schedule have low scheduling overhead. A static schedule can be expected to give good performance when the work per iterations is more or less the same for all iterations. Many, but not all loops have this property, although the time per iterations can be influenced heavily by the memory and cache access patterns even for code where the iterations incur the same number of instructions to be executed. Dynamic and guided schedules might be preferable for loops with conditions depending on the iteration index and also otherwise having varying amount of work per iteration. The guided schedule is motivated by the idea that, when a thread becomes ready to execute a next chunk, the work in the remaining iterations is more or less the same per iterations, in which case it makes sense to divide these iterations evenly into p chunks. Both dynamic and guided schedules have a higher scheduling overhead than static schedules. For instance, dynamic scheduling could be implemented by the OpenMP runtime system with a simple *work pool* that maintains the next, not yet executed loop iteration index. Implementing such a work pool would require just an *atomic counter*:

```
do {
  start = atomic_fetch_and_add(&i,chunksize);
  if (start>=n) break;
  end = min(start+chunksize,n);
  for (j = start; j<end; j++) {
    // execute chunk
  }
} while (1);
```

Here, `chunksize` is the (fixed) chunksize c, and it was tacitly assumed that the loop increment was 1.

In OpenMP, the particular schedule type for a parallel for loop is determined by an explicit *schedule clause* that can take an optional, explicit chunk size parameter. For static and dynamic schedules this optional chunk size is then the exact size in number of iterations of the chunks, whereas for guided schedules, the explicit chunk size is a lower limit for the smaller and smaller chunks.

```
schedule(static[,chunksize])
schedule(dynamic[,chunksize])
schedule(guided[,chunksize])
```

If no chunk size is given, a default chunk size is used. For a static schedule for a loop range with n iterations this is approximately n/p, such that there are exactly p chunks, one for each thread, with one or more chunks having one or more extra iterations if p does not divide n. The OpenMP specification

deliberately does not specify which chunks will get the extra iterations. For a dynamic schedule, the default chunk size is 1.

For simple loops over arrays it can make sense to let the chunksize *c* be some multiple of the *cache line* (block) size in order to avoid *false sharing*.

There are two additional schedule types that can be given with the schedule clause.

```
schedule(auto)
schedule(runtime)
```

With the `runtime` type schedule, the schedule can be set externally by the `OMP_SCHEDULE` environment variable, which can be very useful for tuning and experimenting with different schedules. Some examples of `OMP_SCHEDULE` settings are as follows.

```
"static,1"
"static,8"
"dynamic"
"guided,100"
```

With the `auto` type schedule, the choice of "best" schedule is left to the OpenMP compiler and runtime system.

2.3.11 *Collapsing Nested Loops*

Many computations, for instance computations involving matrices, are often expressed with (doubly, triply) nested loops. If all the loops in the loop nests are loops of independent iterations, either of them can be parallelized with the parallel for directive. Deciding which one to parallelize may not be obvious, depending (among other things) on the amount of work per iteration and the number of iterations per loop. Often it makes sense to parallelize the loop with the larges number of iterations, but this may cause the code to blow up with different parallelizations, depending on which loop has the most iterations. A sometimes good solution is to treat the nested loops as one larger loop, that is to transform code of the form

```
for (i=0; i<n; i++) {   // parallelize this loop?
  for (j=0; j<m; j++) { // or this loop?
    x[i][j] = f(i,j);
  }
}
```

into

```
for (ij=0; ij<n*m; ij++) {
  i = ij/m; j = ij%m;
  x[i][j] = f(i,j);
}
```

This transformation is valid in the sense that each iteration of the nested loop is performed exactly once by the transformed loop; but under the condition that all loop bounds can be computed before the two loops and do not change during the iterations.

The outlined transformation can be performed automatically (to any nesting depth) by the OpenMP compiler with the `collapse(<nesting depth>)` clause to the `for` directive. The loops must be perfectly nested which means that the body of an outer loop must consist of only the next inner loop. As for all OpenMP parallelizable loops, the iteration ranges must be in the canonical form prescribed by OpenMP. The two nested loops can then be parallelized as follows.

```
#pragma omp parallel for collapse(2)
for (i=0; i<n; i++) {   // OpenMP will make one loop out of two
  for (j=0; j<m; j++) {
    x[i][j] = f(i,j);
  }
}
```

The `schedule()` and all other clauses allowed for parallel for loops can be used and will be interpreted as if the loop has been transformed (collapsed, flattened) as outlined. According to the OpenMP specification, the sequential execution order of the iterations in uncollapsed loops determines the order of the iterations for the collapsed range iteration range.

2.3.12 *Reductions*

Two frequently occurring loop patterns are the following:

- Prefix-sums

```
for (i=1; i<n; i++) {
  a[i] = a[i-1]+a[i];
}
```

- Reduction

```
sum = a[0];
for (i=1; i<n; i++) {
  sum += a[i];
}
```

Both of these loop patterns are loops of essentially dependent iterations, and therefore cannot be correctly parallelized with the OpenMP constructs for loop parallelization seen so far. The computations expressed by the two loops (parallel prefix-sums, and simple reductions) require different, parallel algorithms in order that they can be executed with any speed-up by a set of

threads working together. Thus, either non-trivial transformations of the loop patterns by the compiler into better, parallel algorithms (consisting for instance of sequences of easily parallelizable loops of independent iterations), or the execution of preimplemented algorithms at runtime is required to handle such loop patterns well. Good parallel algorithms require the binary operator used in the patterns (here: +) to be at least associative.

The latter, reduction pattern loop can be handled, that is, parallelized efficiently with OpenMP by using the `reduction()` clause with the parallel `for` directive. How well the parallelization works will depend on the OpenMP compiler and runtime system among other things.

The `reduction()` clause is quite flexible. It takes a binary reduction operator, and a list of reduction variables on which reduction with this operator is to be performed in the loop. The order of the reductions follow the loop iteration order, but it is not defined where brackets are put: associativity is exploited. Different reduction operators can be used in the same loop by giving a reduction clause for each.

```
reduction(<reduction operator>:<reduction variables>)
```

The allowed operators are `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`, as well as special `min` and `max` operators. Minimum and maximum operations are expressed either with special operators or code patterns like for instance C expressions like

```
mi = (x<mi) ? x : mi;
if (x>ma) ma = x;
```

that will be recognized by the compiler as minimum (maximum, respectively) computations. Here `mi` and `ma` are global variables declared by the programmer.

The reduction clause can also be used with parallel regions, and the sections work sharing construct. In such cases the reduction will be performed in thread or section order.

With OpenMP 5.0 the former scan/prefix-sum pattern can be also handled. This is expressed by modifying a reduction in a parallelizable loop to "capture" the reduced result for the current iteration, that is the prefix sum for that iteration, and looks as follows:

```
reduction(inscan,<reduction operator>:<reduction variables>)
```

A reduction is performed with the reduction operator on the reduction variables, and the corresponding prefix sum for that operator is captured with either a

```
#pragma omp scan exclusive(<reduction variables>)
```

directive for a structured block (for the exclusive prefix sums), or a

```
#pragma omp scan inclusive(<reduction variables>)
```

directive for a structured block (for the inclusive prefix sums). For the inclusive prefix sums computation, the reduction variables can be used in the block of

the `scan` directive and will contain the result of applying the reduction operator up to and including the current iteration of the parallel loop; conversely, the result of the reduction for the current iteration used before the `scan` directive will be the exclusive prefix sum up to but not including the current iteration. There can be only one `scan` directive in a parallel loop, and in such a loop, scheduling clauses cannot be used. The following example shows how to compute inclusive and exclusive prefix sums for an input array `a` with the result stored in `b`

```
x = 0;
#pragma omp parallel for reduction(inscan,+:x)
for (i=0; i<n; i++) {
  x += a[i]; // reduce
#pragma omp scan inclusive(x)
  b[i] = x;  // and save the prefix (current value)
}

x = 10;
#pragma omp parallel for reduction(inscan,+:x)
for (i=0; i<n; i++) {
  b[i] = x;  // save the prefix
#pragma omp scan exclusive(x)
  x += a[i]; // and reduce for next iteration
}
```

A convenient use of reduction with the scan directive is for array compaction, as discussed for Quicksort. The marked elements of an input array `b` has to be compacted into a shorter array `a`, and what is needed for that is a running sum (exclusive prefix sum):

```
int mark[n];
// mark[i] == 0/1 determines whether element i of b shall be taken
...

int ix = 0;
#pragma omp parallel for reduction(inscan,+:ix)
for (i=0; i<n; i++) {
  if (mark[i]) a[ix] = b[i];
#pragma omp scan exclusive(ix)
  ix += mark[i];
}
```

2.3.13 *Work sharing: Tasks and Task Graphs*

Another way of expressing substantial, dynamically evolving work is by the way of a Directed Acyclic task Graph (DAG). The OpenMP *tasking* work sharing constructs makes it possibly to express such computations.

Consider the recursive Quicksort algorithm as discussed in Section 1.3.1. In each Quicksort invocation the input array is partitioned into two (assume for simplicity) roughly equally large parts, each of which can be Quicksorted independently of the other. With several threads available as in an OpenMP parallel region, each Quicksort call can be wrapped as a *task* to be executed by a thread that may happen to be available and has no other work to do. In a parallel region, any piece of code, like a procedure call (Quicksort), a function call, or even a structured block, can be marked as a *task* by the corresponding OpenMP work sharing construct.

```
#pragma omp task [clauses]
<structured statement>
```

The code designated as a task will be prepared and wrapped by the thread executing the `omp task` pragma (with help at compile time by the OpenMP compiler), but the task itself will (may) be executed by any (other) thread in the parallel region at a later time. Tasks thus generated will be completed at the latest where completion is requested. One such point of completion is the implicit barrier at the end of the parallel region. All generated tasks can also be completed by an explicit `#pragma omp barrier` construct. In the terminology of Section 1.3.1, the tasks being wrapped by a thread are *ready*, but they do have dependencies on (private and shared) variables of the thread that generated the task. Thus, for correct OpenMP task programs, after a task has been generated, the generating task shall not update any variable that can be referred to by the generated tasks. If it does, data races which are illegal in OpenMP, may arise.

A thread that generates one or more tasks (a thread can have many `omp task` directives, for instance through recursive calls) may depend on these tasks to complete before it can continue its computation, for instance on values computed by the tasks. Waiting for completion of immediately generated tasks can be enforced by the `taskwait` construct.

```
#pragma omp taskwait [clauses]
```

The only allowed clauses are `depend()` clauses, expressing dependencies on other tasks. Dependencies are not treated in these lectures.

Here is a standard example of an algorithm that can be parallelized with tasks. The problem is to count the number of occurrences of some value x in an unordered array `a` of n elements. The algorithm is recursive. If $n = 1$ the problem is trivial: There is an occurrence if $\texttt{a}[0] = x$, otherwise not. If $n > 0$ the array is split into two halves, the number of occurrences in both halves counted and added together. This idea can obviously be formulated as a computation on a task graph, and be implemented in OpenMP as shown here.

```
int search(int x, int a[], int n)
{
```

```
  if (n==1) {
    return (a[0]==x) ? 1 : 0;
  } else {
    int s0, int s1;
#pragma omp task shared(s0,a)
    s0 = search(x,a,n/2);
#pragma omp task shared(s1,a)
    s1 = search(x,a+n/2,n-n/2);
#pragma omp taskwait

    return s0+s1;
  }
}

int main(...)
{
  int a[n];
  int x;
  int s;

#pragma omp parallel shared(x) shared(a)
  {

#pragma omp single
    s = search(x,a,n);
  }
}
```

Here, each recursive call is marked as an `omp task`. In order to sum the number of occurrences for each half of the array, an explicit `omp taskwait` is necessary. Also, the computed results (and the array pointer) are classified as `shared` which is crucial, since the tasks can be executed by any of the threads, in particular by a thread that is different from the one that allocated the variable. The thread that executes a task must be able to update the variable that was possibly allocated by another thread which is possible only if the variable is shared among the two different threads.

In the main program, the treads are activated by the `parallel` region, but only one, here some arbitrary `single`, thread shall initiate the search. If `single` (or `master`) is forgotten, all threads will start performing the search operation, which leads to superfluous work (by a factor of the number of threads p) and possibly (in the search example: definitely) data races.

In the example, the recursion is done all the way down to the bottom $n = 1$ condition. This is rarely a good choice, neither sequentially, nor in parallel. Finding a good cut-off for recursive algorithms is in general a difficult problem, which we will not solve here. In order to prevent too many, too small tasks (fast completion), a task can be designated as `final`, meaning that the task will

generate no additional tasks. Together with with a conditional `if`-clause this can possibly be used as a substitute for an explicit cut-off programmed into the recursive task.

The `omp task` work sharing construct offers further possibilities for controlling when a task will be ready for execution. Input-output dependencies can be expressed with `depend()` clauses. By the `priority()` clause, tasks can be given priorities as hints to the OpenMP runtime system in which order the tasks should preferably be executed.

```
void quicksort(int a[], int n)
{
  int i, j;
  int aa;

  if (n<2) return;

  // partition
  int pivot = a[0]; // choose an element (non-randomly...)
  i = 0; j = n;
  for (;;) {
    while (++i<j&&a[i]<pivot); // has one advantage
    while (a[--j]>pivot);
    if (i>=j) break;
    aa = a[i]; a[i] = a[j]; a[j] = aa;
  }
  // swap pivot
  aa = a[0]; a[0] = a[j]; a[j] = aa;

#pragma omp task shared(a) untied if (n>1000)
  quicksort(a,j);
#pragma omp task shared(a) untied if (n>1000)
  quicksort(a+j+1,n-j-1);
//#pragma omp taskwait
}

int main(int argc, char *argv[])
{
  ...

  start = omp_get_wtime();
#pragma omp parallel
  {
#pragma omp single nowait
    quicksort(a,n);
    //#pragma omp taskwait
  }
  stop = omp_get_wtime();
```

```
}
```

2.3.14 *Mutual Exclusion Constructs*

In order to prevent data races in parallel regions, OpenMP provides direct support for *mutual exclusion* by named critical sections.

```
#pragma omp critical [(name)]
<structured statement>
```

Threads that encounter a (named) *critical section* will all execute the code in the critical section, but under mutual exclusion, that is at most one thread at a time can execute the code for its critical section. In a critical section, one or more shared variables can be updated, shared variables can be read and the thread can make decisions based on the read values. Since no other threads will be executing code for the named critical section at the same time, such updates are technically not data races, and it is possible to ensure a definite outcome of the parallel execution of the threads. The order in which the threads will be able to enter the critical section is undefined, and will depend on the relative speeds of the threads, when they encounter the critical section, how many threads arrive "at the same time", and how the runtime system mutual exclusion (locking) algorithms resolve the conflicts. Thus, relying on some specific behavior of the critical section construct will lead to incorrect programs. A concrete case is the implementation of reduction like operations: Implementations with critical sections will be correct only when the reduction operators being used are commutative.

Critical sections are always (relatively) expensive constructs and will therefore have an impact on the overall performance of a parallel program, in particular since they may lead to serialization between the threads. They should be used sparingly and with care.

In case the update and work to be done in a critical section has particular simple form, it may be possible to use a hardware assisted atomic operation instead. OpenMP provides access to certain types of *atomic operations* by the following construct.

```
#pragma omp atomic [read|write|update|capture]
<atomic statement>
```

Atomic updates and capture operations allow the use of fetch and add (FAA) type atomic operations. The atomic statement is restricted to be of the form `x++;`, `++x;`, `x-;`, `-x;`, and `x = x binop expr;` etc. for the `atomic update` clause, and `y = x++;`, etc. for the `atomic capture` clause. Here `x` and `y` are variables where the C operators apply, `binop` one of the word wise C operations `+`, `*`, `-`, `/`, `&`, `^`, `|`, «, and ».

The hardware compare-and-swap (CAS) operation is not supported as an OpenMP atomic operation. This makes the implementation of certain kinds of

concurrent algorithms and data structures impossible in OpenMP, but this is beyond this lecture, see for instance [33].

2.3.15 *Locks*

Sometimes (named) critical sections are insufficient, for instance for implementing list based algorithms with hands-over locking where a lock (critical section) is needed for each element of the list. For that reason, OpenMP provides locks that can be allocated dynamically similar to the `pthreads` locks.

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

Locks in OpenMP do not have condition variables. OpenMP provides nested (recursive) locks, but locks do not have a try-lock operation. OpenMP also does not provide reader-and-writers locks. Thus, OpenMP is not intended for involved programming with locks the same way that `pthreads` and other threads interfaces are.

2.3.16 *Special loops*

Loops of independent iterations where the operation(s) per iteration have a particularly simple form, for instance expressing a simple n-element vector addition like:

```
for (i=0; i<n; i++)
  c[i] = a[i]+b[i];
```

can benefit from hardware capabilities for operating on small vectors with a single instruction. Modern processors typically have such capabilities in the form of extended vector instructions for operating on 2, 4, 8, 16 float or double elements with one instruction (SSE, AVX). Such instructions are called SIMD instructions. With OpenMP, the compiler can be instructed to try using SIMD instructions with the following three loop parallelization constructs.

A sequential loop to be executed by one thread with SIMD instructions is designated with the `simd` pragma.

```
#pragma omp simd [clauses]
for (<canonical form loop range>)
<loop body>
```

A loop within a parallel region to be shared among the threads of the region with each chunk executed with SIMD instructions is designated with the `for simd` pragma.

```
#pragma omp for simd [clauses]
for (<canonical form loop range>)
<loop body>
```

A parallel region with SIMD loop sharing can be written with a shorthand, composite construct.

```
#pragma omp parallel for simd [clauses]
for (<canonical form loop range>)
<loop body>
```

For the compiler to be able to exploit SIMD instructions, often certain constraints must be observed that can be expressed by additional clauses. Such conditions and constraints are beyond these OpenMP lectures.

A different way of parallelizing a loop of independent iterations is to (recursively) break the iteration range into smaller ranges that are executed as tasks as the threads become available. Such loop parallelization can easily be done by hand; but OpenMP provides a construct for automatically performing the transformation into a set of tasks for the parts of the iteration range.

```
#pragma omp taskloop [clauses]
for (<canonical form loop range>)
<loop body>
```

A taskloop is initiated by a single thread in a parallel region. Evidently, a taskloop does not take a `schedule()` clause; instead, the size of the parts of the iteration range can be controlled by a `grainsize()` clause. Alternatively, the number of tasks across which the loop is split can be set with the `num_tasks()` clause.

2.3.17 *Parallelizing Loops with Hopeless Dependencies*

Instead of completely giving up on (and not parallelizing) loops with dependency patterns that cannot be handled by reductions or any other of the available means, OpenMP as a last resort makes it possible to mark a part of the code for the loop iterations as having to be executed in the sequential iteration order. This is done by giving the `ordered` clause to the parallel for loop, and marking the section of code that has to be done in the sequential iteration order with the corresponding OpenMP construct.

```
#pragma omp ordered
<structured statement>
```

In a parallel loop there can be only one ordered block of code. This construct usually brings only overhead, but could allow other parts of the iterations to be performed in parallel. It use cannot be recommended.

2.3.18 *Example: Parallelizing a sequential algorithm with dependencies*

```
int primesieve(int n, int primes[])
{
  int i, j, k;

  unsigned char *mark;

  // can save half the space/time for even non-primes
  mark = (unsigned char*)malloc(n*sizeof(unsigned char));

  for (i=2; i<n; i++) mark[i] = 0x1;

  k = 0;
  for (i=2; i*i<n; i++) {
    if (mark[i]) {
      primes[k++] = i;
      for (j=i*i; j<n; j+=i) mark[j] = 0x0;
    }
  }
  for (; i<n; i++) {
    if (mark[i]) primes[k++] = i;
  }

  free(mark);

  return k;
}
```

Solution with ordered clause.
Solution with scan-reduction.
Hnad-written prefix-sums.

2.3.19 *Cilk: A Task Parallel C extension*

Cilk (alluding to "silk" and C) is (was) a C language extension for task parallel programming originally developed at MIT in the mid-90ties, with focus on provably efficient execution of the generated acyclic task graphs by the runtime system [14, 45, 13, 6]. Cilk was supported with `gcc` and other compilers for a number of years, but is unfortunately being deprecated since 2018 (due to issues with Intel). The OpenMP task model has surely been inspired by

Cilk, and Cilk programs can now easily be reimplemented with OpenMP. Cilk provides three new keywords to C.

```
cilk_spawn <function call>
cilk_sync
cilk_for (<canonical form iteration space>) <loop body>
```

Generation of tasks is called *spawning* in Cilk, and the `cilk_spawn` keyword marks a function or procedure call as ready for being executed as a task. This corresponds to the `omp task` construct which is however more general: With OpenMP a whole code block can be wrapped as a task. Immediately spawned child tasks will be waited for at the end of the statement block doing the task spawns. If waiting for the immediately spawned tasks to complete is required (as in the search program discussed in Section 2.3.13) the keyword `cilk_sync` can be used, much in the same way as `omp taskwait`. Finally, the `cilk_for` keyword is used as a shorthand for parallelizing loops as collections of tasks, much in the same way as `omp taskloop`.

Cilk has no (explicit) concept of threads. The `cilk_spawn` construct indicates that a function or procedure call *may* be executed in parallel with the code following the spawn (called the *continuation*); but not *how*. The `cilk_sync` construct introduces a dependency point where the execution must wait for the spawned calls to have completed. Thus, a Cilk program is also a correct sequential program (same holds, btw., for an OpenMP program). The Cilk runtime system executes spawned threads by a clever *work-stealing algorithm*. In the multi-threaded runtime system, threads execute spawned tasks from a local task-queue, and when running out of local tasks, *steal* tasks from other runtime threads; until there are no more threads to be executed. The Cilk constructs give rise to highly structured, acyclic tasks graphs, so-called *(fully) strict computations*. For (fully) strict computations, it can be shown that such a task graph with $T_1(n)$ total work and work on the longest path of $T\infty(n)$ can be executed in $O(T_1(n)/p + T\infty(n))$ expected time step by the work-stealing runtime system on a dedicated parallel shared-memory computing system with p processors running the p worker threads [6]. This is a constant factor of optimal, and in this sense, Cilk comes with a provably efficient runtime system. The Cilk runtime work-stealing algorithm implements a randomized, greedy scheduling strategy. A work-stealing algorithm is most likely also the basis for the OpenMP runtime system for executing OpenMP tasks (but this is not specified by the OpenMP standard).

As seen with the OpenMP examples, task parallel programs often follow from recursive, divide-and-conquer algorithms if the recursive calls are independent of each other. This was the case with the search algorithm, the Quicksort example, and also sorting by merging can be expressed in this way. Runtime bounds for recursive algorithms, both with regard to the total number of work, and the work of a single path of recursive calls down to the base case, can often be expressed as recurrence relations and sometimes the solutions

follow directly from the Master Theorem 9; if not, the recurrence must be solved by (induction by) hand.

Such analyses reveal for standard implementations of Quicksort and Mergesort that $T_(n) = O(n \log n)$ and $T\infty(n) = O(n)$. The parallelism is modest $O(\log n)$, meaning that linear speed-up can be achieved only for a modest range of processor-cores and threads. The bottleneck in the two cases were the sequential partitioning step, and the sequential merge operation. To achieve more parallelism, parallel algorithms for the bottleneck operations must be found.

In Section 1.4.1, several parallel approaches were given for merging in parallel in $O(n/p + \log n)$ time steps. A drawback of these algorithms for implementation as task parallel algorithms (with no explicit notion of threads) is that the number of processors p is used and must be known. The final algorithm in this part of the lecture script is therefore a different, recursive divide-and-conquer merging algorithm that can readily be implemented with Cilk and OpenMP tasks.

```
void parmerge(int A[], int n, int B[], int m, int C[])
{
  if (n<m) {
    int k;
    int *X;
    k = n; n = m; m = k;
    X = A; A = B; B = X;
  }
  if (n==0) {
    parcopy(B,m,C); return;
  }

  int r = n/2; // assume n>=m
  int s = binsearch(A[r],B,m); // rank
  C[r+s] = A[r];
  cilk_spawn parmerge(A,r,B,s,C);
  cilk_spawn parmerge(A+r+1,n-r-1,B+s,m-s,C+r+s+1);
  cilk_sync; // not necessary, implicit in Cilk
}
```

The algorithm ranks the middle element of one of the arrays in the other array, that is computes rank(a$[\lfloor n/2 \rfloor]$,B) by binary search, which gives two pairs of smaller subarrays that can be (recursively) merged together. In case a pair has an array without any elements, a parallel (recursive) copy operation is used to copy the other array to the output array. For the parallel recursion to terminate, the element A$[\lfloor n/2 \rfloor]$ which is larger than or equal to all previous elements in A, and larger than or equal to B$[s]$ and all previous elements in B, is written immediately to its correct position in the output array, ensuring that the parts of the A array are strictly smaller in both spawned calls.

The recurrences, assuming for the input size of the two arrays that $n = m$, are as follows.

- $$T_1(2n) \;=\; T_1(n/2 + \alpha n) + T_1(n/2 + (1-\alpha)n) + O(\log n)$$

  for some $\alpha, 0 \leq \alpha \leq 1$ that can vary throughout the evaluation of the recurrence, corresponding to the found rank in the smaller array.

- $$\mathrm{T}\infty(2n) \;=\; \mathrm{T}\infty(3/2n) + O(\log n)$$

  since the larger of the input arrays is always halved and in the worst case merged (recursively) with the smaller array.

The second recurrence can be solved by the Master Theorem (Case 2 with $a = 1, b = \frac{2n}{3/2n} = 4/3, d = 0, e = 1$), whereas the first requires a direct induction proof to give $T_1(n) = O(n)$. To see this, conjecture the solution to be

$$T_1(n) \;\leq\; Cn - c\log_2 n$$

for constants C and c where the time to rank an element in a sequence of length n is at most $c \log n$. Using this as induction hypothesis, the recurrence relation now gives

$$\begin{aligned} T_1(2n) &\leq T_1(n/2 + \alpha n) + T_1(n/2 + (1-\alpha)n) + c\log_2 n \\ &= C(n/2 + \alpha n) - c\log_2(n/2 + \alpha n) + \\ &\quad C(n/2 + (1-\alpha)n) - c\log_2(n/2 + (1-\alpha)n) + c\log_2 n \quad . \end{aligned}$$

Assuming the worst case in both logarithmic terms, that is $\alpha = 1$ and $\alpha = 0$, respectively, gives

$$\begin{aligned} & C(n/2 + \alpha n) - c\log_2(n/2 + \alpha n) + \\ & C(n/2 + (1-\alpha)n) - c\log_2(n/2 + (1-\alpha)n) + c\log_2 n \\ = \;& C2n - 2c\log_2(3/2n) + c\log_2 n \\ = \;& C2n - 2c\log_2(3/2) - 2c\log_2 n + c\log_2 n \\ = \;& C2n - 2c\log_2 3 + 2c - c\log_2 n \\ = \;& C2n - 2c\log_2 3 + 2c - c(\log_2 2n - 1) \\ = \;& C2n - 2c\log_2 3 + c - c\log_2 2n \\ \leq \;& C2n - c(2\log_2 3 - 1) - c\log_2 2n \\ \leq \;& C2n - c\log_2 2n \end{aligned}$$

using $\log 2n = \log 2 + \log n$ and $2\log_2 3 - 1 > 0$ which then establishes the induction hypothesis.

