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

```
do i=1,500
   a(i)=c*b(i)
enddo
```
P1

```
do i=501,1000
   a(i)=c*b(i)
enddo
```
P2

```
do i=1,1000
   a(i)=c*b(i)
enddo
```

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

Domain 1

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).
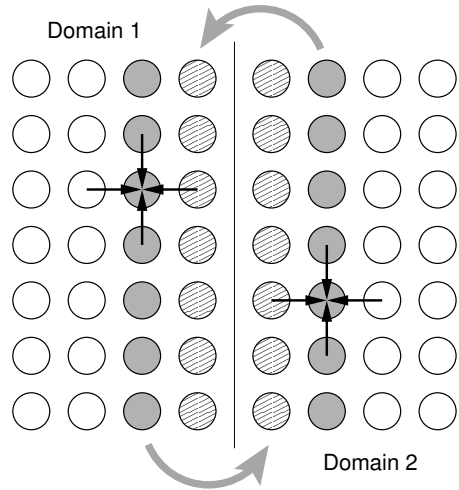
Domain 2

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 $\mathcal{O}(n(N-1))$ for stripe domains, whereas an optimal decomposition into square subdomains leads to a cost of $\mathcal{O}(2n(\sqrt{N}-1))$. Hence, for large N the optimal decomposition has an advantage in communication cost of $\mathcal{O}(2/\sqrt{N})$. 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-
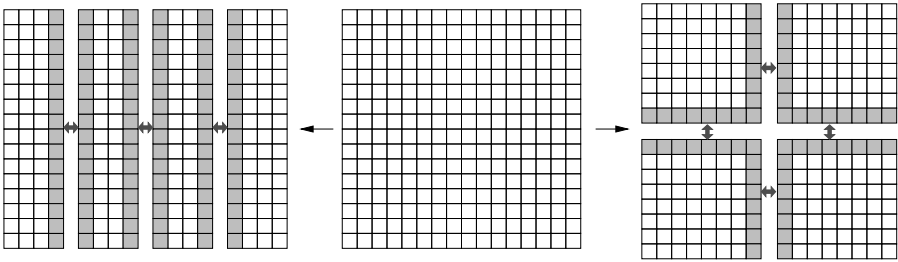
Figure 5.3: Domain decomposition of a two-dimensional Jacobi solver, which requires next-neighbor interactions. Cutting into stripes (left) is simple but incurs more communication than optimal decomposition (right). Shaded cells participate in network communication.

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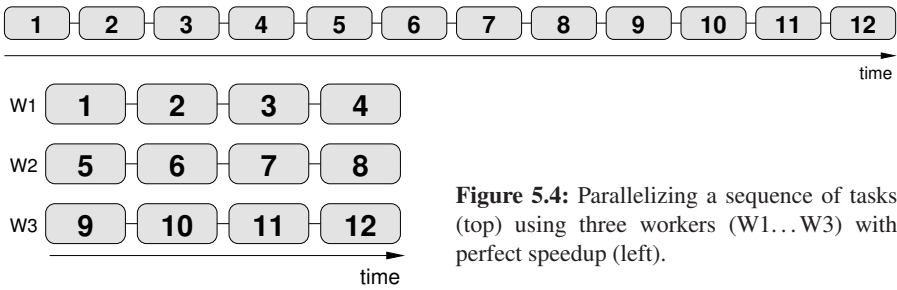
Figure 5.4: Parallelizing a sequence of tasks (top) using three workers (W1…W3) with perfect speedup (left).

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.
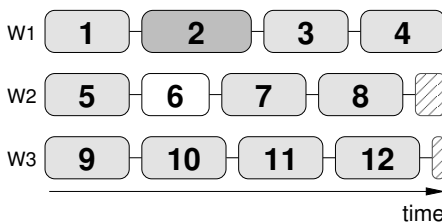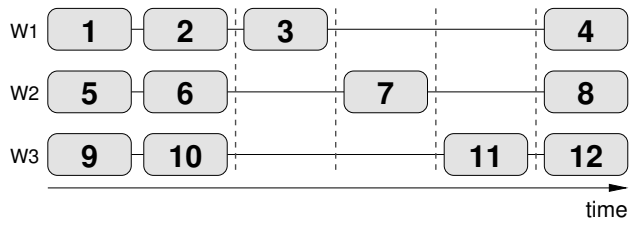

Figure 5.5: Some tasks executed by different workers at different speeds lead to *load imbalance*. Hatched regions indicate unused resources.

