Chapter 5

Basics of parallelization

Before actually engaging in parallel programming it is vital to know about some fundamental rules in parallelization. This pertains to the available parallelization options and, even more importantly, to performance limitations. It is one of the most common misconceptions that the more "hardware" is put into executing a parallel program, the faster it will run. Billions of CPU hours are wasted every year because supercomputer users have no idea about the limitations of parallel execution.

In this chapter we will first identify and categorize the most common strategies for parallelization, and then investigate parallelism on a theoretical level: Simple mathematical models will be derived that allow insight into the factors that hamper parallel performance. Although the applicability and predictive power of such models is limited, they provide unique insights that are largely independent of concrete parallel programming paradigms. Practical programming standards for writing parallel programs will be introduced in the subsequent chapters.

5.1 Why parallelize?

With all the different kinds of parallel hardware that exists, from massively parallel supercomputers down to multicore laptops, parallelism seems to be a ubiquitous phenomenon. However, many scientific users may even today not be required to actually write parallel programs, because a single core is sufficient to fulfill their demands. If such demands outgrow the single core's capabilities, they can do so for two quite distinct reasons:

- A single core may be too slow to perform the required task(s) in a "tolerable" amount of time. The definition of "tolerable" certainly varies, but "overnight" is often a reasonable estimate. Depending on the requirements, "over lunch" or "duration of a PhD thesis" may also be valid.

- The memory requirements cannot be met by the amount of main memory which is available on a single system, because larger problems (with higher resolution, more physics, more particles, etc.) need to be solved.

The first problem is likely to occur more often in the future because of the irreversible multicore transition. For a long time, before the advent of parallel computers, the second problem was tackled by so-called *out-of-core* techniques, tailoring the algorithm

so that large parts of the data set could be held on mass storage and loaded on demand with a (hopefully) minor impact on performance. However, the chasm between peak performance and available I/O bandwidth (and latency) is bound to grow even faster than the DRAM gap, and it is questionable whether out-of-core can play a major role for serial computing in the future. High-speed I/O resources in parallel computers are today mostly available in the form of parallel file systems, which unfold their superior performance only if used with parallel data streams from different sources.

Of course, the reason for "going parallel" may strongly influence the chosen method of parallelization. The following section provides an overview on the latter.

5.2 Parallelism

Writing a parallel program must always start by identifying the parallelism inherent in the algorithm at hand. Different variants of parallelism induce different methods of parallelization. This section can only give a coarse summary on available parallelization methods, but it should enable the reader to consult more advanced literature on the topic. Mattson et al. [S6] have given a comprehensive overview on parallel programming patterns. We will restrict ourselves to methods for exploiting parallelism using multiple cores or compute nodes. The fine-grained concurrency implemented with superscalar processors and SIMD capabilities has been introduced in Chapters 1 and 2.

5.2.1 Data parallelism

Many problems in scientific computing involve processing of large quantities of data stored on a computer. If this manipulation can be performed in parallel, i.e., by multiple processors working on different parts of the data, we speak of *data parallelism*. As a matter of fact, this is the dominant parallelization concept in scientific computing on MIMD-type computers. It also goes under the name of *SPMD* (Single Program Multiple Data), as usually the same code is executed on all processors, with independent instruction pointers. It is thus not to be confused with SIMD parallelism.

Example: Medium-grained loop parallelism

Processing of array data by loops or loop nests is a central component in most scientific codes. A typical example are linear algebra operations on vectors or matrices, as implemented in the standard BLAS library [N50]. Often the computations performed on individual array elements are independent of each other and are hence typical candidates for parallel execution by several processors in shared memory (see Figure 5.1). The reason why this variant of parallel computing is often called "medium-grained" is that the distribution of work across processors is flexible and easily changeable down to the single data element: In contrast to what is shown in

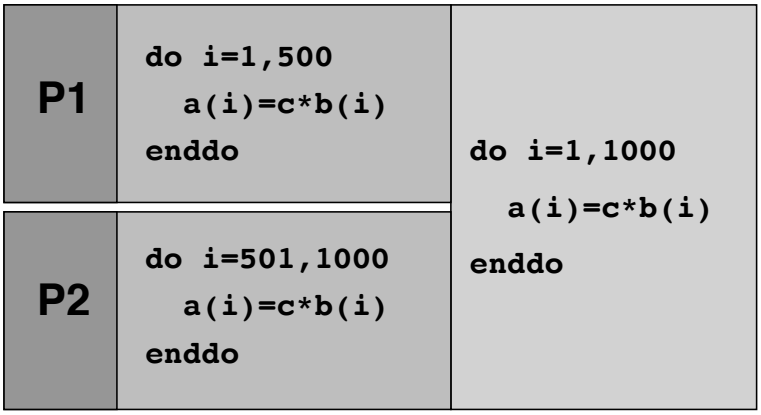

Figure 5.1: An example for medium-grained parallelism: The iterations of a loop are distributed to two processors P1 and P2 (in shared memory) for concurrent execution.

Figure 5.1, one could choose an interleaved pattern where all odd-(even-)indexed elements are processed by P1 (P2).

OpenMP, a compiler extension based on directives and a simple API, supports, among other things, data parallelism on loops. See Chapter 6 for an introduction to OpenMP.

Example: Coarse-grained parallelism by domain decomposition

Simulations of physical processes (like, e.g., fluid flow, mechanical stress, quantum fields) often work with a simplified picture of reality in which a *computational domain*, e.g., some volume of a fluid, is represented as a *grid* that defines discrete positions for the physical quantities under consideration (the Jacobi algorithm as introduced in Section 3.3 is an example). Such grids are not necessarily Cartesian but are often adapted to the numerical constraints of the algorithms used. The goal of the simulation is usually the computation of observables on this grid. A straightforward way to distribute the work involved across workers, i.e., processors, is to assign a part of the grid to each worker. This is called *domain decomposition*. As an example consider a two-dimensional Jacobi solver, which updates physical variables on a $n \times n$ grid. Domain decomposition for N workers subdivides the computational domain into N subdomains. If, e.g., the grid is divided into strips along the y direction (index `k` in Listing 3.1), each worker performs a single sweep on its local strip, updating the array for time step T_1. On a shared-memory parallel computer, all grid sites in all domains can be updated before the processors have to synchronize at the end of the sweep. However, on a distributed-memory system, updating the boundary sites of one domain requires data from one or more adjacent domains. Therefore, before a domain update, all boundary values needed for the upcoming sweep must be communicated to the relevant neighboring domains. In order to store this data, each domain must be equipped with some extra grid points, the so-called *halo* or *ghost layers* (see Figure 5.2). After the exchange, each domain is ready for the next sweep. The whole parallel algorithm is completely equivalent to purely serial execution. Section 9.3 will show in detail how this algorithm can be implemented using MPI, the Message Passing Interface.

How exactly the subdomains should be formed out of the complete grid may be a

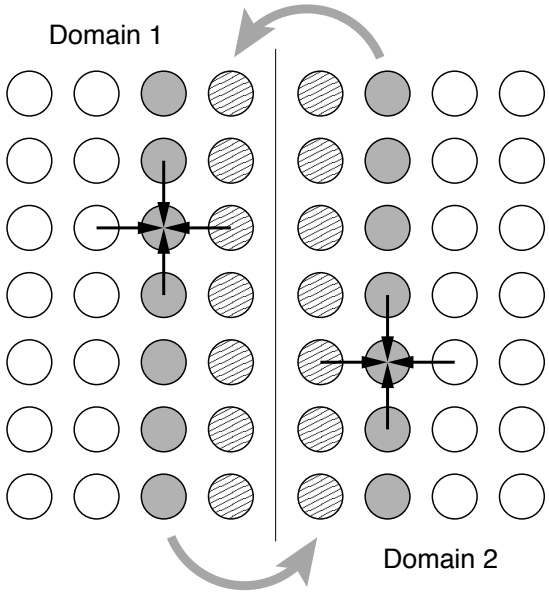

Figure 5.2: Using halo ("ghost") layers for communication across domain boundaries in a distributed-memory parallel Jacobi solver. After the local updates in each domain, the boundary layers (shaded) are copied to the halo of the neighboring domain (hatched).

difficult problem to solve, because several factors influence the optimal choice. First and foremost, the computational effort should be equal for all domains to prevent some workers from idling while others still update their own domains. This is called *load balancing* (see Figure 5.5 and Section 5.3.9). After load imbalance has been eliminated one should care about reducing the communication overhead. The data volume to be communicated is proportional to the overall area of the domain cuts. Comparing the two alternatives for 2D domain decomposition of an $n \times n$ grid to N workers in Figure 5.3, one arrives at a communication cost of $\mathcal{O}\left(n(N-1)\right)$ for stripe domains, whereas an optimal decomposition into square subdomains leads to a cost of $\mathcal{O}\left(2n(\sqrt{N}-1)\right)$. Hence, for large N the optimal decomposition has an advantage in communication cost of $\mathcal{O}\left(2/\sqrt{N}\right)$. Whether this difference is significant or not in reality depends on the problem size and other factors, of course. Communication must be counted as overhead that reduces a program's performance. In practice one should thus try to minimize boundary area as far as possible unless there are very good reasons to do otherwise. See Section 10.4.1 for a more general discussion.

Note that the calculation of communication overhead depends crucially on the *locality* of data dependencies, in the sense that communication cost grows linearly with the distance that has to be bridged in order to calculate observables at a certain site of the grid. For example, to get the first or second derivative of some quantity with respect to the coordinates, only a next-neighbor relation has to be implemented and the communication layers in Figure 5.3 have a width of one. For higher-order derivatives this changes significantly, and if there is some long-ranged interaction like a Coulomb potential (1/distance), the layers would encompass the complete computational domain, making communication dominant. In such a case, domain decomposition is usually not applicable and one has to revert to other parallelization strategies.

Domain decomposition has the attractive property that domain boundary area grows more slowly than volume if the problem size increases with N constant. There-

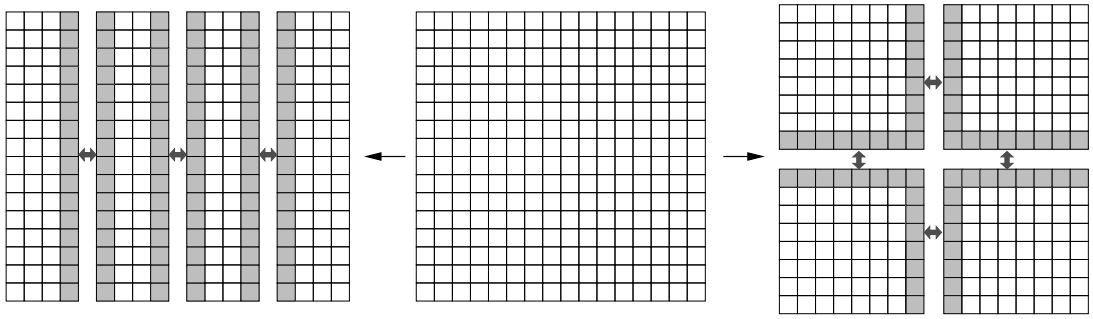

Figure 5.3: Domain decomposition of a two-dimensional Jacobi solver, which requires next-neighbor interactions. Cutting into stripes (left) is simple but incurs more communication than optimal decomposition (right). Shaded cells participate in network communication.

fore, one can sometimes alleviate communication bottlenecks just by choosing a larger problem size. The expected effects of scaling problem size and/or the number of workers with optimal domain decomposition in three dimensions will be discussed in Section 5.3 below.

The details about how the parallel Jacobi solver with domain decomposition can be implemented in reality will be revealed in Section 9.3, after the introduction of the Message Passing Interface (MPI).

Although the Jacobi method is quite inefficient in terms of convergence properties, it is very instructive and serves as a prototype for more advanced algorithms. Moreover, it lends itself to a host of scalar optimization techniques, some of which have been demonstrated in Section 3.4 in the context of matrix transposition (see also Problem 3.4 on page 92).

5.2.2 Functional parallelism

Sometimes the solution of a "big" numerical problem can be split into more or less disparate subtasks, which work together by data exchange and synchronization. In this case, the subtasks execute completely different code on different data items, which is why functional parallelism is also called *MPMD* (Multiple Program Multiple Data). This does not rule out, however, that each subtask could be executed in parallel by several processors in an SPMD fashion.

Functional parallelism bears pros and cons, mainly because of performance reasons. When different parts of the problem have different performance properties and hardware requirements, bottlenecks and load imbalance can easily arise. On the other hand, overlapping tasks that would otherwise be executed sequentially could accelerate execution considerably.

In the face of the increasing number of processor cores on a chip to spend on different tasks one may speculate whether we are experiencing the dawn of functional parallelism. In the following we briefly describe some important variants of functional parallelism. See also Section 11.1.2 for another example in the context of hybrid programming.

Example: Master-worker scheme

Reserving one compute element for administrative tasks while all others solve the actual problem is called the *master-worker* scheme. The master distributes work and collects results. A typical example is a parallel ray tracing program: A ray tracer computes a photorealistic image from a mathematical representation of a scene. For each pixel to be rendered, a "ray" is sent from the imaginary observer's eye into the scene, hits surfaces, gets reflected, etc., picking up color components. If all compute elements have a copy of the scene, all pixels are independent and can be computed in parallel. Due to efficiency concerns, the picture is usually divided into "work packages" (rows or tiles). Whenever a worker has finished a package, it requests a new one from the master, who keeps lists of finished and yet to be completed tiles. In case of a distributed-memory system, the finished tile must also be communicated over the network. See Refs. [A80, A81] for an implementation and a detailed performance analysis of parallel raytracing in a master-worker setting.

A drawback of the master-worker scheme is the potential communication and performance bottleneck that may appear with a single master when the number of workers is large.

Example: Functional decomposition

Multiphysics simulations are prominent applications for parallelization by functional decomposition. For instance, the airflow around a racing car could be simulated using a parallel CFD (Computational Fluid Dynamics) code. On the other hand, a parallel finite element simulation could describe the reaction of the flexible structures of the car body to the flow, according to their geometry and material properties. Both codes have to be coupled using an appropriate communication layer.

Although multiphysics codes are gaining popularity, there is often a big load balancing problem because it is hard in practice to dynamically shift resources between the different functional domains. See Section 5.3.9 for more information on load imbalance.

5.3 Parallel scalability

5.3.1 Factors that limit parallel execution

As shown in Section 5.2 above, parallelism may be exploited in a multitude of ways. Finding parallelism is not only a common problem in computing but also in many other areas like manufacturing, traffic flow and even business processes. In a very simplistic view, all execution units (workers, assembly lines, waiting queues, CPUs,...) execute their assigned work in exactly the same amount of time. Under such conditions, using N workers, a problem that takes a time T to be solved sequentially will now ideally take only T/N (see Figure 5.4). We call this a *speedup* of N.

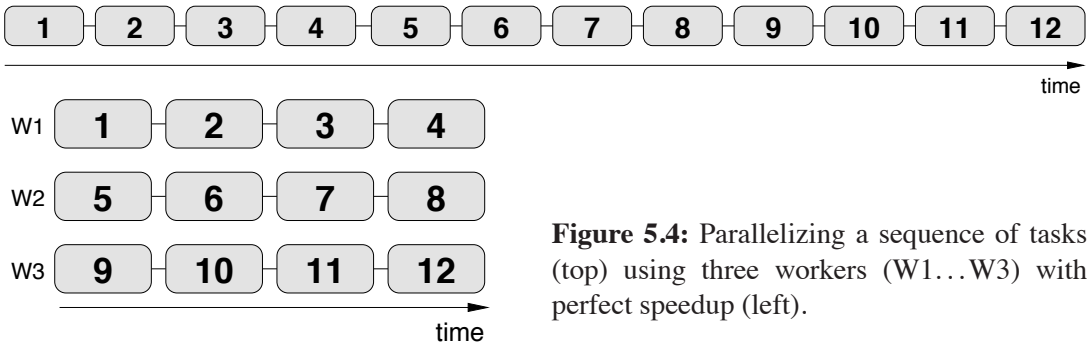

Figure 5.4: Parallelizing a sequence of tasks (top) using three workers (W1...W3) with perfect speedup (left).