We summarize in the following Theorem.

Theorem 13 *The merging problem can be solved work-optimally with* $T_1(n) = O(n)$ *and* $\mathrm{T}\infty(n) = O(\log^2 n)$.

The recursive Cilk merging algorithm can now be plugged into a recursive algorithm for sorting by merging.

2.4 EXERCISES

OpenMP programming exercises.

1. Quicksort. Benchmark. Cut-off
2. Prime sieve. Benchmark.
3. Recursive matrix-matrix multiplication.
4. Loop-based matrix-maztrix multiplication. Benchmark and compare.

3

DISTRIBUTED MEMORY PARALLEL SYSTEMS AND MPI

3.1 EIGHTH BLOCK (1 LECTURE)

This lecture is an introduction to performance relevant aspects of "real", parallel, distributed memory systems.

A naive, parallel distributed memory system model consists of a set of p processors each with local memory for program and data (MIMD architecture). Processors execute independently and asynchronously, and exchange information through explicit communication through an *interconnection network*. Communication is (significantly) more expensive than accessing data in local memory. The network may provide means for synchronizing processors.

In a corresponding distributed memory programming model, processes (or threads) communicate explicitly by executing commutation operations, either pairwise, or in more complex, collective patterns. Distributed memory programming models also offer means for synchronizing processes.

The concrete, distributed memory programming interface will be MPI (the *Message-Passing Interface*), which is treated in depth in the next lectures.

3.1.1 *Network Properties: Structure and Topology*

The distinguishing, new feature of distributed memory systems is the interconnection network (sometimes called just *interconnect*) needed for communication between processors, which can be individual cores, multi-core CPU's, or larger entities consisting of many multi-core CPU's, see Section 3.1.5. These entities are physically connected (electric or optical cables or other, often just called *links*), and not all these elements may be immediately, directly connected with each other; typically they are not. Also, some elements in the network may not be processors used for computation, but simply *network switches* serving communication between other network elements. It is clear that the (physical) properties of the network (speed of the connections, processing capabilities, the composition and structure of the network) plays a decisive role for the performance of algorithms and programs running on distributed memory systems. It is also clear that without a powerful interconnect, there can be no parallel computing: We are interested in non-trivial problems requiring non-trivial communication and interaction between processors.

An interconnect where the processors are also the communication elements, and in which there are no switches, is called a *direct network*. An interconnect, in which there are also special switch elements (special communication processors with connections to other elements) is on the other hand called an *indirect network*.

First we are interested in investigating how structural properties of the network influence the communication performance, and the capability to solve problems that we are interested in.

The structure or *topology* of a communication network, both direct and indirect, can be modeled as a(n un)directed, unweighted graph $G = (V, E)$, where the vertices (nodes) V denote processors or network communication elements, and the edges E model the immediate connection between communication elements. Two elements (processors or switches) $u, v \in V$ are immediately connected (adjacent neighbors) if there is a (directed) edge (arc) $(u, v) \in E$. For most communication networks, if network element u can send data directly to network element v via a link (u, v), then also v can send data directly to u, that is, communication networks are most often undirected (or bidirected), and the edge (u, v) can be used in both directions. It can nevertheless be relevant, sometimes, to consider directed graphs; and indeed there has been (few) examples of real, parallel distributed memory systems built on directed interconnection networks. When two processors u and v are not adjacent in the network, a path between u and v must be found along which u and v can then communicate. Let a path between nodes u and v have length l. Communicating some data from u to v along this path will take $l - 1$ successive communication operations.

Recall that the *diameter* of a graph $G = (V, E)$ is the maximum over all shortest paths between pairs of nodes $u, v \in V$.

$$\text{diam}(G) \quad = \quad \max\{\text{dist}(u, v) | u, v \in V\}$$

Here, $\text{dist}(u, v)$ denotes the distance in number of links that have to be traversed to get from u to v in G (shortest path length in number of edges to traverse). The diameter is a lower bound on the number of communication steps for communication operations and algorithms that involve message transmission between nodes u and v which have the longest distance in the communication network.

The degree $\text{degree}(G)$ of a graph $G = (V, E)$ is the largest number of outgoing edges from a node in G, that is the largest node degree of a node in G.

$$\text{degree}(G) \quad = \quad \max\{\text{degree}(u) | u \in V\}$$

where the node degree of $u \in V$ is given by $\text{degree}(u) = |\{v \in V | (u, v) \in E\}|$.

The *bisection width* of a graph $G = (V, E)$ is the smallest number of edges that must be removed in order for the graph to fall apart into two approximately

equally large parts, that is to partition the vertices of G into two disjoint subsets with no edges between pairs of vertices in the two subsets.

$$\text{bisec}(G) \quad = \quad \min_{V',V''\subset V, V'\cup V''=V, V'\cap V''=\emptyset, ||V'|-|V''||\leq 1} |\{(u,v)\in E, u\in V', v\in V''\}|$$

While both $\text{diam}(G)$ and $\text{degree}(G)$ can be easily computed in polynomial time for any given network topology graph G, $\text{bisec}(G)$ can (most likely) not. The problem of finding $\text{bisec}(G)$ is essentially the *Graph Partitioning* problem, one of the classical, standard NP-complete problems [28, ND14].

The best possible communication network in terms of diameter and bi-section width is the *fully connected network* $G = (V, E)$ where $(u, v) \in E$ for all $u, v \in V$ (assume either $(u, u) \in E$ or $(u, u) \notin E$ as convenient). For a fully connected network G, $\text{diam}(G) = 1$ and $\text{bisec}(G) = |V|^2/4$ (for $|V|$ even). The significant drawbacks of the fully connected network is the large (maximum) number of links, namely $|V|(|V| - 1)$ and the high degree, namely $\text{degree}(G) = |V| - 1$.

The worst possible communication networks that can support parallel computing are the *linear processor array* and the *processor ring* which are graphs $A, R = (V, E)$ consisting of either a single path from two vertices $u, v \in V$ both having degree 1 with all other vertices inbetween having degree 2, or a single cycle spanning all vertices $v \in V$ each of which have degree 2. For the linear array, $\text{diam}(A) = |V| - 1$ and $\text{bisec}(A) = 1$, and for the ring $\text{diam}(R) = \lfloor |V|/2 \rfloor$ and $\text{bisec}(R) = 2$. A significant advantage of linear arrays and rings is the small(est possible) number of links (to keep the graph connected) and the low degree.

Number of communication edges and node degrees entail concrete, physical costs (space and money) when building parallel computing systems with given network properties, as do other factors like for instance the necessary physical lengths of cables. It is therefore interesting, relevant, and highly challenging to find good compromises between costs and structural network properties desirable for supporting non-trivial parallel computing. Many different (with and with no commercial potential) solutions have been given, see for instance the aforementioned http://www.top500.org.

Numerous networks between the two extremes have been proposed and studied, see for instance [44], and are not the topic of this lecture. Only three classes of communication networks shall be mentioned, namely *trees*, d-dimensional *tori/meshes*, and *hypercubes*.

In a *tree network*, the topology graph $T = (V, E)$ is a tree (minimal connected graph over the nodes in V), most often with logarithmic diameter as in balanced binary or k-ary trees, binomial trees, etc.. Being minimal connected, tree networks have $\text{bisec}(T) = 1$, since removing any one link will make the network fall apart, and are in that sense no better than linear processor arrays or rings.

In a d-dimensional *mesh network* with dimension sizes (or orders) $r_0, \ldots, r_{d-1}$, the processors are identified with the set of d-element integer vectors $V = \{(x_0, \ldots, x_{d-1}) | x_i \in \{0, 1, \ldots, r_i - 1\}\}$. The number of processors in such a d-dimensional mesh is therefore $|V| = \prod_{i=0}^{d-1} r_i$. There is a bidirected link (u, v) between two processors $u = (x_0, \ldots, x_{d-1})$ and $v = (y_0, \ldots, y_{d-1})$ if $|x_i - y_i| = 1$ for some coordinate $i, 0 \leq i < d$ and $x_j = y_j$ for all other coordinates. A *torus network* or *torus* is a mesh network with additional "wrap-around" edges between processors at the "borders" of the mesh, that is between two processors $u = (x_0, \ldots, x_{d-1})$ and $v = (y_0, \ldots, y_{d-1})$ if $x_i = 0$ and $y_i = d_i - 1$ for some ith coordinate and $x_j = y_j$ for all other coordinates $j \neq i$. The diameter of a mesh $M = (V, E)$ is $\text{diam}(M) = \sum_{i=0}^{d-1}(r_i - 1)$, and the degree is $\text{degree}(M) = 2d$. The diameter of a torus $T = (V, E)$ is $\text{diam}(T) = \sum_{i=0}^{d-1} \lfloor r_i/2 \rfloor$, and the degree likewise $\text{degree}(T) = 2d$.

A uniform (symmetric, homogeneous) mesh or torus network have the same order for all dimensions, $r = \sqrt[d]{p}$. The bisection width of a symmetric mesh is $\text{bisec}(M) = p^{\frac{d-1}{d}} = p/\sqrt[d]{p} = p/r$ and of a symmetric torus $\text{bisec}(T) = 2p^{\frac{d-1}{d}} = 2p/r$ (for r even).

A *hypercube network* $H = (V, E)$ is a special case of a uniform torus (or mesh) network in which all coordinates are either $x_i = 0$ or $x_i = 1$ (note that in this case, mesh and torus coincide, the torus has no more edges than the mesh). Thus, the number of processors is $p = 2^d$ for some d, that is a power of 2, or, the other way around, the dimension of a p-processor hypercube is $d = \log_2 p$. Each processor has d neighboring processors which for processor $u = (x_0, \ldots, x_{d-1})$ are found by changing one of the i coordinates from x_i to $1 - x_i$ (flipping the ith bit in u when viewed as a binary number). Both the degree and the diameter of a hypercube is $\text{degree}(H) = \text{diam}(H) = d = \log_2 p$. The bisection width is $\text{bisec}(H) = p/2$.

Modern high-performance systems are often built as torus networks of $d = 3, 5, 6$ dimensions, or as indirect networks with multiple switches of small, fully-connected networks, often called *multi-stage networks* of which there are many examples. Hypercube networks were once popular, but are currently not built (what could some reasons be?).

3.1.2 *Communication algorithms in networks*

Communication from a processor u to another processor v in a given network $G = (V, E)$ requires at least $\text{dist}(u, v)$ communication steps in which processor u sends data to a neighboring processor that is closer to v (along an edge in E), that in turns sends data to a neighboring processor that is closer to v (along an edge in E), etc., until the data reaches v. This is regardless of the amount of data to be transferred and the concrete costs incurred by sending and receiving some amount of data (see later). It is relevant to study the number of such communication steps that may be required for other, more complex

communication operations, apart from just the transmission of information from one processor to another. We therefore first assume that data to be communicated are all of some small unit, and that each communication step takes the same unit of time.

In a communication step, a processor $u \in V$ can communicate with a neighbor in the communication network $G = (V, E)$. What exactly a processor can do in a communication step depends on the *capabilities* of the communication system. We say that a communication system is *one-ported* (or *single-ported*) if a processor can engage in at most one communication operation in a step. A communication system where at processor can be involved in up to k communication operations in the same step (that is, concurrently) is called *k-ported* (or just multi-ported).

If communication in a step between neighboring processors $u \in V$ and $v \in V$ with $(u, v) \in E$ is only in one direction from u to v or from v to u, communication is *unidirectional*, and the communication system is said to be unidirectional if it can support only unidirectional communication in a step. Communication in both directions, from u to v and from v to u is bidirectional, telephone-like (in an old sense of "telephone" where two parties can speak at once), and a communication system that can support such communication is said to be bidirectional. Communication where a processor u receives from a processor w and sends to a processor v is general, bidirectional, send-receive, and a system that can support such communication in a step is said to be bidirectional in the general, send-receive sense.

Most modern communication systems and networks can, roughly, support general, bidirectional send-receive communication. Systems with indirect, multi-stage communication networks are often one-ported, whereas torus-based systems are most often $2d$-ported and can therefore, roughly, support communication with all torus neighbors in a step.

Processors in a communication network can work independently and concurrently. For the analysis of communication algorithms, we count the total number of steps in which processors are communicating that are required for solving the given problem, that is, for the last processor to finish. In each step, some or all of the processors in the network may be involved. Sometimes, such steps are called rounds.

Interesting communication problems often correspond to parallelization patterns (see Section 1.3.4) that are useful in complex algorithms and applications, for instance broadcasting data from one processor to other processors, exchanging information between all processors, etc.. In any such communication pattern that involves transmission of data from a processor u to a processor v where $\text{dist}(u, v) = \text{diam}(G)$, an obvious lower bound on the number of steps required to complete the pattern operation is $\text{diam}(G)$. One such pattern is the broadcast operation which we formalize as the following communication problem.

Definition 11 (Broadcast problem) *Let $G = (V, E)$ be a communication network, and $r \in V$ a given* root processor *which has some data that needs to be transmitted to all other processors $u \in V$. The* broadcast problem *is to devise for a given network $G = (V, E)$ and any root $r \in V$ an algorithm with the smallest possible number communication steps that transmits the data from r to the other processors of G.*

Both the (structure and capabilities of the) network G and the chosen root processor r are known to all processors and can be used in the algorithm. A solution to the broadcast problem for some class of communication networks is usually an algorithm with the communication steps for each processor that solves the problem for any r. In particular, G is not part of the input but fixed and given and can be used in the algorithm design, whereas the root processor r is usually taken to be an input parameter, that is, however, known to all processors.

In tree, torus, and hypercube networks, the diameter lower bound argument gives a non-constant bound, depending on the number of processors in the network, for solving the broadcast problem. But even in a fully-connected network with constant diameter one, the number of communication rounds is non-constant, as captured by the following, important statement.

Theorem 14 *In a fully-connected network, p-processor network $G = (V, E), p = |V|$ with k-ported, unidirectional communication capabilities for $k \geq 1$, the number of communication rounds necessary and sufficient for solving the broadcast problem is $\lceil \log_{k+1} p \rceil$.*

The proof for the lower bound part of the claim is the following information-theoretic argument. The best that an algorithm that solves the broadcast problem can do is the following. In the first communication step, only the root processor has the data, and can disseminate the data it has to at most k new processors that so far did not have the data. In the next round, the best that each of the $k + 1$ processors that now have the data can do is to disseminate the data to $k + 1$ new processors that so far did not have the data. Thus, from one communication round to the next, the best than an algorithm can achieve is that a factor of $k + 1$ more processors now have the data. The smallest number of communication rounds i that are required for all processors to eventually receive the data is found by solving $(k + 1)^i \geq p$ which by taking the (natural, any) logarithm on both sides gives $i \ln(k + 1) \geq \ln p$ which is $i \geq \lceil \log_{k+1} p \rceil$ since the solution (number of rounds) must be integral.

The argument almost also leads to an algorithm that matches this lower bound, namely by the following idea. Partition the communication network into $k + 1$ pieces of roughly the same number of processors. The root processor

r belongs to one of these pieces; for the other pieces, a virtual root processor is chosen (the processors must be able to do this with no communication, based on the information they have on the identity of r and the fact that G is fully connected). The "real" root sends the data it has to the k virtual roots. The broadcast problem has now been reduced to $k+1$ proportionally smaller broadcast problems (still on fully-connected networks), and these can be solved recursively, in parallel. The number of recursive steps needed for all pieces to have been reduced to a single processor is $\lceil \log_{k+1} p \rceil$.

Good and even optimal solutions for the broadcast problem, in the sense of matching a known lower bound, for many types and classes of networks, like trees, tori, and hypercubes (and many, many others) are known, but not always trivial, and not the subject of these lectures.

The broadcast problem for an arbitrary graph G (as part of the input) is NP-complete [28, ND49].

3.1.3 *Concrete communication costs*

Communication mostly involves not only small, indivisible units of information, but (complex) data of some size m (Bytes, integers, other relevant, but stated unit). What is the cost (in time) of transmitting such data between processors in the network?

As a first shot, often a simple linear time cost model is adopted for the concrete costs of transmitting data of m units (Bytes) from processor u to processor v. The linear transmission cost model states that transmitting m units from u to v along a communication edge $(u, v) \in E$ takes

$$\alpha + \beta m$$

time units, where α is a fixed, *start-up latency* (for the given network) and β a *time per unit* of data transmitted.

The linear time cost model is a crude, first, and perhaps even misleading approximation of the cost of communication between processors in a network, or distributed-memory parallel computing system. When at all, such a model is (tacitly) assumed in the analysis of the distributed memory algorithms in this lecture. The model correctly emphasizes that communication takes time, both in terms of cost per unit and latency, and both of these terms can be considerable.

3.1.4 *Routing and Switching*

In a not fully connected network $G = (V, E)$ where not every processor can communicate directly with any other processor, a general purpose *routing system* (*routing algorithm*, *routing protocol*) shall make it possible for any processor $u \in V$ to send data to any other processor $v \in V$ via some path of intermediate

processors in V. In a sense, the routing system turns a not fully connected network of processors into a virtually fully connected network where any processor can communicate directly with any other other processor, however, not necessarily at the same cost of communication (see Section 3.1.3). A routing algorithm could be *centralized*, but is rather a set of local, per processor/switch algorithms, each making decisions on what to do with a received message based on its own state and possibly the state of some of the immediately adjacent processors or switches (local information). Some parallel algorithms are designed entirely without a routing system by explicitly (pre)computing how processors communicate along which paths with each other. Such an approach can make it possible to give more precise, better bounds on the expected running time, but is not general purpose and comes with a high design cost (specialized algorithm). A routing system may be realized in hardware, in software, or in a combination of hard- and software (therefore the term routing *system*). Designing and analyzing routing algorithms for different types of graphs is a typical distributed computing topic (recall Definition 3), but routing systems and algorithms are not a topic of this lecture. A few terms are useful, though.

The most important requirement to a routing system (algorithm, protocol), is *deadlock freedom*: A message sent from a processor u to a processor v must eventually arrive correctly (uncorrupted) at processor v, *regardless* of any other traffic in the communication network. A deadlock could arise when two processors or network elements at the same time require a resource, for instance, want to send data to the same processor or switch, possibly over the same edge, and the conflict cannot be resolved. It may also be seen as the task of the routing system to ensure *reliable communication*: no data lost, no data corrupted, perhaps even that data are delivered in some specific order (as must for instance be guaranteed by MPI, see Section 3.2.10). This is important, since network hardware does not always guarantee such properties.

A routing systems should be (as) fast (as possible). In the linear time cost model, routing data of m units from processor u to processor v along a path of length l would take $l(\alpha + \beta m)$ time units. For larger number of data units, this can be improved by *pipelining* as follows: The m units are separated into smaller *packets* of some maximum size of b units (assuming $m > b$) that are sent one after the other. The time for the last packet to arrive at the destination processor v would then be

$$\begin{aligned} l(\alpha+\beta b)+(\lceil m/b\rceil-1)(\alpha+\beta b) &= (l+\lceil m/b\rceil-1)\alpha+\beta(l-1)b+\beta m \\ &= (l-1)\alpha+\lceil m/b\rceil\alpha+\beta(l-1)b+\beta m \end{aligned}$$

The first term on the left-hand side is the time for the first packet to arrive at v. The last term is the time for each following packet, of which there are $\lceil m/b\rceil - 1$ in total. Sending all $\lceil m/b\rceil$ packets has a cost of βm since the last packet may be smaller than b units. If the packet size b can be chosen freely, a

best possible packet size minimizing the total transmission time can be found (by calculus, or) by balancing the terms $\lceil m/b \rceil \alpha$ and $\beta(l-1)b$ which both depend on b. This yields a best packet size b of

$$b = \sqrt{\frac{m}{l-1}}\sqrt{\frac{\alpha}{\beta}}$$

and a shortest transmission time of

$$(l-1)\alpha + 2\sqrt{(l-1)m}\sqrt{\alpha\beta} + \beta m$$

provided that $l > 1$. The important result is that the last βm term with pipelining is no longer linearly dependent on the path length l. Routing with pipelining is sometimes called *packet switching*, whereas routing along a path without pipelining is called *store-and-forward*. Both store-and-forward and packet switching routing require some intermediate buffer space in the routing system, either for all m data units, or for a block of up to b units. These and other terms are used somewhat differently in different fields, depending on the level at which the network is examined, the use (internet computing is different from parallel computing!), tradition, and many other factors [67].

In a communication network there may be several, partially different paths from a processor u to a processor v. When data are to be sent from processor u to processor v, the routing system chooses an appropriate path. This choice of course depends on u and v and the network topology $G = (V, E)$, but may also depend on the current traffic in the system, that is concurrent communication between other processors.

With *deterministic (oblivious) routing*, the route is determined solely by the endpoints u and v and the structure of the network G, whereas network traffic plays no role. With *adaptive routing*, the routing system takes other communication into account, and thus the route from u to v can be different from time to time. A routing algorithm uses *minimal routing* when routing from u to v is always along a shortest path (of length $\text{dist}(u, v)$). When several paths are possible, and pipelining (packet-switching) is employed, it may be that different blocks (packets) are taking different routes. In such cases, packets could potentially arrive at the destination processor v in a different order than the order in which they were sent from the source processor u. It is then the task of the routing system to correctly assemble the packets in the right order at the destination.

In the presence of traffic in the communication network due to many pairs of processors communicating at the same time, the optimistic, model based estimate of the transmission time from u to v will of course not hold, and data communication times will (for some pairs of processors) be higher. This is due to network *contention*, for instance of edges (u, v) that occur in multiple paths and are needed by several pairs of processors (the best that can be hoped for is a serialization slowdown), and resource *congestion* by too high load on, say,

intermediate buffers or processors. The routing system can apply different strategies to alleviate and control contention and congestion, typically some form of *flow control*.

3.1.5 *Hierarchical, Distributed Memory Systems*

In modern parallel computing systems, the communication system has a more complex, hybrid structure, consisting of communication networks at different levels. Thus, a single, unweighted graph that alone describes the topology of the whole system may not be adequate or helpful.

A two-level hierarchical system, for instance, could consist in a number of shared-memory "compute nodes" interconnected by a, typically, indirect network. Thus, processor-cores within the same shared-memory compute node may have different communication characteristics than processor-cores residing on different compute nodes. In particular, if several processor-cores on the same compute node need to communicate with processor-cores on other compute nodes, they will have to share the network that interconnects the compute nodes.

3.1.6 *Programming Models for Distributed Memory Systems*

Programming models for distributed memory systems usually abstract away from concrete network properties as discussed in the previous sections, and assume that the active entities of the model (processes, threads, ...) can freely communicate as in a fully connected network. Processes are usually not synchronized, operate on local data that are invisible to other processes (shared nothing), according to local programs (SPMD or MIMD), and cooperate (exchange data, synchronize) with the other processes by explicit or implicit communication. Programming models also usually assume that message transmission between processes and threads is reliable (deadlock free, correct), and sometimes ordered according to certain constraints and rules. For the implementation of such programming models, it is the task of the runtime system and routing algorithms to ensure reliable message delivery between any processes in the model. Distributed memory programming models sometimes provide means for reflecting and exploiting properties of the underlying, hierarchical communication system. The programming model underlying MPI is a good example.

Pragmatically, if one measures communication under different loads and between processes residing in different parts of the system, network and system properties will become manifest (and can sometimes be concretely inferred). Such differences may be reflected in the cost models for the programming model. MPI does not come with a cost model. Strictly speaking, all analysis of MPI programs will be based on model external assumptions, benchmarking results, and known system properties.

Distributed system programming models can be classified as either *data distribution centric* or *communication centric*. In a data distribution centric model, the data structures allowed by the model (arrays, multi-dimensional arrays, vectors, matrices, tensors, complex objects, ...) are distributed according to given rules across the processes. When a process accesses or updates a part of a distributed data structure that is residing with another process, communication is implied. A communication centric model usually does not define distributed data structures, and the model instead focusses on properties of explicit communication and synchronization operations.

Examples of data distribution centric models are so-called *Partitioned Global Address Space* models (*PGAS*). In such models, data structures, typically simple arrays, can be distributed across threads (processes), and access to non thread local parts of arrays implicitly leads to communication. An example implementation of a PGAS model is *Unified Parallel C* (*UPC*) [23]. PGAS models and languages will not be treated further in these lectures.

MPI is on the other hand a communication centric model.

3.2 NINTH BLOCK (3 LECTURES)

Our concrete example of a distributed-memory programming interface implementing a distributed-memory programming model is *MPI*, the *Message-Passing Interface* [51]. MPI is an older interface dating back to around 1992, widely used (especially in HPC), and relevant to study and learn because of the concepts it introduces and its still widespread use. MPI is an interface for C and Fortran (still an important programming language in HPC). MPI is maintained and developed further by the so-called MPI Forum, an open forum of academic institutions, laboratories, compute centers and industry; incidentally, many of the MPI Forum members are also part of the OpenMP ARB. The standard is freely available and can be found via `www.mpi-forum.org`. These pages also gives information on the standardization process (currently towards MPI 4.0).

The reference for programming (and learning) MPI is the latest version of the standard [51]. Some helpful reading are the series of books on "Using MPI" [29, 30, 31].

This lecture (and the next ones) gives an introduction to MPI for Parallel Computing, covering all its fundamental concepts and features. Some aspects of MPI will not be dealt with, most notably support for I/O, dynamic processes, and tools' building. Our HPC Master lecture deals with some of these things.

3.2.1 *The Message-passing Programming Model*

The message-passing programming model goes way back at least to papers by Dijkstra and Hoare in the 60ties and 70ties. The idea is to structure parallel computations as sequential processes with no shared information

that communicate explicitly by sending and receiving *messages* between each other [36, 37], as a means to develop (provably) correct, parallel and concurrent programs. The message-passing model is called *Communicating Sequential Processes* (*CSP*). CSP programs in particular cannot have data races. The programming model that is implicitly behind MPI is much wider in scope than CSP in that it incorporates both synchronous and asynchronous point-to-point communication (CSP focussed on synchronous, handshaking communication), one-sided communication and collective communication, and provides features for data layout description, interaction with the communication system and external environment (I/O).

Some main characteristics of the MPI message-passing programming model are:

1. Finite sets of processes (in communication domains) that can communicate.
2. Processes ordered by rank in communication domains.
3. Ranks are successive $0, \ldots p-1$, with p the number of processes in the communication domain (size).
4. More than one communication domain possible, and created relative to default domain of all started processes. Processes can belong to several communication domains.
5. Processes operate on local data, all communication between processes explicit.
6. Communication is reliable and ordered.
7. Network oblivious, communication between all processes is possible.
8. Three basic communication models:
   (a) Point-to-point communication, different modes, non-local and local completion semantics.
   (b) One-sided communication, different synchronization mechanisms, local completion mechanisms.
   (c) Collective operations, non-local (and local) completion semantics.
9. Structure of communicated data orthogonal to communication model and mode.
10. Communication domains may reflect physical topology.