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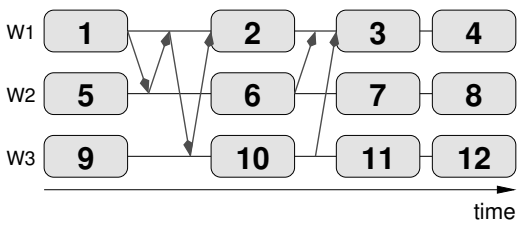
First we assume a fixed problem, which is to be solved by N workers. We normalize the single-worker (serial) runtime

$$T_f^s = s + p \tag{5.1}$$

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

$$T_f^p = s + \frac{p}{N} . \tag{5.2}$$

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

$$T_v^s = s + pN^\alpha . \tag{5.3}$$

Consequently, the parallel runtime is

$$T_v^p = s + pN^{\alpha-1} . \tag{5.4}$$

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

$$P_f^s = \frac{s+p}{T_f^s} = 1 , \tag{5.5}$$

as expected. Parallel performance is in this case

$$P_f^p = \frac{s+p}{T_f^p(N)} = \frac{1}{s + \frac{1-s}{N}} , \tag{5.6}$$

and application speedup ("scalability") is

$$S_f = \frac{P_f^p}{P_f^s} = \frac{1}{s + \frac{1-s}{N}} \quad \text{"Amdahl's Law"} \tag{5.7}$$

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

$$P_f^{sp} = \frac{p}{T_f^s} = p , \tag{5.8}$$

and parallel performance is

$$P_f^{pp} = \frac{p}{T_f^p(N)} = \frac{1-s}{s + \frac{1-s}{N}} . \tag{5.9}$$

Calculation of application speedup finally yields

$$S_f^p = \frac{P_f^{pp}}{P_f^{sp}} = \frac{1}{s + \frac{1-s}{N}} , \tag{5.10}$$

which is Amdahl's Law again. Strikingly, P_f^{pp} and $S_f^p(N)$ 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

$$P_v^s = \frac{s+p}{T_f^s} = 1 , \tag{5.11}$$

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

$$P_v^p = \frac{s + pN^\alpha}{T_v^p(N)} = \frac{s + (1-s)N^\alpha}{s + (1-s)N^{\alpha-1}} = S_v , \tag{5.12}$$

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

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

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 $s \gg p$.
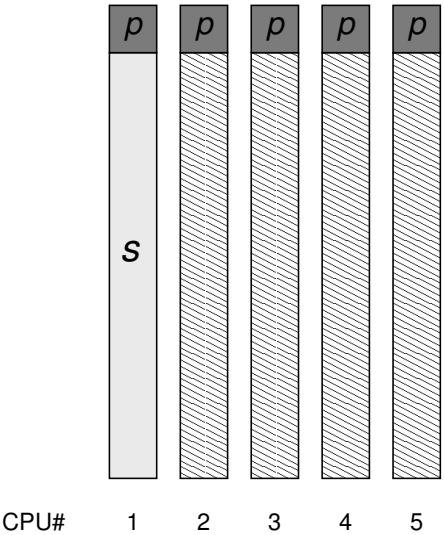
For $\alpha = 0$ this yields $1/(sN + (1 - s))$, which is the expected ratio for the Amdahl case and approaches zero with large N. For $\alpha = 1$ we get $s/N + (1 - s)$, which is also correct because the more CPUs are used the more CPU cycles are wasted, and, starting from $\varepsilon = s + p = 1$ 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,