Whatever parallelization scheme is chosen, this perfect picture will most probably not hold in reality. Some of the reasons for this have already been mentioned above: Not all workers might execute their tasks in the same amount of time because the problem was not (or could not) be partitioned into pieces with equal complexity. Hence, there are times when all but a few have nothing to do but wait for the latecomers to arrive (see Figure 5.5). This *load imbalance* hampers performance because some resources are underutilized. Moreover there might be shared resources like, e.g., tools that only exist once but are needed by all workers. This will effectively *serialize* part of the concurrent execution (Figure 5.6). And finally, the parallel workflow may require some communication between workers, adding overhead that would not be present in the serial case (Figure 5.7). All these effects can impose limits on speedup. How well a task can be parallelized is usually quantified by some *scalability* metric. Using such metrics, one can answer questions like:

- How much faster can a given problem be solved with N workers instead of one?
- How much more *work* can be done with N workers instead of one?
- What impact do the communication requirements of the parallel application have on performance and scalability?
- What fraction of the resources is actually used productively for solving the problem?

The following sections introduce the most important metrics and develops models that allow us to pinpoint the influence of some of the roadblocks just mentioned.

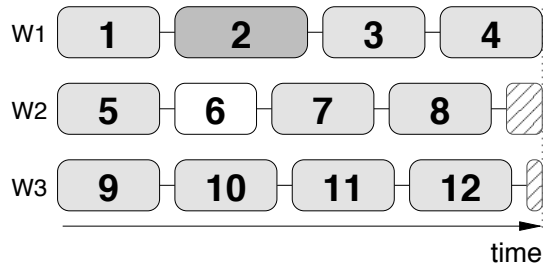

Figure 5.5: Some tasks executed by different workers at different speeds lead to *load imbalance*. Hatched regions indicate unused resources.

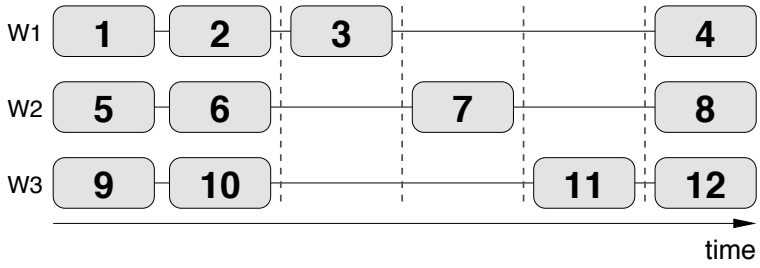

Figure 5.6: Parallelization with a bottleneck. Tasks 3, 7 and 11 cannot overlap with anything else across the dashed "barriers."

5.3.2 Scalability metrics

In order to be able to define scalability we first have to identify the basic measurements on which derived performance metrics are built. In a simple model, the overall problem size ("amount of work") shall be $s + p = 1$, where s is the serial (nonparallelizable) part and p is the perfectly parallelizable fraction. There can be many reasons for a nonvanishing serial part:

- *Algorithmic limitations*. Operations that cannot be done in parallel because of, e.g., mutual dependencies, can only be performed one after another, or even in a certain order.

- *Bottlenecks*. Shared resources are common in computer systems: Execution units in the core, shared paths to memory in multicore chips, I/O devices. Access to a shared resource *serializes* execution. Even if the algorithm itself could be performed completely in parallel, concurrency may be limited by bottlenecks.

- *Startup overhead*. Starting a parallel program, regardless of the technical details, takes time. Of course, system designs try to minimize startup time, especially in massively parallel systems, but there is always a nonvanishing serial part. If a parallel application's overall runtime is too short, startup will have a strong impact.

- *Communication*. Fully concurrent communication between different parts of a parallel system cannot be taken for granted, as was shown in Section 4.5. If solving a problem in parallel requires communication, some serialization is usually unavoidable. We will see in Section 5.3.6 below how to incorporate communication into scalability metrics in a more elaborate way than just adding a constant to the serial fraction.

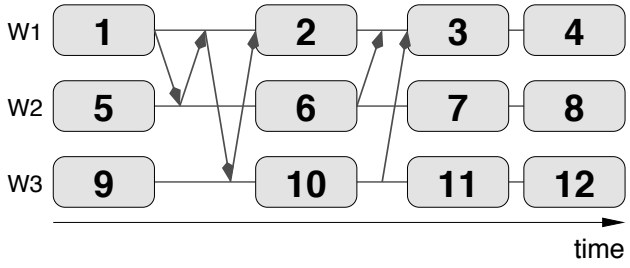

Figure 5.7: Communication processes (arrows represent messages) limit scalability if they cannot be overlapped with each other or with calculation.

First we assume a fixed problem, which is to be solved by N workers. We normalize the single-worker (serial) runtime

$$T_{\mathrm{f}}^{\mathrm{s}} = s + p \tag{5.1}$$

to one. Solving the same problem on N workers will require a runtime of

$$T_{\mathrm{f}}^{\mathrm{p}} = s + \frac{p}{N} \, . \tag{5.2}$$

This is called *strong scaling* because the amount of work stays constant no matter how many workers are used. Here the goal of parallelization is minimization of time to solution for a given problem.

If time to solution is not the primary objective because larger problem sizes (for which available memory is the limiting factor) are of interest, it is appropriate to scale the problem size with some power of N so that the total amount of work is $s + pN^{\alpha}$, where α is a positive but otherwise free parameter. Here we use the implicit assumption that the serial fraction s is a constant. We define the serial runtime for the scaled (variably-sized) problem as

$$T_{\mathrm{v}}^{\mathrm{s}} = s + pN^{\alpha} \, . \tag{5.3}$$

Consequently, the parallel runtime is

$$T_{\mathrm{v}}^{\mathrm{p}} = s + pN^{\alpha-1} \, . \tag{5.4}$$

The term *weak scaling* has been coined for this approach, although it is commonly used only for the special case $\alpha = 1$. One should add that other ways of scaling work with N are possible, but the N^{α} dependency will suffice for what we want to show further on.

We will see that different scalability metrics with different emphasis on what "performance" really means can lead to some counterintuitive results.

5.3.3 Simple scalability laws

In a simple ansatz, *application speedup* can be defined as the quotient of parallel and serial performance for fixed problem size. In the following we define "performance" as "work over time," unless otherwise noted. Serial performance for fixed problem size (work) $s + p$ is thus

$$P_{\mathrm{f}}^{\mathrm{s}} = \frac{s+p}{T_{\mathrm{f}}^{\mathrm{s}}} = 1 \, , \tag{5.5}$$

as expected. Parallel performance is in this case

$$P_{\mathrm{f}}^{\mathrm{p}} = \frac{s+p}{T_{\mathrm{f}}^{\mathrm{p}}(N)} = \frac{1}{s + \frac{1-s}{N}} \, , \tag{5.6}$$

and application speedup ("scalability") is

$$S_\mathrm{f} = \frac{P_\mathrm{f}^\mathrm{p}}{P_\mathrm{f}^\mathrm{s}} = \frac{1}{s + \frac{1-s}{N}} \quad \text{"Amdahl's Law"} \tag{5.7}$$

We have derived *Amdahl's Law*, which was first conceived by Gene Amdahl in 1967 [M45]. It limits application speedup for $N \to \infty$ to $1/s$. This well-known function answers the question "How much faster (in terms of runtime) does my application run when I put the same problem on N CPUs?" As one might imagine, the answer to this question depends heavily on how the term "work" is defined. If, in contrast to what has been done above, we define "work" as only the parallelizable part of the calculation (for which there may be sound reasons at first sight), the results for constant work are slightly different. Serial performance is

$$P_\mathrm{f}^{\mathrm{s}p} = \frac{p}{T_\mathrm{f}^\mathrm{s}} = p \ , \tag{5.8}$$

and parallel performance is

$$P_\mathrm{f}^{\mathrm{p}p} = \frac{p}{T_\mathrm{f}^\mathrm{p}(N)} = \frac{1-s}{s + \frac{1-s}{N}} \ . \tag{5.9}$$

Calculation of application speedup finally yields

$$S_\mathrm{f}^p = \frac{P_\mathrm{f}^{\mathrm{p}p}}{P_\mathrm{f}^{\mathrm{s}p}} = \frac{1}{s + \frac{1-s}{N}} \ , \tag{5.10}$$

which is Amdahl's Law again. Strikingly, $P_\mathrm{f}^{\mathrm{p}p}$ and $S_\mathrm{f}^p(N)$ are not identical any more. Although *scalability* does not change with this different notion of "work," *performance* does, and is a factor of p smaller.

In the case of *weak scaling* where workload grows with CPU count, the question to ask is "How much more work can my program do in a given amount of time when I put a larger problem on N CPUs?" Serial performance as defined above is again

$$P_\mathrm{v}^\mathrm{s} = \frac{s+p}{T_\mathrm{f}^\mathrm{s}} = 1 \ , \tag{5.11}$$

as $N = 1$. Based on (5.3) and (5.4), Parallel performance (work over time) is

$$P_\mathrm{v}^\mathrm{p} = \frac{s + pN^\alpha}{T_\mathrm{v}^\mathrm{p}(N)} = \frac{s + (1-s)N^\alpha}{s + (1-s)N^{\alpha-1}} = S_\mathrm{v} \ , \tag{5.12}$$

again identical to application speedup. In the special case $\alpha = 0$ (strong scaling) we recover Amdahl's Law. With $0 < \alpha < 1$, we get for large CPU counts

$$S_\mathrm{v} \xrightarrow{N \gg 1} \frac{s + (1-s)N^\alpha}{s} = 1 + \frac{p}{s} N^\alpha \ , \tag{5.13}$$

which is linear in N^α. As a result, weak scaling allows us to cross the Amdahl Barrier

and get unlimited performance, even for small α. In the ideal case $\alpha = 1$, (5.12) simplifies to

$$S_{\mathrm{v}}(\alpha = 1) = s + (1-s)N\ , \quad \text{“Gustafson's Law”} \tag{5.14}$$

and speedup is linear in N, even for small N. This is called *Gustafson's Law* [M46]. Keep in mind that the terms with N or N^α in the previous formulas always bear a prefactor that depends on the serial fraction s, thus a large serial fraction can lead to a very small slope.

As previously demonstrated with Amdahl scaling we will now shift our focus to the other definition of “work” that only includes the parallel fraction p. Serial performance is

$$P_{\mathrm{v}}^{sp} = p \tag{5.15}$$

and parallel performance is

$$P_{\mathrm{v}}^{pp} = \frac{pN^\alpha}{T_{\mathrm{v}}^{\mathrm{p}}(N)} = \frac{(1-s)N^\alpha}{s + (1-s)N^{\alpha-1}}\ , \tag{5.16}$$

which leads to an application speedup of

$$S_{\mathrm{v}}^{p} = \frac{P_{\mathrm{v}}^{pp}}{P_{\mathrm{v}}^{sp}} = \frac{N^\alpha}{s + (1-s)N^{\alpha-1}}\ . \tag{5.17}$$

Not surprisingly, speedup and performance are again not identical and differ by a factor of p. The important fact is that, in contrast to (5.14), for $\alpha = 1$ application speedup becomes purely linear in N *with a slope of one*. So even though the overall work to be done (serial and parallel part) has not changed, scalability as defined in (5.17) makes us believe that suddenly all is well and the application scales perfectly. If some performance metric is applied that is only relevant in the parallel part of the program (e.g., “number of lattice site updates” instead of “CPU cycles”), this mistake can easily go unnoticed, and CPU power is wasted (see next section).

5.3.4 Parallel efficiency

In the light of the considerations about scalability, one other point of interest is the question how effectively a given resource, i.e., CPU computational power, can be used in a parallel program (in the following we assume that the serial part of the program is executed on one single worker while all others have to wait). Usually, parallel efficiency is then defined as

$$\varepsilon = \frac{\text{performance on } N \text{ CPUs}}{N\times \text{ performance on one CPU}} = \frac{\text{speedup}}{N}. \tag{5.18}$$

We will only consider weak scaling, since the limit $\alpha \to 0$ will always recover the Amdahl case. In the case where “work” is defined as $s + pN^\alpha$, we get

$$\varepsilon = \frac{S_{\mathrm{v}}}{N} = \frac{sN^{-\alpha} + (1-s)}{sN^{1-\alpha} + (1-s)}\ . \tag{5.19}$$

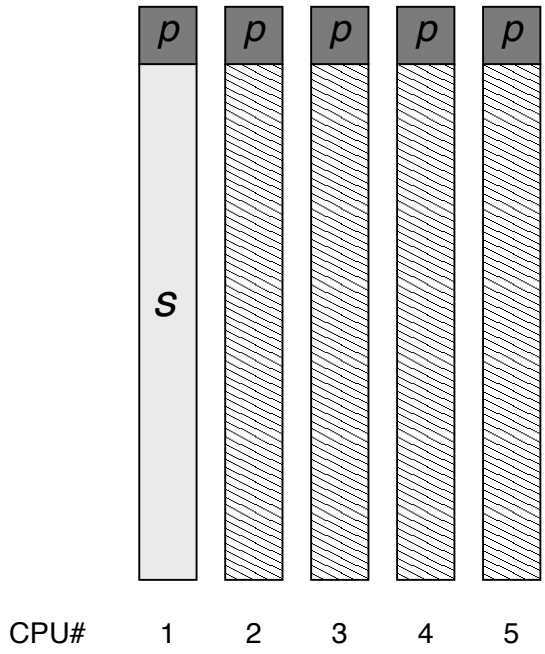

Figure 5.8: Weak scaling with an inappropriate definition of "work" that includes only the parallelizable part. Although "work over time" scales perfectly with CPU count, i.e., $\varepsilon_p = 1$, most of the resources (hatched boxes) are unused because $s \gg p$.

For $\alpha = 0$ this yields $1/(sN + (1-s))$, which is the expected ratio for the Amdahl case and approaches zero with large N. For $\alpha = 1$ we get $s/N + (1-s)$, which is also correct because the more CPUs are used the more CPU cycles are wasted, and, starting from $\varepsilon = s + p = 1$ for $N = 1$, efficiency reaches a limit of $1 - s = p$ for large N. Weak scaling enables us to use at least a certain fraction of CPU power, even when the CPU count is very large. Wasted CPU time grows linearly with N, though, but this issue is clearly visible with the definitions used.

Results change completely when our other definition of "work" (pN^α) is applied. Here,

$$\varepsilon_p = \frac{S_\mathrm{v}^p}{N} = \frac{N^{\alpha-1}}{s + (1-s)N^{\alpha-1}} \ . \tag{5.20}$$

For $\alpha = 1$ we now get $\varepsilon_p = 1$, which should mean perfect efficiency. We are fooled into believing that no cycles are wasted with weak scaling, although if s is large most of the CPU power is unused. A simple example will exemplify this danger: Assume that some code performs floating-point operations only within its parallelized part, which takes about 10% of execution time in the serial case. Using weak scaling with $\alpha = 1$, one could now report MFlops/sec performance numbers vs. CPU count (see Figure 5.8). Although all processors except one are idle 90% of their time, the MFlops/sec rate is a factor of N higher when using N CPUs. Performance behavior that is presented in this way should always raise suspicion.