MPI has no performance model, and there is no prescriptions in the MPI standard on how the many, many different MPI constructs are to be implemented nor on which algorithms are to be used (for instance for the collective communication operations).

However, MPI is designed, so is the intention, to make high-performance implementations possible on wide ranges of parallel computing systems, meaning that the functionality and semantics is close to what an underlying communication system will offer, that preprocessing and communication of meta-information is not necessary (or strictly confined), and that memory required by library internals is limited and/or can be controlled. These design objectives explain the concrete "look-and-feel" of many of the MPI functions.

3.2.2 *MPI in C*

MPI is a library and MPI functionality can be used by linking the code against an MPI library. There are several such libraries available, notably the open source libraries `mpich`, `mvapich`, and OpenMPI as well as vendor libraries, often for specific High-Performance Computing systems. C code using MPI must include the function prototype header with the `#include <mpi.h>` preprocessor directive. All MPI relevant functions and predefined objects are prefixed with `MPI_` which identifies the MPI "name space". It is considered illegal to use the `MPI_` prefix for own functions or objects in the code. MPI programs are usually compiled with a special compiler (wrapper, mostly) that takes care of proper linking against the MPI library, for example `mpicc`, which will accept also standard optimization options and arguments.

We explain the MPI functions by listing the C prototypes that give the types of all arguments, and explain the outcome for given inputs (loose before-after explanation).

MPI functions return an error code, and it is good practice to check the error code (which is often not done). The error code `MPI_SUCCESS` means "success".

3.2.3 *Compiling and Running MPI programs*

An MPI program is, unlike an OpenMP program, simply a C (or Fortran) program with library calls to the MPI functions, and therefore MPI programs can be compiled with a standard C compiler. To ease linking against the MPI library, normally an `mpicc` compiler command is provided that is just a wrapper around the C compiler command and will therefore take the standard C compiler flags and options.

Running an MPI program with a desired number of processes is more complex. Resources, cores, compute nodes, for the processes must be allocated, and the processes started at the allocated computing resources. For small, stand-alone systems (your laptop or workstation) this can sometimes be done with a command-line command like `mpirun`. More commonly, and on larger systems, a batch scheduling system like `slurm` is used.

When processes have been started, they become MPI processes after having initialized the MPI library. In the MPI context, processes are most often bound

("pinned") to a specific processor-core, or at least compute node. This binding is outside the control of MPI.

It is (usually) possible to start more MPI processes than there are physical processor-cores in the system. But as with OpenMP and `pthreads`, such *oversubscription* must be used with care.

3.2.4 *Initializing the MPI Library*

After the processes are started on the system, the internal data structures of the MPI library must be initialized. This done by the `MPI_Init` call which takes the standard C argument count and array as arguments. After use, all activity of the MPI library is completed and resources freed with an `MPI_Finalize` call. Prior to `MPI_Init`, and after `MPI_Finalize`, no MPI calls can be performed, except for the two check calls `MPI_Initialized` and `MPI_Finalized` that tells the caller (perhaps an application specific library written with MPI with its own initialization function) whether MPI has been initialized or completed. When the MPI library has been finalized, it cannot be initialized again within the same program.

```
int MPI_Init(int *argc, char ***argv);
int MPI_Finalize(void);
int MPI_Finalized(int *flag);
int MPI_Initialized(int *flag);

int MPI_Abort(MPI_Comm comm, int errorcode);
```

The `MPI_Abort` call can be used to force termination of the running MPI program in an emergency situation.

An MPI library can provide (limited) information about itself (and its environment) by the following operations.

```
int MPI_Get_version(int *version, int *subversion);
int MPI_Get_library_version(char *version, int *resultlen);

int MPI_Get_processor_name(char *name, int *resultlen);
```

These calls illustrate the tediousness of MPI being a library (and the shortcomings of C for manipulating strings): For the strings `version` and `name`, the user must reserve space of at least `MPI_MAX_LIBRARY_VERSION_STRING` and `MPI_MAX_PROCESSOR_NAME` characters, respectively. The strings are copied into these arrays, in C properly terminated by a null character, and the number of actual characters, excluding the trailing null character, stored in `resultlen`. Thus, in C, output argument (result values) are always of pointer type.

A process can read the wall-clock time from some point in the past (in seconds). The timers are local, and (usually) not synchronized. The call can be used to time process local operations, and is heavily used for this.

```
double MPI_Wtime(void);
double MPI_Wtick(void);
```

Whether the timers are synchronized (global) can be queried by reading an attribute. The attribute mechanism of MPI is not covered in these lectures, although it is important for library building with MPI. The existence of the attribute mechanism illustrates again how MPI supports portable application specific library building, but also the tediousness of MPI being a library and not an integrated part of a programming language. Information must flow in and out of the library to and from the application (specific library).

3.2.5 *Failures and Error Checking in MPI*

The MPI interface functions are, at first sight, often quite involved and take many arguments that have to be used correctly. If an argument (precondition) is not as specified, there is no guarantee that the function will have the specified effect and produce the desired outcome. MPI performs only rudimentary argument (precondition) checks, but the extent of this is not specified in the standard, and MPI libraries differ in the amount and kinds of such checks done; sometimes tools or options can be used to perform more extensive checking which can of course be helpful in the development phase of an application. But the programmer can most surely not rely on the MPI library to catch mistakes and errors. The MPI standard specifically states [51, page 340] that "An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. ... The set of errors that are handled by MPI is implementation dependent. ... Specifically, text [in the standard] that states that errors *will* be handled should be read as *may* be handled" (emphasis original).

As mentioned, almost all MPI functions return an error code, and it can make sense to check this and try to take action on certain error return codes. But there is no guarantee that this will be possible, the application may (and often will) have crashed and no error code will ever be seen. MPI programs therefore typically do only a limited amount of error code checking. In particular, communication failures due to processor/node crashes or failures in the communication system are typically not handled, and will in most cases cause the whole application to abort. The own, most common reason for your application to crash is memory corruption through wrong use of MPI functions (here memory tools can be useful).

Part of the reason for MPI not doing extensive error checking and handling is that MPI is designed to allow high-performance implementations, and therefore do not impose (expensive) checks for errors and wrong usage of the MPI functions.

MPI aims to make it possible to control the response of the library in case of failures. This is accomplished through *error handlers* which are special

functions that can be attached to communicator objects (see next section) and are invoked by the MPI library when an error condition occurs in an MPI call on that communicator object. Error handlers are beyond the scope of this lecture. The quotes from the MPI standard cited above still apply.

```
int MPI_Errhandler_create(MPI_Handler_function *function,
                          MPI_Errhandler *errhandler);
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler);
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler);
int MPI_Errhandler_free(MPI_Errhandler *errhandler);
```

3.2.6 *MPI Concepts: Communicators*

After processes have been started and the MPI library initialized, the started processes are put into the communication domain called `MPI_COMM_WORLD` (in addition, each process is also put into a domain by itself called `MPI_COMM_SELF`). Communication domains are called *communicators* in MPI. A *communicator* is a distributed object that can be operated upon by all processes belonging to the communicator. A communicator is referenced by a *handle* of type `MPI_Comm`. In particular, processes can look up the *size*, that is the number of processes, in a communicator `comm`, and each process can determine its own *rank* in a communicator by the following functions.

```
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

Thus, the code snippet

```
int rank, size;

MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

will, when executed by any of the started MPI processes, identify the process relative to all other started processes by its serial number (rank) in the `MPI_COMM_WORLD` communicator. A typical MPI program will have such a code sequence somewhere after the `MPI_Init` call, and the processes decide what to do based on `rank` and `size`. Here (as always), the error return codes of the two function calls are ignored.

This trivial piece of code illustrates a number of important MPI concepts and principles.

- Processes belong to communication domains which are called communicators in MPI, in particular to the `MPI_COMM_WORLD` communicator consisting of all started processes.

- Processes have a *rank* (serial number) in a communicator, ranks are successive from 0 to $p-1$ where p is the size of the communicator ($0 \leq$ `rank` $<$ `size`).
- The rank of a given process in `MPI_COMM_WORLD` is determined by external factors (how the processes were started). The rank of a process in a communicator will never change.
- Communicators are identified by handles of type `MPI_Comm`, and are opaque objects on which certain operations are defined.
- There can be several communicators in an application, and the same process can belong to many communicators, possibly with a different rank in each.
- The communicator is a (the) most fundamental object/concept in MPI: All communication is relative to a communicator, all collective operations (see later) are relative to a communicator. In particular, processes from different communicators cannot communicate, and simultaneous communication on different communicators can never interfere.
- MPI objects are static objects, they cannot be changed (only given free), but new objects, for instance communicators, can be created from already existing ones by appropriate functionality.

For any communicator there is a special process rank `MPI_PROC_NULL` outside the range form 0 to `size` -1 that can actually be referenced and used for non-communication: A communication with `MPI_PROC_NULL` has no effect (see later).

The principle that communication is relative to a communicator, and that communication between processes in one communicator can never interfere with communication between processes in another communicator is fundamental. It is what allows construction of *safe, parallel libraries*. If each library used in an application uses its own communicator(s), communication going on in different libraries can never interfere.

For library construction, the fundamental operation on communicators is the creation of a duplicate communicator. The duplicate represents a communication domain with the same set of processes in the same order, but nevertheless a different domain. Thus, communication on a communicator and its duplicate can never interfere. The `MPI_Comm_dup` operation is shown below. It is the first example of a so-called *collective operation*, meaning that it has to be called by all processes in the communicator `comm`.

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm);
int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm
                           *newcomm);
```

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm);

int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,
                          MPI_Comm *newcomm);

int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
                        MPI_Info info, MPI_Comm *newcomm);
```

The `MPI_Comm_split` and `MPI_Comm_create` functions allow to create new communicators from existing ones possibly with fewer processes, and possibly with a different order. Both calls are collective, so all processes in the `comm` argument have to make the call. In particular, `MPI_Comm_split` takes an integer `color` argument, and all processes giving the same color will end up in a same `newcomm` communicator. The `key` argument can be used to control the numbering of the processes in the new communicator. Processes with the same color are sorted after the `key` argument, and this determines their ranks in `newcomm`. The special `MPI_UNDEFINED` argument as color, indicates that a process calling with this color is not going to belong to any communicator. Again, this discussion illustrates some fundamental principles.

- MPI functions have input and output arguments. Output arguments in C have pointer type (we saw this already with `MPI_Comm_rank` and `MPI_Comm_size`).

- There are functions in MPI that are *collective*, meaning that they have to be called by all processes in the input communicator. Collective functions are *always* called symmetrically, that is, all processes (in the communicator) makes the same call, but possibly with different arguments. The input arguments given by a process determine the role of that process in the call.

- On return from an `MPI_Comm_split` call, all processes will have, in addition to the still existing, unchanged input communicator `comm`, a new communicator `newcomm` to which they belong together with all the other processes that called with the same `color` argument, and with rank determined by the position in the list of processes with the same color sorted after the `key` argument.

- After completion of a communicator creating operation, each calling process will (in case of `MPI_Comm_split`) belong to two communicators, `comm` and `newcomm`, possibly of different sizes, and possibly with a different rank in each.

- New processes cannot be created or started (by this functionality). The communicator creating functions operate on a given set of processes

represented by an input communicator, only ranks and sizes can be different in the created communicators.

The `MPI_Comm_create` call likewise allows to create arbitrary new communicators from old ones. This is based on process groups, a new concept that will be explained briefly later. A difference is that the `newcomm` returned to some processes can be an invalid `MPI_COMM_NULL` communicator, a communicator with no operations and that can mostly not be used as input argument to MPI functions. The last two operations, albeit useful, are not treated in these lectures.

In the following, there will be concrete examples of the use of `MPI_Comm_split` and `MPI_Comm_create`, for instance in the implementations of Quicksort-like algorithms.

After use, a communicator is freed by the `MPI_Comm_free` call. Since communicators are distributed objects, all processes in the communicator have to eventually call `MPI_Comm_free` on the communicator.

```
int MPI_Comm_free(MPI_Comm *comm);
```

A communicator is typically a "costly object" in MPI in terms of required memory space (depending on the quality of the MPI library implementation), so also for that reason it is *always* good practice to free MPI objects that are no longer going to be used.

There is sometimes-helpful functionality in MPI for comparing two communicators.

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result);
```

The possible outcomes are `MPI_IDENT`, meaning that the two input communicators are indeed referring to the same object, `MPI_CONGRUENT`, meaning that the two input communicators represent the same processes in the same order, `MPI_SIMILAR`, meaning that the two input communicators represent the same processes but not necessarily in the same order, and `MPI_UNEQUAL` for anything else. A communicator and its duplicate would thus be `MPI_CONGRUENT`, but not `MPI_IDENT`. This functionality is typically for use in application specific libraries, and most likely never used directly in an application.

3.2.7 *Organizing Processes*

In these lectures we touch briefly on functionality to give more structure to the organization of MPI processes than just the rank in a communicator.

A running example through the lectures is the stencil computation. In a large d-dimensional "matrix", all entries have to be updated according to the same stencil rule for each entry, for instance an average over neighboring elements "up, down, left, right, front, rear" (3-dimensional example) [22], and this update is iterated a large number of times, until some convergence criterion

is met. In a distributed memory, message-passing setting, the "matrix" is conveniently cut into "rectangular" submatrices, a submatrix for each process, with all submatrices being of roughly the same size. We will return to this example shortly.

For the communication that is needed for a parallel implementation of the stencil update, it can be convenient to be able to think of the processes as points in a d-dimensional, integer grid. MPI communicator creation functionality makes it possible organize the processes into such a d-dimensional grid by giving each process a d-dimensional coordinate vector describing its position in the grid. A communicator with an imposed grid structure is called a *Cartesian communicator*, and Cartesian communicators are created and used with the functionality listed below.

```
int MPI_Cart_create(MPI_Comm comm_old,
                    int ndims, const int dims[], const int periods[],
                    int reorder, MPI_Comm *comm_cart);
int MPI_Cartdim_get(MPI_Comm comm, int *ndims);

int MPI_Cart_get(MPI_Comm comm,
                 int maxdims, int dims[], int periods[], int coords[]);

int MPI_Dims_create(int nnodes, int ndims, int dims[]);

int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm);

int MPI_Topo_test(MPI_Comm comm, int *status);
```

A Cartesian communicator, the `newcomm` returned by the `MPI_Cart_create` call, is like any other communicator, and can be used wherever a "normal" communicator can be used, but carries additional information about the size of the grid, namely the number of dimensions d and the size along each dimension. The number of dimensions d is given as input `ndims`, and the size of the dimensions is stored in the input array `dims[]` with d entries. It must hold that $\prod_{i=0}^{d-1} \texttt{dims}[i] \leq p$ where p is the size of the communicator `comm_old`. If $\prod_{i=0}^{d-1} \texttt{dims}[i] < p$ some processes in `comm_old` will not be part of the new `comm_cart` communicator, and these processes will be returned the value `MPI_COMM_NULL`. The Cartesian grid is the set of integer vector coordinates

$$\{(c_0, c_1, \ldots, c_{d-1}) | 0 \leq c_i < \texttt{dims}[i], 0 \leq i < d\}$$

and each process in `cart_comm` is uniquely associated with one such vector. The association of processes with vectors is by *row major* assignment ("last coordinate change the fastest"). More precisely, a process with coordinates $(c_0, c_1, \ldots, c_{d-1})$ thus has rank r with

$$r = \sum_{i=0}^{d-1} c_i \prod_{j=i+1}^{d-1} d_j$$

where d_j = `dims[j]` (and the empty product $\prod_{j=i+1}^{d-1} d_j$ for $i = d-1$ being 1). The rank r can of course be computed in $O(d)$ steps (better than the $O(d^2)$ of the formula).

The `periods` array is a Boolean (0/1) array indicating whether the grid is periodic in the ith dimension, $0 \leq i < d$. Periodic in the ith dimension means that a coordinate vector $(c_0, \ldots, d_i, \ldots, c_{d-1})$ is treated as $(c_0, \ldots, 0, \ldots, c_{d-1})$, and $(c_0, \ldots, -1, \ldots, c_{d-1})$ as $(c_0, \ldots, d_i - 1, \ldots, c_{d-1})$. The grid "wraps around" in the ith dimension. A full torus is a grid that is periodic in all d dimensions.

The placement of the MPI processes in a grid via the `MPI_Cart_create` operation carries with it an implied, preferred communication pattern, namely that each process is likely (in the application) to communicate with its immediate neighbors in the grid along the d dimensions. It is implied that a process with coordinate vector $(c_0, \ldots, c_i, \ldots, c_{d-1})$ will most likely communicate (only, in a preferred way) with the $2d$ processes $(c_0, \ldots, c_i \pm 1, \ldots, c_{d-1})$ for each $i, 0 \leq i < d$. If the grid is not periodic in dimension i, then some neighbors might be non-existing, which is represented by `MPI_PROC_NULL`, the non-existing process mentioned in Section 3.2.6.

The `MPI_Cart_create` takes a new type of argument, the `reorder` flag. Setting this flag allows the MPI library to attempt to reorder (rerank) the processes in the input communicator, so as to better reflect the process communication pattern that is implied by the process grid organization, namely that processes that are neighbors in the grid (see also Section 3.1.1) will communicate most intensively. More concretely, the idea is that processes that are expected to communicate by being grid neighbors, are ranked to processes on processor-cores in the physical system that are also close to each other, for instance by having a direct communication link. Whether, how and to what extent an MPI library does such a reranking and what the benefits will be in concrete applications is entirely implementation dependent.

The `MPI_Cart_get` operations are again for library building purposes and can be used to query a communicator created with `MPI_Cart_create`. Whether this is the case can be checked with the `MPI_Topo_test` operation which will in that case return the value `MPI_CART`.

For setting up Cartesian communicators over an existing communicator of size p (that is, with p MPI processes), the `MPI_Dims_create` function can be helpful for factoring p into d factors that are close to each other. The factors are returned in non-increasing order in the `dims` array that must be initialized to a non-negative value. Positive entries indicate factors that are already set and fixed, so only

$$\frac{p}{\prod_{\texttt{dims[}i\texttt{]}>0} \texttt{dims[}i\texttt{]}}$$

will be factored over the zero entries in `dims`.

```
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank);
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[]);
```

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
                   int *rank_source, int *rank_dest);
```

The functions `MPI_Cart_rank` and `MPI_Cart_coords` are used to translate between ranks and coordinate vectors. Cartesian communicators, in combination with `MPI_Comm_split`, are used later in the lectures to ease the implementation of the SUMMA matrix-matrix multiplication algorithm (see Section 3.2.28). The shift operation `MPI_Cart_shift` can be used to compute the ranks of processes along the *i*th `direction` (dimension) by giving an integer displacement. We will see an example in Section 3.2.9. Here is a first, small, full-fledged MPI program setting up (and freeing) new, Cartesian communicators, and verifying the row-major placement of the MPI processes in each of the created, non-periodic Cartesian grids:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#include <assert.h>

int main(int argc, char *argv[])
{
  int d, p;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&p);

  for (d=1; d<=p; d++) {
    int dims[d], periods[d];
    int coords[d];

    MPI_Comm cartcomm;
    int rank, size;
    int r, dd, i;

    for (i=0; i<d; i++) dims[i] = 0;
    MPI_Dims_create(p,d,dims);

    for (i=0; i<d; i++) periods[i] = 0;
    MPI_Cart_create(MPI_COMM_WORLD,d,dims,periods,0,&cartcomm);
    assert(cartcomm!=MPI_COMM_NULL);

    MPI_Comm_rank(cartcomm,&rank);
    MPI_Cart_coords(cartcomm,rank,d,coords);

    r = 0; dd = 1;
```

```
    for (i=d-1; i>=0; i--) {
      r += coords[i]*dd;
      dd *= dims[i];
    }
    assert(r==rank);

    MPI_Comm_free(&cartcomm);
  }

  MPI_Finalize();

  return 0;
}
```

The idea of specifying a likely pattern of most intense communication based on which the MPI library can attempt to rerank processes is generalized with the so-called *distributed graph communicators*. Such communicators are created by specifying a communication graph of possibly weighted communication edges between processes. The specified communication pattern is used for two purposes in the MPI library. First, by setting the `reorder` flag to true (1), the MPI library can attempt to place the processes such that processes that are adjacent in the communication graph by a (heavy) communication edge are placed "close" to each other. Second, the communication graphs defines the so-called *neighborhoods* for a special kind of collective operations, the so-called *neighborhood collectives* that are explained briefly in Section 3.2.31. The functionality is listed here for completeness, but not treated further in these lectures (see our HPC lecture).

```
int MPI_Dist_graph_create(MPI_Comm comm_old,
                          int n, const int sources[], const int degrees[],
                          const int destinations[], const int weights[],
                          MPI_Info info, int reorder,
                          MPI_Comm *comm_dist_graph);
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old,
                                   int indegree,
                                   const int sources[],
                                   const int sourceweights[],
                                   int outdegree,
                                   const int destinations[],
                                   const int destweights[],
                                   MPI_Info info, int reorder,
                                   MPI_Comm *comm_dist_graph);

int MPI_Dist_graph_neighbors_count(MPI_Comm comm,
                                   int *indegree, int *outdegree,
                                   int *weighted);
int MPI_Dist_graph_neighbors(MPI_Comm comm,
```

```
                    int maxindegree,
                    int sources[], int sourceweights[],
                    int maxoutdegree,
                    int destinations[], int destweights[]);
```

A distributed graph communicator can, like the case was for Cartesian communicators, be queried. The `MPI_Topo_test` operation will return the value `MPI_DIST_GRAPH`.

Process reordering in MPI (sometimes called *process mapping*) via `MPI_Comm_split`, `MPI_Cart_create`, and `MPI_Dist_graph_create` is always realized in the following way. The MPI processes are bound to processor-cores and compute nodes in the system, and are contained in one or more communicators. Processes are statically bound to some part of the system and do not move. What can be changed from one communicator to another is only the rank that a process may have, so not the processes but the ranks are reordered. Assume that two processes in the input communicator `comm_old` have rank i and rank j and are adjacent (neighbors) in a distributed graph or Cartesian grid. In the resulting, reordered communicator, the ranks i and j may now be the ranks of processes (in `comm_old`) that happen to close in the system, for instance by residing on the same compute node. Thus, process reordering and process mapping are both misnomers. The MPI mechanisms are purely doing rank reordering.

Since processes themselves do not move, this means that possibly data from the process with rank i in the input communicator `comm_old` may have to be transferred to the process that now has rank i in the resulting communicator. Should such data transfer be necessary, the application programmer must implement it explicitly. Therefore, programs often do the process mapping early in the application before the processes generate or read much data.

To support mapping of data between communicators where the same process may have different ranks in the communicators, MPI provides mechanisms for translating ranks from one communicator to another. Some will be described in Section 3.2.9. The communicator comparing function `MPI_Comm_compare` may be of some use here also.

3.2.8 *MPI Concepts: Objects and Handles*

The most important MPI object is the *communicator* which is the concrete representation of an ordered domain of MPI processes that can communicate with each other. A communicator is a *distributed object*, meaning that it can be accessed and used by all the processes that have a reference to the object. MPI objects are referenced via predefined MPI handle types, of which there are quite a few (but not all that many). MPI objects can, as for the communicators, be distributed and accessible by a whole set of processes, or be *local objects* that are only accessible by the single process having the handle to the object.

Handles are mostly opaque (with one important exception that will be treated next), and their implementation unspecified in the MPI standard. An object referenced by a handle can be accessed and used only through the functions defined on the corresponding type of handle. The most important MPI objects and corresponding handles are the following.

- `MPI_Comm` for communicators, distributed.
- `MPI_Win` for communication windows, represents a communication domain and associated pieces of memory, distributed.
- `MPI_Datatype` for so-called datatypes that describe process local layout and structure of data to be communicated, local.
- `MPI_Group` for ordered sets of processes as an object that can be manipulated by process local operations, local.
- `MPI_Status` for information returned from a (point-to-point) communication operation. This is the exception to the opaqueness property of handles (see shortly). Local.
- `MPI_Request` for information about an open, possibly not yet completed communication operation (mostly point-to-point, but also collective). Local.
- `MPI_Op` for binary operators for the reduction collectives. Local.
- `MPI_Errhandler` for action to be taken on discovery of an error or failure, see remark on error handling in MPI. Local, and not treated in this lecture.
- `MPI_Info` for specifying additional information when creating (certain kinds of) objects like distributed graph communicators. Local, and not treated in this lecture.

3.2.9 *MPI Concept: Process Groups*

Process groups are local objects with handle type `MPI_Group` that represent ordered sets of processes. No communication operations are defined on process groups; the groups are for processes to locally compute other ordered sets of processes. Groups are used as input to a number of other (often collective) MPI functions that involve many processes in a communicator.

Initialization of the MPI library does not initially construct any process groups (in the way that `MPI_COMM_WORLD` is constructed). Instead, a local group object can be extracted from a distributed communicator object. The `MPI_Comm_group` operation is a local operation that a process can perform on a communicator. It returns the ordered set of processes of the communicator as

a local group object. A process can query its rank in a group. If it does not belong to the group, the special value `MPI_UNDEFINED` is returned.

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group);

int MPI_Group_size(MPI_Group group, int *size);
int MPI_Group_rank(MPI_Group group, int *rank);
```

Operations on groups are somewhat set like, but the order plays a role.

```
int MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[],
                              MPI_Group group2, int ranks2[]);

int MPI_Group_union(MPI_Group group1, MPI_Group group2,
                    MPI_Group *newgroup);
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
                           MPI_Group *newgroup);
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
                         MPI_Group *newgroup);

int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
                   MPI_Group *newgroup);
int MPI_Group_excl(MPI_Group group, int n, const int ranks[],
                   MPI_Group *newgroup);

int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
                   MPI_Group *newgroup);
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
                         MPI_Group *newgroup);

int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result);

int MPI_Group_free(MPI_Group *group);
```

We give three examples of important uses of MPI process groups. The first example shows how to create a communicator that does not contain a certain, specified process. This is helpful and sometimes needed for applications following the *master-worker* pattern (see Section 1.3.4) where one `master` process (rank) has a special role and should be excluded from communication between the non-masters (worker processes).

```
MPI_Group group, workers;
MPI_Comm work;

master = ...; // some arbitrary master (rank) in comm
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

MPI_Comm_group(comm,&group);
```

```
// exclude the master
MPI_Group_excl(group,1,&master,&workers);
MPI_Comm_create(comm,workers,&work);
if (rank==master) assert(work==MPI_COMM_NULL);
else {
  int r;
  MPI_Comm_rank(work,&r);
  if (rank<master) assert(r==rank); else assert(r==rank-1);
}
MPI_Group_free(&group);
```

The group of processes from the given communicator `comm` is extracted, each process computes a group excluding the given master process (given as a process rank between 0 and the number of processes in `comm`), and this group is used as input argument to the `MPI_Comm_create` function. Each process computes the same group. The `master` process that is not part of the group is returned the `MPI_COMM_NULL` value, whereas the workers are returned a handle to a new `work` communicator. This communicator can now be used for all kinds of communication as supported by MPI.