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

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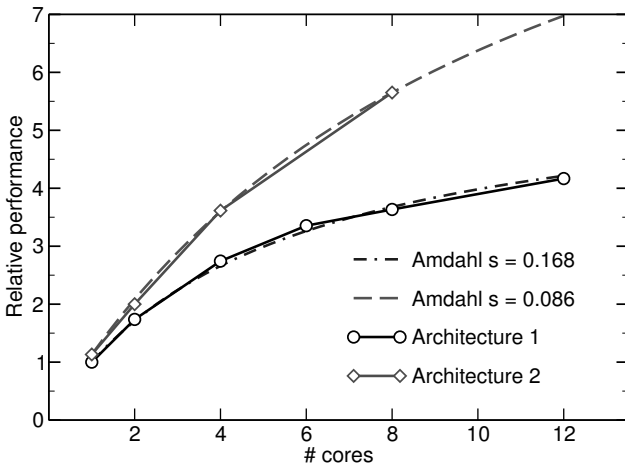
Figure 5.9: Performance of a benchmark code versus the number of processors (strong scaling) on two different architectures. Although the single-thread performance is nearly identical on both machines, the serial fraction s is much smaller on architecture 2, leading to superior strong scalability.

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,

$$S_v^c = \frac{1}{s + \frac{1-s}{N} + (\kappa + \lambda)N} \xrightarrow{N \gg 1} \frac{1}{(\kappa + \lambda)N} \, , \qquad (5.26)$$

  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

$$S_v^c = \frac{1}{s + \frac{1-s}{N} + \kappa + \lambda} \xrightarrow{N \gg 1} \frac{1}{s + \kappa + \lambda} \, . \qquad (5.27)$$

  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:

$$S_v^c = \frac{1}{s + \frac{1-s}{N} + \kappa N^{-\beta} + \lambda} \xrightarrow{N \gg 1} \frac{1}{s + \lambda} \, . \qquad (5.28)$$

  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):

$$S_v^c = \frac{s + pN}{s + p + \kappa + \lambda} \xrightarrow{N \gg 1} \frac{s + (1-s)N}{1 + \kappa + \lambda} \, . \qquad (5.29)$$

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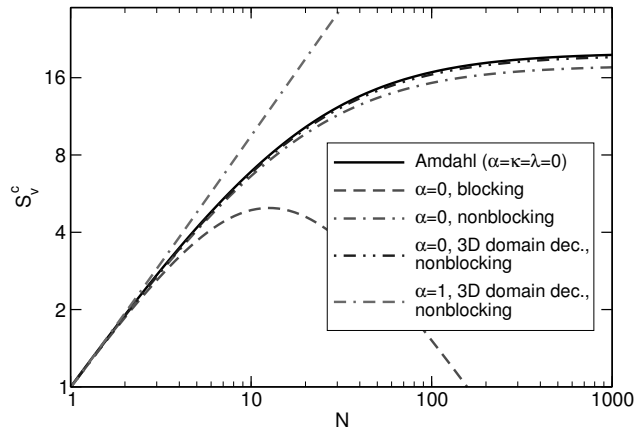Amdahl case, which is
shown for reference.