5.3.5 Serial performance versus strong scalability

In order to check whether some performance model is appropriate for the code at hand, one should measure scalability for some processor numbers and fix the free model parameters by least-squares fitting. Figure 5.9 shows an example where the

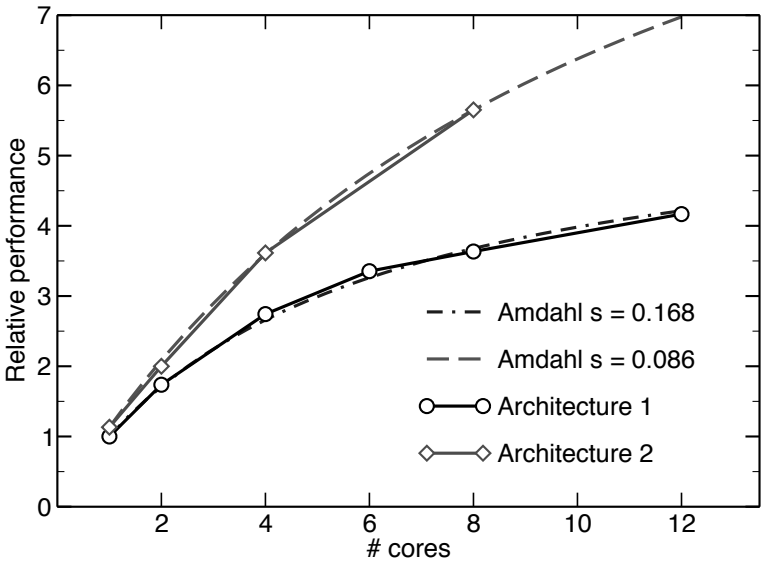

Figure 5.9: Performance of a benchmark code versus the number of processors (strong scaling) on two different architectures. Although the single-thread performance is nearly identical on both machines, the serial fraction s is much smaller on architecture 2, leading to superior strong scalability.

same code was run in a strong scaling scenario on two different parallel architectures. The measured performance data was normalized to the single-core case on architecture 1, and the serial fraction s was determined by a least-squares fit to Amdahl's Law (5.7).

Judging from the small performance difference on a single core it is quite surprising that architecture 2 shows such a large advantage in scalability, with only about half the serial fraction. This behavior can be explained by the fact that the parallel part of the calculation is purely compute-bound, whereas the serial part is limited by memory bandwidth. Although the peak performance per core is identical on both systems, architecture 2 has a much wider path to memory. As the number of workers increases, performance ceases to be governed by computational speed and the memory-bound serial fraction starts to dominate. Hence, the significant advantage in scalability for architecture 2. This example shows that it is vital to not only be aware of the existence of a nonparallelizable part but also of its specific demands on the architecture under consideration: One should not infer the scalability behavior on one architecture from data obtained on the other. One may also argue that parallel computers that are targeted towards strong scaling should have a heterogeneous architecture, with some of the hardware dedicated exclusively to executing serial code as fast as possible. This pertains to multicore chips as well [R39, M47, M48].

In view of optimization, strong scaling has the unfortunate side effect that using more and more processors leads to performance being governed by code that was not subject to parallelization efforts (this is one variant of the "law of diminishing returns"). If standard scalar optimizations like those shown in Chapters 2 and 3 can be applied to the serial part of an application, they can thus truly improve strong scalability, although serial performance will hardly change. The question whether one should invest scalar optimization effort into the serial or the parallel part of an application seems to be answered by this observation. However, one must keep in mind that *performance*, and not scalability is the relevant metric; fortunately, Amdahl's Law can provide an approximate guideline. Assuming that the serial part can

be accelerated by a factor of $\xi > 1$, parallel performance (see (5.6)) becomes

$$P_{\mathrm{f}}^{s,\xi} = \frac{1}{\frac{s}{\xi} + \frac{1-s}{N}} \; . \tag{5.21}$$

On the other hand, if only the parallel part gets optimized (by the same factor) we get

$$P_{\mathrm{f}}^{p,\xi} = \frac{1}{s + \frac{1-s}{\xi N}} \; . \tag{5.22}$$

The ratio of those two expressions determines the crossover point, i.e., the number of workers at which optimizing the serial part pays off more:

$$\frac{P_{\mathrm{f}}^{s,\xi}}{P_{\mathrm{f}}^{p,\xi}} = \frac{\xi s + \frac{1-s}{N}}{s + \xi \frac{1-s}{N}} \geq 1 \quad \Longrightarrow \quad N \geq \frac{1}{s} - 1 \; . \tag{5.23}$$

This result does not depend on ξ, and it is exactly the number of workers where the speedup is half the maximum asymptotic value predicted by Amdahl's Law. If $s \ll 1$, parallel efficiency $\varepsilon = (1-s)^{-1}/2$ is already close to 0.5 at this point, and it would not make sense to enlarge N even further anyway. Thus, one should try to optimize the parallelizable part first, unless the code is used in a region of very bad parallel efficiency (probably because the main reason for going parallel was lack of memory).

Note, however, that in reality it will not be possible to achieve the same speedup ξ for both the serial and the parallel part, so the crossover point will be shifted accordingly. In the example above (see Figure 5.9) the parallel part is dominated by matrix-matrix multiplications, which run close to peak performance anyway. Accelerating the sequential part is hence the only option to improve performance at a given N.

5.3.6 Refined performance models

There are situations where Amdahl's and Gustafson's Laws are not appropriate because the underlying model does not encompass components like communication, load imbalance, parallel startup overhead, etc. As an example for possible refinements we will include a basic communication model. For simplicity we presuppose that communication cannot be overlapped with computation (see Figure 5.7), an assumption that is actually true for many parallel architectures. In a parallel calculation, communication must thus be accounted for as a correction term in parallel runtime (5.4):

$$T_{\mathrm{v}}^{\mathrm{pc}} = s + pN^{\alpha-1} + c_\alpha(N) \; . \tag{5.24}$$

The communication overhead $c_\alpha(N)$ must not be included into the definition of "work" that is used to derive performance as it emerges from processes that are solely a result of the parallelization. Parallel speedup is then

$$S_{\mathrm{v}}^{\mathrm{c}} = \frac{s + pN^\alpha}{T_{\mathrm{v}}^{\mathrm{pc}}(N)} = \frac{s + (1-s)N^\alpha}{s + (1-s)N^{\alpha-1} + c_\alpha(N)} \; . \tag{5.25}$$

There are many possibilities for the functional dependence $c_\alpha(N)$; it may be some simple function, or it may not be possible to write it in closed form at all. Furthermore we assume that the amount of communication is the same for all workers. As with processor-memory communication, the time a message transfer requires is the sum of the latency λ for setting up the communication and a "streaming" part $\kappa = n/B$, where n is the message size and B is the bandwidth (see Section 4.5.1 for real-world examples). A few special cases are described below:

- $\alpha = 0$, *blocking network:* If the communication network has a "bus-like" structure (see Section 4.5.2), i.e., only one message can be in flight at any time, and the communication overhead per CPU is independent of N then $c_\alpha(N) = (\kappa+\lambda)N$. Thus,

$$S_{\mathrm{v}}^{\mathrm{c}} = \frac{1}{s + \frac{1-s}{N} + (\kappa+\lambda)N} \xrightarrow{N\gg 1} \frac{1}{(\kappa+\lambda)N} , \tag{5.26}$$

  i.e., performance is dominated by communication and even goes to zero for large CPU numbers. This is a very common situation as it also applies to the presence of shared resources like memory paths, I/O devices and even on-chip arithmetic units.

- $\alpha = 0$, *nonblocking network, constant communication cost:* If the communication network can sustain $N/2$ concurrent messages with no collisions (see Section 4.5.3), and message size is independent of N, then $c_\alpha(N) = \kappa+\lambda$ and

$$S_{\mathrm{v}}^{\mathrm{c}} = \frac{1}{s + \frac{1-s}{N} + \kappa + \lambda} \xrightarrow{N\gg 1} \frac{1}{s+\kappa+\lambda} . \tag{5.27}$$

  Here the situation is quite similar to the Amdahl case and performance will saturate at a lower value than without communication.

- $\alpha = 0$, *nonblocking network, domain decomposition with ghost layer communication:* In this case communication overhead decreases with N for strong scaling, e.g., like $c_\alpha(N) = \kappa N^{-\beta} + \lambda$. For any $\beta > 0$ performance at large N will be dominated by s and the latency:

$$S_{\mathrm{v}}^{\mathrm{c}} = \frac{1}{s + \frac{1-s}{N} + \kappa N^{-\beta} + \lambda} \xrightarrow{N\gg 1} \frac{1}{s+\lambda} . \tag{5.28}$$

  This arises, e.g., when domain decomposition (see page 117) is employed on a computational domain along all three coordinate axes. In this case $\beta = 2/3$.

- $\alpha = 1$, *nonblocking network, domain decomposition with ghost layer communication:* Finally, when the problem size grows linearly with N, one may end up in a situation where communication per CPU stays independent of N. As this is weak scaling, the numerator leads to linear scalability with an overall performance penalty (prefactor):

$$S_{\mathrm{v}}^{\mathrm{c}} = \frac{s+pN}{s+p+\kappa+\lambda} \xrightarrow{N\gg 1} \frac{s+(1-s)N}{1+\kappa+\lambda} . \tag{5.29}$$

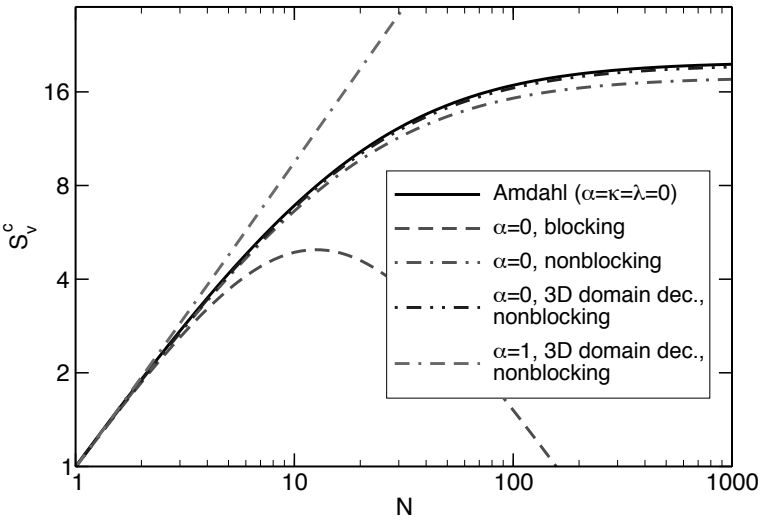

Figure 5.10: Predicted parallel scalability for different models at $s = 0.05$. In general, $\kappa = 0.005$ and $\lambda = 0.001$ except for the Amdahl case, which is shown for reference.

Figure 5.10 illustrates the four cases at $\kappa = 0.005$, $\lambda = 0.001$, and $s = 0.05$, and compares with Amdahl's Law. Note that the simplified models we have covered in this section are far from accurate for many applications. As an example, consider an application that is large enough to not fit into a single processor's cache but small enough to fit into the aggregate caches of N_c CPUs. Performance limitations like serial fractions, communication, etc., could then be ameliorated, or even overcompensated, so that $S_v^c(N) > N$ for some range of N. This is called *superlinear speedup* and can only occur if problem size grows more slowly than N, i.e., at $\alpha < 1$. See also Section 6.2 and Problem 7.2.

One must also keep in mind that those models are valid only for $N > 1$ as there is usually no communication in the serial case. A fitting procedure that tries to fix the parameters for some specific code should thus ignore the point $N = 1$.

When running application codes on parallel computers, there is often the question about the "optimal" choice for N. From the user's perspective, N should be as large as possible, minimizing time to solution. This would generally be a waste of resources, however, because parallel efficiency is low near the performance maximum. See Problem 5.2 for a possible cost model that aims to resolve this conflict. Note that if the main reason for parallelization is the need for large memory, low efficiency may be acceptable nevertheless.

5.3.7 Choosing the right scaling baseline

Today's high performance computers are all massively parallel. In the previous sections we have described the different ways a parallel computer can be built: There are multicore chips, sitting in multisocket shared-memory nodes, which are again connected by multilevel networks. Hence, a parallel system always comprises a number of *hierarchy levels*. Scaling a parallel code from one to many CPUs can lead to false conclusions if the hierarchical structure is not taken into account.

Figure 5.11 shows an example for strong scaling of an application on a system with four processors per node. Assuming that the code follows a communication

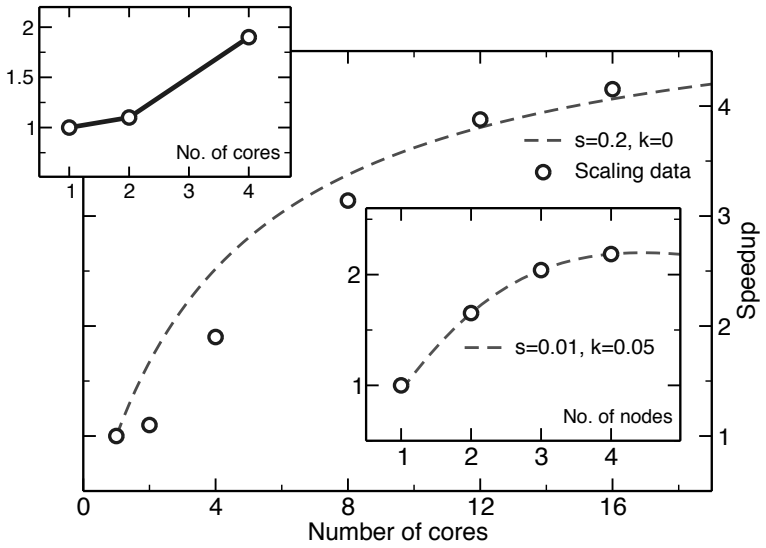

Figure 5.11: Speedup versus number of CPUs used for a hypothetical code on a hierarchical system with four CPUs per node. Depending on the chosen scaling baseline, fits to the model (5.26) can lead to vastly different results. Right inset: Scalability across nodes. Left inset: Scalability inside one node.

model as in (5.26), a least-squares fitting was used to determine the serial fraction s and the communication time per process, $k = \kappa + \lambda$ (main panel). As there is only a speedup of ≈ 4 at 16 cores, $s = 0.2$ does seem plausible, and communication apparently plays no significant role. However, the quality of the fit is mediocre, especially for small numbers of cores. Thus one may arrive at the conclusion that scalability inside a node is governed by factors different from serial fraction and communication, and that (5.26) is not valid for all numbers of cores. The right inset in Figure 5.11 shows scalability data normalized to the 4-core (one-node) performance, i.e., we have chosen a different *scaling baseline*. Obviously the model (5.26) is well suited for this situation and yields completely different fitting parameters, which indicate that communication plays a major role ($s = 0.01$, $k = 0.05$). The left inset in Figure 5.11 extracts the intranode behavior only; the data is typical for a memory-bound situation. On a node architecture as in Figure 4.4, using two cores on the same socket may lead to a bandwidth bottleneck, which is evident from the small speedup when going from one to two cores. Using the second socket as well gives a strong performance boost, however.

In conclusion, scalability on a given parallel architecture should always be reported in relation to a relevant scaling baseline. On typical compute clusters, where shared-memory multiprocessor nodes are coupled via a high-performance interconnect, this means that intranode and internode scaling behavior should be strictly separated. This principle also applies to other hierarchy levels like in, e.g., modern multisocket multicore shared memory systems (see Section 4.2), and even to the the complex cache group and thread structure in current multicore processors (see Section 1.4).