The same effect could be achieved less tediously with the following call(s), the only difference being that now also the master process is in a communicator, all by itself, though.

```
MPI_Comm workerormaster;

MPI_Comm_rank(comm,&rank);

int color = (rank==master) ? 0 : 1;
MPI_Comm_split(comm,color,0,&workerormaster);
```

The second example computes, for each process in a d-dimensional Cartesian grid (communicator), a group consisting the $2d+1$ neighboring processes along the d dimensions, including the process itself. It is assumed that the arrays `dims` and `periods` have been correctly and sensibly initialized (see previous example) prior to the `MPI_Cart_create` call by all processes. All other variables are likewise assumed to have been declared and sensibly initialized.

```
MPI_Cart_create(comm,d,dims,periods,0,&cartcomm);
assert(cartcomm!=MPI_COMM_NULL);

MPI_Comm_group(cartcomm,&group);

MPI_Comm_rank(cartcomm,&r);
k = 0;
neighbors[k++] = r;
for (i=0; i<d; i++) {
  MPI_Cart_shift(cartcomm,i,1,&r1,&r2);
  if (r1!=MPI_PROC_NULL) neighbors[k++] = r1;
```

```
  if (r2!=MPI_PROC_NULL&&r1!=r2) neighbors[k++] = r2;
}
assert(k<=2*d+1);
MPI_Group_incl(group,k,neighbors,&neighborgroup);
// neighborgroup now ready for use
```

The `neighborgroup` computed for each process contains a group of the local, implied grid neighborhood. This will be used later for synchronizing one-sided communication operations, see Section 3.2.21.

The third and last example shows how to translate ranks between two communicators. The problem here is the following: A new communicator `comm_new` has been created out of an old one `comm_old` (with `MPI_Comm_split`, `MPI_Cart_create`, `MPI_Dist_graph_create` or other operation), possibly with ranks being reordered, and possibly having fewer processes. For process rank i in the old communicator, what is the rank j in the old communicator of the process that has rank i in the new communicator? This information is needed in case data have to be transferred from process i in the old communicator to the process that now has rank i in the new communicator.

```
MPI_Comm_rank(comm_old,&i);

MPI_Comm_group(comm_old,&group_old);
MPI_Comm_group(comm_new,&group_new);

MPI_Group_translate_ranks(group_new,1,&i,group_old,&j);

MPI_Group_free(&group_old);
MPI_Group_free(&group_new);
```

Now, process i in the old communicator can send its data to process j (also in the old communicator), because process j is the process that has rank i in the new communicator `comm_new`.

3.2.10 *Point-to-point Communication*

Processes that belong to the same communication domain by having a handle to the same communicator can communicate with each other. We first describe the more or less classical MPI message-passing model of point-to-point communication between pairs of processes.

It is important that MPI communication between processes in a communicator has no connectivity restrictions. Any process can communicate with any other processes, as if the processes would be running on processors in a fully-connected network (See Section 3.1.1). It is the task of the MPI library and runtime (routing) system to facilitate such communication. Recall that MPI does not provide a cost model for communication between processes. The actual costs (measured time) by sending data from one process to another in

a pair of processes can be different from the communication costs between processes in any other pair of processes. Also, costs can (and do) depend on the overall communication activity between the running MPI processes.

It is also important that communication in MPI is always reliable. This means that a transmitted message can *always* be assumed to arrive uncorrupted and in full. In case the parallel computing system and communication network on which the MPI program is running are not reliable, it is again the task of the MPI library and runtime system to ensure reliable communication.

Finally, point-to-point communication is *ordered*. This means that a sequences of messages sent from one process to another will (eventually) become available at the other, receiving process in that order.

In point-to-point communication, two processes are explicitly involved. A sending process in a communication domain specifies an amount of data to be sent to a *determinate* receiving process which must be prepared to receive at least the sent amount of data. The next two functions are the basic MPI point-to-point communication operations.

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status);
```

Data to be sent and received are specified by the first three arguments: A buffer address pointing to the part of memory where data are (to be) located, an element count, and an argument describing the structure of each element (see next section).

By posting the `MPI_Send` call, a sending process initiates and completes sending data (number of elements of given structure) to the receiving process. The sending process returns from the call when the data are safely underway and the send buffer can be used again for other data. By posting the `MPI_Recv` call, a receiving process declares itself ready to receive up to the described amount of data (number of elements of given structure) from a sending process. The call completes when data sent have been received (correctly and without loss, see discussion above). Thus, for point-to-point communication to take place, both sending and receiving process are explicitly involved. The receiving process must specify *and* have allocated enough buffer space for the data that are being sent. For communication to take place, sending and receiving process must give the same *message tag* to the message. The sending process must give the rank of the receiving process. Receiving processes must be prepared to receive from that process, however, wildcards are possible, see later. Thus, sending of messages is *determinate*, but receiving is not.

The send-receive functionality illustrates another important MPI principle. All(most all) space for MPI data, notably data buffers but also argument lists etc. is in *user space* and managed by the application programmer. It is important (sometimes forgotten, with dire consequences) to always have allocated enough

buffer space for data that are being sent and received, and later to free this space to avoid running out of memory.

To illustrate point-to-point communication between processes in a communicator, here is an MPI implementation of the broadcast operation described in Definition 11 and discussed intensively later (Section 3.2.27). The process with rank `root` is the process having the data, and `count` is the number of elements. The elements that are being communicated are simple C integers of type `int` which in MPI are described by the MPI datatype `MPI_INT`. The program is written to work for any number of processes larger than one.

```
#define TAG 1000

MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

assert(size>1);

if (rank==root) {
  MPI_Send(buffer,count,MPI_INT,(rank+1)%size,TAG,comm);
} else if (rank==(root-1+size)%size) {
  MPI_Status status;

  MPI_Recv(buffer,count,MPI_INT,(rank-1+size)%size,TAG,
           comm,&status);
} else {
  MPI_Status status;

  MPI_Recv(buffer,count,MPI_INT,(rank-1+size)%size,TAG,
           comm,&status);
  MPI_Send(buffer,count,MPI_INT,(rank+1)%size,TAG,comm);
}
```

The processes in this algorithm are organized as a processor ring , and the number of communication steps (rounds) for the algorithm is $p-1$ where p is the number of MPI processes, see Section 3.2.12 for more on possible analysis of message-passing algorithms. The ith process in the ring (counting from the root process) needs to wait for process $i-1$ to have received the data, etc.. Theorem 14 tells that this algorithm is poor. Here is another implementation of the broadcast operation that is likewise poor, but not equally so, and not for the same reasons (why?).

```
#define TAG 1000

MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

if (rank==root) {
  int i;
```

```
  for (i=0; i<size; i++) {
    if (i==root) continue;
    MPI_Send(buffer,count,MPI_INT,i,TAG,comm);
  }
} else {
  MPI_Recv(buffer,count,MPI_INT,root,TAG,comm,MPI_STATUS_IGNORE);
}
```

This example shows the use of a special value as `status` argument, namely `MPI_STATUS_IGNORE`. This value can be given when the `status` of the receive operation is not needed. Otherwise, the information in the `status` object (handle) contains information on the completion of the receive operation, namely whether an error occurred, from which process the data were received, and on the amount of data received.

Since receive calls like `MPI_Recv` can specify more elements than actually sent in a send call, functionality is needed for the receiving process to find out how much data was sent. This information is available in the process local status object via the `status` handle. The two following functions `MPI_Get_count` and `MPI_Get_elements` that operate on `status` objects are defined for this purpose. The `datatype` is an input argument, which imposes an interpretation of the received data, and is needed in order to compute correctly the number (count) of such datatype elements that were received (`MPI_Get_count`). For complex datatypes, the `MPI_Get_elements` call, instead computes the number of elements of a *basic datatype* that were received (see Section 3.2.14). For simple (non-complex), basic datatypes like `MPI_INT` as used in the code examples above, the `MPI_Get_count` and `MPI_Get_elements` calls are equivalent.

```
int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype,
                  int *count);
int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,
                     int *count);
```

The `status` object/handle is peculiar in MPI. Handles were said to be *opaque*, but handles of type `MPI_Status` are only half so. Status objects have three predefined fields, namely `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`, and these are important for non-determinate communication as explained in Section 3.2.11.

Algorithms often desire or even require that a process in a communication round can both send and receive a message, as for instance permitted in the one-ported, fully bidirectional send-receive communication model. The example below, where the processes are organized in a ring like in the first broadcast implementation above, will obviously lead to a *communication deadlock*. Each process is waiting to receive data from the previous process in the ring, but these data cannot be sent, since also this process is waiting to receive data from its previous process, etc..

```
#define TAG 1000
```

```
MPI_Status status;
int *a = ...;
int *b = ...;

MPI_Recv(a,count,MPI_INT,(rank-1+size)%size,TAG,
         comm,&status);
MPI_Send(b,count,MPI_INT,(rank+1)%size,TAG,comm);
```

MPI provides an `MPI_Sendrecv` operation to handle such situations that combines the functionality and parameters of a blocking send and a blocking receive operation. With `MPI_Sendrecv`, a process can at the same time, concurrently, send and receive data to and from two other processes in the communicator — that could actually be the same process — without the risk of deadlocking. When an `MPI_Sendrecv` call returns, data have left the send buffer (as with `MPI_Send`) which can then be reused for other data, and received into the receive buffer (as with `MPI_Recv`). The status of the receive part is recorded in the `status` object.

```
int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status);

int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,
                         int dest, int sendtag, int source, int recvtag,
                         MPI_Comm comm, MPI_Status *status);
```

Send and receive buffers must not overlap in any way, since this would lead to an indeterminate situation: did the send part take place, in part or in total, before or after the receive part? It is the programmers responsibility to make sure that this is indeed guaranteed, neither compiler nor MPI library will or can check this. Such unintentionally overlapping buffers are another common source of often very hard to find errors in MPI programs. In case data should be sent from some buffer, and (later) be received into the same buffer, the `MPI_Sendrecv_replace` operation can be used (which most likely will allocate some intermediate space for the receive part, and later copy this back: therefore entailing potentially significant extra costs. Of course, the MPI standard neither prescribes or forbids any particular implementation. If one needs to know, only benchmarking and MPI library code inspection, if open, helps).

With `MPI_Sendrecv` the deadlock situation from above is resolved:

```
#define TAG 1000

MPI_Status status;
```

```
int *a = ...;
int *b = ...;

MPI_Sendrecv(b,count,MPI_INT,(rank+1)%size,TAG,
             a,count,MPI_INT,(rank-1+size)%size,TAG,
             comm,&status);
```

3.2.11 *Determinate vs. Non-determinate Communication*

A sending process always specifies a determinate, specific receiver by its rank in the communicator. A sending process also gives each message sent a specific tag. In MPI, a *message tag* is just a non-negative integer that is attached as a label to a message (up to a specified upper bound given by `MPI_TAG_UB`). The message tag can be used by the receiver to distinguish one kind of tagged message from other kinds of tagged messages, and to select which message is to be received by an `MPI_Recv` call in case more than one message has been sent from one or more other processes.

As seen above, a receiving process can specify explicitly the rank of the sending process from which it wants to receive a specific message with a specific tag. In contrast to sending processes, receiving processes can also receive from a *non-determinate* process. This is done by specifying a wildcard `MPI_ANY_SOURCE` for the rank argument, and will enable the receiving process to receive the message from any of the processes in the communicator. Likewise, the tag argument can be given a wildcard `MPI_ANY_TAG`.

Whereas programs with determinate ranks in the communication operations are communication deterministic, programs using the `MPI_ANY_SOURCE` wildcard can be non-deterministic. Non-deterministic programs can cause problems not encountered with deterministic programs. The following examples illustrate some of these points.

Point to point-communication is ordered. If data messages are sent in sequence with the same tag by a sequence of `MPI_Send` operations, the data will be ready to be received by the destination process in that order. This is referred to as *ordered communication* in MPI. The program below illustrates the advantages of the ordering constraint. Data from two buffers with different numbers of elements, the first 500 integers (`MPI_INT`), and different element types, the second with 100 doubles (`MPI_DOUBLE`), are sent from process 0 to the last process with rank $p-1$ (p being the number of processes in communicator `comm`). It is good SPMD style to write MPI program so that they work for any number of processes, which is done here for any number of processes larger than one, as asserted. An open `else` instead of the `else if (rank==size-1)` conditional would lead to a deadlock.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);
```

```
assert(size>1);

if (rank==0) {
  int buf1[500];
  double buf2[100];

  MPI_Send(buf1,500,MPI_INT,size-1,TAG,comm);
  MPI_Send(buf2,100,MPI_DOUBLE,size-1,TAG,comm);
} else if (rank==size-1) {
  MPI_Status status;

  int buf1[1000];
  double buf2[200];
  int cc;

  MPI_Recv(buf1,1000,MPI_INT,0,TAG,comm,&status);
  MPI_Get_elements(&status,MPI_INT,&cc);
  assert(cc<1000);
  assert(cc==500);

  MPI_Recv(buf2,200,MPI_DOUBLE,0,TAG,comm,&status);
  MPI_Get_elements(&status,MPI_DOUBLE,&cc);
  assert(cc<200);
  assert(cc==100);
}
```

Since less data are sent in each message than expected by the receiving process, the exact number of data elements received in each of the messages is computed by the `MPI_Get_elements` operation. The assertions assert that the (stack) allocated buffers are not overflowing. The `count` argument is an upper bound on the number of elements that can be received, and this upper bound should of course be no larger that the actual number of elements in the buffer used for reception. Again, the compiler can and will not check this, and it is entirely the programmer's responsibility to ensure that buffers are not overwritten (which will most likely cause segmentation faults at some point in the program execution). It is also worth noticing that the message tag has nothing to do with the type of the message being communicated: the same tag is used for `MPI_INT` and `MPI_DOUBLE` messages.

In the next example, the data to be sent to process $p-1$ come from two different processes. In order to avoid waiting times, the receiving process uses `MPI_ANY_SOURCE` to be able to receive the message from the source process that becomes ready first. Here, both buffers contain C integers, and both sending processes uses the same message tag. Since the two sent messages have different numbers of elements, the receiving process must ensure that both receive buffers are large enough to hold the number of elements in the

largest message. This is the price of non-determinacy. The special `MPI_Status` field `MPI_SOURCE` is used to distinguish the messages based on the source of origin.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

assert(size>2);

if (rank==0) {
  int buf1[500];

  MPI_Send(buf1,500,MPI_INT,size-1,TAG,comm);
} else if (rank==1) {
  int buf2[100];

  MPI_Send(buf2,100,MPI_INT,size-1,TAG,comm);
} else if (rank==size-1) {
  MPI_Status status;

  int buf1[1000];
  int buf2[1000];
  int cc;

  MPI_Recv(buf1,1000,MPI_INT,MPI_ANY_SOURCE,TAG,comm,&status);
  MPI_Get_elements(&status,MPI_INT,&cc);
  assert(cc<1000);
  if (status.MPI_SOURCE==0) {
    assert(cc==500);
  } else {
    assert(cc==100);
  }

  MPI_Recv(buf2,1000,MPI_INT,MPI_ANY_SOURCE,TAG,comm,&status);
  MPI_Get_elements(&status,MPI_INT,&cc);
  assert(cc<1000);
  if (status.MPI_SOURCE==0) {
    assert(cc==500);
  } else {
    assert(cc==100);
  }
}
```

Non-determinacy can easily lead to incorrect, possibly crashing programs. In the next program, the sending processes send different types and numbers of elements (`MPI_INT` and `MPI_DOUBLE`), but for the receiving process it has been forgotten that these two messages may arrive in any order depending on the relative timing of the two sending processes and possibly other factors.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

assert(size>2);
if (rank==0) {
  int buf1[500];

  MPI_Send(buf1,500,MPI_INT,size-1,TAG,comm);
} else if (rank==1) {
  double buf2[100];

  MPI_Send(buf2,100,MPI_DOUBLE,size-1,TAG,comm);
} else if (rank==size-1) {
  MPI_Status status;

  int buf1[1000];
  double buf2[200];
  int cc;

  MPI_Recv(buf1,1000,MPI_INT,MPI_ANY_SOURCE,TAG,comm,&status);
  MPI_Get_elements(&status,MPI_INT,&cc);
  assert(cc<1000);
  if (status.MPI_SOURCE==0) {
    assert(cc==500);
  } else {
    assert(cc==100);
  }

  MPI_Recv(buf2,200,MPI_DOUBLE,MPI_ANY_SOURCE,TAG,comm,&status);
  MPI_Get_elements(&status,MPI_DOUBLE,&cc);
  assert(cc<1000);
  if (status.MPI_SOURCE==0) {
    assert(cc==500);
  } else {
    assert(cc==100);
  }
}
```

The program may crash, possibly with an MPI error message that a received message has been truncated, or by one of the assertions being violated.

The correct order of the received messages can be enforced by using different tags for the two messages. Of course, this sacrifices the potential performance advantage of non-determinacy. The program below is correct. The first `MPI_Recv` operation by process $p-1$ can only receive a message with tag `TAG0`, and such a message will eventually be sent by process 0. The next

to be received message must have tag `TAG1` and also such a message will eventually be sent by process 1.

```
#define TAG0 500
#define TAG1 501

if (rank==0) {
  int buf1[500];

  MPI_Send(buf1,500,MPI_INT,size-1,TAG0,comm);
} else if (rank==1) {
  double buf2[100];

  MPI_Send(buf2,100,MPI_DOUBLE,size-1,TAG1,comm);
} else if (rank==size-1) {
  MPI_Status status;

  int buf1[1000];
  DOUBLE buf2[200];
  int cc;

  MPI_Recv(buf1,1000,MPI_INT,MPI_ANY_SOURCE,TAG0,comm,&status);
  MPI_Get_elements(&status,MPI_INT,&cc);
  assert(cc<1000);
  assert(cc==500);

  MPI_Recv(buf2,200,MPI_DOUBLE,MPI_ANY_SOURCE,TAG1,comm,&status);
  MPI_Get_elements(&status,MPI_DOUBLE,&cc);
  assert(cc<1000);
  assert(cc==100);
}
```

Message tags are a specialty of point-to-point communication, where there is a need to be able to label and distinguish messages. One-sided communication (Section 3.2.21) and collective communication (Section 3.2.27) do not need and do not utilize message tags.

In order to find out whether a message from a determinate or non-determinate source with a given or wildcard tag is ready to be received, MPI provides calls to probe for such possible messages.

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
```

Such a call returns when a message with the specified characteristics (source and tag) is ready to be received. Upon return, an `MPI_Recv` or other point-to-point message receive operation must be executed in order to actually receive the message. Advanced note: This separation into the probing for a message and the actual reception of the message can cause problems (race condition)

when MPI is used in a multi-threaded program, for instance OpenMP or `pthreads`.

3.2.12 *Point-to-point Communication Complexity and Performance*

When two MPI processes at the same time (whatever that may mean) become ready to communicate, the one process sending data of m units, the other one ready to receive at least m units, time for transmitting the m data units may naively be modeled as $\alpha + \beta m$ as done in Section 3.1.3. We can also account for this data transmission as one *communication step*, independently of the amount of data being transmitted.

Several independent pairs of processes can, if the underlying communication network is strong enough, communicate independently, and if all processes communicate the same amount of data m, we can also count such concurrent communication operations as one communication step.

The *communication step complexity* for a full message-passing computation can, under the assumption that in each communication step, all communicating processes communicate the same amount of data, be counted as the number of steps (sometimes also called *communication rounds*) required from the start of all processes until the last process has completed. This amounts to finding the longest (weighted) path from a first to a last process in the communication DAG (Directed Acyclic Graph) describing the communication operations. In the communication DAG, there is an edge from process i to process j when process i sends a message that is received by process j.

The linear array broadcast implementation from Section 3.2.10 was claimed to take $p - 1$ communication steps. This can be seen by an inductive argument. If there is only two processes, one communication step is obviously required and suffices. With $p > 2$ processes, the root process first sends the data to the next process in one communication step. This process now behaves like a root for $p - 1$ processes, for which case the broadcast can inductively be completed in $p - 2$ steps, for a total of $(p - 2) + 1 = p - 1$ steps.

Below is a better broadcast algorithm that completes in $\lceil \log_2 p \rceil$ steps, matching the lower bound of Theorem 14.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

virt = (rank-root+size)%size;

if (virt!=0) {
  int d = 1;
  while (virt>=d) {
    dist = d; d <<= 1; // multiply by two
  }
  MPI_Recv(buffer,count,datatype,(virt-dist+root+size)%size,TAG,comm,
```

```
          MPI_STATUS_IGNORE);
  dist = d;
} else dist = 1;
while (virt+dist<size) {
  MPI_Send(buffer,count,datatype,(virt+dist+root)%size,TAG,comm);
  dist <<= 1; // multiply by two
}
```

There is much more on (the analysis) of such algorithms in our HPC lecture.

3.2.13 *MPI Concepts: Semantic terms*

The simple send and receive operations, as well as all the other operations discussed in the previous sections are *blocking*. This is a specific MPI semantic term which means that the call returns when the operation is locally complete, from the calling process' point of view. With `MPI_Send` in particular, data are out of the send-buffer which can now be reused for other purposes, for instance as buffer for the next `MPI_Send`. Also other resources have been given free and can be reused. When, e.g., `MPI_Comm_split` returns, a new communicator has been created and is ready for use from the calling process' point of view. Note that *blocking* does not imply anything about what other processes have done, it is simply the condition that an operation has been completed process locally according to the semantics of the operation. For instance, return from an `MPI_Send` call does not mean that the data have been received by the receiving process which might not even have posted its `MPI_Recv` call, not even that data are anywhere near the receiving process. Data could simply have been buffered somewhere by the MPI library, a technique that is often used for small messages and can have some performance advantages. On the other hand, for a blocking operation to complete, some action by other processes *may* be necessary. For instance, very large data are typically not buffered by MPI libraries, and in such cases point-to-point communication can complete only when sending and receiving processes are both active.

As counterpart MPI defines operations that are *non-blocking*. Such operations will return immediately (whatever that means), always independently of action being taken by other processes, and are therefore also called *immediate operations* and prefixed with an `I` (Capital "I") in MPI.

An MPI operation is said to have *local completion* if it can always complete independently of action by other processes. Trivial examples so far were the blocking operations `MPI_Comm_rank` and `MPI_Comm_size`. The `MPI_Send` operation is blocking, but does not have local completion. The `MPI_Comm_create` operation is blocking (and collective) and does not have local completion: Some action may be required by the other processes in order to create the new communicators.

The counterpart to local completion is *non-local completion* which means that in order for an operation to complete, action by other processes *may* be

needed. Here, *action* means that other processes are performing MPI calls that enables the operation to complete.

Per definition, non-blocking calls have local completion.

As discussed for the blocking `MPI_Send` operation, an implementation of MPI (that does intermediate buffering) may make it possible for an `MPI_Send` call to complete and return even without the receiving process having posted a suitable (matching) `MPI_Recv` call. But it may also not. Relying on such implementation specific behavior is bad and dangerous practice, since it makes programs non-portable. The program may run on one machine with one MPI library, but it may stop working on the next machine with a different MPI library. The practice of (perhaps unbeknownst) relying on implementation dependent behavior is called *unsafe programming*.

Concretely, for blocking `MPI_Send-MPI_Recv` communication, one should write the application such that there will always, eventually be a matching `MPI_Recv` call for any `MPI_Send` call executed by some process.

Here is a typical example of unsafe communication in a ring. All processes initiates a (blocking) `MPI_Send` call to the next process in the ring, after which they receive data from the previous process in the ring. The `MPI_Send` may — or may *not* — be able to complete, depending on the message count and on implementation details of the MPI library. If it cannot complete, a deadlock ensues. The nasty thing about this kind of codes is that they may well work under the right circumstances, and then suddenly not when conditions change. That is why this style of programming is called *unsafe*. Unsafe programs are in particular not portable.

```
#define TAG 1000

MPI_Status status;
int *a = ...;
int *b = ...;

MPI_Send(b,count,MPI_INT,(rank+1)%size,TAG,comm);
MPI_Recv(a,count,MPI_INT,(rank-1+size)%size,TAG,
         comm,&status);
```

In some of the examples above, message tags were used to enforce a certain order on received messages. This usage can easily result in an unsafe program as the example below shows.

```
#define TAGbuf1 100
#define TAGbuf2 101

if (rank==0) {
  int buf1[500];
  double buf2[100];

  MPI_Send(buf2,100,MPI_DOUBLE,size-1,TAGbuf2,comm);
```

```
  MPI_Send(buf1,500,MPI_INT,size-1,TAGbuf1,comm);

} else if (rank==size-1) {
  // order, buf2 smaller than buf1, but no overflow
  MPI_Status status;

  int buf1[1000];
  double buf2[200];
  int cc;

  MPI_Recv(buf1,1000,MPI_INT,0,TAGbuf1,comm,&status);
  MPI_Get_elements(&status,MPI_INT,&cc);
  assert(cc<=1000);
  assert(cc==500);

  MPI_Recv(buf2,200,MPI_DOUBLE,0,TAGbuf2,comm,&status);
  MPI_Get_elements(&status,MPI_DOUBLE,&cc);
  assert(cc<=200);
    assert(cc==100);
  }
```

Care is needed to ensure that a program is not unsafe. Sometimes this can be difficult, as the two-dimensional stencil code below shows. Here, the processes have been organized as a two-dimensional, Cartesian communicator, see Section 3.2.7. It is a good exercise to reflect on this example, and on how the code can be made safe and portable.

```
int left, right;
int up, down;

MPI_Cart_shift(cartcomm,1,1,&left,&right);
MPI_Cart_shift(cartcomm,0,1,&up,&down);

double *out_left, *out_right, *out_up, *out_down;
double *in_left, *in_right, *in_up, *in_down;
int c = ...;

int done = 0;
while (!done) { // iterate until convergence
  MPI_Send(out_left,c,MPI_DOUBLE,left,TAG,cartcomm);
  MPI_Send(out_right,c,MPI_DOUBLE,right,TAG,cartcomm);
  MPI_Send(out_up,c,MPI_DOUBLE,up,TAG,cartcomm);
  MPI_Send(out_down,c,MPI_DOUBLE,down,TAG,cartcomm);

  MPI_Recv(in_left,c,MPI_DOUBLE,right,TAG,cartcomm,MPI_STATUS_IGNORE);
  MPI_Recv(in_right,c,MPI_DOUBLE,left,TAG,cartcomm,MPI_STATUS_IGNORE);
  MPI_Recv(in_up,c,MPI_DOUBLE,down,TAG,cartcomm,MPI_STATUS_IGNORE);
  MPI_Recv(in_down,c,MPI_DOUBLE,up,TAG,cartcomm,MPI_STATUS_IGNORE);
```

Table 1: Some C datatypes and their corresponding `MPI_Datatype`.

C language type	Corresponding MPI datatype
`char`	`MPI_CHAR`
`int`	`MPI_INT`
`long`	`MPI_LONG`
`float`	`MPI_FLOAT`
`double`	`MPI_DOUBLE`

```
}
```

3.2.14 *MPI Concepts: Specifying Data*

Data to be communicated in MPI are always specified the same way. A block of elements is described by a triple consisting of starting address (or offset) in memory (buffer), number of elements (count), and structure/layout of the elements (datatype). As a mnemonic for the MPI communication operations it is helpful to keep in mind that data are always specified by triples of `buffer,count,datatype`; this greatly reduces the number and meaning of arguments one has to think of, and makes it easy to guess/reconstruct the signature of many MPI operations.