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
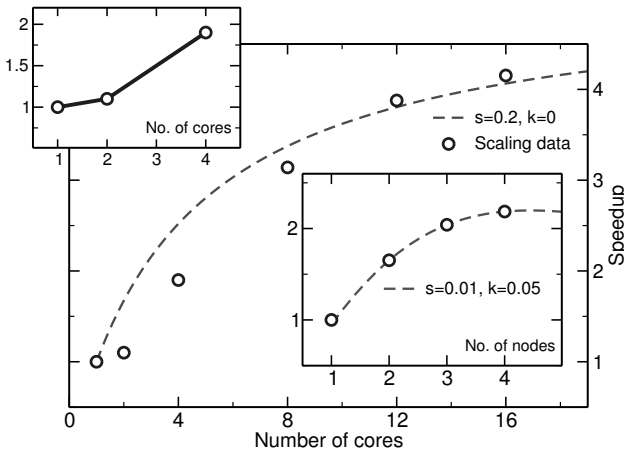
Figure 5.11: Speedup versus number of CPUs used for a hypothetical code on a hierarchical system with four CPUs per node. Depending on the chosen scaling baseline, fits to the model (5.26) can lead to vastly different results. Right inset: Scalability across nodes. Left inset: Scalability inside one 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-
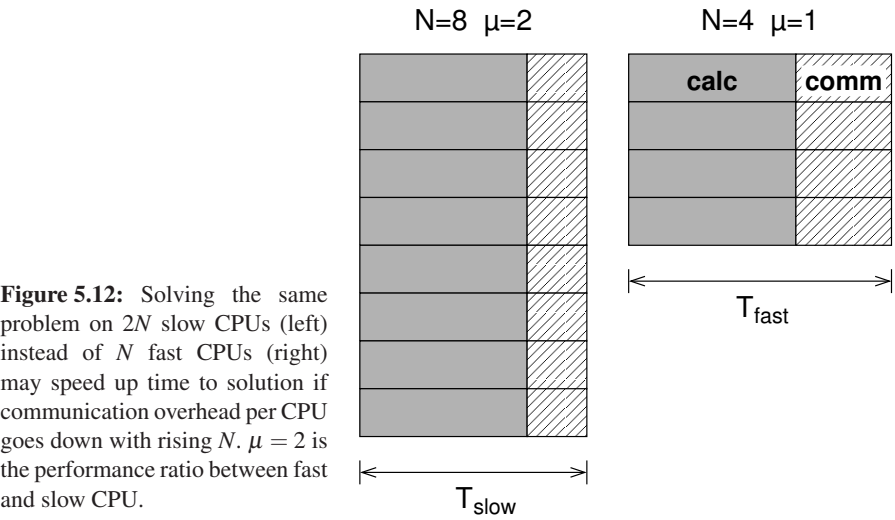
Figure 5.12: Solving the same problem on 2*N* slow CPUs (left) instead of *N* fast CPUs (right) may speed up time to solution if communication overhead per CPU goes down with rising *N*. $\mu = 2$ is the performance ratio between fast and slow CPU.

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:

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

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

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

We can distinguish several cases here:

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

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

  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

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

  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 $A_\mu^s(N) > 1$ 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 band-width overhead with increasing N, is large, especially if the latency is low:

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

  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

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

  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 $1.5 \lesssim A^{s,\max} \lesssim 2.5$.

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

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

which leads to the advantage function

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

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

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

Again we consider special cases:

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

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

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:

$$A_\mu^w(N) = \frac{1+\tilde\lambda}{1+\tilde\lambda/\mu} = \begin{cases} \xrightarrow{\mu\gg\tilde\lambda} 1+\tilde\lambda \\ \xrightarrow{\tilde\lambda\gg1} \mu \end{cases} \tag{5.43}$$

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 $1.33 \lesssim A^w \lesssim 1.6$ 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 $A_\mu^w \lesssim \mu$, 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,

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

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.
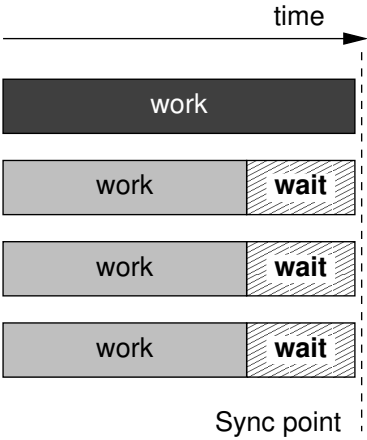
Figure 5.13: Load imbalance with few (one in this case) "laggers": A lot of resources are underutilized (hatched areas).
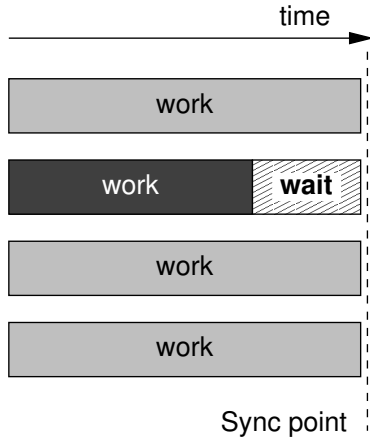
Figure 5.14: Load imbalance with few (one in this case) "speeders": Underutilization may be acceptable.