5.3.8 Case study: Can slower processors compute faster?

It is often stated that, all else being equal, using a slower processor in a parallel computer (or a less-optimized single processor code) improves scalability of applica-

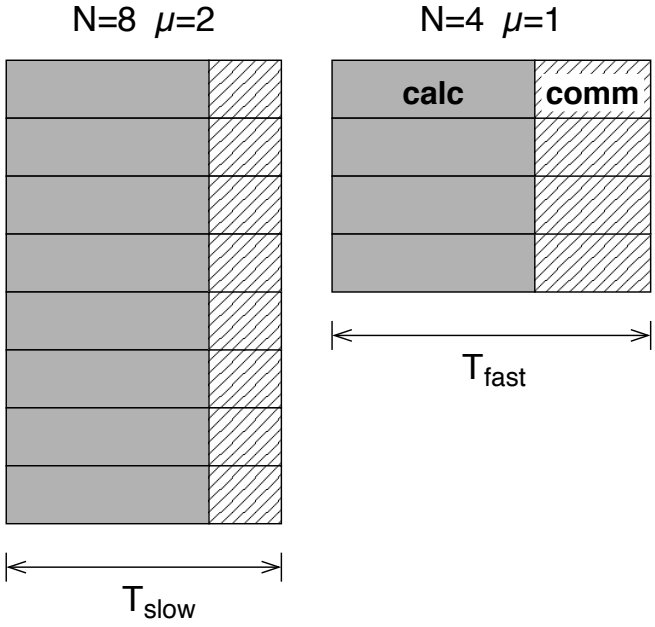

Figure 5.12: Solving the same problem on $2N$ slow CPUs (left) instead of N fast CPUs (right) may speed up time to solution if communication overhead per CPU goes down with rising N. $\mu = 2$ is the performance ratio between fast and slow CPU.

tions because the adverse effects of communication overhead are reduced in relation to "useful" computation. A "truly scalable" computer may thus be built from slow CPUs and a reasonable network. In order to find the truth behind this concept we will establish a performance model for "slow" computers. In this context, "slow" shall mean that the baseline serial execution time is $\mu \geq 1$ instead of 1, i.e., CPU speed is quantified as μ^{-1}. Figure 5.12 demonstrates how "slow computing" may work. If the same problem is solved by μN slow instead of N fast processors, overall runtime may be shortened if communication overhead per CPU goes down as well. How strong this effect is and whether it makes sense to build a parallel computer based on it remains to be seen. Interesting questions to ask are:

1. Does it make sense to use μN "slow" processors instead of N standard CPUs in order to get better overall performance?

2. What conditions must be fulfilled by communication overhead to achieve better performance with slow processors?

3. Does the concept work in strong and weak scaling scenarios alike?

4. What is the expected performance gain?

5. Can a "slow" machine deliver more performance than a machine with standard processors within the same power envelope?

For the last question, in the absence of communication overhead we already know the answer because the situation is very similar to the multicore transition whose consequences were described in Section 1.4. The additional inefficiencies connected with communication might change those results significantly, however. More importantly, the CPU cores only contribute a part of the whole system's power consumption; a

power/performance model that merely comprises the CPU components is necessarily incomplete and will be of very limited use. Hence, we will concentrate on the other questions. We already expect from Figure 5.12 that a sensible performance model must include a realistic communication component.

Strong scaling

Assuming a fixed problem size and a generic communication model as in (5.24), the speedup for the slow computer is

$$S_\mu(N) = \frac{1}{s+(1-s)/N+c(N)/\mu} \; . \tag{5.30}$$

For $\mu > 1$ and $N > 1$ this is clearly larger than $S_{\mu=1}(N)$ whenever $c(N) \neq 0$: A machine with slow processors "scales better," but only if there is communication overhead.

Of course, scalability alone is no appropriate measure for *application performance* since it relates parallel performance to serial performance on one CPU of the same speed μ^{-1}. We want to compare the absolute performance advantage of μN slow CPUs over N *standard* processors:

$$A^{\mathrm{s}}_\mu(N) := \frac{S_\mu(\mu N)}{\mu S_{\mu=1}(N)} = \frac{s+(1-s)/N+c(N)}{\mu s+(1-s)/N+c(\mu N)} \tag{5.31}$$

If $\mu > 1$, this is greater than one if

$$c(\mu N) - c(N) < -s(\mu-1) \; . \tag{5.32}$$

Hence, if we assume that the condition should hold for all μ, $c(N)$ must be a decreasing function of N with a minimum slope. At $s = 0$ a negative slope is sufficient, i.e., communication overhead must decrease if N is increased. This result was expected from the simple observation in Figure 5.12.

In order to estimate the achievable gains we look at the special case of Cartesian domain decomposition on a nonblocking network as in (5.28). The advantage function is then

$$A^{\mathrm{s}}_\mu(N) := \frac{S_\mu(\mu N)}{\mu S_{\mu=1}(N)} = \frac{s+(1-s)/N+\lambda+\kappa N^{-\beta}}{\mu s+(1-s)/N+\lambda+\kappa(N\mu)^{-\beta}} \tag{5.33}$$

We can distinguish several cases here:

- $\kappa = 0$: With no communication bandwidth overhead,

$$A^{\mathrm{s}}_\mu(N) = \frac{s+(1-s)/N+\lambda}{\mu s+(1-s)/N+\lambda} \xrightarrow{N\to\infty} \frac{s+\lambda}{\mu s+\lambda} \; , \tag{5.34}$$

  which is always smaller than one. In this limit there is no performance to be gained with slow CPUs, and the pure power advantage from using many slow processors is even partly neutralized.

- $\kappa \neq 0, \lambda = 0$: To leading order in $N^{-\beta}$ (5.33) can be approximated as

$$A_\mu^{\mathrm{s}}(N) = \frac{1}{\mu s}\left(s + \kappa N^{-\beta}(1-\mu^{-\beta})\right) + \mathscr{O}(N^{-2\beta}) \xrightarrow{N\to\infty} \frac{1}{\mu}\,. \tag{5.35}$$

  Evidently, $s \neq 0$ and $\kappa \neq 0$ lead to opposite effects: For very large N, the serial fraction dominates and $A_\mu(N) < 1$. At smaller N, there may be a chance to get $A_\mu^{\mathrm{s}}(N) > 1$ if s is not too large.

- $s = 0$: In a strong scaling scenario, this case is somewhat unrealistic. However, it is the limit in which a machine with slow CPUs performs best: The positive effect of the κ-dependent terms, i.e., the reduction of communication bandwidth overhead with increasing N, is large, especially if the latency is low:

$$A_\mu^{\mathrm{s}}(N) = \frac{N^{-1} + \lambda + \kappa N^{-\beta}}{N^{-1} + \lambda + \kappa (N\mu)^{-\beta}} \xrightarrow{N\to\infty,\lambda>0} 1^{+} \tag{5.36}$$

  In the generic case $\kappa \neq 0$, $\lambda \neq 0$ and $0 < \beta < 1$ this function approaches 1 from above as $N \to \infty$ and has a maximum at $N_{\mathrm{MA}} = (1-\beta)/\beta\lambda$. Hence, the largest possible advantage is

$$A_\mu^{\mathrm{s,max}} = A_\mu^{\mathrm{s}}(N_{\mathrm{MA}}) = \frac{1 + \kappa\beta^\beta X^{\beta-1}}{1 + \kappa\beta^\beta X^{\beta-1}\mu^{-\beta}}, \quad \text{with} \quad X = \frac{\lambda}{1-\beta}\,. \tag{5.37}$$

  This approaches μ^β as $\lambda \to 0$. At the time of writing, typical "scalable" HPC systems with slow CPUs operate at $2 \lesssim \mu \lesssim 4$, so for optimal 3D domain decomposition along all coordinate axes ($\beta = 2/3$) the maximum theoretical advantage is $1.5 \lesssim A^{\mathrm{s,max}} \lesssim 2.5$.

It must be emphasized that the assumption $s = 0$ cannot really be upheld for strong scaling because its influence will dominate scalability in the same way as network latency for large N. Thus, we must conclude that the region of applicability for "slow" machines is very narrow in strong scaling scenarios.

Even if it may seem technically feasible to take the limit of very large μ and achieve even grater gains, it must be stressed that applications must provide sufficient parallelism to profit from more and more processors. This does not only refer to Amdahl's Law, which predicts that the influence of the serial fraction s, however small it may be initially, will dominate as N increases (as shown in (5.34) and (5.35)); in most cases there is some "granularity" inherent to the model used to implement the numerics (e.g., number of fluid cells, number of particles, etc.), which strictly limits the number of workers.

Strictly weak scaling

We choose Gustafson scaling (work $\propto N$) and a generic communication model. The speedup function for the slow computer is

$$S_\mu(N) = \frac{[s + (1-s)N]/(\mu + c(N))}{\mu^{-1}} = \frac{s + (1-s)N}{1 + c(N)/\mu}\,, \tag{5.38}$$

which leads to the advantage function

$$A_\mu^{\mathrm{w}}(N) := \frac{S_\mu(\mu N)}{\mu S_{\mu=1}(N)} = \frac{[s+(1-s)\mu N]\,[1+c(N)]}{[s+(1-s)N]\,[\mu+c(\mu N)]} \,. \tag{5.39}$$

Neglecting the serial part s, this is greater than one if $c(N) > c(\mu N)/\mu$: Communication overhead may even increase with N, but this increase must be slower than linear.

For a more quantitative analysis we turn again to the concrete case of Cartesian domain decomposition, where weak scaling incurs communication overhead that is independent of N as in (5.29). We choose $\tilde{\lambda} := \kappa + \lambda$ because the bandwidth overhead enters as latency. The speedup function for the slow computer is

$$S_\mu(N) = \frac{s+(1-s)N}{1+\tilde{\lambda}\,\mu^{-1}} \,, \tag{5.40}$$

hence the performance advantage is

$$A_\mu^{\mathrm{w}}(N) := \frac{S_\mu(\mu N)}{\mu S_{\mu=1}(N)} = \frac{(1+\tilde{\lambda})\,[s+(1-s)\mu N]}{[(1-s)N+s]\,(\tilde{\lambda}+\mu)} \,. \tag{5.41}$$

Again we consider special cases:

- $\tilde{\lambda} = 0$: In the absence of communication overhead,

$$A_\mu^{\mathrm{w}}(N) = \frac{(1-s)N+s/\mu}{(1-s)N+s} = 1 - \frac{\mu-1}{\mu N}\,s + \mathscr{O}(s^2) \,, \tag{5.42}$$

  which is clearly smaller than one, as expected. The situation is very similar to the strong scaling case (5.34).

- $s = 0$: With perfect parallelizability the performance advantage is always larger than one for $\mu > 1$, and independent of N:

$$A_\mu^{\mathrm{w}}(N) = \frac{1+\tilde{\lambda}}{1+\tilde{\lambda}/\mu} = \begin{cases} \xrightarrow{\mu \gg \tilde{\lambda}} 1+\tilde{\lambda} \\ \xrightarrow{\tilde{\lambda} \gg 1} \mu \end{cases} \tag{5.43}$$

  However, there is no significant gain to be expected for small $\tilde{\lambda}$. Even if $\tilde{\lambda} = 1$, i.e., if communication overhead is comparable to the serial runtime, we only get $1.33 \lesssim A^{\mathrm{w}} \lesssim 1.6$ in the typical range $2 \lesssim \mu \lesssim 4$.

In this scenario we have assumed that the "slow" machine with μ times more processors works on a problem which is μ times as large — hence the term "strictly weak scaling." We are comparing "fast" and "slow" machines across different problem sizes, which may not be what is desirable in reality, especially because the actual runtime grows accordingly. From this point of view the performance advantage A_μ^{w}, even if it can be greater than one, misses the important aspect of "time to solution." This disadvantage could be compensated if $A_\mu^{\mathrm{w}} \lesssim \mu$, but this is impossible according to (5.43).

Modified weak scaling

In reality, one would rather scale the amount of work with N (the number of standard CPUs) instead of μN so that the amount of memory per slow CPU can be μ times smaller. Indeed, this is the way such "scalable" HPC systems are usually built. The performance model thus encompasses both weak and strong scaling.

The advantage function to look at must separate the notions for "number of workers" and "amount of work." Therefore, we start with the speedup function

$$\begin{aligned} S_\mu^{\text{mod}}(N,W) &= \frac{[s+(1-s)W]\,/\,[\mu s+\mu(1-s)W/N+c(N/W)]}{[s+(1-s)]\,/\,\mu} \\ &= \frac{s+(1-s)W}{s+(1-s)W/N+c(N/W)\mu^{-1}}\ , \end{aligned} \tag{5.44}$$

where N is the number of workers and W denotes the amount of parallel work to be done. This expression specializes to strictly weak scaling if $W=N$ and $c(1)=\tilde{\lambda}$. The term $c(N/W)$ reflects the strong scaling component, effectively reducing communication overhead when $N>W$. Now we can derive the advantage function for modified weak scaling:

$$A_\mu^{\text{mod}}(N) := \frac{S_\mu^{\text{mod}}(\mu N,N)}{\mu S_{\mu=1}^{\text{mod}}(N,N)} = \frac{1+c(1)}{1+s(\mu-1)+c(\mu)}\ . \tag{5.45}$$

This is independent of N, which is not surprising since we keep the problem size constant when going from N to μN workers. The condition for a true advantage is the same as for strong scaling at $N=1$ (see (5.32)):

$$c(\mu)-c(1) < -s(\mu-1)\ . \tag{5.46}$$

In case of Cartesian domain decomposition we have $c(\mu)=\lambda+\kappa\mu^{-\beta}$, hence

$$A_\mu^{\text{mod}}(N) = \frac{1+\lambda+\kappa}{1+s(\mu-1)+\lambda+\kappa\mu^{-\beta}}\ . \tag{5.47}$$

For $s=0$ and to leading order in κ and λ,

$$A_\mu^{\text{mod}}(N) = 1+\left(1-\mu^{-\beta}\right)\kappa-\left(1+\mu^{-\beta}\right)\lambda\kappa+\mathscr{O}(\lambda^2,\kappa^2)\ , \tag{5.48}$$

which shows that communication bandwidth overhead (κ) dominates the gain. So in contrast to the strictly weak scaling case (5.43), latency enters only in a second-order term. Even for $\kappa=1$, at $\lambda=0$, $\beta=2/3$ and $2\lesssim\mu\lesssim 4$ we get $1.2\lesssim A^{\text{mod}}\lesssim 1.4$. In general, in the limit of large bandwidth overhead and small latency, modified weak scaling is the favorable mode of operation for parallel computers with slow processors. The dependence on μ is quite weak, and the advantage goes to $1+\kappa$ as $\mu\to\infty$.

In conclusion, we have found theoretical evidence that it can really be useful to

build large machines with many slow processors. Together with the expected reduction in power consumption vs. application performance, they may provide an attractive solution to the "power-performance dilemma," and the successful line of IBM Blue Gene supercomputers [V114, V115] shows that the concept works in practice. However, one must keep in mind that not all applications are well suited for massive parallelism, and that compromises must be made that may impede scalability (e.g., building fully nonblocking fat-tree networks becomes prohibitively expensive in very large systems). The need for a "sufficiently" strong single chip prevails if *all* applications are to profit from the blessings of Moore's Law.

5.3.9 Load imbalance

Inexperienced HPC users usually try to find the reasons for bad scalability of their parallel programs in the hardware details of the platform used and the specific drawbacks of the chosen parallelization method: Communication overhead, synchronization loss, false sharing, NUMA locality, bandwidth bottlenecks, etc. While all these are possible reasons for bad scalability (and are covered in due detail elsewhere in this book), load imbalance is often overlooked. Load imbalance occurs when synchronization points are reached by some workers earlier than by others (see Figure 5.5), leading to at least one worker idling while others still do useful work. As a consequence, resources are underutilized.

The consequences of load imbalance are hard to characterize in a simple model without further assumptions about the work distribution. Also, the actual impact on performance is not easily judged: As Figure 5.13 shows, having a few workers that take longer to reach the synchronization point ("laggers") leaves the rest, i.e., the majority of workers, idling for some time, incurring significant loss. On the other hand, a few "speeders," i.e., workers that finish their tasks early, may be harmless because the accumulated waiting time is negligible (see Figure 5.14).