The third argument in the triple, the `MPI_Datatype`, describes the structure or layout of data to be communicated (sent or received) locally, at the process. For basic, simple, non-complex objects like the `int`'s and `double`'s in a C program, there are corresponding, predefined handles like `MPI_INT` and `MPI_DOUBLE` that describe to the MPI library that the bits and bytes in a data buffer represents these kinds of objects.

For the simple, and most common case of elements from a consecutive buffer, for instance an array, of some simple elementary programming language type being communicated, the datatype argument just tells the MPI library that the bytes are to be interpreted as the corresponding programming language type is represented in memory. There is therefore an MPI datatype for each simple, elementary programming language datatype. Some correspondences for C are shown in Table 1; Fortran has Fortran-like names for the corresponding MPI types.

Correct MPI programs require that data elements of some programming language type that are sent as a sequence of MPI datatypes are received as a sequence of the same MPI datatypes. Observing this requirement ensures that the bits and bytes that are sent and received are interpreted and handled in the intended way both by the sending and by the receiving process. It is important to understand that the programming language type of objects are

not known to the MPI library (therefore, the library has to be instructed in each communication operation), and that the MPI datatype information is *not* in any way part of the transmitted data. It is entirely the programmer's responsibility to ensure that all communicated data are given the right MPI datatype for both sending and receiving processes. Neither compiler nor MPI library can and will (for performance reasons) check this. For this same reason, MPI does not perform type conversion (as known from, e.g., C). If a data buffer is sent as a sequence of `MPI_INT` objects and received as a sequence of `MPI_FLOAT` objects, no useful outcome can be expected. Most certainly, the `int`'s will not be converted to `double`'s in a semantically meaningful way!

The next three small examples illustrate this. In the first example, some `long`'s are sent correctly as `MPI_LONG`, but wrongly received into a (large enough, presumably) buffer of `double`'s as `MPI_DOUBLE`.

```
if (rank==0) {
  long a[n];
  MPI_Send(a,n,MPI_LONG,size-1,TAG,comm);
} else if (rank==size-1) {
  double a[n];
  MPI_Recv(a,n,MPI_DOUBLE,0,TAG,comm,MPI_STATUS_IGNORE);
}
```

In the second example, `double`'s sent correctly are received as a sequence of `MPI_BYTE`. This may or may not give correct results, is nevertheless a dangerously incorrect MPI programming style.

```
double a[n];
if (rank==0) {
  MPI_Send(a,n,MPI_DOUBLE,size-1,TAG,comm);
} else if (rank==size-1) {
  MPI_Recv(a,n*sizeof(double),MPI_BYTE,0,TAG,comm,MPI_STATUS_IGNORE);
}
```

In the third and last example, the objects are sent and received as streams of uninterpreted bytes. This is not technically wrong, but any type information on how `double`'s are to be handled (e.g., Endianness) is lost.

```
double a[n];
if (rank==0) {
  MPI_Send(a,n*sizeof(double),MPI_BYTE,size-1,TAG,comm);
} else if (rank==size-1) {
  MPI_Recv(a,n*sizeof(double),MPI_BYTE,0,TAG,comm,MPI_STATUS_IGNORE);
}
```

The next purpose of MPI datatypes is to be able to describe layouts of complex data in process local memory in order to give the MPI library the possibility to read and write data elements from specific locations and not necessarily as a consecutive stream of elements in a simple, linear buffer

(array). Simple examples are the columns of a two-dimensional matrix; a submatrix of some d-dimensional matrix; complex C structures with many different component types, etc.. The MPI concept of a datatype is thus different from the same-named, semantic type programming language concept. In MPI, a datatype describes the (local, spatial) structure of data objects to be communicated.

The idea of the MPI *user-defined datatype*, or *derived datatype* mechanism is to be able to encapsulate such complex data layouts into a single unit which can then be used as the unit of communication in *all* MPI communication operations. A derived datatype represents an ordered list of simple, basic datatypes (as we have seen: `MPI_INT`, `MPI_DOUBLE`, `MPI_CHAR`, etc.) together with a displacement or relative offset for each simple element. The offset for an element gives the linear position of the element in memory relative to a given base address, e.g., the buffer argument supplied in the MPI communication calls.

An explicit list of basic element datatypes with displacements is in MPI terms called a *type map*. The type map is used locally by communicating MPI processes to access the basic elements in the order implied by the list in local memory, regardless of whether processes are sending or receiving data. A type map is thus a purely process local construct and the type maps of one process are not known to any other processes. Identical type maps for different processes can of course be constructed by the programmer, but MPI itself cannot and does not exchange type maps or any other type information (datatypes and type maps are not *first-class citizens* in MPI).

Communication in MPI can be thought of as a stream of elements described by a corresponding stream of simple, basic datatypes. Such an ordered sequence of basic datatypes is in MPI terms called a *type signature*. When two processes are communicating, the signature of the data that are sent must be a prefix of the signature of the data that the receiving process is prepared to receive. Again, the signatures are not part of the data that are being communicated. It is purely the programmers responsibility to guarantee that the signature rule is obeyed. By this choice, it is possible with the help of the programmer to do type safe communication in MPI, but without the burden (and performance disadvantage) of having to communicate any type meta information.

In MPI, type maps are not represented explicitly by lists of basic datatypes and displacements. Instead, MPI provides a number of constructors for compactly describing more and more irregular layouts of data in memory. Layouts described by this mechanism are called derived or user-defined datatypes. In Section 3.2.19 the MPI constructors will be briefly explained. A derived datatype can be used in any MPI communication operation and in all operations that take `MPI_Datatype` arguments.

A type map as represented by a derived datatype is a complex object encompassing possibly many basic datatypes together with their displacements.

The *size* of a derived datatype is the number of Bytes required (locally, for the process) to represent all the basic datatypes in the derived datatype. The *extent* of a derived datatype is a quantity in Bytes associated with a derived datatype which is necessary when a derived datatype is used in communication operations with an element count that is larger than one. The signature of the derived datatype is the unit of communication. A count $c, c > 1$ tells that more than one element of this unit is to be communicated. The ith element, $0 \leq i < c$ is taken from relative offset $i \cdot$ `extent` from the given communication buffer address, where `extent` is the extent of the datatype. The following MPI calls return the size and extent of both simple, basic, predefined datatypes and user-defined datatypes.

```
int MPI_Type_size(MPI_Datatype datatype, int *size);
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent);

int MPI_Type_get_extent(MPI_Datatype datatype,
                        MPI_Aint *lb, MPI_Aint *extent);
int MPI_Type_get_true_extent(MPI_Datatype datatype,
                             MPI_Aint *true_lb, MPI_Aint *true_extent);
```

Often, but not always, the extent of a derived datatype correspond to the "footprint" in memory of the type layout described by that datatype, that is the linear difference between the element with the smallest displacement and the element with the largest displacement (plus the size of that element). The datatype constructors all have associated rules for how the extent of the resulting derived datatype is computed. There are, however, special type constructors for creating datatypes with a different (arbitrary) extent, a feature that is extremely powerful for advanced usage of MPI, and therefore the extent is not simply the "memory footprint" of the layout. However, the memory footprint is needed in case new memory for some complex layout need to be allocated, and for this, the special call `MPI_Type_get_true_extent` is defined. Unfortunately, even this is not always sufficient for computing the right amount of memory space. Memory allocation and derived datatypes need care.

The calls returning an extent have arguments of type pointer to `MPI_Aint`. This argument type is not an MPI handle, but the type of an object that can represent an *address-sized integer*. In many cases (compilers, systems), an `MPI_Aint` is indeed different from a C `int` (64 versus 32 Bits). The `MPI_Aint` type is used for many MPI operations where it is important that an argument is a process local address; but is is not used very consistently in the MPI standard.

3.2.15 *MPI Concept: Matching Communication Operations*

In order for point-to-point communication between two processes to be successful, the `MPI_Send` and `MPI_Recv` operations must *match*. First of all, the two processes must make their calls on the same communicator: In MPI, commu-

nication on one communicator can never interfere with communication on another communicator, so the situation with an MPI_Send on one communicator and an otherwise correct MPI_Recv operation on another communicator will be a deadlock. The destination rank given by the sending process must match the rank given by the receiving process. Either the receiving process gives explicitly the rank of the sending process, or the MPI_ANY_SOURCE wildcard. Likewise, the message tags must match. As mentioned, it is perfectly legal for a process to communicate with itself. However, with blocking operations only, it is not possible to do this in a safe way; at least one of either the send or receive operations has to be non-blocking, or the MPI_Sendrecv operation must be used. Also, care has to be taken in such cases, that receive and send buffer (which are on the same process) do not overlap in anyway, since in that case the result would be unspecified.

Second, the amount of data sent in the MPI_Send operation must be at most the amount of data that the MPI_Recv operation is prepared to receive, as specified by count and datatype arguments. The MPI types of the sent and received elements must correspond, technically this means that the signature of the sent data must be a prefix of the signature of the data specified in the receive call. As discussed, MPI cannot and does not check for this.

When an MPI_Send and an MPI_Recv call match, communication can take place and the MPI implementation guarantees that data are correctly received. There is no need for low-level consistency or correctness checks.

Communication with the special MPI_PROC_NULL process always matches, but has no effect, neither in an MPI_Send nor in an MPI_Recv operation.

3.2.16 *Non-blocking Point-to-point Communication*

MPI defines non-blocking point-to-point communication counterparts for the simple MPI_Send, MPI_Recv, and MPI_Probe operations (but no non-blocking MPI_Sendrecv operation).

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Request *request);

int MPI_Iprobe(int source, int tag, MPI_Comm comm,
               int *flag, MPI_Status *status);
```

All these operations return "immediately": what exactly this means (how fast is "immediate"?) is, by the nature of the MPI standard specification, not defined, but the important point is that the operations have entirely local completion semantics and return independently of any MPI actions taken by any other processes. Non-blocking point-to-point operations can therefore be

used to avoid situation that might otherwise lead to a deadlock (unsafe code) with blocking communication.

These non-blocking send and receive operations take the same input parameters as their blocking counterparts, but has a new output argument, the `MPI_Request` object. The `MPI_Request` object can be used to query the completion status of the corresponding operation, and to enforce completion. A non-blocking `MPI_Isend` call with ensuing, enforced completion has the same effect (semantics) as a blocking `MPI_Send` call. That is, enforced completion means only that the send operation has been completed from the process's point of view, and does not imply that the receiving process has even reached the point in the code of a matching receive call. The non-blocking counterpart of the probe operation, the `MPI_Iprobe`, does not return an `MPI_Request` object. Instead, the completion of the probe for a matching, incoming message is indicated in the `flag` return argument (pointer).

There is a whole repertoire of operations for checking and enforcing completion of immediate, pending `MPI_Isend` and `MPI_Irecv` communication operations. These calls can either test whether an operation, referred to by an `MPI_Request` object, is complete which is signalled in a `flag` return argument, or enforce, that is wait for completion of an operation. There are calls that operate on a set of request objects, rather than a single object, and can test for/enforce completion of either some single (arbitrary) operation, some, or all operations in the set of requests (given as an input array). For complete operations, their status is returned in corresponding `MPI_Status` objects, just as was the case for the blocking `MPI_Send` and `MPI_Recv` calls.

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

int MPI_Waitany(int count, MPI_Request array_of_requests[], int *indx,
                MPI_Status *status);
int MPI_Testany(int count, MPI_Request array_of_requests[], int *indx,
                int *flag, MPI_Status *status);
int MPI_Waitall(int count, MPI_Request array_of_requests[],
                MPI_Status array_of_statuses[]);
int MPI_Testall(int count, MPI_Request array_of_requests[],
                int *flag, MPI_Status array_of_statuses[]);
int MPI_Waitsome(int incount, MPI_Request array_of_requests[], int *outcount,
                 int array_of_indices[], MPI_Status array_of_statuses[]);
int MPI_Testsome(int incount, MPI_Request array_of_requests[], int *outcount,
                 int array_of_indices[], MPI_Status array_of_statuses[]);

int MPI_Request_free(MPI_Request *request);
```

The non-blocking communication operations separate the initialization and the completion of an operation, and can be most convenient for writing safe

programs that cannot deadlock in any possible situation. For instance, the `MPI_Sendrecv` operation is equivalent to either

```
MPI_Isend(sendbuf,sendcount,sendtype,dest,sendtag,comm,&request);
MPI_Recv(recvbuf,recvcount,recvtype,source,recvtag,comm,&status);
MPI_Wait(&request,MPI_STATUS_IGNORE);
```

or

```
MPI_Irecv(recvbuf,recvcount,recvtype,source,recvtag,comm,&request);
MPI_Send(sendbuf,sendcount,sendtype,dest,sendtag,comm);
MPI_Wait(&request,&status);
```

or even

```
MPI_Irecv(recvbuf,recvcount,recvtype,source,recvtag,comm,&request[0]);
MPI_Isend(sendbuf,sendcount,sendtype,dest,sendtag,comm,&request[1]);
MPI_Waitall(2,request,status);
```

where for the last code snippet, the status of the receive operation is in `status[0]`).

The unsafe, two-dimensional stencil code can be made safe and deadlock-free simply by using non-blocking send and receive operations.

```
MPI_Request request[8];
while (!done) {
  int k = 0;

  MPI_Isend(out_left,c,MPI_DOUBLE,left,TAG,cartcomm,&request[k++]);
  MPI_Isend(out_right,c,MPI_DOUBLE,right,TAG,cartcomm,&request[k++]);
  MPI_Isend(out_up,c,MPI_DOUBLE,up,TAG,cartcomm,&request[k++]);
  MPI_Isend(out_down,c,MPI_DOUBLE,down,TAG,cartcomm,&request[k++]);

  MPI_Irecv(in_left,c,MPI_DOUBLE,right,TAG,cartcomm,&request[k++]);
  MPI_Irecv(in_right,c,MPI_DOUBLE,left,TAG,cartcomm,&request[k++]);
  MPI_Irecv(in_up,c,MPI_DOUBLE,down,TAG,cartcomm,&request[k++]);
  MPI_Irecv(in_down,c,MPI_DOUBLE,up,TAG,cartcomm,&request[k++]);

  MPI_Waitall(k,request,MPI_STATUSES_IGNORE);
}
```

The special value `MPI_STATUSES_IGNORE` indicates that all status'es in an array should be ignored.

3.2.17 *Exotic send operations*⋆

MPI provides a few more send operations with additional semantic content. These operations come in both blocking and non-blocking variants. There is a *synchronous send* operation, where local completion implies that the receiving

process has indeed started reception of the message by a matching receive operation. There is a *buffered send* operations, where data are explicitly stored in a local buffer in order to provide local completion semantics. The local buffer is allocated in user space, and needs to be explicitly attached for this use to the MPI library. Finally, there is a *ready send* operation, which can be used provided that a matching receive operation has already been posted before the buffered send. Send-receive communication can possibly be implemented more efficiently under this precondition, and was included in MPI to enable such implementations. Using it correctly require additional explicit or implicit synchronization, and is rather left to experts.

These more exotic send functions are listed below, in order of exoticness, but are not covered further in these lectures.

```
int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm);
int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm);
int MPI_Rsend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm);

int MPI_Buffer_attach(void *buffer, int size);
int MPI_Buffer_detach(void *buffer_addr, int *size);
```

The non-blocking counterparts are listed below.

```
int MPI_Issend(const void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *request);
```

Any send function and send operations can match with a receive operation, whether blocking or non-blocking. There is only one kind of receive operation in MPI, and completion signal that data have been received correctly from a matching, sending process.

For completeness, we mention that it is/should be technically possible to cancel a message. However, the semantics and guarantees are not clear, and relying on this functionality is never recommended in MPI programs.

```
int MPI_Cancel(MPI_Request *request);
int MPI_Test_cancelled(const MPI_Status *status, int *flag);
```

3.2.18 *MPI Concept: Persistence⋆*

A new MPI concept which we do not cover in this lecture is *persistent (point-to-point communication) operations*. The idea is to be able to split the initialization

of a communication operations (argument parsing, reservation of memory and communication resources) from the operations itself, and to make it possible to execute the operation many times with the same arguments. Persistent operations aims to make it possible to amortize possibly expensive set-up costs over many uses of the same operation.

Concretely, MPI reuses the concept of `MPI_Request` handles as objects to store the precomputed information for a communication operation. The MPI standard defines a persistent counterpart for all the different types of send operations, and for the receive operations. New operations are used to (re)start any single or a whole set of persistent communication operations.

```
int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype,
                  int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Bsend_init(const void *buf, int count, MPI_Datatype datatype,
                   int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype,
                   int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Rsend_init(const void *buf, int count, MPI_Datatype datatype,
                   int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype,
                  int source, int tag, MPI_Comm comm, MPI_Request *request);

int MPI_Start(MPI_Request *request);
int MPI_Startall(int count, MPI_Request array_of_requests[]);
```

Both of the start calls are non-blocking, thus the persistent communication operations behave like the corresponding non-blocking operations. Completion can be checked or enforced with the same operations on the `MPI_Request` object as explained in Section 3.2.16.

3.2.19 *More on User-defined, Derived Datatypes⋆*

The datatype argument appearing in the communication operations so far describe the process local unit of communication, and the count argument the number of such units. The units we have seen in the small examples hitherto corresponded simply to basic C datatypes like `int`'s by the `MPI_Datatype` `MPI_INT`, etc.. A process local communication unit can be more complex, though, and describe a whole sequence of basic datatypes together with their relative displacement in memory. Such a description was called the *type map*. The rules for matching communication say that the count times the number of basic datatypes in the `MPI_Datatype` unit that are sent must be no larger than the number of elements that the receiving process is prepared to receive, and that the sequence of basic datatypes must be identical.

Type maps are represented in a compact(er) form by the `MPI_Datatype` objects. MPI provides a set of constructors for constructing new, more complex datatypes out of already existing ones (again, MPI objects cannot be changed,

only new objects can be created from existing ones). Such datatype objects are called *derived datatypes*, and are means to describe the structure of complex data in the memory of a process.

A set of fundamental constructors are listed below, in order of increasing generality. That is, the structure than can be described by one constructor can also be described by the following ones, but these describe something that can not be described by a previous one.

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype);
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype);
int MPI_Type_create_indexed_block(int count, int blocklength,
                                  const int array_of_displacements[],
                                  MPI_Datatype oldtype,
                                  MPI_Datatype *newtype);
int MPI_Type_indexed(int count, const int *array_of_blocklengths,
                     const int *array_of_displacements,
                     MPI_Datatype oldtype, MPI_Datatype *newtype);
int MPI_Type_struct(int count, const int *array_of_blocklengths,
                    const MPI_Aint *array_of_displacements,
                    const MPI_Datatype *array_of_types,
                    MPI_Datatype *newtype);
int MPI_Type_create_struct(int count, const int array_of_blocklengths[],
                           const MPI_Aint array_of_displacements[],
                           const MPI_Datatype array_of_types[],
                           MPI_Datatype *newtype);
```

Note that the naming of these type creating functions is somewhat inconsistent. This has historical reasons, and the MPI archeologist can mine out which.

Before a derived datatype can be used in communication operations, it must be *committed* to the MPI library. The `MPI_Type_commit` operation is a handle where the MPI library can perform optimizations on the type map description; such optimizations (that can be costly) can hopefully be amortized over many uses of the same, derived datatype. As with other MPI objects, derived datatypes should be freed after use, as they may take up (rarely, but sometimes considerable) resources.

```
int MPI_Type_commit(MPI_Datatype *datatype);

int MPI_Type_free(MPI_Datatype *datatype);
```

Only derived datatypes created in the application must and can be freed. The predefined datatypes `MPI_INT`, `MPI_DOUBLE`, etc.., cannot be freed.

The constructors describe data layouts of the following kinds. As can be seen from the interface listings, all constructors take (various kinds of) repetition counts, lists of displacements, and previously defined units of communication described as derived datatypes.

1. A *contiguous type* describes a contiguous repetition of an already described unit, where one unit follows immediately after the previous one.

2. A *vector type* describes a regularly strided (spaced) repetition of blocks of an already described unit.

3. A *block index type* describes a sequence of contiguous blocks of previously described units, each with a specific, relative displacement; all blocks have the size in number of units.

4. An *index type* describes a sequence of blocks of previously described units, each with a specific, relative displacement; blocks may have different sizes in number of units.

5. A *structured type* describes a sequence of blocks of previously described units, each with a specific, relative displacement, block may have different sizes in number of units, and the units of each block may be different, previously described units.

The elements in contiguous blocks of units are spaced from each other by the *extent* of the unit, see Section 3.2.14, and likewise all relative displacements are in multiples of the extent the previous unit. Only for structured types, the displacements are in Bytes (since here different units can be given for different blocks). The extent of a constructed, new derived datatype (unit) is the linear distance from the beginning of the first block to the end of the last block in the unit.

It is worth noticing that with the types of constructors described above, it is indeed possible to construct type maps where some data elements have the same displacement, and such type maps are not per se illegal or disallowed. A type map with this property is said to have *overlapping entries*. The rules for matching communication so are intended to enforce that the outcome of a communication operation is determinate. Thus, in particular, datatypes used as arguments for receive operations must not have overlapping entries. For datatypes used as send arguments, this is not a problem, and thus allowed.

In the following example an `MPI_Type_vector` constructor is used to describe an n column submatrix of an $m \times (np)$ matrix with m rows and np columns, where p is the number of MPI processes. In the program, all processes have a matrix of this size, and send their first n columns to the process with rank 0. Matrices are maintained per hand in row-major order. The elements corresponding to n consecutive columns are thus blocks of n elements starting at each multiple inp of np for $i, 0 \leq i < m$. The resulting, full $m \times (np)$ matrix is stored at process 0 in a separately allocated, new matrix. Thus, it cannot happen that a process sends and receives data from overlapping memory regions.

```
int m, n; int i, j;

m = ...;
n = ...;

double *matrix;
matrix = (double*)malloc(m*size*n*sizeof(double));

MPI_Datatype cols;
MPI_Type_vector(m,n,n*size,MPI_DOUBLE,&cols);
MPI_Type_commit(&cols);

MPI_Request request;
MPI_Isend(matrix,1,cols,0,MATTAG,comm,&request);
if (rank==0) {
  double *newmatrix;
  newmatrix = (double*)malloc(m*size*n*sizeof(double));

  for (i=0; i<size; i++) {
    MPI_Recv(newmatrix+i*n,1,cols,i,MATTAG,comm,MPI_STATUS_IGNORE);
  }
}
MPI_Wait(&request,MPI_STATUS_IGNORE);

MPI_Type_free(&cols);
```

In the example, where communication is of the individual $m \times n$ submatrices by point-to-point communication, the extent of the vector datatype does not play a role. This is not always so, and sometimes the default extent of a derived datatype is not what is effectively needed in order to access the data in the right locations. An important type creating function for controlling the extent of a datatype, outside the scope of this lecture, is the resizing function in which a new datatype with arbitrary extent is created from an existing, derived datatype.

```
int MPI_Type_create_resized(MPI_Datatype oldtype,
                            MPI_Aint lb, MPI_Aint extent,
                            MPI_Datatype *newtype);
```

Should displacements in multiples of the extent of the `MPI_Datatype` old unit not be sufficient, also constructors where all strides and displacements are in Bytes are provided.

```
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
                     MPI_Datatype oldtype, MPI_Datatype *newtype);
int MPI_Type_hindexed(int count, const int *array_of_blocklengths,
                      const MPI_Aint *array_of_displacements,
                      MPI_Datatype oldtype, MPI_Datatype *newtype);
```

```
int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
                            MPI_Datatype oldtype, MPI_Datatype *newtype);
int MPI_Type_create_hindexed_block(int count, int blocklength,
                                   const MPI_Aint array_of_displacements[],
                                   MPI_Datatype oldtype,
                                   MPI_Datatype *newtype);
int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],
                             const MPI_Aint array_of_displacements[],
                             MPI_Datatype oldtype, MPI_Datatype *newtype);
```

Complex (composite) layouts corresponding to distributed arrays and subarrays can be decribed with the following two, composite derived datatype constructors; that are also beyond the scope of this lecture.

```
int MPI_Type_create_darray(int size, int rank, int ndims,
                           const int array_of_gsizes[],
                           const int array_of_distribs[],
                           const int array_of_dargs[],
                           const int array_of_psizes[],
                           int order,
                           MPI_Datatype oldtype, MPI_Datatype *newtype);

int MPI_Type_create_subarray(int ndims,
                             const int array_of_sizes[],
                             const int array_of_subsizes[],
                             const int array_of_starts[],
                             int order,
                             MPI_Datatype oldtype, MPI_Datatype *newtype);
```

We finally mention that MPI provides a special datatype for opaque, compact storage of data described by derived datatypes. The datatype for such data is `MPI_PACKED`, and three functions make it possible to pack and unpack data into this format. This functionality should ideally never be needed.