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].
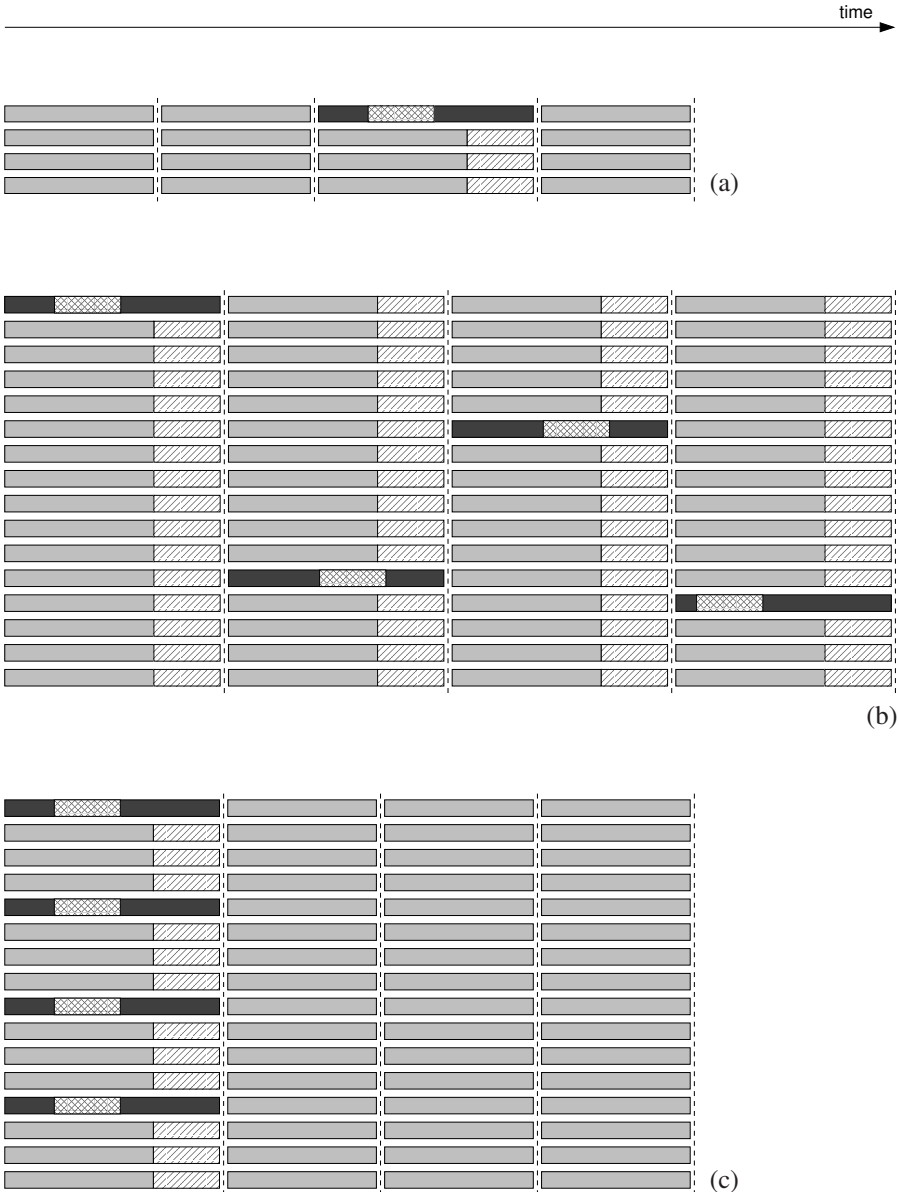
Figure 5.15: If an OS-related delay (cross-hatched boxes) occurs with some given probability per time, the impact on parallel performance may be small when the number of workers is small (a). Increasing the number of workers (here in a weak scaling scenario) increases the probability of the delay occurring 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.