The possible reasons for load imbalance are diverse, and can generally be divided into algorithmic issues, which should be tackled by choosing a modified or completely different algorithm, and optimization problems, which could be solved by code changes only. Sometimes the two are not easily distinguishable:

- The method chosen for distributing work among the workers may not be compatible with the structure of the problem. For example, in case of the blocked JDS sparse matrix-vector multiply algorithm introduced in Section 3.6.2, one could go about and assign a contiguous chunk of the loop over blocks (loop variable `ib`) to each worker. Owing to the JDS storage scheme, this could (depending on the actual matrix structure) cause load imbalance because the last iterations of the `ib` loop work on the lower parts of the matrix, where the number of diagonals is smaller. In this situation it might be better to use a cyclic or even dynamic distribution. This is especially easy to do with shared-memory parallel programming; see Section 6.1.6.

- No matter what variant of parallelism is exploited (see Section 5.2), it may not be known at compile time how much time a "chunk" of work actually takes.

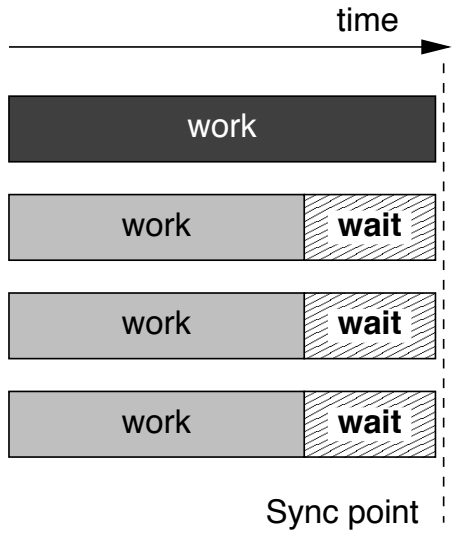

Figure 5.13: Load imbalance with few (one in this case) "laggers": A lot of resources are underutilized (hatched areas).

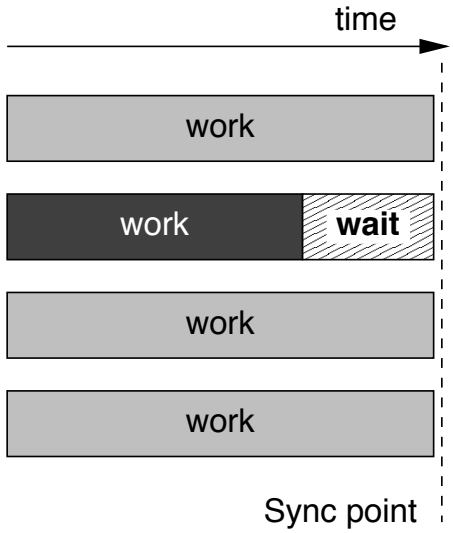

Figure 5.14: Load imbalance with few (one in this case) "speeders": Underutilization may be acceptable.

For example, an algorithm that requires each worker to perform a number of iterations in order to reach some convergence limit could be inherently load imbalanced because a different number of iterations may be needed on each worker.

- There may be a coarse granularity to the problem, limiting the available parallelism. This happens usually when the number of workers is not significantly smaller than the number of work packages. Exploiting additional levels of parallelism (if they exist) can help mitigate this problem.

- Although load imbalance is most often caused by uneven work distribution as described above, there may be other reasons. If a worker has to wait for resources like, e.g., I/O or communication devices, the time spent with such waiting does not count as useful work but can nevertheless lead to a delay, which turns the worker into a "lagger" (this is not to be confused with OS jitter; see below). Additionally, overhead of this kind is often statistical in nature, causing erratic load imbalance behavior.

If load imbalance is identified as a major performance problem, it should be checked whether a different strategy for work distribution could eliminate or at least reduce it. When a completely even distribution is impossible, it may suffice to get rid of "laggers" to substantially improve scalability. Furthermore, hiding I/O and communication costs by overlapping with useful work are also possible means to avoid load imbalance [A82].

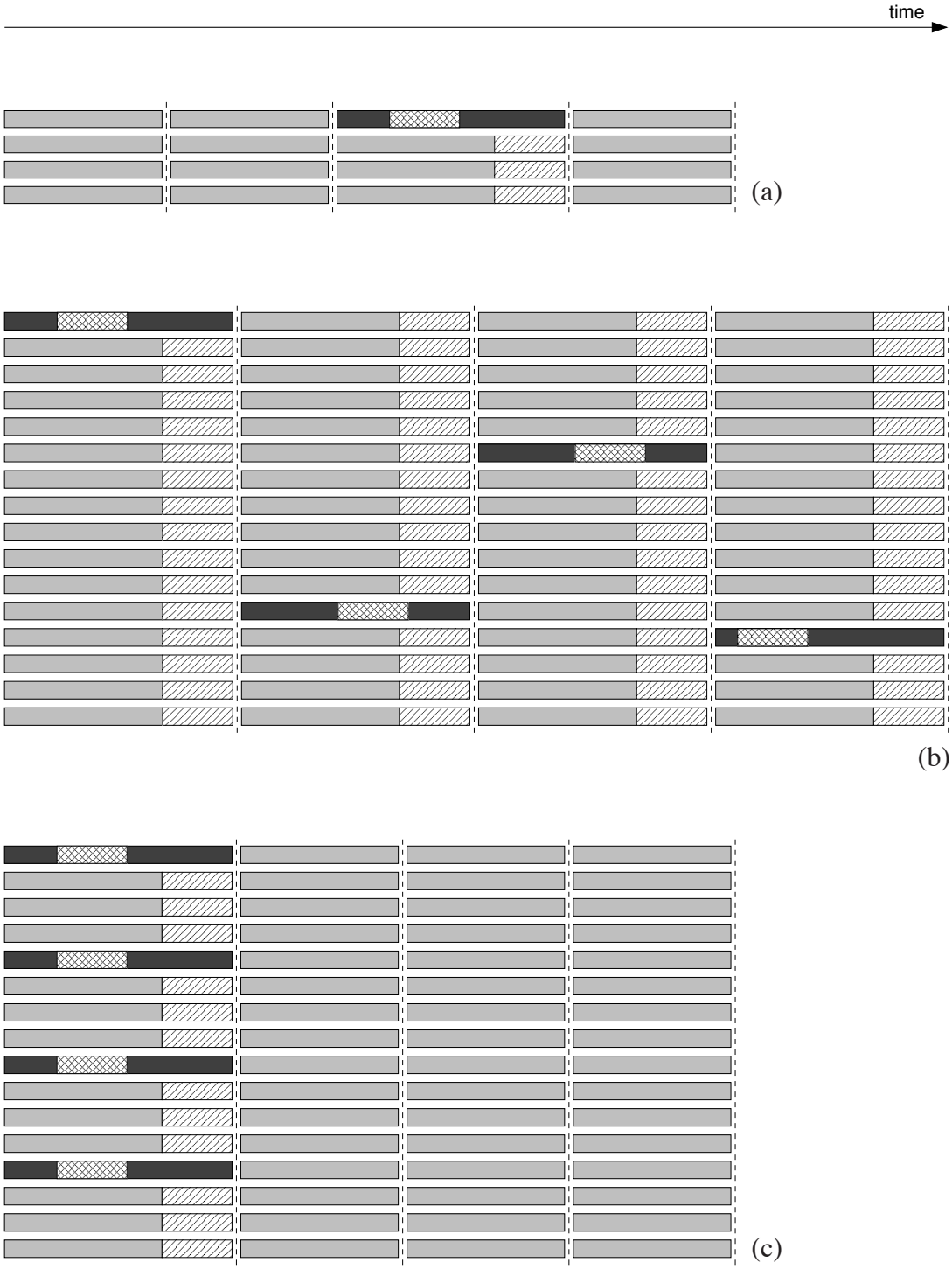

Figure 5.15: If an OS-related delay (cross-hatched boxes) occurs with some given probability per time, the impact on parallel performance may be small when the number of workers is small (a). Increasing the number of workers (here in a weak scaling scenario) increases the probability of the delay occurring before the next synchronization point, lengthening overall runtime (b). Synchronizing OS activity on all operating systems in the machine eliminates "OS jitter," leading to improved performance (c). (Pictures adapted from [L77]).

OS jitter

A peculiar and interesting source of load imbalance with surprising consequences has recently been identified in large-scale parallel systems built from commodity components [L77]. Most standard installations of distributed-memory parallel computers run individual, independent operating system instances on all nodes. An operating system has many routine chores, of which running user programs is only one. Whenever a regular task like writing to a log file, delivering performance metrics, flushing disk caches, starting cron jobs, etc., kicks in, a running application process may be delayed by some amount. On the next synchronization point, this "lagger" will delay the parallel program slightly due to load imbalance, but this is usually negligible if it happens infrequently *and* the number of processes is small (see Figure 5.15 (a)). Certainly, the exact delay will depend on the duration of the OS activity and the frequency of synchronizations.

Unfortunately, the situation changes when the number of workers is massively increased. This is because "OS noise" is of statistical nature over all workers; the more workers there are, the larger the probability that a delay will occur between two successive synchronization points. Load imbalance will thus start to happen more frequently when the frequency of synchronization points in the code comes near the average frequency of noise-caused delays, which may be described as a "resonance" phenomenon [L77]. This is shown in Figure 5.15 (b) for a weak scaling scenario. Note that this effect is strictly limited to massively parallel systems; in practice, it will not show up with only tens or even hundreds of compute nodes. There are sources of performance variability in those smaller systems, too, but they are unrelated to OS jitter.

Apart from trying to reduce OS activity as far as possible (by, e.g., deactivating unused daemons, polling, and logging activities, or leaving one processor per node free for OS tasks), an effective means of reducing OS jitter is to *synchronize* unavoidable periodic activities on all workers (see Figure 5.15 (c)). This aligns the delays on all workers at the same synchronization point, and the performance penalty is not larger than in case (a). However, such measures are not standard procedures and require substantial changes to the operating system. Still, as the number of cores and nodes in large-scale parallel computers continues to increase, OS noise containment will most probably soon be a common feature.

Problems

For solutions see page 296 *ff*.

5.1 *Overlapping communication and computation*. How would the strong scaling analysis for slow processors in Section 5.3.8 qualitatively change if communication could overlap with computation (assuming that the hardware supports it and the algorithm is formulated accordingly)? Take into account that the overlap may not be perfect if communication time exceeds computation time.

5.2 *Choosing an optimal number of workers*. If the scalability characteristics of a parallel program are such that performance saturates or even declines with growing N, the question arises what the "optimal" number of workers is. Usually one would not want to choose the point where performance is at its maximum (or close to saturation), because parallel efficiency will already be low there. What is needed is a "cost model" that discourages the use of too many workers. Most computing centers charge for compute time in units of *CPU wallclock hours*, i.e., an N-CPU job running for a time T_w will be charged as an amount proportional to NT_w. For the user, minimizing the product of walltime (i.e., time to solution) and cost should provide a sensible balance. Derive a condition for the optimal number of workers N_opt, assuming strong scaling with a constant communication overhead (i.e., a latency-bound situation). What is the speedup with N_opt workers?

5.3 *The impact of synchronization*. Synchronizing all workers can be very time-consuming, since it incurs costs that are usually somewhere between logarithmic and linear in the number of workers. What is the impact of synchronization on strong and weak scalability?

5.4 *Accelerator devices*. Accelerator devices for standard compute nodes are becoming increasingly popular today. A common variant of this idea is to outfit standard compute nodes (comprising, e.g., two multicore chips) with special hardware sitting in an I/O slot. Accelerator hardware is capable of executing certain operations orders of magnitude faster than the host system's CPUs, but the amount of available memory is usually much smaller than the host's. Porting an application to an accelerator involves identifying suitable code parts to execute on the special hardware. If the speedup for the accelerated code parts is α, how much of the original code (in terms of runtime) must be ported to get at least 90% efficiency on the accelerator hardware? What is the significance of the memory size restrictions?

5.5 *Fooling the masses with performance data*. The reader is strongly encouraged to read David H. Bailey's humorous article "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers" [S7]. Although the paper was written already in 1991, many of the points made are still highly relevant.

Chapter 6

Shared-memory parallel programming with OpenMP

In the multicore world, one-socket single-core systems have all but vanished except for the embedded market. The price vs. performance "sweet spot" lies mostly in the two-socket regime, with multiple cores (and possibly multiple chips) on a socket. Parallel programming in a basic form should thus start at the shared-memory level, although it is entirely possible to run multiple processes without a concept of shared memory. See Chapter 9 for details on distributed-memory parallel programming.

However, shared-memory programming is not an invention of the multicore era. Systems with multiple (single-core) processors have been around for decades, and appropriate portable programming interfaces, most notably POSIX threads [P9], have been developed in the 1990s. The basic principles, limitations and bottlenecks of shared-memory parallel programming are certainly the same as with any other parallel model (see Chapter 5), although there are some peculiarities which will be covered in Chapter 7. The purpose of the current chapter is to give a nonexhaustive overview of OpenMP, which is the dominant shared-memory programming standard today. OpenMP bindings are defined for the C, C++, and Fortran languages as of the current version of the standard (3.0). Some OpenMP constructs that are mainly used for optimization will be introduced in Chapter 7.

We should add that there are specific solutions for the C++ language like, e.g., Intel Threading Building Blocks (TBB) [P10], which may provide better functionality than OpenMP in some respects. We also deliberately ignore compiler-based automatic shared-memory parallelization because it has up to now not lived up to expectations except for trivial cases.

6.1 Short introduction to OpenMP

Shared memory opens the possibility to have immediate access to all data from all processors without explicit communication. Unfortunately, POSIX threads are not a comfortable parallel programming model for most scientific software, which is typically loop-centric. For this reason, a joint effort was made by compiler vendors to establish a standard in this field, called OpenMP [P11]. OpenMP is a set of *compiler directives* that a non-OpenMP-capable compiler would just regard as comments and ignore. Hence, a well-written parallel OpenMP program is also a valid serial program

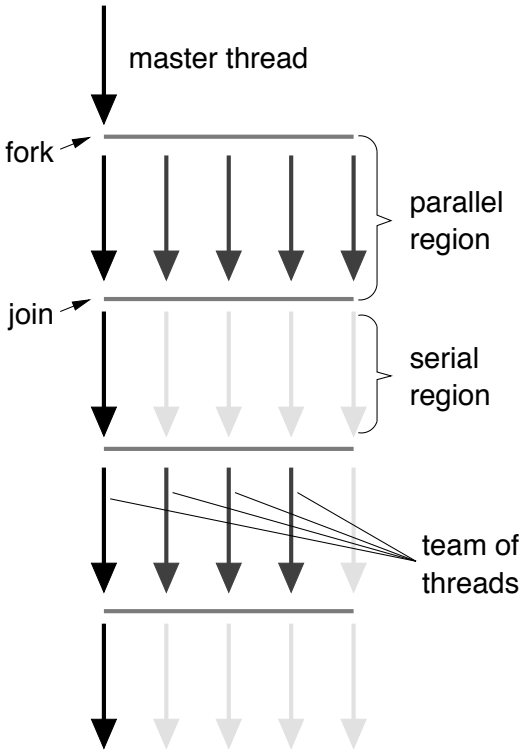

Figure 6.1: Model for OpenMP thread operations: The master thread "forks" team of threads, which work on shared memory in a parallel region. After the parallel region, the threads are "joined," i.e., terminated or put to sleep, until the next parallel region starts. The number of running threads may vary among parallel regions.

(this is certainly not a requirement, but it simplifies development and debugging considerably). The central entity in an OpenMP program is not a process but a *thread*. Threads are also called "lightweight processes" because several of them can share a common address space and mutually access data. Spawning a thread is much less costly than forking a new process, because threads share everything but instruction pointer (the address of the next instruction to be executed), stack pointer and register state. Each thread can, by means of its local stack pointer, also have "private" variables, but since all data is accessible via the common address space, it is only a matter of taking the address of an item to make it accessible to all other threads as well. However, the OpenMP standard actually *forbids* making a private object available to other threads via its address. It will become clear later that this is actually a good idea.

We will concentrate on the Fortran interface for OpenMP here, and point out important differences to the C/C++ bindings as appropriate.