```
int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype,
             void *outbuf, int outsize, int *position, MPI_Comm comm);
int MPI_Unpack(const void *inbuf, int insize, int *position,
               void *outbuf, int outcount, MPI_Datatype datatype,
               MPI_Comm comm);
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
                  int *size);
```

3.2.20 *MPI Concept: Progress*

When is (point-to-point) communication that is eventually to happen, for instance by a pair of correctly matching send and receive operations, actually

happening? The MPI does not prescribe how the underlying system is to be implemented. The loosely stated rule, is that correct communication that could happen, eventually should happen, at the very latest when `MPI_Finalize` or some other MPI operation is invoked. This give a lot of freedom to MPI library implementers, and implementors are taking this freedom. There are three basic implementations alternative to ensure progress in MPI.

1. Hardware (communication network, and network processor)
2. Separate thread in the MPI runtime system
3. With MPI library calls

To ensure progress, it is commonly good advice to make MPI calls regularly in the application.

3.2.21 *One-sided Communication*

With the two-sided, point-to-point communication seen so far, the two communicating processes (which may under circumstances be the same process) are both explicitly involved, one specifying where the data to be sent are located in the process local memory of the sending process and how they are structured, and the other one specifying where the data to be received are to go and how they are structured in that process' local memory. Communication can take place when both processes have posted their respective calls, and complete according to the semantics described so far as part of the communication modes and operations that are being used.

In contrast, with MPI *one-sided communication*, one process alone is explicitly initiating the communication, and therefore has to specify what is happening at both sides. MPI provides one-sided communication operations for retrieving data (`MPI_Get`) from another process, for transferring data to another process (`MPI_Put`), for transferring data to and performing an (`MPI_Op`) operation at another process (`MPI_Accumulate`), as well as a number of special, atomic operations on data at another process. The process that initiates the communication operation is in MPI terms referred to as the *origin process* and the process to which data are transferred or from which data are retrieved as the *target process*. In order to ensure that a data transfer has taken place and is completed, whether at origin or at target process, an explicit synchronization must be performed which can involve both origin and target processes. With one-sided communication, synchronization is thus explicit and decoupled from the communication operation. This was different for point-to-point communication where synchronization and completion is coupled to the communication operation, regardless of whether this is blocking or non-blocking. Also, in contrast to point-to-point communication, all one-sided communication calls are *non-blocking* in the MPI sense.

In the distributed-memory programming model, processes do not share address spaces in any way, and an address (pointer) at one process has no meaning for another process. Thus, means are needed to make it possible for an origin process to address data at a target process. The means in MPI is that processes participating in one-sided communication expose parts of their memory in a special, distributed data structure called a *communication window* for which a handle of type `MPI_Win` is defined. Data at target processes are referenced by (non-negative) displacements and translated into addresses into the exposed memory at the target processes. MPI provides the `MPI_Win_create` collective operation for creating a communication window in which each process gives the process local address and the size (in Bytes) of the memory it will expose, together with a process local displacement unit that is used when translating displacements into addresses. The MPI operations for managing windows and memory are shown below.

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
                   MPI_Comm comm, MPI_Win *win);
int MPI_Win_free(MPI_Win *win);

int MPI_Win_get_group(MPI_Win win, MPI_Group *group);

int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
                     MPI_Comm comm, void *baseptr, MPI_Win *win);
int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,
                            MPI_Comm comm, void *baseptr, MPI_Win *win);
int MPI_Win_shared_query(MPI_Win win,
                         int rank, MPI_Aint *size, int *disp_unit,
                         void *baseptr);
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win);
int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size);
int MPI_Win_detach(MPI_Win win, const void *base);
```

Window creation is a collective operations for the processes in the communicator used in the call. Memory per process that is to be exposed to other processes must have been allocated in advance, either with a C standard memory allocator like `malloc()` or with a special, dedicated memory-allocator that is provided by the MPI library implementation. Using stack allocated data in a communication window is dangerous practice since this memory can go away before the window is freed. The rationale for having special allocators is that a HPC system may have special regions of memory that are particularly suited to one-sided communication, e.g., that can be read and written by other processors with special instructions, or that some memory can be shared between some MPI processes (for instance processes on the same shared-memory compute node). The special allocators (with special free operation) makes it possible to enforce in a portable way the use of such memory regions. Window objects should, as always, be freed when no longer used in the application;

allocated and exposed memory must be freed explicitly; freeing is not done by `MPI_Win_free`.

The `MPI_Info` object makes it possible to provide additional information on the use of the communication window to the MPI library. A valid argument is always `MPI_INFO_NULL`.

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr);
int MPI_Free_mem(void *base);
```

```
int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used);
int MPI_Win_set_info(MPI_Win win, MPI_Info info);
```

The one-sided communication operations are listed below.

```
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
            int target_rank, MPI_Aint target_disp, int target_count,
            MPI_Datatype target_datatype, MPI_Win win);
int MPI_Put(const void *origin_addr, int origin_count,
            MPI_Datatype origin_datatype,
            int target_rank, MPI_Aint target_disp, int target_count,
            MPI_Datatype target_datatype, MPI_Win win);

int MPI_Accumulate(const void *origin_addr, int origin_count,
                   MPI_Datatype origin_datatype,
                   int target_rank, MPI_Aint target_disp, int target_count,
                   MPI_Datatype target_datatype, MPI_Op op, MPI_Win win);

int MPI_Get_accumulate(const void *origin_addr, int origin_count,
                       MPI_Datatype origin_datatype,
                       void *result_addr, int result_count,
                       MPI_Datatype result_datatype,
                       int target_rank, MPI_Aint target_disp,
                       int target_count, MPI_Datatype target_datatype,
                       MPI_Op op, MPI_Win win);

int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
                     MPI_Datatype datatype, int target_rank,
                     MPI_Aint target_disp,
                     MPI_Op op, MPI_Win win);

int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,
                         void *result_addr, MPI_Datatype datatype,
                         int target_rank, MPI_Aint target_disp, MPI_Win win);
```

The `MPI_Get` and `MPI_Put` calls are the two basic one-sided communication calls. Each specifies data for the operation at the calling, *origin process* in the usual form of a base address, an element count, and a datatype that describes the kind and structure of the elements. What is to happen at the

target process is likewise given with the operation in the form of a relative displacement, an element count, and a datatype. Data at both origin and target processes can be arbitrarily structured, and any (committed) predefined or user-defined derived datatype can be used for both `origin_datatype` and `target_datatype`. The two datatypes can even be different. However, for a one-sided communication call to be correct, the signature of the data to be transmitted must be a prefix of the signature of the data to be received. Thus, for `MPI_Get`, the `target_count` and `target_datatype` must be a prefix of the `origin_count` and `origin_datatype`, and for `MPI_Put` the other way around. This is similar to the rule for point-to-point communication. As with point-to-point communication, also `MPI_PROC_NULL` can be used as rank for the target process; no communication will take place.

The one-sided communication calls are like the non-blocking point-to-point operations: They only indicate that communication eventually is to take place. When this exactly happens is dependent on the synchronization mechanisms that will be used, and, to a very large extent, on the MPI library implementation. In order to be able to write provably correct programs, MPI poses strict conditions on which data elements can be written. These rules in effect states that no data element may possibly be written by more than one one-sided communication operation before synchronization has taken place; programs that violate this rule are simply erroneous. As with so many other things in MPI, it is solely the programmer's responsibility to ensure that this cannot happen. Thus, two or more `MPI_Put` operations are not allowed to put any data to the same target address, and two or more `MPI_Get` operations are not allowed to retrieve data into the same origin address. Concurrent `MPI_Get` and `MPI_Put` operations that reference the same address are also not allowed; this situation is a classical *data race*. Different one-sided communication operations cannot be kept separate from each other by means of message tags, as was the case for point-to-point communication.

A one-sided communication operation that access data at a target process with some displacement `disp`, will access the address

$$\texttt{base} + \texttt{disp} \cdot \texttt{disp\_unit}$$

where both `base` and `disp_unit` are the value provided in the `MPI_Win_create` call by the target process. In most common uses of one-sided communication, all processes give the same `disp_unit`.

The `MPI_Accumulate` call is like an `MPI_Put` operation, but will apply the supplied MPI binary `MPI_Op` operator on the origin and target elements. The `MPI_Accumulate` operation is an exception to the stated rules: several concurrent operations can update the same elements. Such concurrent updates are performed like atomic operations, but are atomic only per element. The `MPI_Get_accumulate` retrieves the old values from the target memory before doing the accumulation. Only the predefined `MPI_Op` operators can be used, and not user-defined operators (think about why this is the case).

The atomic *fecth-and-op* and *compare-and-swap* operations provide atomic operation functionality to MPI, and can be used (only) on single elements of a predefined datatype. An efficient MPI library implementation may be able to execute these calls by native, atomic operations, at least under some circumstances.

3.2.22 *One-sided communication completion and synchronization*

A one-sided communication operation by itself is non-blocking and neither determines when data are transferred between origin and target processes, nor when data will be available at either of the processes. This must be enforced by explicit synchronization operations.

In order to understand, work with, and reason about one-sided communication, MPI uses a so-called *communication epoch* model. From each process' point of view, one-sided communication takes place in disjoint epochs. Epochs are opened and closed by synchronization operations. A process that wants to access the window memory of some other process must open a next epoch for access to that process. A process whose window memory may be accessed by another process must open an epoch for exposure to that process.

The MPI one-sided communication model provides two kinds of synchronization operations for opening epochs: With *active synchronization*, both origin and target processes actively open their respective access and exposure epochs. With *passive synchronization*, the origin process alone is will open epoch for access (at the origin process) and exposure (at the target process). Epochs must be explicitly closed. When an origin process closes its access epoch, all one-sided communication operations will be completed from the origin process' point-of-view. In particular, all data elements retrieved by `MPI_Get` or `MPI_Get_accumulate` operations will be available for use. When a target process closes its exposure epoch, all one-sided communication operations on that target will be complete at the target. In particular, data transferred with `MPI_Put` will be available for use.

MPI provides two kinds of operations for active synchronization. The `MPI_Win_fence` is a collective operation over all processes belonging to the window. An `MPI_Win_fence` will close a preceding epoch, and for each process open an access epoch with access to all other processes, and an exposure epoch giving exposure to all other processes. Dedicated, more specific control over access and exposure is provided by the `MPI_Win_start` and `MPI_Win_post` operations, the first one providing access to a group of processes (represented as `MPI_Group` objects, see Section 3.2.9), the second one granting exposure to a group of processes. Access and exposure epochs are explicitly closed with `MPI_Win_complete` and `MPI_Win_wait`, respectively. The `MPI_Win_test` operation is a non-blocking version of `MPI_Win_wait`.

```
int MPI_Win_fence(int assert, MPI_Win win);
```

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win);
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win);
int MPI_Win_complete(MPI_Win win);
int MPI_Win_wait(MPI_Win win);
int MPI_Win_test(MPI_Win win, int *flag);

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win);
int MPI_Win_unlock(int rank, MPI_Win win);

int MPI_Win_lock_all(int assert, MPI_Win win);
int MPI_Win_unlock_all(MPI_Win win);
```

The `MPI_Win_lock` and `MPI_Win_unlock` operations passively opens a target exposure epoch and an origin access epoch. These operation have nothing to do with locks in the sense seen so far.

3.2.23 *Example: One-sided stencil updates*

As an example, we implement the stencil update that we saw before with blocking and non-blocking point-to-point communication now using one sided communication instead.

The window is created from the Cartesian communicator that was created for defining the neighborhoods, see Section 3.2.13. An advantage over the point-to-point implementations could for instance be if in some iteration there is not an update to all for the four neighbors.

```
double halo[4*c];
MPI_Win win;
MPI_Win_create(halo,4*c*sizeof(double),sizeof(double),MPI_INFO_NULL,
               cartcomm,&win);

done = 1;
while (!done) {
  MPI_Win_fence(MPI_MODE_NOPRECEDE,win);
  MPI_Put(out_left,c,MPI_DOUBLE,left,0,c,MPI_DOUBLE,win);
  MPI_Put(out_right,c,MPI_DOUBLE,right,c,c,MPI_DOUBLE,win);
  MPI_Put(out_up,c,MPI_DOUBLE,up,2*c,c,MPI_DOUBLE,win);
  MPI_Put(out_down,c,MPI_DOUBLE,down,3*c,c,MPI_DOUBLE,win);
  MPI_Win_fence(MPI_MODE_NOSUCCEED,win);

  // now use information in halo array
  done = 1;
}

MPI_Win_free(&win);
```

With the same window, but using the dedicated synchronization mechanism.

```
MPI_Group accessexposure;
MPI_Group group;

int neighbors[4] = {left,right,up,down}; // the ranks must be different
MPI_Comm_group(cartcomm,&group);
MPI_Group_incl(group,4,neighbors,&accessexposure);

while (!done) {
  MPI_Win_start(accessexposure,0,win);
  MPI_Win_post(accessexposure,0,win);

  MPI_Get(in_left,c,MPI_DOUBLE,left,0,c,MPI_DOUBLE,win);
  MPI_Get(in_right,c,MPI_DOUBLE,right,c,c,MPI_DOUBLE,win);
  MPI_Get(in_up,c,MPI_DOUBLE,up,2*c,c,MPI_DOUBLE,win);
  MPI_Get(in_down,c,MPI_DOUBLE,down,3*c,c,MPI_DOUBLE,win);

  MPI_Win_wait(win);
  MPI_Win_complete(win);

  // now use information in the input buffers
  done = 1;
}

MPI_Group_free(&accessexposure);
MPI_Group_free(&group);

MPI_Win_free(&win);
```

3.2.24 *Example: Distributed-memory Binary Search*

Slide Notes

Slide examples explains how binary search and merging by co-ranking can be implemented with one-sided communication.

3.2.25 *Additional one-sided communication operations⋆*

```
int MPI_Rput(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype,
             int target_rank, MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request);
int MPI_Rget(void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
             int target_rank, MPI_Aint target_disp, int target_count,
```

```
            MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request);
int MPI_Raccumulate(const void *origin_addr, int origin_count,
                    MPI_Datatype origin_datatype,
                    int target_rank, MPI_Aint target_disp, int
                    target_count, MPI_Datatype target_datatype,
                    MPI_Op op, MPI_Win win,MPI_Request *request);
int MPI_Rget_accumulate(const void *origin_addr, int origin_count,
                       MPI_Datatype origin_datatype,
                       void *result_addr, int result_count,
                       MPI_Datatype result_datatype,
                       int target_rank, MPI_Aint target_disp, int
                       target_count, MPI_Datatype target_datatype,
                       MPI_Op op, MPI_Win win, MPI_Request *request);
```

3.2.26 *MPI Concepts: Collective Semantics*

We have seen many examples of MPI operations that are collective in the sense that they have to be called by all processes belonging to the input communicator. More concretely, for a collective operations *C* that is to be used on a communicator `comm`, if some process calls *C*, then all other processes in `comm` must also eventually call *C*, and no other collective before *C*. By this rule, for each communicator the application programmer must ensure that all collective calls are done in the same order by all processes in the communicator. As with other calls and operations in MPI, disregarding this rule and doing something else is plain wrong and the outcome undefined. Concretely, this means that any behavior is possible: deadlock, memory corruption, immediate program crash, and even successful completion with apparently sensible results. The latter is the most misleading and dangerous behavior!

Collective operations like *C* are always called *symmetrically*, that is the same function *C* is called by all processes, but the processes can give different parameters, and the arguments can have a different meaning on the different processes (see shortly). For all collectives, arguments must be given *consistently* over the calling processes, This means different things for different collectives, but disregarding the rules on consistent arguments is wrong, and there is no guarantee on how an MPI library may react (deadlock, crash, weird results, ...). For instance, for the `MPI_Comm_create` collective operation (see Section 3.2.6) there are rules for the input group arguments, namely that all processes that belong to a group given as input by some process must call with an equivalent group argument (recall that groups are process local objects; all processes in a group must have created a group for the same set of processes in the same order).

Here two examples illustrating the consistency rules (anticipating the collective operations of the next section). The `MPI_Bcast` operation broadcasts a buffer of some number of elements from a `root` process to all other pro-

cesses in the communicator. It is a consistency requirement that all processes specify exactly the same number of elements (adhering to the type signature rules). In the first example, inadvertently the non-root processes gave a larger element count than the root process. The program may well run with most MPI libraries, but the outcome will sooner or later prove fatal: the last, fourth element in the `dims` array has never been received by the non-root processes, and anything may be in `dims[3]`.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

if (rank==root)
  MPI_Bcast(&dims[0],3,MPI_INT,root,comm);
} else {
  MPI_Bcast(&dims[0],4,MPI_INT,root,comm);
}
```

In the second example, the non-roots gave the fixed root value 0 for the fourth argument of the `MPI_Bcast` call. The consistency requirement for `MPI_Bcast` is, however, that all processes must give exactly the same value for the `root` argument. The program will most likely hang with most MPI libraries when `root` is *not* process 0 in the communicator.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

if (rank==root)
  MPI_Bcast(&dims[0],4,MPI_INT,root,comm);
} else {
  MPI_Bcast(&dims[0],4,MPI_INT,0,comm);
}
```

In contrast, point-to-point communication is *non-symmetric*: There are distinct send operation, like `MPI_Send`, `MPI_Isend`, ..., and different, distinct receive operations, like `MPI_Recv`, `MPI_Irecv`, ...

Collective operations seen so far, and also those that will be introduced in the next section, are all *blocking* in the MPI sense. When a process returns from a collective call *C*, the operation has been completed from that process' point of view. All resources needed on the process for the call have been given free by the call, and can be reused. In collective operations for exchanging information between processes, this in particular means that data are out of send buffers, and have been delivered in receive buffers. Send buffers can be used freely again to store new data for following communication operations, and values in receive buffers can be used for computation by the process. Like for point-to-point communication, also (some) non-blocking collective operations have been defined in MPI. The semantic rules are slightly different from those for non-blocking point-to-point communication. Non-blocking

collective operations are beyond these lectures (some will be mentioned for completeness in the script, though, see Section 3.2.30).

Blocking collective operations have *non-local completion* which means (as for point-to-point communication) that for a process to complete a collective call, it may require, and in most cases does require(!) that the other processes in the communicator be actively engaged in the operation. The rules for correct usage of collective operations exactly ensure that for any collective call C made by some process, eventually all processes in the communicator will have made the collective call to C, and at the latest at that point, C can be completed on the processes.

On the other hand, collective operations are, or should by the application programmer be thought of as *non-synchronizing*. A process returning from its blocking collective call C cannot make any inference on what any of the other processes have done or not done. Some processes may not even have reached the point in their code where they perform the C call! There is one conspicuous, obvious exception to this rule (think ahead).

A program using collective operations that relies on synchronizing behavior, or makes any such assumptions is called *unsafe*. Unsafe programming is a pernicious practice: A program may well run under some circumstances (MPI library, system, number of compute-nodes, ...), and then suddenly not run anymore (or produce wrong results) when circumstances change. Unsafe programs are non-portable programs!

3.2.27 *Collective Communication and Reduction Operations*

Collective communication in MPI, the third important communication model, more specifically refers to the small set of 17 functions or patterns (see Section 1.3.4) for data exchange and reduction operations over all processes in a communicator. These 17 collective operations are what is commonly meant by the term (MPI) *collectives*.

The MPI collectives are broadly of the following kind. They are invoked symmetrically by all processes belonging to a communicator.

- A *barrier operation* ensures that all processes have reached the same point in their execution.
- A *broadcast operation* transfers the same data from one designated process to all other processes.
- A *gather operation* collects data from all processes to one designated process.
- A *scatter operation* transfers individual data from one designated process to each of the other processes.

- An *allgather operation*, also known as alltoall broadcast, gathers data from all processes to all processes, or, equivalently, broadcasts data from each process to all other processes.
- An *alltoall operation*, also known as *personalized exchange*, or transpose, transfers individual data from each process to each of the other processes.
- A *reduction operation* applies a binary, associative operation to data contributed by the processes, and makes the result available to one or all processes in total or in part.
- A *scan operation* performs a prefix-sums computation on data contributed by the processes.

The designated process for the broadcast, gather and scatter operations is called the *root process* or just *root*. The operations exist in different variants according to the amount of data that are supplied and collected by the processes. Variants of the operations where each process either receives or sends the same amount of data to other processes are called *regular*. Variants where different processes may send and/or receive amounts of data that are different from other processes' amounts are called *irregular*. For historical reasons, the irregular variants of the MPI collective operations are sometimes (not in all cases) called "vector" variants. Data are always specified as block of elements, each block by a count and (derived) datatype argument. It is sometimes helpful, especially for the reduction and scan operations, to think of input and output as vectors of elements (often of the same, basic datatype like `MPI_INT`, `MPI_FLOAT`, etc.).

It is sometimes helpful as a mnemonic to classify the collectives along dimensions of data exchanged, and whether some process has a special role: Regular vs. irregular ("vector"), and rooted (asymmetric) vs. non-rooted (symmetric). See Table 2 for such a classification using the names given to the collectives by MPI.

The performance and concrete implementation of the collectives are as for everything else in MPI *not* specified by the MPI standard. In order to say something about what can be expected, assumptions have to be imposed from the outside.

Complexity of the regular collectives in a simple, homogeneous, linear-cost transmission model (see Section 3.1.3) on fully-connected networks with one-ported communication capabilities, with p processors and total data m is as stated in Table 3. On networks that are not fully connected, having diameter larger than one (see Section 3.1.1), the complexities are as stated in Table 4. Finding the algorithms that achieve these bounds is not in all cases at all trivial; see our HPC lecture. A good starting point for the interested reader is [17] and [15] with interesting trade-offs for alltoall communication. For collective algorithms, it is important that the dominating terms in the upper bounds which often correspond to the number of communication rounds or

Table 2: Classification of the MPI collective operations.

	Regular `MPI_Barrier`	Irregular (vector)

Rooted (asymmetric)	`MPI_Bcast`
	`MPI_Gather`
	`MPI_Scatter`
	`MPI_Reduce`
		`MPI_Gatherv`
		`MPI_Scatterv`

Non-rooted (symmetric)	`MPI_Allgather`
	`MPI_Alltoall`
	`MPI_Allreduce`
	`MPI_Reduce_scatter_block`
	`MPI_Scan`
	`MPI_Exscan`
		`MPI_Allgatherv`
		`MPI_Alltoallv`
		`MPI_Alltoallw`
		`MPI_Reduce_scatter`

critical path length have small constants, and analyzing (and improving) these constant terms is important.

The interface specifications for the regular communication/data exchange collectives are listed below. The `MPI_Barrier` operation is special: It does not communicate any data, but has the sole effect of logically synchronizing the processes. All processes in the communicator must eventually call the barrier operation, and no process is allowed to return from this blocking call before all other processes have made their call to `MPI_Barrier`. This is the only (blocking) collective with synchronizing behavior and where a process that returns from its call can infer and rely on (all) other processes also having made the call. For all other blocking collectives, the return from a call by a process means only that the operation has been completed from that process' point of view. It is not possible to infer anything about the other processes in general, some may not even have made the corresponding call. Relying on synchronizing behavior of collectives is another example of *unsafe programming*, a style that can lead to unpleasant surprises with errors that can be very hard to debug.

```
int MPI_Barrier(MPI_Comm comm);

int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm);

int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm);
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
```

Table 3: Complexity of the MPI collective operations in the linear-cost communication model under fully-connected network (and one-ported) communication assumptions. The total problem size is m and the number of processes p.

Collective	Complexity
`MPI_Barrier`	$O(\log p)$
`MPI_Bcast`	$O(m + \log p)$
`MPI_Gather`	$O(m + \log p)$
`MPI_Scatter`	$O(m + \log p)$
`MPI_Allgather`	$O(m + \log p)$
`MPI_Alltoall`	Between $O(m + p)$ and $O(m \log p)$
`MPI_Reduce`	$O(m + \log p)$
`MPI_Allreduce`	$O(m + \log p)$
`MPI_Reduce_scatter_block`	$O(m + \log p)$
`MPI_Scan`	$O(m + \log p)$
`MPI_Exscan`	$O(m + \log p)$

Table 4: Complexity of the MPI collective operations in the linear-cost communication model under non-fully connected network assumptions. The total problem size is m and the number of processes p and the network diameter d.

Collective	Complexity
`MPI_Barrier`	$O(d)$
`MPI_Bcast`	$O(m + d)$
`MPI_Gather`	$O(m + d)$
`MPI_Scatter`	$O(m + d)$
`MPI_Allgather`	$O(m + d)$
`MPI_Alltoall`	$O(m + pd)$
`MPI_Reduce`	$O(m + d)$
`MPI_Allreduce`	$O(m + d)$
`MPI_Reduce_scatter_block`	$O(m + d)$
`MPI_Scan`	$O(m + d)$
`MPI_Exscan`	$O(m + d)$

```
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);

int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm);
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm);
```

For the `MPI_Bcast` operation, the designated *root process* (the process with rank equal to `root`) transfers the data stored beginning at the address `buffer` to the other processes in the communicator used in the call. Data consist of `count` elements of type and structure described by the `datatype` argument. The processes can give different datatype and count arguments, but all processes must specify the same *type signature*: the same lists of elements of a basic datatype. The collective rule are stricter than the signature rules for point-to-point and one-sided communication. Also, all processes must give the same value for the `root` argument; if they do not, a deadlock is likely to occur (such things depend on the concrete MPI library implementation and on the circumstances).

For the other collectives, similar rules apply. Data leaving a process are specified in the send buffer arguments, and data to be received by a process in the receive buffer arguments. Signatures between processes where a data transfer is to take place must be identical. For the rooted collectives, all processes must give the same root argument.

The `MPI_Gather` operation collects data from all processes to the designated root process. The data to be stored at the root process are stored starting at the `recvbuf` address. The data from each process will consist of `recvcount` elements, all of the type and structure described by the `recvtype` (derived) datatype. The data from the processes are stored in rank order, with the data from process i at the address

$$\texttt{recvbuf} + i \cdot \texttt{recvcount} \cdot \texttt{extent}$$

where `extent` is the extent of the `recvtype` datatype, as defined by the `MPI_Type_get_extent` call (explained Section 3.2.14). The data that a process contributes to the root are stored starting at the `sendbuf` address and each process contributes `sendcount` elements of type and structure given by `sendtype`. Each process send signature must be identical to the signature of the received data. For all non-root processes, the receive buffer arguments are not significant. All processes contribute data to the root, including the root itself! The data from the root to the root are stored at the address $\texttt{recvbuf} + \texttt{root} \cdot \texttt{recvcount} \cdot \texttt{extent}$, and this incurs a memory copy operation at the root process. Such a perhaps costly (perhaps not) memory copy can be avoided by letting the root process give the special address argument `MPI_IN_PLACE` for the `sendbuf` ar-

gument. Many other collective operations have the same "problem", and the MPI_IN_PLACE argument can be applied in many cases.

The MPI_Scatter operation is the "dual" of the MPI_Gather operations. Data from the root process to the other processes are stored, in rank order, at the root process' sendbuffer, and are transmitted from this buffer. The data for process i are stored at the address

$$\texttt{sendbuf} + i \cdot \texttt{sendcount} \cdot \texttt{extent}$$

where here extent is the extent of the sendtype (derived) datatype. Same rules and considerations as for MPI_Gather apply. Also here, the MPI_IN_PLACE argument can be given as the recvbuf argument at the root to prevent that data are copied from the send buffer to the receive buffer at the root.

Here is an example illustrating the use of the MPI_Gather collective together with derived datatypes. An $m \times (np)$ matrix is to be put together from column submatrices of n columns (out of np columns in total) at the root process which is done by gathering the column submatrices at the root. It is a good exercise to recap the extent rules for MPI_Gather and figure out why it is necessary to modify the extent of the receive datatype (by creating a new datatype with the MPI_Type_create_resized operation).

```
int m, n;

m = ...;
n = ...

double *matrix;
matrix = (double*)malloc(m*n*size*sizeof(double));

MPI_Datatype vec, cols;
MPI_Type_vector(m,n,n*size,MPI_DOUBLE,&vec);
MPI_Type_create_resized(vec,0,n*sizeof(double),&cols);
MPI_Type_commit(&cols);

double *newmatrix;
if (rank==0) {
  newmatrix = (double*)malloc(m*n*size*sizeof(double));
}

MPI_Gather(matrix,1,cols,newmatrix,1,cols,0,comm);

MPI_Type_free(&vec);
MPI_Type_free(&cols);
```

The MPI_Allgather operation has the same effect as would each process perform an MPI_Gather operation, that is, as p gather operations with root arguments $i = 0, \ldots, p-1$ (where p is the number of MPI processes in the

communicator argument). Equivalently, MPI_Allgather has the effect as would each process i copy its data from its sendbuf into the address recvbuf $+ i \cdot$ recvcount $\cdot$ extent and perform a broadcast operation out of this buffer of recvcount elements of type and structure described by the recvtype datatype, with all other processes also giving this buffer address (the copy would be unnecessary if the MPI_IN_PLACE argument is given. The MPI rules for MPI_IN_PLACE for MPI_Allgather are strict, though, and require that if some process give MPI_IN_PLACE as sendbuf argument, then all processes must do so). In other words, data from all processes are gather in rank order by all processes.