6.1.1 Parallel execution

In any OpenMP program, a single thread, the *master thread*, runs immediately after startup. Truly parallel execution happens inside *parallel regions*, of which an arbitrary number can exist in a program. Between two parallel regions, no thread except the master thread executes any code. This is also called the "fork-join model" (see Figure 6.1). Inside a parallel region, a *team of threads* executes instruction streams concurrently. The number of threads in a team may vary among parallel regions.

OpenMP is a layer that adapts the raw OS thread interface to make it more us-

able with the typical structures that numerical software tends to employ. In practice, parallel regions in Fortran are initiated by `!$OMP PARALLEL` and ended by `!$OMP END PARALLEL` directives, respectively. The `!$OMP` string is a so-called *sentinel*, which starts an OpenMP directive (in C/C++, `#pragma omp` is used instead). Inside a parallel region, each thread carries a unique identifier, its *thread ID*, which runs from zero to the number of threads minus one, and can be obtained by the `omp_get_thread_num()` API function:

```
1   use omp_lib      ! module with API declarations
2
3   print *,'I am the master, and I am alone'
4  !$OMP PARALLEL
5   call do_work_package(omp_get_thread_num(),omp_get_num_threads())
6  !$OMP END PARALLEL
```

The `omp_get_num_threads()` function returns the number of active threads in the current parallel region. The `omp_lib` module contains the API definitions (in Fortran 77 and C/C++ there are include files `mpif.h` and `omp.h`, respectively). Code between `OMP PARALLEL` and `OMP END PARALLEL`, including subroutine calls, is executed by every thread. In the simplest case, the thread ID can be used to distinguish the tasks to be performed on different threads; this is done by calling the `do_work_package()` subroutine in above example with the thread ID and the overall number of threads as parameters. Using OpenMP in this way is mostly equivalent to the POSIX threads programming model.

An important difference between the Fortran and C/C++ OpenMP bindings must be stressed here. In C/C++, there is no `end parallel` directive, because all directives apply to the following statement or structured block. The example above would thus look like this in C++:

```
1   #include <omp.h>
2
3   std::cout << "I am the master, and I am alone";
4  #pragma omp parallel
5  {
6   do_work_package(omp_get_thread_num(),omp_get_num_threads());
7  }
```

The curly braces could actually be omitted in this particular case, but the fact that a structured block is subject to parallel execution has consequences for data scoping (see below).

The actual number of running threads does not have to be known at compile time. It can be set by the environment variable prior to running the executable:

```
1  $ export OMP_NUM_THREADS=4
2  $ ./a.out
```

Although there are also means to set or alter the number of running threads under program control, an OpenMP program should always be written so that it does not assume a specific number of threads.

Listing 6.1: "Manual" loop parallelization and variable privatization. Note that this is *not* the intended mode for OpenMP.

```
1   integer :: bstart, bend, blen, numth, tid, i
2   integer :: N
3   double precision, dimension(N) :: a,b,c
4   ...
5  !$OMP PARALLEL PRIVATE(bstart,bend,blen,numth,tid,i)
6   numth = omp_get_num_threads()
7   tid = omp_get_thread_num()
8   blen = N/numth
9   if(tid.lt.mod(N,numth)) then
10    blen = blen + 1
11    bstart = blen * tid + 1
12  else
13    bstart = blen * tid + mod(N,numth) + 1
14  endif
15  bend = bstart + blen - 1
16  do i = bstart,bend
17    a(i) = b(i) + c(i)
18  enddo
19 !$OMP END PARALLEL
```

6.1.2 Data scoping

Any variables that existed before a parallel region still exist inside, and are by default shared between all threads. True work sharing, however, makes sense only if each thread can have its own, *private* variables. OpenMP supports this concept by defining a separate stack for every thread. There are three ways to make private variables:

1. A variable that exists before entry to a parallel construct can be privatized, i.e., made available as a private instance for every thread, by a `PRIVATE` clause to the `OMP PARALLEL` directive. The private variable's scope extends until the end of the parallel construct.

2. The index variable of a worksharing loop (see next section) is automatically made private.

3. Local variables in a subroutine called from a parallel region are private to each calling thread. This pertains also to copies of actual arguments generated by the call-by-value semantics, and to variables declared inside structured blocks in C/C++. However, local variables carrying the `SAVE` attribute in Fortran (or the `static` storage class in C/C++) will be shared.

Shared variables that are not modified in the parallel region do not have to be made private.

A simple loop that adds two arrays could thus be parallelized as shown in Listing 6.1. The actual loop is contained in lines 16–18, and everything before that is

just for calculating the loop bounds for each thread. In line 5 the `PRIVATE` clause to the `PARALLEL` directive privatizes all specified variables, i.e., each thread gets its own instance of each variable on its local stack, with an undefined initial value (C++ objects will be instantiated using the default constructor). Using `FIRSTPRIVATE` instead of `PRIVATE` would initialize the privatize instances with the contents of the shared instance (in C++, the copy constructor is employed). After the parallel region, the original values of the privatized variables are retained if they are not modified on purpose. Note that there are separate clauses (`THREADPRIVATE` and `COPYIN`, respectively [P11]) for privatization of global or static data (`SAVE` variables, common block elements, `static` variables).

In C/C++, there is actually less need for using the `private` clause in many cases, because the `parallel` directive applies to a structured block. Instead of privatizing shared instances, one can simply declare local variables:

```
1 #pragma omp parallel
2 {
3   int bstart, bend, blen, numth, tid, i;
4   ...              // calculate loop boundaries
5   for(i=bstart; i<=bend; ++i)
6     a[i] = b[i] + c[i];
7 }
```

Manual loop parallelization as shown here is certainly not the intended mode of operation in OpenMP. The standard defines much more advanced means of distributing work among threads (see below).

6.1.3 OpenMP worksharing for loops

Being the omnipresent programming structure in scientific codes, loops are natural candidates for parallelization if individual iterations are independent. This corresponds to the medium-grained data parallelism described in Section 5.2.1. As an example, consider a parallel version of a simple program for the calculation of π by integration:

$$\pi = \int_0^1 \mathrm{d}x \, \frac{4}{1+x^2} \tag{6.1}$$

Listing 6.2 shows a possible implementation. In contrast to the previous examples, this is also valid serial code. The initial value of `sum` is copied to the private instances via the `FIRSTPRIVATE` clause on the `PARALLEL` directive. Then, a `DO` directive in front of a `do` loop starts a *worksharing* construct: The iterations of the loop are distributed among the threads (which are running because we are in a parallel region). Each thread gets its own iteration space, i.e., is assigned to a different set of `i` values. How threads are mapped to iterations is implementation-dependent by default, but can be influenced by the programmer (see Section 6.1.6 below). Although shared in the enclosing parallel region, the loop counter `i` is privatized automatically. The final `END DO` directive after the loop is not strictly necessary here, but may be required in cases where the `NOWAIT` clause is specified; see Section 7.2.1 on page 170 for

Listing 6.2: A simple program for numerical integration of a function in OpenMP.

```
1   double precision :: pi,w,sum,x
2   integer :: i,N=1000000
3
4   pi = 0.d0
5   w = 1.d0/N
6   sum = 0.d0
7  !$OMP PARALLEL PRIVATE(x) FIRSTPRIVATE(sum)
8  !$OMP DO
9     do i=1,n
10      x = w*(i-0.5d0)
11      sum = sum + 4.d0/(1.d0+x*x)
12    enddo
13 !$OMP END DO
14 !$OMP CRITICAL
15    pi= pi + w*sum
16 !$OMP END CRITICAL
17 !$OMP END PARALLEL
```

details. A `DO` directive must be followed by a `do` loop, and applies to this loop only. In C/C++, the `for` directive serves the same purpose. Loop counters are restricted to integers (signed or unsigned), pointers, or random access iterators.

In a parallel loop, each thread executes "its" share of the loop's iteration space, accumulating into its private `sum` variable (line 11). After the loop, and still inside the parallel region, the partial sums must be added to get the final result (line 15), because the private instances of `sum` will be gone once the region is left. There is a problem, however: Without any countermeasures, threads would write to the result variable `pi` concurrently. The result would depend on the exact order the threads access `pi`, and it would most probably be wrong. This is called a *race condition*, and the next section will explain what one can do to prevent it.

Loop worksharing works even if the parallel loop itself resides in a subroutine called from the enclosing parallel region. The `DO` directive is then called *orphaned*, because it is outside the lexical extent of a parallel region. If such a directive is encountered while no parallel region is active, the loop will not be workshared.

Finally, if a separation of the parallel region from the workshared loop is not required, the two directives can be combined:

```
1  !$OMP PARALLEL DO
2    do i=1,N
3      a(i) = b(i) + c(i) * d(i)
4    enddo
5  !$OMP END PARALLEL DO
```

The set of clauses allowed for this *combined parallel worksharing directive* is the union of all clauses allowed on each directive separately.

6.1.4 Synchronization

Critical regions

Concurrent write access to a shared variable or, in more general terms, a shared resource, must be avoided by all means to circumvent race conditions. *Critical regions* solve this problem by making sure that at most one thread at a time executes some piece of code. If a thread is executing code inside a critical region, and another thread wants to enter, the latter must wait (block) until the former has left the region. In the integration example (Listing 6.2), the `CRITICAL` and `END CRITICAL` directives (lines 14 and 16) bracket the update to `pi` so that the result is always correct. Note that the order in which threads enter the critical region is undefined, and can change from run to run. Consequently, the definition of a "correct result" must encompass the possibility that the partial sums are accumulated in a random order, and the usual reservations regarding floating-point accuracy do apply [135]. (If strong sequential equivalence, i.e., bitwise identical results compared to a serial code is required, OpenMP provides a possible solution with the `ORDERED` construct, which we do not cover here.)

Critical regions hold the danger of *deadlocks* when used inappropriately. A deadlock arises when one or more "agents" (threads in this case) wait for resources that will never become available, a situation that is easily generated with badly arranged `CRITICAL` directives. When a thread encounters a `CRITICAL` directive inside a critical region, it will block forever. Since this could happen in a deeply nested subroutine, deadlocks are sometimes hard to pin down.

OpenMP has a simple solution for this problem: A critical region may be given a *name* that distinguishes it from others. The name is specified in parentheses after the `CRITICAL` directive:

```
1  !$OMP PARALLEL DO PRIVATE(x)
2    do i=1,N
3      x = SIN(2*PI*x/N)
4  !$OMP CRITICAL (psum)
5      sum = sum + func(x)
6  !$OMP END CRITICAL (psum)
7    enddo
8  !$OMP END PARALLEL DO
9    ...
10   double precision func(v)
11   double precision :: v
12 !$OMP CRITICAL (prand)
13   func = v + random_func()
14 !$OMP END CRITICAL (prand)
15   END SUBROUTINE func
```

The update to `sum` in line 5 is protected by a critical region. In subroutine `func()` there is another critical region because it is not allowed to call `random_func()` (line 13) by more than one thread at a time; it probably contains a random seed with a `SAVE` attribute. Such a function is not *thread safe*, i.e., its concurrent execution would incur a race condition.

Without the names on the two different critical regions, this code would deadlock because a thread that has just called `func()`, already in a critical region, would immediately encounter the second critical region and wait for itself indefinitely to free the resource. With the names, the second critical region is understood to protect a different resource than the first.

A disadvantage of named critical regions is that the names are unique identifiers. It is not possible to have them indexed by an integer variable, for instance. There are OpenMP API functions that support the use of *locks* for protecting shared resources. The advantage of locks is that they are ordinary variables that can be arranged as arrays or in structures. That way it is possible to protect each single element of an array of resources individually, even if their number is not known at compile time. See Section 7.2.3 for an example.

Barriers

If, at a certain point in the parallel execution, it is necessary to synchronize *all* threads, a `BARRIER` can be used:

```
1 !$OMP BARRIER
```

The barrier is a *synchronization point*, which guarantees that all threads have reached it before any thread goes on executing the code below it. Certainly it must be ensured that every thread hits the barrier, or a deadlock may occur.

Barriers should be used with caution in OpenMP programs, partly because of their potential to cause deadlocks, but also due to their performance impact (synchronization is overhead). Note also that every parallel region executes an implicit barrier at its end, which cannot be removed. There is also a default implicit barrier at the end of worksharing loops and some other constructs to prevent race conditions. It can be eliminated by specifying the `NOWAIT` clause. See Section 7.2.1 for details.

6.1.5 Reductions

The example in Listing 6.3 shows a loop code that adds some random noise to the elements of an array `a()` and calculates its vector norm. The `RANDOM_NUMBER()` subroutine may be assumed to be thread safe, according to the OpenMP standard.

Similar to the integration code in Listing 6.2, the loop implements a *reduction* operation: Many contributions (the updated elements of `a()`) are accumulated into a single variable. We have previously solved this problem with a critical region, but OpenMP provides a more elegant alternative by supporting reductions directly via the `REDUCTION` clause (end of line 5). It automatically privatizes the specified variable(s) (`s` in this case) and initializes the private instances with a sensible starting value. At the end of the construct, all partial results are accumulated into the shared instance of `s`, using the specified operator (+ here) to get the final result.

There is a set of supported operators for OpenMP reductions (slightly different for Fortran and C/C++), which cannot be extended. C++ overloaded operators are not allowed. However, the most common cases (addition, subtraction, multiplication,

Listing 6.3: Example with reduction clause for adding noise to the elements of an array and calculating its vector norm.

```
1   double precision :: r,s
2   double precision, dimension(N) :: a
3
4   call RANDOM_SEED()
5  !$OMP PARALLEL DO PRIVATE(r) REDUCTION(+:s)
6   do i=1,N
7     call RANDOM_NUMBER(r)  ! thread safe
8     a(i) = a(i) + func(r)  ! func() is thread safe
9     s = s + a(i) * a(i)
10  enddo
11 !$OMP END PARALLEL DO
12
13  print *,'Sum = ',s
```

logical, etc.) are covered. If a required operator is not available, one must revert to the "manual" method as shown in the Listing 6.2.

Note that the automatic initialization for reduction variables, though convenient, bears the danger of producing invalid serial, i.e., non-OpenMP code. Compiling the example above without OpenMP support will leave `s` uninitialized.

6.1.6 Loop scheduling

As mentioned earlier, the mapping of loop iterations to threads is configurable. It can be controlled by the argument of a `SCHEDULE` clause to the loop worksharing directive:

```
1  !$OMP DO SCHEDULE(STATIC)
2    do i=1,N
3      a(i) = calculate(i)
4    enddo
5  !$OMP END DO
```

The simplest possibility is `STATIC`, which divides the loop into contiguous chunks of (roughly) equal size. Each thread then executes on exactly one chunk. If for some reason the amount of work per loop iteration is not constant but, e.g., decreases with loop index, this strategy is suboptimal because different threads will get vastly different workloads, which leads to load imbalance. One solution would be to use a *chunksize* like in "`STATIC,1`," dictating that chunks of size 1 should be distributed across threads in a round-robin manner. The chunksize may not only be a constant but any valid integer-valued expression.

There are alternatives to the static schedule for other types of workload (see Figure 6.2). *Dynamic* scheduling assigns a chunk of work, whose size is defined by the chunksize, to the next thread that has finished its current chunk. This allows for a very flexible distribution which is usually not reproduced from run to run. Threads that get assigned "easier" chunks will end up completing more of them, and load

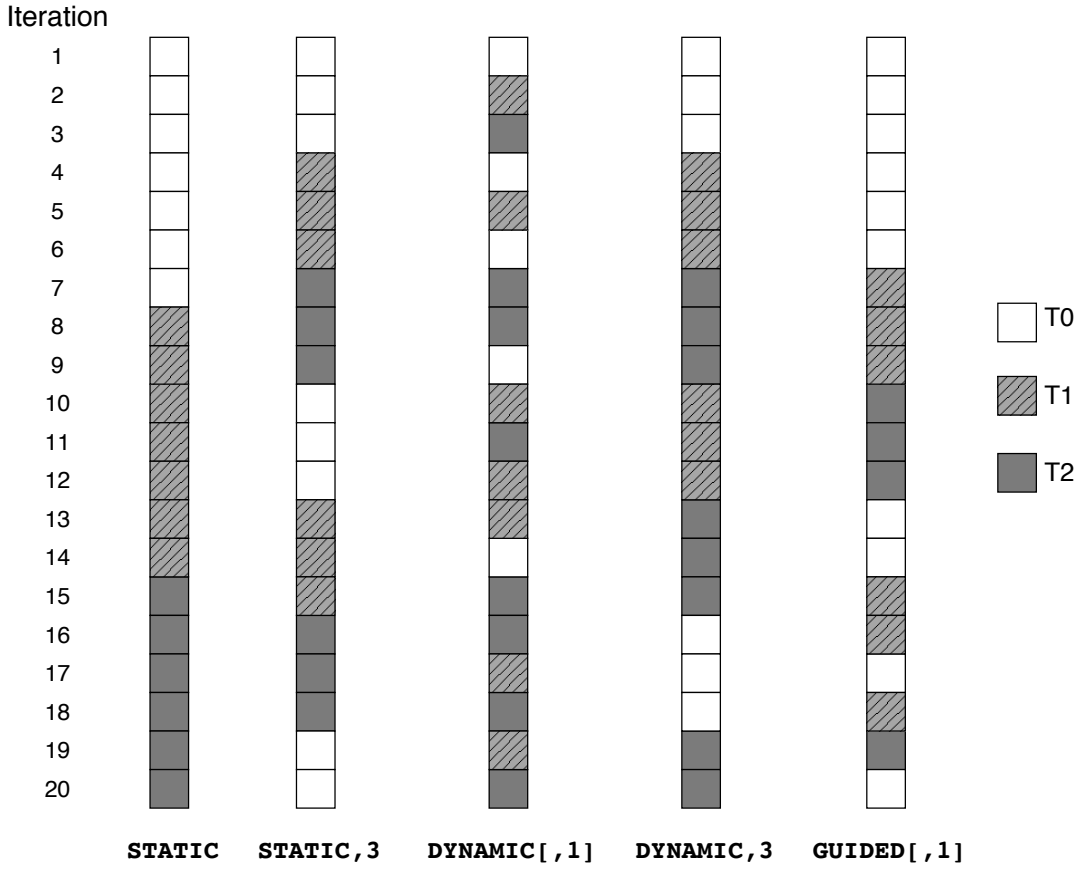

Figure 6.2: Loop schedules in OpenMP. The example loop has 20 iterations and is executed by three threads (T0, T1, T2). The default chunksize for `DYNAMIC` and `GUIDED` is one. If a chunksize is specified, the last chunk may be shorter. Note that only the `STATIC` schedules guarantee that the distribution of chunks among threads stays the same from run to run.

imbalance is greatly reduced. The downside is that dynamic scheduling generates significant overhead if the chunks are too small in terms of execution time (see Section 7.2.1 for an assessment of scheduling overhead). This is why it is often desirable to use a moderately large chunksize on tight loops, which in turn leads to more load imbalance. In cases where this is a problem, the *guided* schedule may help. Again, threads request new chunks dynamically, but the chunksize is always proportional to the remaining number of iterations divided by the number of threads. The smallest chunksize is specified in the schedule clause (default is 1). Despite the dynamic assignment of chunks, scheduling overhead is kept under control. However, a word of caution is in order regarding dynamic and guided schedules: Due to the indeterministic nature of the assignment of threads to chunks, applications that are limited by memory bandwidth may suffer from insufficient access locality on ccNUMA systems (see Section 4.2.3 for an introduction to ccNUMA architecture and Chapter 8 for ccNUMA-specific performance effects and optimization methods). The static schedule is thus the only choice under such circumstances, if the standard worksharing

directives are used. Of course there is also the possibility of "explicit" scheduling, using the thread ID number to assign work to threads as shown in Section 6.1.2.

For debugging and profiling purposes, OpenMP provides a facility to determine the loop scheduling at runtime. If the scheduling clause specifies "`RUNTIME`," the loop is scheduled according to the contents of the `OMP_SCHEDULE` shell variable. However, there is no way to set different schedulings for different loops that use the `SCHEDULE(RUNTIME)` clause.

6.1.7 Tasking

In early versions of the standard, parallel worksharing in OpenMP was mainly concerned with loop structures. However, not all parallel work comes in loops; a typical example is a linear list of objects (probably arranged in a `std::list<>` STL container), which should be processed in parallel. Since a list is not easily addressable by an integer index or a random-access iterator, a loop worksharing construct is ruled out, or could only be used with considerable programming effort.

OpenMP 3.0 provides the *task* concept to circumvent this limitation. A task is defined by the `TASK` directive, and contains code to be executed.[1] When a thread encounters a task construct, it may execute it right away or set up the appropriate data environment and defer its execution. The task is then ready to be executed later by any thread of the team.

As a simple example, consider a loop in which some function must be called for each loop index with some probability:

```
1    integer i,N=1000000
2    type(object), dimension(N) :: p
3    double precision :: r
4    ...
5  !$OMP PARALLEL PRIVATE(r,i)
6  !$OMP SINGLE
7    do i=1,N
8      call RANDOM_NUMBER(r)
9      if(p(i)%weight > r) then
10 !$OMP TASK
11       ! i is automatically firstprivate
12       ! p() is shared
13       call do_work_with(p(i))
14 !$OMP END TASK
15     endif
16   enddo
17 !$OMP END SINGLE
18 !$OMP END PARALLEL
```

The actual number of calls to `do_work_with()` is unknown, so tasking is a natural choice here. A `do` loop over all elements of `p()` is executed in a `SINGLE` region (lines 6–17). A `SINGLE` region will be entered by one thread only, namely the one that reaches the `SINGLE` directive first. All others skip the code until the

[1] In OpenMP terminology, "task" is actually a more general term; the definition given here is sufficient for our purpose.

`END SINGLE` directive and wait there in an implicit barrier. With a probability determined by the current object's content, a `TASK` construct is entered. One task consists in the call to `do_work_with()` (line 13) together with the appropriate data environment, which comprises the array of types `p()` and the index `i`. Of course, the index is unique for each task, so it should actually be subject to a `FIRSTPRIVATE` clause. OpenMP specifies that variables that are private in the enclosing context are automatically made `FIRSTPRIVATE` inside the task, while shared data stays shared (except if an additional data scoping clause is present). This is exactly what we want here, so no additional clause is required.

All the tasks generated by the thread in the `SINGLE` region are subject to dynamic execution by the thread team. Actually, the generating thread may also be forced to suspend execution of the loop at the `TASK` construct (which is one example of a *task scheduling point*) in order to participate in running queued tasks. This can happen when the (implementation-dependent) internal limit of queued tasks is reached. After some tasks have been run, the generating thread will return to the loop. Note that there are complexities involved in task scheduling that our simple example cannot fathom; multiple threads can generate tasks concurrently, and tasks can be declared *untied* so that a different thread may take up execution at a task scheduling point. The OpenMP standard provides excessive examples.

Task parallelism with its indeterministic execution poses the same problems for ccNUMA access locality as dynamic or guided loop scheduling. Programming techniques to ameliorate these difficulties do exist [O58], but their applicability is limited.

6.1.8 Miscellaneous

Conditional compilation

In some cases it may be useful to write different code depending on OpenMP being enabled or not. The directives themselves are no problem here because they will be ignored gracefully. Beyond this default behavior one may want to mask out, e.g., calls to API functions or any code that makes no sense without OpenMP enabled. This is supported in C/C++ by the preprocessor symbol `_OPENMP`, which is defined only if OpenMP is available. In Fortran the special sentinel "`!$`" acts as a comment only if OpenMP is not enabled (see Listing 6.4).

Memory consistency

In the code shown in Listing 6.4, the second API call (line 8) is located in a `SINGLE` region. This is done because `numthreads` is global and should be written to only by one thread. In the critical region each thread just prints a message, but a necessary requirement for the `numthreads` variable to have the updated value is that no thread leaves the `SINGLE` region before the update has been "promoted" to memory. The `END SINGLE` directive acts as an implicit barrier, i.e., no thread can continue executing code before all threads have reached the same point. The OpenMP memory model ensures that barriers enforce memory consistency: Variables that have been held in registers are written out so that cache coherence can make sure that

Listing 6.4: Fortran sentinels and conditional compilation with OpenMP combined.

```
1  !$ use omp_lib
2    myid=0
3    numthreads=1
4  #ifdef _OPENMP
5  !$OMP PARALLEL PRIVATE(myid)
6    myid = omp_get_thread_num()
7  !$OMP SINGLE
8    numthreads = omp_get_num_threads()
9  !$OMP END SINGLE
10 !$OMP CRITICAL
11   write(*,*) 'Parallel program - this is thread ',myid,&
12                                          ' of ',numthreads
13 !$OMP END CRITICAL
14 !$OMP END PARALLEL
15 #else
16   write(*,*) 'Serial program'
17 #endif
```

all caches get updated values. This can also be initiated under program control via the `FLUSH` directive, but most OpenMP worksharing and synchronization constructs perform implicit barriers, and hence flushes, at the end.

Note that compiler optimizations can prevent modified variable contents to be seen by other threads immediately. If in doubt, use the `FLUSH` directive or declare the variable as `volatile` (only available in C/C++ and Fortran 2003).

Thread safety

The `write` statement in line 11 is serialized (i.e., protected by a critical region) so that its output does not get clobbered when multiple threads write to the console. As a general rule, I/O operations and general OS functionality, but also common library functions should be serialized because they may not be thread safe. A prominent example is the `rand()` function from the C library, as it uses a static variable to store its hidden state (the seed).

Affinity

One should note that the OpenMP standard gives no hints as to how threads are to be bound to the cores in a system, and there are no provisions for implementing locality constraints. One cannot rely at all on the OS to make a good choice regarding placement of threads, so it makes sense (especially on multicore architectures and ccNUMA systems) to use OS-level tools, compiler support or library functions to explicitly pin threads to cores. See Appendix A for technical details.

Environment variables

Some aspects of OpenMP program execution can be influenced by environment variables. `OMP_NUM_THREADS` and `OMP_SCHEDULE` have already been described above.

Concerning thread-local variables, one must keep in mind that usually the OS shell restricts the maximum size of all stack variables of its processes, and there may also be a system limit on each thread's stack size. This limit can be adjusted via the `OMP_STACKSIZE` environment variable. Setting it to, e.g., "`100M`" will set a stack size of 100 MB per thread (excluding the initial program thread, whose stack size is still set by the shell). Stack overflows are a frequent source of problems with OpenMP programs.

The OpenMP standard allows for the number of active threads to dynamically change between parallel regions in order to adapt to available system resources (dynamic thread number adjustment). This feature can be switched on or off by setting the `OMP_DYNAMIC` environment variable to `true` or `false`, respectively. It is unspecified what the OpenMP runtime implements as the default.

6.2 Case study: OpenMP-parallel Jacobi algorithm

The Jacobi algorithm studied in Section 3.3 can be parallelized in a straightforward way. We add a slight modification, however: A sensible convergence criterion shall ensure that the code actually produces a converged result. To do this we introduce a new variable `maxdelta`, which stores the maximum absolute difference over all lattice sites between the values before and after each sweep (see Listing 6.5). If `maxdelta` drops below some threshold `eps`, convergence is reached.

Fortunately the OpenMP Fortran interface permits using the `MAX()` intrinsic function in `REDUCTION` clauses, which simplifies the convergence check (lines 7 and 15 in Listing 6.5). Figure 6.3 shows performance data for one, two, and four threads on an Intel dual-socket Xeon 5160 3.0 GHz node. In this node, the two cores in a socket share a common 4 MB L2 cache and a frontside bus (FSB) to the chipset. The results exemplify several key aspects of parallel programming in multicore environments:

- With increasing N there is the expected performance breakdown when the working set ($2 \times N^2 \times 8$ bytes) does not fit into cache any more. This breakdown occurs at the same N for single-thread and dual-thread runs if the two threads run in the same L2 group (filled symbols). If the threads run on different sockets (open symbols), this limit is a factor of $\sqrt{2}$ larger because the aggregate cache size is doubled (dashed lines in Figure 6.3). The second breakdown at very large N, i.e., when two successive lattice rows exceed the L2 cache size, cannot be seen here as we use a square lattice (see Section 3.3).

- A single thread can saturate a socket's FSB for a memory-bound situation, i.e.,

Listing 6.5: OpenMP implementation of the 2D Jacobi algorithm on an $N \times N$ lattice, with a convergence criterion added.

```
1   double precision, dimension(0:N+1,0:N+1,0:1) :: phi
2   double precision :: maxdelta,eps
3   integer :: t0,t1
4   eps = 1.d-14       ! convergence threshold
5   t0 = 0 ; t1 = 1
6   maxdelta = 2.d0*eps
7   do while(maxdelta.gt.eps)
8     maxdelta = 0.d0
9  !$OMP PARALLEL DO REDUCTION(max:maxdelta)
10    do k = 1,N
11      do i = 1,N
12         ! four flops, one store, four loads
13         phi(i,k,t1) = (  phi(i+1,k,t0) + phi(i-1,k,t0)
14                        + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
15         maxdelta = max(maxdelta,abs(phi(i,k,t1)-phi(i,k,t0)))
16      enddo
17    enddo
18 !$OMP END PARALLEL DO
19    ! swap arrays
20    i = t0 ; t0=t1 ; t1=i
21  enddo
```

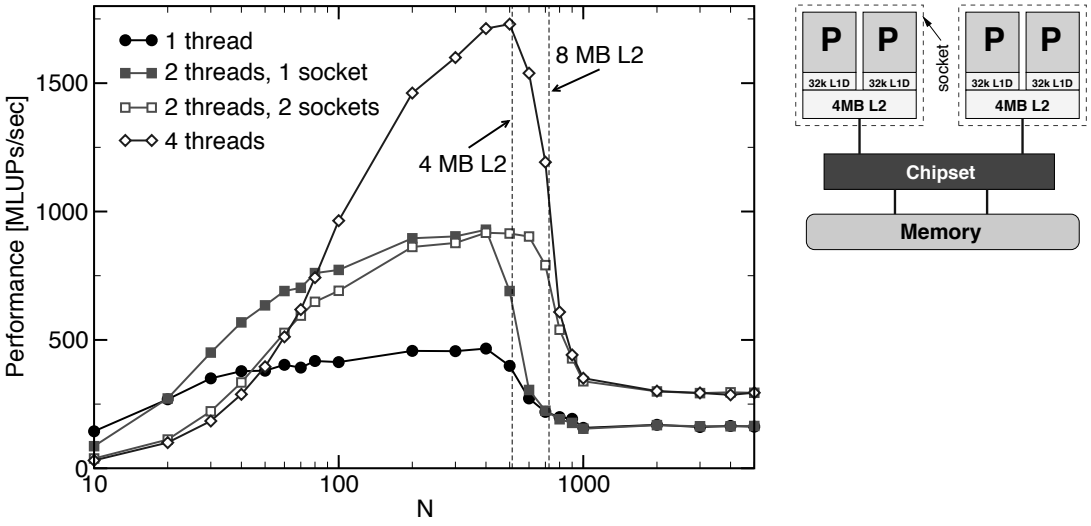

Figure 6.3: Performance versus problem size of a 2D Jacobi solver on an $N \times N$ lattice with OpenMP parallelization at one, two, and four threads on an Intel dual-core dual-socket Xeon 5160 node at 3.0 GHz (right). For two threads, there is a choice to place them on one socket (filled squares) or on different sockets (open squares).

at large N. Running two threads on the same socket has no benefit whatsoever in this limit because contention will occur on the frontside bus. Adding a second socket gives an 80% boost, as two FSBs become available. Scalability is not perfect because of deficiencies in the chipset and FSB architecture. Note that bandwidth scalability behavior on all memory hierarchy levels is strongly architecture-dependent; there are multicore chips on which it takes two or more threads to saturate the memory interface.