Finally, in MPI_Alltoall operation, each process has individual data to each other process. The data for process i are stored starting from address

$$\texttt{sendbuf} + i \cdot \texttt{sendcount} \cdot \texttt{sendextent}$$

and the data from process j are received and stored starting at address

$$\texttt{recvbuf} + j \cdot \texttt{recvcount} \cdot \texttt{recvextent} \quad .$$

The data sent to each process consist of sendcount elements of type and structure described by sendtype, and the data received of recvcount elements as described by recvtype. As can be seen, the MPI_Alltoall operation has the same effect as p MPI_Scatter operations with roots $i = 0, \ldots, p-1$, or as p MPI_Gather operations with roots $i = 0, \ldots, p-1$. For completeness, we mention that also for MPI_Alltoall, the MPI_IN_PLACE argument can be used, but with a quite different meaning and flavor. The MPI_IN_PLACE argument can be given for the sendbuf argument in which cases data are sent and received (replaced) from the recvbuf address (in rank order). If used, all processes must call with the MPI_IN_PLACE argument.

For the gather/scatter, allgather and alltoall operations, also so-called irregular or "vector" variants are defined in MPI. The interface specifications for these irregular communication/data exchange collectives are listed below.

```
int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, const int *recvcounts, const int *displs,
                MPI_Datatype recvtype, int root, MPI_Comm comm);
int MPI_Scatterv(const void *sendbuf, const int *sendcounts,
                 const int *displs, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm);

int MPI_Allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                   void *recvbuf, const int *recvcounts,
                   const int *displs, MPI_Datatype recvtype,
                   MPI_Comm comm);

int MPI_Alltoallv(const void *sendbuf, const int *sendcounts,
                  const int *sdispls, MPI_Datatype sendtype,
```

```
                  void *recvbuf, const int *recvcounts,
                  const int *rdispls, MPI_Datatype recvtype,
                  MPI_Comm comm);
int MPI_Alltoallw(const void *sendbuf, const int sendcounts[],
                  const int sdispls[], const MPI_Datatype sendtypes[],
                  void *recvbuf, const int recvcounts[],
                  const int rdispls[], const MPI_Datatype recvtypes[],
                  MPI_Comm comm);
```

Each of these operations perform the same kind of communication/data exchange operations as their regular counterpart, but the amount of data contributed among processes can vary. For instance, the `MPI_Gatherv` operations transfers data from all processes to a given root process. Data to be transferred are specified by the send buffer argument triple (`sendbuf`, `sendcount` and `sendtype`) and the processes may, in contrast to the `MPI_Gather` operation, specify different numbers of elements to be transferred. The root process has a vector (hence the "vector" suffix `v` to these operations) of counts where `recvcounts[`i`]` specifies the count of elements (of type `recvtype`) from process i. The signature of process i specified by process' i `sendcount` and `sendtype` arguments must be identical to the signature at the root process given by `recvcounts[`i`]` and `recvtype`. At the root the data are gathered starting from memory address `recvbuf`. More precisely, the data from process i are stored starting from address

$$\texttt{recvbuf} + \texttt{displs[i]} \cdot \texttt{extent}$$

where `extent` is the extent of the `recvtype` derived datatype. Thus, the displacement vector `displs` is the relative offset or displacement of the data from each process in units the extent of the receive type.

The `MPI_Scatterv`, `MPI_Allgatherv` and `MPI_Alltoallv` operations are similar. Where more data are to be transferred to other processes, there are `sendcounts` and send `displs` vectors in the argument lists, and where data are to be transferred from other processes there are `recvcounts` and receive `displs` vectors in the argument lists. There is a single datatype argument, either a `sendtype` or a `recvtype` describing the type and structure of all data sent or received. The `MPI_Alltoallw` operation is different in this respect. This special collective has a separate datatype argument for data to and from each of the other processes.

Using irregular collectives can be tedious. Assume a root process has to gather different amounts of data from the other processes (like the column vector `MPI_Gather` application above, but now with possibly different numbers of columns from each process) but actually does not know in advance how much data it is going to receive from each of the other processes. Since the `MPI_Gatherv` collective needs the `recvcounts` and `displs` vectors to be set up correctly, the element counts must first be collected from all processes. But, this is what the regular `MPI_Gather` operation is for. So, first the element counts are

gathered at the root into the recvcounts vector, based on which appropriate displacements are computed (in the example, data are stored consecutively, but this must not necessarily always be so), and then finally the data can be correctly collected with the MPI_Gatherv operation.

```
// gather counts from all processes
MPI_Gather(&sendcount,1,MPI_INT,recvcounts,1,MPI_INT,root,comm);
if (rank==root) {
  // compute displacements, on root only
  displs[0] = 0;
  for (i=1; i<size; i++) {
    displs[i] = displs[i-1]+recvcounts[i-1];
  }
}
// gather the possibly different amounts of data from all processes
MPI_Gatherv(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs,recvtype,
            root,comm);
```

Also for the irregular communication/data exchange collectives, the MPI_IN_PLACE argument can be used. Sometimes this is convenient, and can sometimes even give a performance benefit.

The reduction collectives perform an additional computation on the data supplied by the processes making the collective call. Here it is convenient to think of the processes as all supplying a vector of some count of elements of a basic datatype (like MPI_INT, MPI_FLOAT, MPI_LONG, MPI_DOUBLE, etc.), although derived datatypes can be used in some circumstances. These vectors are "reduced" elementwise in pairs using a binary operator supplied in the call. The interface specifications for the reduction type collectives are listed below.

```
int MPI_Reduce(const void *sendbuf,
               void *recvbuf, int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm);
int MPI_Allreduce(const void *sendbuf,
                  void *recvbuf, int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm);
int MPI_Reduce_scatter_block(const void *sendbuf,
                             void *recvbuf, int recvcount,
                             MPI_Datatype datatype, MPI_Op op,
                             MPI_Comm comm);
int MPI_Reduce_scatter(const void *sendbuf,
                       void *recvbuf, const int recvcounts[],
                       MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);

int MPI_Scan(const void *sendbuf,
             void *recvbuf, int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm);
int MPI_Exscan(const void *sendbuf,
               void *recvbuf, int count, MPI_Datatype datatype,
```

Table 5: Binary operators for collective reduction operations.

Operator	MPI
Sum	`MPI_SUM`
Product	`MPI_PROD`
Minimum	`MPI_MIN`
Maximum	`MPI_MAX`
Logical (wordwise) and, or, exclusive or	`MPI_LAND`, `MPI_LOR`, `MPI_LXOR`
Bitwise and, or, exclusive or	`MPI_BAND`, `MPI_BOR`, `MPI_BXOR`
Minimum with location	`MPI_MINLOC`
Maximum with location	`MPI_MAXLOC`

```
                MPI_Op op, MPI_Comm comm);
```

Let $\oplus$ be an associative, binary operator operating elementwise on vectors x and y with the same number of elements c. The reduction collective operations perform a reduction like

$$z \quad = \quad x_0 \oplus x_1 \oplus \cdots \oplus x_{p-1}$$

where x_i is the vector supplied by MPI process i, p the number of processes, and brackets can be left out by associativity; $x \oplus (y \oplus z) = (x \oplus y) \oplus z$. Operators are not assumed to be commutative, and commutativity is (often, usually) not exploited by MPI library implementations.

MPI provides a number of predefined operators working on vectors of basic datatypes stored consecutively in send and receive buffers with a count of elements. Operators are identified by the `MPI_Op` handle. It is also possible for the application programmer to define own operators by attaching a function with a predefined signature to an operator handle, but this is beyond the scope of these lectures. The standard MPI operators are are listed in Table 5.

In the reduction and scan collectives, all processes must give the same `MPI_Op` argument, otherwise the results are undefined (as can be imagined). All processes must give input vectors with the same number of elements (of the same basic datatype).

Elementwise binary reduction by some operator $\oplus$ on two input vectors means, for instance, that

$$\begin{pmatrix} x_{c-1} \\ \vdots \\ x_1 \\ x_0 \end{pmatrix} + \begin{pmatrix} y_{c-1} \\ \vdots \\ y_1 \\ y_0 \end{pmatrix} = \begin{pmatrix} x_{c-1} + y_{c-1} \\ \vdots \\ x_1 + y_1 \\ x_0 + y_0 \end{pmatrix}$$

for the + operator `MPI_SUM`, and

$$\min\left\{\begin{pmatrix} x_{c-1} \\ \vdots \\ x_1 \\ x_0 \end{pmatrix}, \begin{pmatrix} y_{c-1} \\ \vdots \\ y_1 \\ y_0 \end{pmatrix}\right\} = \begin{pmatrix} \min\{x_{c-1}, y_{c-1}\} \\ \vdots \\ \min\{x_1, y_1\} \\ \min\{x_0, y_0\} \end{pmatrix}$$

for the minimum operator `MPI_MIN`.

The reduction collectives differ in the way the output vector is stored. For the `MPI_Reduce` operation which takes a root argument, the computed result z is stored in the receive buffer at the root, and the `recvbuf` argument is significant only for the root process. For the `MPI_Allreduce` operation, all processes receive the computed result z in their respective receive buffers. With the `MPI_Reduce_scatter_block` and `MPI_Reduce_scatter` operations, the result vector z is split into subvectors $z^0, z^1, \ldots z^{p-1}$ of $c_0, c_1, \ldots c_{p-1}$ elements, respectively, with $c = \sum_{i=0}^{p-1} c_i$, and the vector z_i stored in the receive buffer at process i. For `MPI_Reduce_scatter_block`, all c_i are equal, so subvectors have the same number of elements, whereas for `MPI_Reduce_scatter` the c_i counts are stored in the input vector `recvcounts` with `recvcounts[`i`]` $= c_i$. All processes must give the same `recvcounts` vector as input. The `MPI_Reduce_scatter` operation is the irregular ("vector" variant), and `MPI_Reduce_scatter_block` the regular variant of this collective operation. The `MPI_IN_PLACE` argument can be given as `sendbuf` argument is some cases. For `MPI_Reduce`, the root can specify that data are to be taken from the `recvbuf` address (where the result is also stored) by giving `MPI_IN_PLACE` as `sendbuf` argument. For `MPI_Allreduce`, `MPI_Reduce_scatter_block`, and `MPI_Reduce_scatter`, all processes must give the `MPI_IN_PLACE` argument.

A simple, common application of collective reduction operations is for checking for agreement on some Boolean outcome. Say that all processes need to agree on some convergence criterion by all having locally satisfied the criterion. Agreement can be checked by performing a reduction with a Boolean (logical) and operation, and making sure that all processes receive the result. The case could occur in a stencil computation, which is iterated until convergence by all processes, and implemented with an `MPI_Allreduce` operation with the logical and operation `MPI_LAND`; the `MPI_IN_PLACE` argument is convenient here.

```
while (!done) {
  int k = 0;

  MPI_Isend(out_left,c,MPI_DOUBLE,left,TAG,cartcomm,&request[k++]);
  MPI_Isend(out_right,c,MPI_DOUBLE,right,TAG,cartcomm,&request[k++]);
  MPI_Isend(out_up,c,MPI_DOUBLE,up,TAG,cartcomm,&request[k++]);
  MPI_Isend(out_down,c,MPI_DOUBLE,down,TAG,cartcomm,&request[k++]);

  MPI_Irecv(in_left,c,MPI_DOUBLE,right,TAG,cartcomm,&request[k++]);
```

```
  MPI_Irecv(in_right,c,MPI_DOUBLE,left,TAG,cartcomm,&request[k++]);
  MPI_Irecv(in_up,c,MPI_DOUBLE,down,TAG,cartcomm,&request[k++]);
  MPI_Irecv(in_down,c,MPI_DOUBLE,up,TAG,cartcomm,&request[k++]);

  MPI_Waitall(k,request,MPI_STATUSES_IGNORE);

  done = 1; // some real local convergence criterion
  MPI_Allreduce(MPI_IN_PLACE,&done,1,MPI_INT,MPI_LAND,cartcomm);
}
```

The two "scan" collective operations `MPI_Scan` and `MPI_Exscan` implement the *inclusive prefix-sums* and *exclusive prefix-sums* operations (elementwise, on vectors), respectively, see Section 1.4.5. The ith elementwise inclusive or exclusive prefix-sum is stored at process i. Processes can use the `MPI_IN_PLACE` argument to indicate that input is to be taken from the `recvbuf` address (where the result is also placed).

A sometimes important, late addition to MPI, is the capability to locally execute a binary operator on two input vectors where the operator can be one of the predefined `MPI_Op` operators. This local operation is shown below; the second argument is both the second input and the address where the result is stored. This is sometimes convenient, and sometimes not; there is (unfortunately) no three-argument version of this local operation in MPI.

```
int MPI_Reduce_local(const void *inbuf,
                     void *inoutbuf, int count, MPI_Datatype datatype,
                     MPI_Op op);

int MPI_Op_commutative(MPI_Op op, int *commute);
```

Below is an implementation of a $p-1$ communication round algorithm for `MPI_Scan`, which illustrates It illustrates the use of `MPI_Reduce_local`. A copy from input in the recvbuffer to the send buffer is needed, and implemented by an `MPI_Sendrecv` operation where each process sends the input data to itself. This operation is here done here on the special `MPI_COMM_SELF` communicator which is a predefined singleton communicator handle for all processes that consist of the process itself only. This copy would be unnecessary if, in the `MPI_Scan` operation, the `MPI_IN_PLACE` argument would have been given.

```
MPI_Sendrecv(sendbuf,c,MPI_FLOAT,0,SCANTAG,
             recvbuf,c,MPI_FLOAT,0,SCANTAG,MPI_COMM_SELF,MPI_STATUS_IGNORE);
if (rank>0) {
  MPI_Recv(tempbuf,c,MPI_FLOAT,rank-1,SCANTAG,comm,MPI_STATUS_IGNORE);
  MPI_Reduce_local(tempbuf,recvbuf,c,MPI_FLOAT,MPI_SUM);
}
if (rank<size-1) {
  MPI_Send(recvbuf,c,MPI_FLOAT,rank+1,SCANTAG,comm);
}
```

For `MPI_COMM_SELF`, the following holds.

```
int rank, size;

MPI_Comm_size(MPI_COMM_SELF,&size);
MPI_Comm_rank(MPI_COMM_SELF,&rank);

assert(size==1);
assert(rank==0);
```

The algorithm is linear, and not efficient. It is a good exercise to consider in which aspects the algorithm is inefficient, and how it can be improved.

As mentioned, the it possible for the application programmer to define an register own, binary functions as `MPI_Op` operations. The functionality for this is listed below.

```
int MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op);
int MPI_Op_free(MPI_Op *op);
```

3.2.28 *Examples: Elementary Linear Algebra*

Matrix-vector multiplication and matrix-matrix multiplication are two elementary operations in linear algebra. The collective operations we have seen in the preceding sections are convenient for solving these problems in parallel without relying on shared-memory access to the input and output matrices and vectors.

In such operations, the input matrices and vectors are distributed in some way over the available processes, and the output is likewise to be distributed over the processes in some (possibly other) way. The distribution of input and output should be considered part of the *problem specification*, and an algorithm/implementation for solving any such problem must respect the prescribed distribution. If the distribution is different, either another algorithm must be developed, or the distribution must be changed (by some algorithm). Distributions are in general balanced, meaning that with p processes, each process will posses $1/p$ of the total input, and compute $1/p$ of the total output. It is obvious that no efficient algorithm can be allowed to gather together the full input or the full output (Amdahl's Law).

We first give two implementations of algorithms for performing matrix-vector multiplication for two different input and output distributions. The total input is a real-valued (`double`) $m \times n$ matrix M and a real-valued n element vector x, and the output a real-valued m element vector y with $y = Mx$. For simplicity, we assume here that p, the number of processes, divides both m and n. It is of course a good exercise to generalize the implementations to arbitrary input sizes m and n.

In the first example, the input matrix is distributed row-wise, meaning that each process has m/p full, consecutive rows of the matrix M. Process 0

the first such m/p rows, process 1 the next m/p rows, and so on. The input vector x is likewise distributed in pieces of n/p consecutive elements. The output vector y is to be distributed in the same manner with m/p consecutive elements per process.

Let M_i be the $(m/p) \times n$ part of the matrix of process i. The part of the output for process i can be computed as $y_i = M_i x$. In order to do this computation, the full x vector must be available at all processes which can be accomplished with an `MPI_Allgather` operation. The rest is easy.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

assert(m%size==0); // regular only
assert(n%size==0);

fullvector = (double*)malloc(n*sizeof(double));

MPI_Allgather(vector,n/size,MPI_DOUBLE,fullvector,n/size,MPI_DOUBLE,comm);
for (i=0; i<m/size; i++) {
  result[i] = matrix[i][0]*fullvector[0];
  for (j=1; j<n; j++) {
    result[i] += matrix[i][j]*fullvector[j];
  }
}
free(fullvector);
```

The run time complexity of this first algorithm can easily be analyzed as follows. Following Table 3, the allgather operation can be done in $O(n + \log p)$ time. The process local matrix-vector product computation takes $O((m/p)n)$ time, for a total $O((m/p)n + n + \log p)$ time steps. This is work-optimal since sequential matrix-vector multiplication takes $O(mn)$ time steps for p processors with p in $O(m)$ processors, if we assume that $n > \log p$.

In the second example, the input matrix is distributed column-wise, meaning that each process has n/p consecutive columns with m rows of the matrix M. Process 0 the first such n/p columns, process 1 the next n/p columns, and so on. The input vector x is likewise distributed in pieces of n/p consecutive elements. The output vector y is to be distributed in the same manner with m/p consecutive elements per process.

Let M'_i be the $m \times (n/p)$ part of the matrix of process i. The full output vector y can be computed as $y = \sum_{i=0}^{p-1} M'_i x_i$, and this y be distributed into the parts y_i of m/p consecutive elements per process. The summation and distribution of the parts can be accomplished by an `MPI_Reduce_scatter_block` operation.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);
```

```
assert(m%size==0); // regular only
assert(n%size==0);

partial = (double*)malloc(m*sizeof(double));

for (i=0; i<m; i++) {
  partial[i] = matrix[i][0]*vector[0];
  for (j=1; j<n/size; j++) {
    partial[i] += matrix[i][j]*vector[j];
  }
}

MPI_Reduce_scatter_block(partial,result,m/size,MPI_DOUBLE,MPI_SUM,comm);
free(partial);
```

The run time complexity of the second algorithm can easily be analyzed as follows. The process local work for the initial matrix-vector multiplication is $O(m(n/p))$. Following Table 3, the reduce-scatter operation can be done in $O(m + \log p)$ time, for a total $O(m(n/p) + m + \log p)$ time steps. Again, this is work-optimal since for p processors with p in $O(n)$, if we assume that $m > \log p$.

Summarizing, we have found the following.

Theorem 15 *Matrix-vector multiplication of an $m \times n$ matrix with an n element vector can be done work-optimally on a p processor system with message-passing communication in $O(mn/p + \min(m, n) + \log p)$ time steps.*

Which of the two algorithms perform better in practice depends on the quality of the implementation of the `MPI_Allgather` and `MPI_Reduce_scatter_block` operations, and on the magnitude of m and n. Keep in mind that the two algorithms assume different distributions of the input matrix! A more scalable algorithm, one for which more processors can be employed with linear speed-up, can be given by combining the two ideas (with a different distribution of the input).

The more challenging looking operation to perform without having the matrices stored in shared memory and being accessible to every thread (process) is matrix-matrix multiplication. Given an $m \times l$ input matrix A, an $l \times n$ input matrix B, compute the $m \times n$ output matrix C as $C = AB$. For simplicity, we assume that the number of processes p is a square (which is not entirely without loss of generality), that is $p = \sqrt{p}\sqrt{p}$ for an integer $\sqrt{p}$, and that $\sqrt{p}$ divides all of m, l, n. The input distribution is balanced such that each process has input submatrices of $(m/\sqrt{p}) \times (l/\sqrt{p}) = ml/p$ and $(l/\sqrt{p}) \times (n/\sqrt{p}) = ln/p$ elements, respectively, and produces an output submatrix of $(m/\sqrt{p}) \times (n/\sqrt{p}) = mn/p$ elements.

We organize the processes in a quadratic, 2-dimensional mesh, and give each processor a coordinate (i, j), for instance by creating a Cartesian commu-

nicator with `MPI_Cart_create`. The submatrices for process i, j are denoted by A_{ij}, B_{ij} and C_{ij}, respectively. Each output submatrix C_{ij} is computed by

$$C_{ij} = \sum_{k=0}^{\sqrt{p}-1} A_{ik} B_{kj}$$

We observe that on each row of processes, the same A_{ik} submatrices are needed by all processes, and on each column of processes, the same B_{kj} submatrices are needed by all processes. This can be accomplished by $\sqrt{p}$ broadcast operations on the rows and on the columns of processes. To implement this conveniently with MPI, communicators for the processes on the rows and on the columns are needed, and we saw how such communicators could be created with the proper `MPI_Comm_split` operations. This potentially expensive communicator creation should be done once and for all. The matrix-matrix multiplication can now easily be implemented as indicated below. The row and column communicators are `rowcomm` and `colcomm`. The multiplication and summation of submatrices is done by an efficient, sequential implementation which is encoded in the "fused-matrix-multiply-add" procedure `fmma`.

```
int rowsize, colsize;
int rowrank, colrank;

MPI_Comm_size(rowcomm,&rowsize);
MPI_Comm_size(colcomm,&colsize);
assert(rowsize==colsize); // size is square

MPI_Comm_rank(rowcomm,&rowrank);
MPI_Comm_rank(colcomm,&colrank);

double **Atmp, **Btmp;
// allocate space for temporary matrices

int i;
for (i=0; i<rowsize; i++) {
  double **AA, **BB;

  AA = (i==rowrank) ? A : Atmp;
  MPI_Bcast(AA[0],m/rowsize*l/rowsize,MPI_DOUBLE,i,rowcomm);

  BB = (i==colrank) ? B : Btmp;
  MPI_Bcast(BB[0],l/rowsize*n/rowsize,MPI_DOUBLE,i,colcomm);

  fmma(m/rowsize,l/rowsize,n/rowsize,C,AA,BB);
}
```

The running time of the matrix-matrix multiplication implementation can be analyzed as follows. As building block, a sequential matrix-matrix algorithm

is used which we assume use $M(m,l,n)$ operations to multiply an $m \times l$ matrix with an $l \times n$ matrix. The cost of adding to matrices is asymptotically smaller, we will assume. The algorithm performs $2\sqrt{p}$ `MPI_Bcast` operations of matrices with $(ml)/p$ and $(ln)/p$ elements accordingly. According to Table 3 this can then be done in

$$O(\sqrt{p}\frac{ml+ln}{p} + \log\sqrt{p}) \quad = \quad O(\frac{l(m+n)}{\sqrt{p}} + \log p)$$

time steps. The number of process local matrix-matrix multiplications is $\sqrt{p}$, each of which takes $M(m/\sqrt{p}, l/\sqrt{p}, n/\sqrt{p})$ time steps. The sequential matrix-matrix multiplication algorithm we have seen has $M(m,l,n) = O(mln)$, so using this algorithm gives

$$\sqrt{p}\, O((m/\sqrt{p})(l/\sqrt{p})(n/\sqrt{p})) \quad = \quad O(\frac{mln}{p})$$

with linear speed-up (for a range of processors p) for the multiplication work.

Summarizing, with the standard sequential matrix-matrix multiplication algorithm as plug-in, we have the following.

Theorem 16 *Matrix-matrix multiplication can be done work-optimally on a p processor system with message-passing communication relative to sequential $M(m,l,n) = O(mln)$ matrix-matrix multiplication in $O(mln/p + l(m+n)/\sqrt{p} + \log p)$ time steps.*

Speed-up is linear as long as p is in $O(((mn)/(m+n))^2)$, assuming that both the first and second term dominate the $\log p$ term.

The algorithm for matrix-matrix multiplication doing broadcast operations on rows and columns of processes (and improvements thereof) is called SUMMA (Scalable Universal Matrix Multiplication Algorithm) [71].

3.2.29 *Examples: Sorting Algorithms*

The Quicksort algorithm idea lends itself well to parallel implementation by point-to-point and collective communication. There are two natural variants. As in the preceding lectures, we assume that good pivots can be found by some means, which is of course crucial for both the theoretical and practical performance; but which is ignored here and to be solved somewhere else (see for instance [7, 8, 58]).

For a distributed-memory implementation, we assume that the input data (elements from some totally ordered set, like integers, floating point numbers, etc.) have been evenly distributed over the available processes. For input of n elements in total, each process will thus have (approximately) n/p elements. The elements are to be sorted and preferably each process will have approximately n/p elements of the output. The output must fulfill that for each

process, the elements in the process' part of the output is sorted, and that the elements of process i are all larger than or equal to the elements of process $i-1$ (for $i > 0$) and smaller than or equal to the elements of process $i+1$ (for $i < p-1$).

For the parallel Quicksort, we assume that the number of processes p is a power of two, $p = 2^k$ for some $k, k \geq 0$. We formulate the algorithm recursively, but recurse on the number of processes which is halved in each recursive call. An implementation for $p, p > 1$ MPI processes in a communicator `comm` would go as follows.

1. Select a global pivot for the n elements, and distribute this pivot to all p processes.
2. Processes locally partition their set of elements into elements smaller than or equal to the global pivot, and elements larger than or equal to the global pivot.
3. The processes pairwise exchange elements, such that half the processes will have elements smaller than or equal to the global pivot, and the other half of processes will have element larger than or equal to the global pivot. Concretely, this will be done such that processes with rank $i, i < p/2$ will have the smaller elements, and processes $i, i \geq p/2$ will have the larger elements.
4. The communicator `comm` with the p processes is split into two communicators with processes smaller than $p/2$ and processes larger than or equal to $p/2$, respectively.
5. Each process recursively calls Quicksort on the new communicator of $p/2$ processes to which it belongs.

With only one process, $p = 1$, a sequential Quicksort is used to sort the process' $n/p = n$ elements. With such an implementation, and a best known implementation of sequential Quicksort, absolute and relative speed-up of the implementation will coincide.