- With two threads, the maximum in-cache performance is the same, no matter whether they run on the same or on different sockets (filled vs. open squares). This indicates that the shared L2 cache can saturate the bandwidth demands of both cores in its group. Note, however, that three of the four loads in the Jacobi kernel are satisfied from L1 cache (see Section 3.3 for an analysis of bandwidth requirements). Performance prediction can be delicate under such conditions [M41, M44].

- At $N < 50$, the location of threads is more important for performance than their number, although the problem fits into the aggregate L1 caches. Using two sockets is roughly a factor of two slower in this case. The reason is that OpenMP overhead like the barrier synchronization at the end of the OpenMP worksharing loop dominates execution time for small N. See Section 7.2 for more information on this problem and how to ameliorate its consequences.

Explaining the performance characteristics of this bandwidth-limited algorithm requires a good understanding of the underlying parallel hardware, including issues specific to multicore chips. Future multicore designs will probably be more "anisotropic" (see, e.g., Figure 1.17) and show a richer, multilevel cache group structure, making it harder to understand performance features of parallel codes [M41].

6.3 Advanced OpenMP: Wavefront parallelization

Up to now we have only encountered problems where OpenMP parallelization was more or less straightforward because the important loops comprised independent iterations. However, in the presence of loop-carried dependencies, which also inhibit pipelining in some cases (see Section 1.2.3), writing a simple worksharing directive in front of a loop leads to unpredictable results. A typical example is the *Gauss–Seidel* algorithm, which can be used for solving systems of linear equations or boundary value problems, and which is also widely employed as a smoother component in multigrid methods. Listing 6.6 shows a possible serial implementation in three spatial dimensions. Like the Jacobi algorithm introduced in Section 3.3, this code solves for the steady state, but there are no separate arrays for the current and the next time step; a stencil update at (i, j, k) directly re-uses the three neighboring sites with smaller coordinates. As those have been updated in the very same sweep

Listing 6.6: A straightforward implementation of the Gauss–Seidel algorithm in three dimensions. The highlighted references cause loop-carried dependencies.

```
1  double precision, parameter :: osth=1/6.d0
2  do it=1,itmax   ! number of iterations (sweeps)
3    ! not parallelizable right away
4    do k=1,kmax
5      do j=1,jmax
6        do i=1,imax
7          phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
8                       + phi(i,j-1,k) + phi(i,j+1,k)
9                       + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
10       enddo
11     enddo
12   enddo
13 enddo
```

before, the Gauss–Seidel algorithm has fundamentally different convergence properties as compared to Jacobi (Stein-Rosenberg Theorem).

Parallelization of the Jacobi algorithm is straightforward (see the previous section) because all updates of a sweep go to a different array, but this is not the case here. Indeed, just writing a `PARALLEL DO` directive in front of the `k` loop would lead to race conditions and yield (wrong) results that most probably vary from run to run.

Still it is possible to parallelize the code with OpenMP. The key idea is to find a way of traversing the lattice that fulfills the dependency constraints imposed by the stencil update. Figures 6.4 and 6.5 show how this can be achieved: Instead of simply cutting the k dimension into chunks to be processed by OpenMP threads, a *wavefront* travels through the lattice in k direction. The dimension along which to parallelize is j, and each of the t threads $\mathrm{T}_0 \ldots \mathrm{T}_{t-1}$ gets assigned a consecutive chunk of size $j_{\max}/t$ along j. This divides the lattice into blocks of size $i_{\max} \times j_{\max}/t \times 1$. The very first block with the lowest k coordinate can only be updated by a single thread (T_0), which forms a "wavefront" by itself (W_1 in Figure 6.4). All other threads have to wait in a barrier until this block is finished. After that, the second wavefront (W_2) can commence, this time with two threads (T_0 and T_1), working on two blocks in parallel. After another barrier, W_3 starts with three threads, and so forth. W_t is the first wavefront to actually utilize all threads, ending the so-called *wind-up phase*. Some time (t wavefronts) before the sweep is complete, the *wind-down phase* begins and the number of working threads is decreased with each successive wavefront. The block with the largest k and j coordinates is finally updated by a single-thread (T_{t-1}) wavefront W_n again. In the end, $n = k_{\max} + t - 1$ wavefronts have traversed the lattice in a "pipeline parallel" pattern. Of those, $2(t-1)$ have utilized less than t threads. The whole scheme can thus only be load balanced if $k_{\max} \gg t$.

Listing 6.7 shows a possible implementation of this algorithm. We assume here for simplicity that $j_{\max}$ is a multiple of the number of threads. Variable `l` counts the

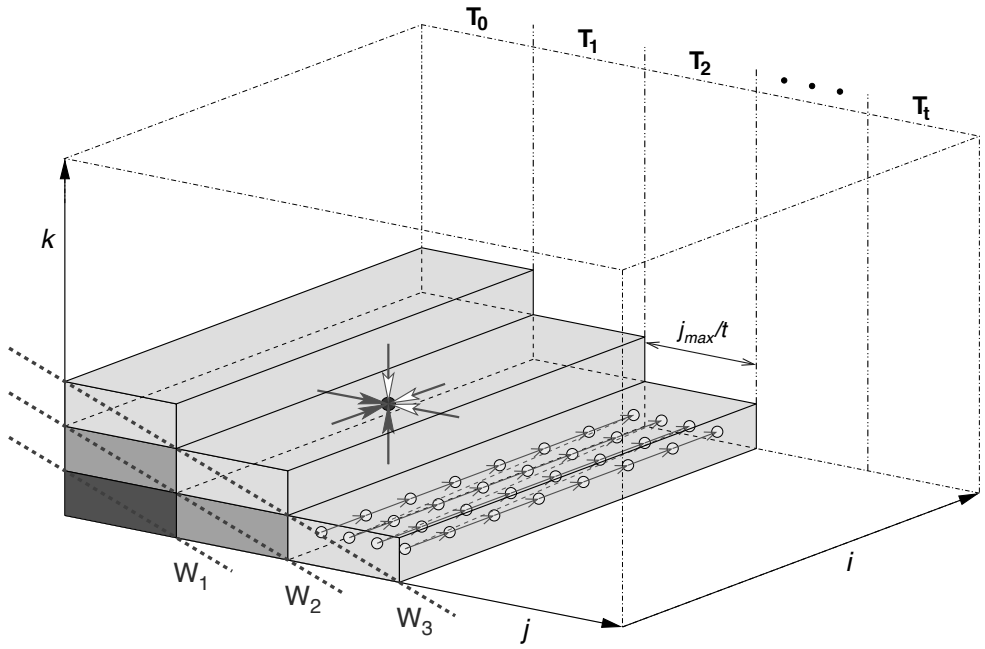

Figure 6.4: Pipeline parallel processing (PPP), a.k.a. wavefront parallelization, for the Gauss–Seidel algorithm in 3D (wind-up phase). In order to fulfill the dependency constraints of each stencil update, successive wavefronts (W_1,W_2,...,W_n) must be performed consecutively, but multiple threads can work in parallel on each individual wavefront. Up until the end of the wind-up phase, only a subset of all t threads can participate.

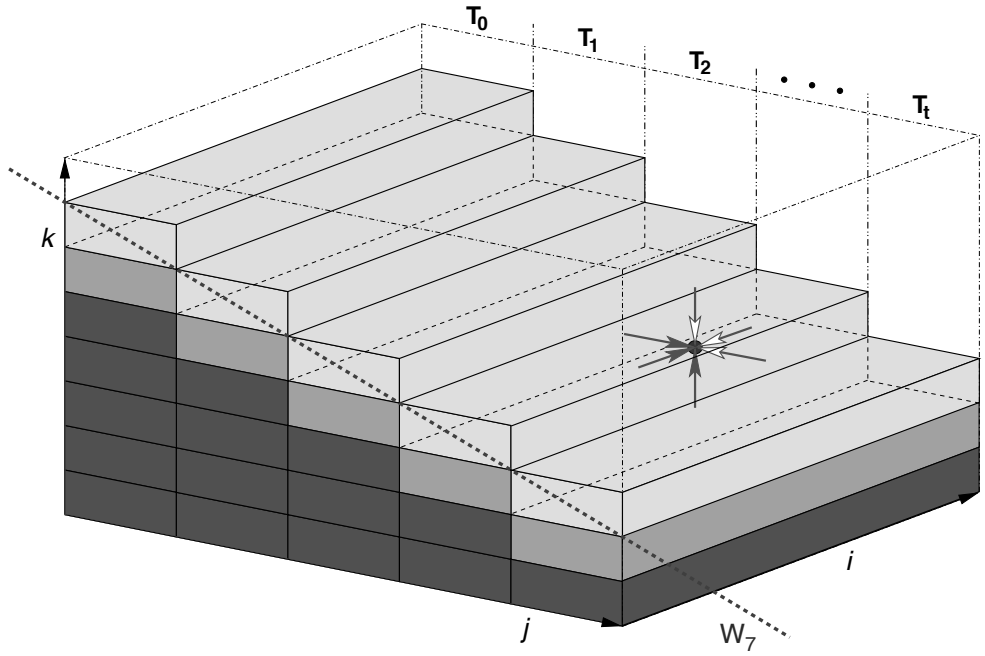

Figure 6.5: Wavefront parallelization for the Gauss–Seidel algorithm in 3D (full pipeline phase). All t threads participate. Wavefront W_7 is shown as an example.

Listing 6.7: The wavefront-parallel Gauss–Seidel algorithm in three dimensions. Loop-carried dependencies are still present, but threads can work in parallel.

```
1  !$OMP PARALLEL PRIVATE(k,j,i,jStart,jEnd,threadID)
2    threadID=OMP_GET_THREAD_NUM()
3  !$OMP SINGLE
4    numThreads=OMP_GET_NUM_THREADS()
5  !$OMP END SINGLE
6    jStart=jmax/numThreads*threadID
7    jEnd=jStart+jmax/numThreads ! jmax is amultiple of numThreads
8    do l=1,kmax+numThreads-1
9      k=l-threadID
10     if((k.ge.1).and.(k.le.kmax)) then
11       do j=jStart,jEnd        ! this is the actual parallel loop
12         do i=1,iMax
13           phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
14                        + phi(i,j-1,k) + phi(i,j+1,k)
15                        + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
16         enddo
17       enddo
18     endif
19 !$OMP BARRIER
20   enddo
21 !$OMP END PARALLEL
```

wavefronts, and `k` is the current k coordinate for each thread. The OpenMP barrier in line 19 is the point where all threads (including possible idle threads) synchronize after a wavefront has been completed.

We have ignored possible scalar optimizations like outer loop unrolling (see the order of site updates illustrated in the T_2 block of Figure 6.4). Note that the stencil update is unchanged from the original version, so there are still loop-carried dependencies. These inhibit fully pipelined execution of the inner loop, but this may be of minor importance if performance is bound by memory bandwidth. See Problem 6.6 for an alternative solution that enables pipelining (and thus vectorization).

Wavefront methods are of utmost importance in High Performance Computing, for massively parallel applications [L76, L78] as well as for optimizing shared-memory codes [O52, O59]. Wavefronts are a natural extension of the pipelining scheme to medium- and coarse-grained parallelism. Unfortunately, mainstream programming languages and parallelization paradigms do not as of now contain any direct support for it. Furthermore, although dependency analysis is a major part of the optimization stage in any modern compiler, very few current compilers are able to perform automatic wavefront parallelization [O59].

Note that stencil algorithms (for which Gauss–Seidel and Jacobi are just two simple examples) are core components in a lot of simulation codes and PDE solvers. Many optimization, parallelization, and vectorization techniques have been devised over the past decades, and there is a vast amount of literature available. More information can be found in the references [O60, O61, O62, O63].

Problems

For solutions see page 298 *ff*.

6.1 *OpenMP correctness*. What is wrong with this OpenMP-parallel Fortran 90 code?

```
1   double precision, dimension(0:360) :: a
2
3  !$OMP PARALLEL DO
4   do i=0,360
5     call f(dble(i)/360*PI, a(i))
6   enddo
7  !$OMP END PARALLEL DO
8
9   ...
10
11  subroutine f(arg, ret)
12    double precision :: arg, ret, noise=1.d-6
13    ret = SIN(arg) + noise
14    noise = -noise
15    return
16  end subroutine
```

6.2 *π by Monte Carlo*. The quarter circle in the first quadrant with origin at (0,0) and radius 1 has an area of $\pi/4$. Look at the random number pairs in $[0,1] \times [0,1]$. The probability that such a point lies inside the quarter circle is $\pi/4$, so given enough statistics we are able to calculate π using this so-called *Monte Carlo* method (see Figure 6.6). Write a parallel OpenMP program that performs this task. Use a suitable subroutine to get separate random number se-

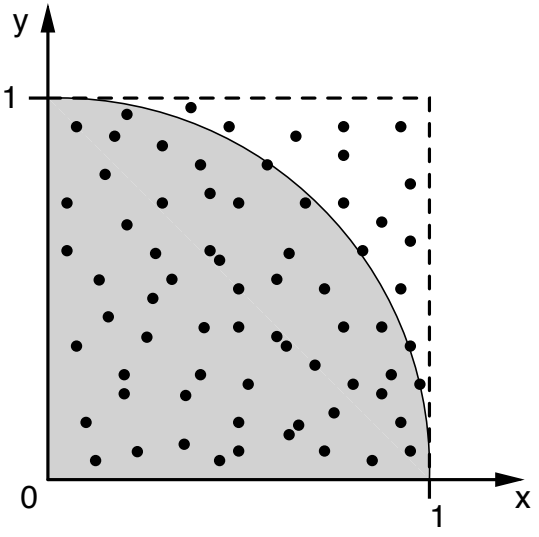

Figure 6.6: Calculating π by a Monte Carlo method (see Problem 6.2). The probability that a random point in the unit square lies inside the quarter circle is $\pi/4$.

quences for all threads. Make sure that adding more threads in a weak scaling scenario actually improves statistics.

6.3 *Disentangling critical regions*. In Section 6.1.4 we demonstrated the use of named critical regions to prevent deadlocks. Which simple modification of the example code would have made named the names obsolete?

6.4 *Synchronization perils*. What is wrong with this code?

```
!$OMP PARALLEL DO SCHEDULE(STATIC) REDUCTION(+:sum)
  do i=1,N
    call do_some_big_stuff(i,x)
    sum = sum + x
    call write_result_to_file(omp_get_thread_num(),x)
!$OMP BARRIER
  enddo
!$OMP END PARALLEL DO
```

6.5 *Unparallelizable?* (This problem appeared on the official OpenMP mailing list in 2007.) Parallelize the loop in the following piece of code using OpenMP:

```
double precision, parameter :: up = 1.00001d0
double precision :: Sn
double precision, dimension(0:len) :: opt

Sn = 1.d0
do n = 0,len
  opt(n) = Sn
  Sn = Sn * up
enddo
```

Simply writing an OpenMP worksharing directive in front of the loop will not work because there is a loop-carried dependency: Each iteration depends on the result from the previous one. The parallelized code should work independently of the OpenMP schedule used. Try to avoid — as far as possible — expensive operations that might impact serial performance.

To solve this problem you may want to consider using the `FIRSTPRIVATE` and `LASTPRIVATE` OpenMP clauses. `LASTPRIVATE` can only be applied to a worksharing loop construct, and has the effect that the listed variables' values are copied from the lexically last loop iteration to the global variable when the parallel loop exits.

6.6 *Gauss–Seidel pipelined*. Devise a reformulation of the Gauss–Seidel sweep (Listing 6.6) so that the inner loop does not contain loop-carried dependencies any more. Hint: Choose some arbitrary site from the lattice and visualize all other sites that can be updated at the same time, obeying the dependency constraints. What would be the performance impact of this formulation on cache-based processors and vector processors (see Section 1.6)?