Step 1 will most likely involve one or more collective operations, e.g., `MPI_Bcast`. For Step 2, where the processes compute locally, a best known sequential implementation for partitioning (in-place) should be used, see for instance [62, 60, 61]. We note that the global pivot for the processes may actually not be in the set of input elements for any one process. For Step 3, point-to-point communication is used, for example like this (for elements of C type `double`):

```
double *a;
double *b;

int n;      // size of local block
```

```
int nn;     // index of pivot
int nl, ns; // larger and smaller elements

int half = size/2;
if (rank<half) {
  nl = n-nn;
  MPI_Sendrecv(&nl,1,MPI_INT,rank+half,QTAG,
               &ns,1,MPI_INT,rank+half,QTAG,comm,MPI_STATUS_IGNORE);
  n = nn+ns;
  b = (double*)malloc(n*sizeof(double));
  assert(n==0||b!=NULL);

  MPI_Sendrecv(a+nn,nl,MPI_DOUBLE,rank+half,QTAG,
               b+nn,ns,MPI_DOUBLE,rank+half,QTAG,comm,MPI_STATUS_IGNORE);
  memcpy(b,a,nn*sizeof(double));
} else {
  ns = nn;
  MPI_Sendrecv(&ns,1,MPI_INT,rank-half,QTAG,
               &nl,1,MPI_INT,rank-half,QTAG,comm,MPI_STATUS_IGNORE);
  n = n-nn+nl;
  b = (double*)malloc(n*sizeof(double));
  assert(n==0||b!=NULL);

  MPI_Sendrecv(a,ns,MPI_DOUBLE,rank-half,QTAG,
               b,nl,MPI_DOUBLE,rank-half,QTAG,comm,MPI_STATUS_IGNORE);
  memcpy(b+nl,a+ns,(n-nl)*sizeof(double));
}
```

The partitioning function shall compute the index `nn` in the array `a`. The processes with rank smaller than $p/2$ shall receive the smaller elements, while the higher ranked processes shall receive the larger elements. The first `MPI_Sendrecv` operation exchanges the number of small and large elements needed for this, based on which a new array `b` can be allocated, and the element exchange proper be done by the second `MPI_Sendrecv` operation. The elements for a process for the recursive call are in the newly allocated `b` array. Some care has to be taken to make sure such intermediate arrays are properly freed.

Step 4 is a typical case for the `MPI_Comm_split` operation. This may introduce overhead that can affect overall performance, and it may be worthwhile to consider whether explicit communicator splitting can be avoided.

Assuming that pivots are selected perfectly, and leads to even partitions at all levels of the recursions, the running time can be asymptotically estimated with the following recurrence relation. The $O(\log p)$ term is for the collective operations for pivot selection, and the $O(n/p)$ term for the element exchange.

$$\begin{aligned} T(n,p) &= O(\log p) + O(n/p) + T(n/2, p/2) \\ T(n,1) &= O(n \log n) \end{aligned}$$

Since $(n/2)(p/2) = n/p$ each level of the recursion will contribute the $O(n/p)$ term, and since $\log p$ recursive calls are needed (p is a power of two), the solution is

$$\begin{aligned} T(n,p) &= O(\log^2 p) + (\log p)O(n/p) + O(n/p \log(n/p)) \\ &= O(\log^2 p) + O(\frac{n \log p}{p} + \frac{n \log n - n \log p}{p})) \\ &= O(\log^2 p) + O((n/p) \log p) \end{aligned}$$

with linear speed-up when n is sufficiently large compared to p.

For well-behaved inputs and pivot selection, this implementation can work well in practice, but it does not guarantee that the output is balanced as block of n/p elements per process, for instance. It is a good exercise to consider how bad the algorithm can behave, and how worst-case input may look, also under different assumptions on the pivot selection.

Another common parallel Quicksort implementation variant which is sometimes referred to as *HyperQuicksort* [73] is to first let the processes sort their n/p elements; this makes perfect pivot selection (per process) trivial, and possibly also easier to find a good overall pivot. To maintain order, a merge step is needed after the element exchange.

These variants, and others that rely solely on collective communication operations for exchanging data are discussed further and implemented in [69].

A drawback of Quicksort as implemented here is that the number of processes must be a power of two, quite a restriction for the system that you may have at hand. Also this is good to think about.

A completely different idea for sorting (non-negative) integers is *counting sort* (or *bucket sort*) which can also be given a parallel, distributed memory implementation. Counting sort is a building block in *radix sort*. Given input of n elements (with integer keys), the idea is to count the number of occurrences for each key, by using the keys as indices into an array of counts, and use this to put the elements into consecutive buckets for each of the keys. When the key range is no larger than $O(n)$ this can be done in linear time by scanning through the elements.

In a distributed memory setting, each process will have n/p of the elements available. The counting, where a process needs to know the total element count for each key, as well as the number of occurrences of each key before its own element, is done by collective allreduce operations and a prefix-sums computation over vectors of counts. Here is a part of such a counting sort (bucket sort) implementation.

```
int n = ...; // number of buckets
int bucketsize[n];
int allsize[n], presize[n];

// do the work, fill into buckets, increment bucketsizes
```

```
MPI_Allreduce(bucketsize,allsize,n,MPI_INT,MPI_SUM,comm);
MPI_Exscan(bucketsize,presize,n,MPI_INT,MPI_SUM,comm);
```

The counts in the `presize` and `allsize` vectors can now be used to compute which elements are to be sent to other processes, and how many elements each process has to receive from other processes. The exchange can be done with `MPI_Alltoall` and `MPI_Alltoallv` operations. To complete, local sorting or reordering is needed. It is a good exercise to try to implement this idea in detail.

3.2.30 *Non-blocking Collective Operations⋆*

The 17 standard collectives explained in the last section are all blocking in the MPI sense. A recent addition to MPI, is a whole set of corresponding, non-blocking collective operations. Non-blocking collectives are not part of this lecture, but the operations are listed here for completeness. The operations complete "immediately", irrespective of any action taken by the other processes in the communicator (this is what non-blocking means), and return an `MPI_Request` object that can be used to query for and enforce completion of any given operation, just as was the case with the non-blocking point-to-point communication operations (Section 3.2.16).

A most important difference to non-blocking point-to-point communication is that blocking and non-blocking collectives cannot be combined. The reason for this is that blocking and non-blocking implementations may use (completely) different algorithms, therefore the steps taken by a process doing a broadcast with `MPI_Ibcast` may not match with the steps taken by another process doing the broadcast with `MPI_Bcast`.

The non-blocking, regular exchange operations are the following.

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request);
int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype,
               int root, MPI_Comm comm, MPI_Request *request);

int MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm, MPI_Request *request);
int MPI_Iscatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm, MPI_Request *request);
int MPI_Iallgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                   void *recvbuf, int recvcount, MPI_Datatype recvtype,
                   MPI_Comm comm, MPI_Request *request);
int MPI_Ialltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm, MPI_Request *request);
```

The non-blocking, regular reduction collectives are the following.

```
int MPI_Ireduce(const void *sendbuf,
                void *recvbuf, int count, MPI_Datatype datatype,
                MPI_Op op, int root, MPI_Comm comm, MPI_Request *request);
int MPI_Iallreduce(const void *sendbuf,
                   void *recvbuf, int count, MPI_Datatype datatype,
                   MPI_Op op, MPI_Comm comm, MPI_Request *request);
int MPI_Ireduce_scatter_block(const void *sendbuf,
                              void *recvbuf, int recvcount,
                              MPI_Datatype datatype,
                              MPI_Op op, MPI_Comm comm, MPI_Request *request);

int MPI_Iscan(const void *sendbuf,
              void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,
              MPI_Comm comm, MPI_Request *request);
int MPI_Iexscan(const void *sendbuf,
                void *recvbuf, int count, MPI_Datatype datatype,
                MPI_Op op, MPI_Comm comm, MPI_Request *request);
```

The irregular, non-blocking data exchange operations are the following.

```
int MPI_Igatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, const int recvcounts[], const int displs[],
                 MPI_Datatype recvtype,
                 int root,  MPI_Comm comm, MPI_Request *request);
int MPI_Iscatterv(const void *sendbuf, const int sendcounts[],
                  const int displs[], MPI_Datatype sendtype,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  int root, MPI_Comm comm, MPI_Request *request);
int MPI_Iallgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                    void *recvbuf, const int recvcounts[], const int displs[],
                    MPI_Datatype recvtype,
                    MPI_Comm comm, MPI_Request *request);
int MPI_Ialltoallv(const void *sendbuf, const int sendcounts[],
                   const int sdispls[], MPI_Datatype sendtype,
                   void *recvbuf, const int recvcounts[],
                   const int rdispls[], MPI_Datatype recvtype,
                   MPI_Comm comm, MPI_Request *request);
int MPI_Ialltoallw(const void *sendbuf, const int sendcounts[],
                   const int sdispls[], const MPI_Datatype sendtypes[],
                   void *recvbuf, const int recvcounts[],
                   const int rdispls[], const MPI_Datatype recvtypes[],
                   MPI_Comm comm, MPI_Request *request);
```

Finally, there is the single, irregular non-blocking reduce-scatter operation.

```
int MPI_Ireduce_scatter(const void *sendbuf,
                        void *recvbuf, const int recvcounts[],
```

```
                        MPI_Datatype datatype,
                        MPI_Op op, MPI_Comm comm, MPI_Request *request);
```

A non-blocking, book keeping communicator duplicate operation is also included in MPI.

```
int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request);
```

The repertoire of non-blocking collective operations may likely grow with time.

3.2.31 *Sparse Collective Communication: Neighborhood collectives*⋆

A recent addition to MPI is a number of collective communication operations that perform data exchanges not over all processes but only among subsets of the processes. These *neighborhood collectives* are not treated in this this lecture, but the functionality mentioned here for completeness.

The idea of sparse, neighborhood collective communication is that each process can perform a data exchange operation with a small set of neighboring processes. What a neighboring process is, is defined by defining the set of neighborhoods, collectively, for all processes. In Section 3.2.7, two ways of defining neighborhoods by creating new communicators with associated neighborhoods were discussed, in detail `MPI_Cart_create`, and briefly touched upon `MPI_Dist_graph_create`.

The collective operations on sparse neighborhoods are of the allgather and alltoall type, and come in both regular and irregular variants, as well as in blocking and non-blocking variants. All neighborhood collectives are strictly collective, that is they have to be called by all processes in the communicators, and no synchronization behavior is implied.

Note that the signatures of these operations are identical to those of the standard collective operations; this be helpful for remembering how these functions look and what they do.

The regular, blocking and non-blocking variants are listed below.

```
int MPI_Neighbor_allgather(const void *sendbuf, int sendcount,
                           MPI_Datatype sendtype,
                           void *recvbuf, int recvcount,
                           MPI_Datatype recvtype, MPI_Comm comm);
int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount,
                          MPI_Datatype sendtype,
                          void *recvbuf, int recvcount,
                          MPI_Datatype recvtype, MPI_Comm comm);

int MPI_Ineighbor_allgather(const void *sendbuf, int sendcount,
                            MPI_Datatype sendtype,
                            void *recvbuf, int recvcount,
                            MPI_Datatype recvtype,
```

```
                            MPI_Comm comm, MPI_Request *request);
int MPI_Ineighbor_alltoall(const void *sendbuf, int sendcount,
                           MPI_Datatype sendtype,
                           void *recvbuf, int recvcount,
                           MPI_Datatype recvtype,
                           MPI_Comm comm, MPI_Request *request);
```

The irregular ("vector"), blocking and non-blocking variants are listed below.

```
int MPI_Neighbor_allgatherv(const void *sendbuf, int sendcount,
                            MPI_Datatype sendtype,
                            void *recvbuf, const int recvcounts[],
                            const int displs[],
                            MPI_Datatype recvtype, MPI_Comm comm);
int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[],
                           const int sdispls[], MPI_Datatype sendtype,
                           void *recvbuf, const int recvcounts[],
                           const int rdispls[], MPI_Datatype recvtype,
                           MPI_Comm comm);
int MPI_Neighbor_alltoallw(const void *sendbuf, const int sendcounts[],
                           const MPI_Aint sdispls[],
                           const MPI_Datatype sendtypes[],
                           void *recvbuf, const int recvcounts[],
                           const MPI_Aint rdispls[],
                           const MPI_Datatype recvtypes[],
                           MPI_Comm comm);

int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount,
                             MPI_Datatype sendtype,
                             void *recvbuf, const int recvcounts[],
                             const int displs[], MPI_Datatype recvtype,
                             MPI_Comm comm, MPI_Request *request);
int MPI_Ineighbor_alltoallv(const void *sendbuf, const int sendcounts[],
                            const int sdispls[], MPI_Datatype sendtype,
                            void *recvbuf, const int recvcounts[],
                            const int rdispls[], MPI_Datatype recvtype,
                            MPI_Comm comm, MPI_Request *request);
int MPI_Ineighbor_alltoallw(const void *sendbuf, const int sendcounts[],
                            const MPI_Aint sdispls[],
                            const MPI_Datatype sendtypes[],
                            void *recvbuf, const int recvcounts[],
                            const MPI_Aint rdispls[],
                            const MPI_Datatype recvtypes[],
                            MPI_Comm comm, MPI_Request *request);
```

3.2.32 *MPI and threads⋆*

MPI can be, and often is together used with thread interfaces like OpenMP or `pthreads`. The idea is, for systems with shared-memory multi-core nodes that are interconnected by a communication network, to let cores on the shared memory node compute as threads, and let only a single or a few MPI processes on the shared-memory node perform communication with processes on other nodes with the MPI functionality. This is a two-level, heterogeneous, hierarchical, programming model. Processes can communicate with other processes with MPI, and threads inside the processes use a thread model to compute in parallel. The threads are the active entities within the processes, and such a two-level model therefore raises the question which threads can or are allowed to perform MPI operations?

MPI answers the question by defining the level of thread support that an MPI library implementation can provide. There are four defined levels of thread support. With `MPI_THREAD_SINGLE`, only a single thread is allowed to execute (essentially: threads parallel programming cannot be used at this level). With `MPI_THREAD_FUNNELED` threads can be used, but only a designated, single *main thread* can perform MPI calls. With `MPI_THREAD_SERIALIZED` all threads are allowed to perform MPI calls, but only one at a time, and it is the users responsibility to ensure that this is the case (by using critical sections and other means). With `MPI_THREAD_MULTIPLE`, all threads can perform MPI calls and may do so concurrently, in parallel. The levels of thread support are ordered, `MPI_THREAD_SINGLE` $<$ `MPI_THREAD_FUNNELED` $<$ `MPI_THREAD_SERIALIZED` $<$ `MPI_THREAD_MULTIPLE`.

Threads levels are controlled and queried by a special initialization function to be used instead of `MPI_Init`. With `MPI_Init_thread`, the user give a required thread level, and the function returns a thread level that can be supported. If the required thread level cannot be supported, the provided level is the highest provided thread level of the application. If the required thread level can be supported, the provided level returned is larger than or equal to the required level.

```
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided);
int MPI_Is_thread_main(int *flag);
int MPI_Query_thread(int *provided);
```

3.2.33 *MPI outlook*

A number of (important) aspects and part of the huge MPI standard were deliberately not treated in this bachelor lecture. These include provisions for input-output and communication with the external file system (MPI-IO), dynamic process management (spawning new MPI processes from an application, connecting running MPI processes), MPI attributes (a very useful mechanism

for library building by attaching information to MPI objects), and a few other things.

The MPI forum is currently preparing MPI 4.0 with a number of major additions to the standard. Some important additions are persistent collective operations (see Section 3.2.18), so-called partitioned (point-to-point) communication, additional support for portably adapting applications to specifics of system topologies (`MPI_Comm_split_type` is one function in this direction), so-called sessions, and provisioning for fault tolerant MPI programming.

3.3 EXERCISES

MPI exercises.

A

PROOFS AND SUPPLEMENTARY MATERIAL

A.1 THE MASTER THEOREM

The "Master Theorem", Theorem 9, gives closed form solutions for a range of divide-and-conquer recurrences of the following form, for constants $a \geq 1, b > 1, d \geq 0, e \geq 0$ (the omitted c is for the constants hidden behind the O) that very often occur in the analysis of (parallel) algorithms:

$$\begin{aligned} T(n) &= aT(n/b) + O(n^d \log^e n) \\ T(1) &= O(1) \end{aligned}$$

The Theorem states a closed-form solution in either of three forms:

1. $T(n) = O(n^d \log^e n)$ if $a/b^d < 1$ (equivalently $b^d/a > 1$),
2. $T(n) = O(n^d \log^{e+1} n)$ if $a/b^d = 1$ (equivalently $b^d/a = 1$), and
3. $T(n) = O(n^{\log_b a})$ if $a/b^d > 1$ (equivalently $b^d/a < 1$).

Let C be a constant at least as large as the leading constant in either of $O(1)$ or $O(n^d \log^e n)$. Then the recurrence takes the form

$$T(n) \leq aT(n/b) + C(n^d \log^e n)$$

First, assume $n = b^k$. With this, the recurrence takes the form

$$T(b^k) \leq aT(b^k/b) + C(b^{kd} k^e)$$

Expanding the recurrence for the first few values of k, $k = 1, 2, 3$ yields:

$$\begin{aligned} T(b) &\leq Ca + C(b^d 1^e) \\ T(b^2) &\leq Ca^2 + Ca(b^d 1^e) + C(b^{2d} 2^e) \\ T(b^3) &\leq Ca^3 + Ca^2(b^d 1^e) + Ca(b^{2d} 2^e) + C(b^{3d} 3^e) \end{aligned}$$

From this, we conjecture that

$$T(b^k) \leq Ca^k(1 + \sum_{i=1}^{k} \left(\frac{b^d}{a}\right)^i i^e)$$

The claim is easily verified by induction. The base case $T(1) \leq C$ holds, since the sum is void (no summands, per definition 0), by the choice of the constant C. Assuming the claim for $k-1$, this gives:

$$\begin{aligned} T(b^k) &\leq aT(b^k/b) + C(b^{kd}k^e) \\ &= aT(b^{k-1}) + C(b^{kd}k^e) \\ &= a(Ca^{k-1}(1 + \sum_{i=1}^{k-1} \left(\frac{b^d}{a}\right)^i i^e)) + C(b^{kd}k^e) \\ &= Ca^k(1 + \sum_{i=1}^{k-1} \left(\frac{b^d}{a}\right)^i i^e)) + Ca^k \left(\frac{b^d}{a}\right)^k k^e) \\ &= Ca^k(1 + \sum_{i=1}^{k} \left(\frac{b^d}{a}\right)^i i^e) \end{aligned}$$

We now distinguish three cases for bounding the sum from above.

1. $b^d/a > 1$:

$$\begin{aligned} \sum_{i=1}^{k} \left(\frac{b^d}{a}\right)^i i^e &\leq k^e \sum_{i=1}^{k} \left(\frac{b^d}{a}\right)^i \\ &= O(k^e \left(\frac{b^d}{a}\right)^{k+1}) \end{aligned}$$

   since the sum is a sum of quotients. Therefore

$$\begin{aligned} T(b^k) &= O(a^k \left(\frac{b^d}{a}\right)^{k+1} k^e) \\ &= O(b^{kd} \left(\frac{b^d}{a}\right) k^e) \\ &= O(n^d \log^e n) \end{aligned}$$

2. $b^d/a = 1$:

$$\begin{aligned} \sum_{i=1}^{k} \left(\frac{b^d}{a}\right)^i i^e &= \sum_{i=1}^{k} i^e \\ &\leq k^{e+1} \end{aligned}$$

   Therefore

$$\begin{aligned} T(b^k) &= O(a^k k^{e+1}) \\ &= O(b^{kd} k^{e+1}) \\ &= O(n^d \log^{e+1} n) \end{aligned}$$

3. $b^d/a < 1$: In this case, we use the fact that an exponential function f^i for $f > 1$ grows faster than the (any) polynomial i^e. We choose a constant $f, f > 1$ with $\left(\frac{b^d}{a}\right) f < 1$. Then, for some constant k', it holds that $i^e < f^i$ for $i \geq k'$.

$$\begin{aligned}
\sum_{i=1}^{k} \left(\frac{b^d}{a}\right)^i i^e &\leq \sum_{i=1}^{k'-1} \left(\frac{b^d}{a}\right)^i i^e + \sum_{i=k'}^{k} \left(\frac{b^d}{a}\right)^i i^e \\
&\leq \sum_{i=1}^{k'-1} \left(\frac{b^d}{a}\right)^i i^e + \sum_{i=k'}^{\infty} \left(\frac{b^d}{a}\right)^i f^i \\
&= \sum_{i=1}^{k'-1} \left(\frac{b^d}{a}\right)^i i^e + \sum_{i=k'}^{\infty} (\left(\frac{b^d}{a}\right) f)^i
\end{aligned}$$

The first sum is finite, and also the second sum which is a quotient series with a qoutient smaller than one is convergent (to a constant). Therefore

$$\begin{aligned}
T(b^k) &= O(a^k) \\
&= O(a^{\log_b n}) \\
&= O(n^{\log_b a})
\end{aligned}$$

When n is not a power of b, it holds that for some k, $b^{k-1} < n < b^k = n'$. Since $T(n)$ is monotone, we have for the three cases

1.

$$\begin{aligned}
T(n) \leq T(n') &= O(n'^d \log^e n') \\
&= O((n'/n)^d n^d \log^e((n'/n)n)) \\
&= O((n'/n)^d n^d (\log^e(n'/n) + \log^e n) \\
&= O(n^d \log^e n)
\end{aligned}$$

since $n'/n < b$ can be upper bounded by the constant b.

2.

$$\begin{aligned}
T(n) \leq T(n') &= O(n'^d \log^{e+1} n') \\
&= O(n^d \log^{e+1} n)
\end{aligned}$$

with the same calculation and argument as in Case 1.

3.

$$\begin{aligned}
T(n) \leq T(n') &= O(n'^{\log_b a}) \\
&= O((n'/n)^{\log_b a} n^{\log_b a}) \\
&= O(b^{\log_b a} n^{\log_b a}) \\
&= O(n^{\log_b a})
\end{aligned}$$

since $n'/n < b$ and also $b^{\log_b a}$ is constant.

The theorem therefore holds for any $n, n \geq 1$. The bounding arguments do not give any useful estimates of the constants incurred by the recurrence; but it can be shown that the bounds are asymptotically tight for recurrences of the form

$$\begin{aligned} T(n) &= aT(n/b) + \Theta(n^d \log^e n) \\ T(1) &= O(1) \end{aligned}$$

The calculations can be improved to give closed-form solutions also for negative values of $e, e < 0$.

BIBLIOGRAPHY

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987. Reprint of 1983 edition with corrections.

[3] M. Ajtai, J. Komlos, and E. Szemeredi. An $O(n \log n)$ sorting network. *Combinatorica*, pages 1–19, 1983.

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computer Conference*, pages 483–485, 1967.

[5] Richard J. Anderson and Gary L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859–868, 1991.

[6] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[7] Michael Axtmann and Peter Sanders. Robust massively parallel sorting. In *19th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 83–97, 2017.

[8] Michael Axtmann, Armin Wiebigke, and Peter Sanders. Lightweight MPI communicators with applications to perfectly balanced quicksort. In *32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[9] Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference*, pages 307–314, 1968.

[10] Arthur J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. Electronic Computers*, 15(5):757–763, 1966.

[11] Gianfranco Bilardi and Franco Preparata. Horizons of parallel computation. *Journal of Parallel and Distributed Computing*, 27:172–182, 1995.

[12] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[13] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37 (1):55–69, 1996.

[15] Jehoshua Bruck, Ching-Tien Ho, Schlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.

[16] Randall E. Bryant and David R. O'Hallaron. *Computer Systems. A Programmer's Perspective*. Prentice-Hall, second edition, 2011.

[17] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert A. van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[18] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2008.

[19] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4): 770–785, 1988.

[20] Richard Cole. Correction parallel merge sort. *SIAM Journal on Computing*, 22(6):1349, 1993.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[22] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors. *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers, 2003.

[23] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, 2005.

[24] V. Faber, Olaf M. Lubeck, and Andrew B. White Jr. Superlinear speedup of an efficient sequential algorithm is not possible. *Parallel Computing*, 3: 259–260, 1986.

[25] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1072.

[26] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–298, 1999.

[27] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1): 4:1–4:22, 2012.

[28] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979. With an addendum, 1991.

[29] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994. Second printing, 1995.

[30] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[31] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. *Using Advanced MPI*. MIT Press, 2014.

[32] David P. Helmbold and Charles E. McDowell. Modeling speedup(n) greater than n. *IEEE Transactions on Parallel and Distributed Systems*, 1(2): 250–256, 1990.

[33] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, revised 1st edition, 2012.

[34] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[35] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[36] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[37] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[38] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[39] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[40] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[41] Donald E. Knuth. *Searching and Sorting*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[42] William Kuszmaul and Charles E. Leiserson. Floors and ceilings in divide-and-conquer recurrences. In *4th Symposium on Simplicity in Algorithms (SOSA)*, pages 133–141, 2021.

[43] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Computer*, 28(9):690–691, 1979.

[44] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architechtures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1992.

[45] Charles E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[46] Timothy G. Mattson, Yun (Helen) He, and Alice E. Koniges. *The OpenMP Common Core. Making OpenMP Simple Again*. MIT Press, 2019.

[47] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[48] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1988.

[49] Daniel J. Sorin Milo M. K. Martin, Mark D. Hill. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55:78–89, 2012.

[50] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[51] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.1*, June 4th 2015. www.mpi-forum.org.

[52] David A. Patterson and John L. Hennessy. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, second edition, 1996.

[53] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann Publishers, third edition, 2004.

[54] Thomas Rauber and Gudula Rünger. *Parallel Programming for Multicore and Cluster Systems*. Springer, second edition, 2010.

[55] Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, 2013.

[56] Tim Roughgarden. *Algorithms Illuminated. Part 1: The Basics*. Soundlikeyourself Publishing, 2017.

[57] Tim Roughgarden, editor. *Beyond the worst-case analysis of algorithms*. Cambridge University Press, 2021.

[58] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. Efficient parallel random sampling – vectorized, cache-efficient, and online. *ACM Transactions on Mathematical Software*, 44(3):29:1–29:14, 2018.

[59] Bertil Schmidt, Jorge Gonzaález-Domínguez, Christian Hundt, and Moritz Schlarb. *Parallel Programming. Concepts and Practice*. Morgan Kaufmann Publishers, 2018.

[60] Robert Sedgewick. Quicksort with equal keys. *SIAM Journal on Computing*, 6(2):240–267, 1977.

[61] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978. Corrigendum *ibidem* **23** (79) 368.

[62] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.

[63] Christian Siebert and Jesper Larsson Träff. Perfectly load-balanced, stable, synchronization-free parallel merge. *Parallel Processing Letters*, 24(1), 2014.

[64] Marc Snir. On parallel searching. *SIAM Journal on Computing*, 14(3): 688–708, 1985.

[65] Marc Snir. Depth-size trade-offs for parallel prefix computation. *Journal of Algorithms*, 7(2):185–201, 1986.

[66] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[67] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Pearson Prentice-Hall, 5th edition, 2011.

[68] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing ans spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[69] Jesper Larsson Träff. Parallel quicksort without pairwise element exchange. arXiv:1804.07494, 2018.

[70] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[71] Robert A. van de Geijn and Jerrell Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency and Computation: Practice and Experience*, 9(4):255–274, 1997.

[72] Ruud van der Pas, Eric Strotzer, and Christian Terboven. *Using OpenMP – The Next Step*. MIT Press, 2017.

[73] Bruce Wagar. Hyperquicksort – a fast sorting algorithm for hypercubes. In *Hypercube Multiprocessors*, pages 292–299. SIAM Press, 1987.

[74] Samuel Williams, Andrew Waterman, and David A. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[75] Haikun Zhu, Chung-Kuan Cheng, and Ronald L. Graham. On the construction of zero-deficiency parallel prefix circuits with minimum depth. *ACM Transactions on Design Automation of Electronic Systystems*, 11(2):387–409, 2006.

INDEX