Regionales Rechenzentrum Erlangen (RRZE)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Concepts of High Performance Computing

Georg Hager[*] Gerhard Wellein[†]

March 2008

[*]georg.hager@rrze.uni-erlangen.de
[†]gerhard.wellein@rrze.uni-erlangen.de

Contents

Part I

Architecture and programming of high performance computing systems

Preface to part I

In the past two decades the accessible compute power for numerical simulations has increased by more than three orders of magnitude. Fields like theoretical physics, theoretical chemistry, engineering science, and materials science to name just a few, have largely benefited from this development because the complex interactions often exceed the capabilities of analytical approaches and require sophisticated numerical simulations. The significance of these simulations, which may require large amounts of data and compute cycles, is frequently determined both by the choice of an appropriate numerical method or solver and the efficient use of modern computers. In particular, the latter point is widely underestimated and requires an understanding of the basic concepts of current (super)computer systems.

In this first part we present a comprehensive introduction to the architectural concepts and performance characteristics of state-of-the art high performance computers, ranging from desktop PCs over "poor man's" Linux clusters to leading edge supercomputers with thousands of processors. In Chapter 1 we discuss basic features of modern "commodity" microprocessors with a slight focus on Intel and AMD products. Vector systems (NEC SX8) are briefly touched. The main emphasis is on the various approaches used for on-chip parallelism and data access, including cache design, and the resulting performance characteristics.

In Chapter 2 we turn to the fundamentals of parallel computing. First we explain the basics and limitations of parallelism without specialization to a concrete method or computer system. Simple performance models are established which help to understand the most severe bottlenecks that will show up with parallel programming.

In terms of concrete manifestations of parallelism we then cover the principles of distributed-memory parallel computers, of which clusters are a variant. These systems are programmed using the widely accepted *message passing* paradigm where processes running on the compute nodes communicate via a library that sends and receives messages between them and thus serves as an abstraction layer to the hardware interconnect. Whether the program is run on an inexpensive cluster with bare Gigabit Ethernet or on a special-purpose vector system featuring a high-performance switch like the NEC IXS does not matter as far as the parallel programming paradigm is concerned. The *Message Passing Interface* (MPI) has emerged as the quasi-standard for message passing libraries. We introduce the most important MPI functionality using some simple examples in Chapter 3. As the network is often a performance-limiting aspect with MPI programming, some comments are made about basic performance characteristics of networks and the influence of bandwidth and latency on overall data transfer efficiency.

Price/performance considerations usually drive distributed-memory parallel systems into a particular direction of design. Compute nodes comprise multiple processors which share the same address space ("shared memory"). Two types of shared memory nodes are in wide use and will be discussed in Chapter 4: The uniform memory architecture (UMA) provides the same view/performance of physical memory for all processors and is used, e.g., in most current Intel-based systems. With the success of AMD Opteron CPUs in combination with HyperTransport technology the cache-coherent non-uniform memory architecture (ccNUMA) has gained increasing attention. The concept of having a single address space on a physically distributed memory (each processor can access local and remote memory) allows for scaling available memory bandwith but requires special care in programming and usage.

Common to all shared-memory systems are mechanisms for establishing cache coherence, i.e. ensuring consistency of the different views to data on different processors in presence of caches. One possible implementation of a cache coherence protocol is chosen to illustrate the potential bottlenecks that coherence traffic may impose. Finally, an introduction to the current standard for shared-memory scientific programming, OpenMP, is given.

1 Architecture and performance characteristics of modern microprocessor systems

1.1 Microprocessor architecture

In the “old days” of scientific supercomputing roughly between 1975 and 1995, leading-edge high performance systems were specially designed for the HPC market by companies like Cray, NEC, Thinking Machines, or Meiko. Those systems were way ahead of standard “commodity” computers in terms of performance and price. Microprocessors, which had been invented in the early 1970s, were only mature enough to hit the HPC market by the end of the 1980s, and it was not until the end of the 1990s that clusters of standard workstation or even PC-based hardware had become competitive at least in terms of peak performance. Today the situation has changed considerably. The HPC world is dominated by cost-effective, off-the-shelf systems with microprocessors that were not primarily designed for scientific computing. A few traditional supercomputer vendors act in a niche market. They offer systems that are designed for high application performance on the single CPU level as well as for highly parallel workloads. Consequently, the scientist is likely to encounter commodity clusters first and only advance to more specialized hardware as requirements grow. For this reason we will mostly be focused on microprocessor-based systems in this paper. *Vector computers* show a different programming paradigm which is in many cases close to the requirements of scientific computation, but they have become rare animals.

Microprocessors are probably the most complicated machinery that man has ever created. Understanding all inner workings of a CPU is out of the question for the scientist and also not required. It is helpful, though, to get a grasp of the high-level features in order to understand potential bottlenecks. Fig. 1.1 shows a very simplified block diagram of a modern microprocessor. The components that actually do “work” for a running application are the arithmetic units for floating-point (FP) and integer (INT) operations and make up for only a very small fraction of chip area. The rest consists of administrative logic that helps to feed those units with operands. All operands must reside in CPU registers which are generally divided into floating-point and integer (or “general purpose”) varieties. Typical CPUs nowadays have between 16 and 128 registers of both kinds. Load (LD) and store (ST) units handle instructions that transfer data to and from registers. Instructions are sorted into several *queues*, waiting to be executed, probably not in the order they were issued (see below). Finally, *caches* hold data and instructions to be (re-)used soon. A lot of additional logic, i.e. branch prediction, reorder buffers, data shortcuts, transaction queues etc. that we cannot touch upon here is built into modern processors. Vendors provide extensive documentation about those details [1, 2].

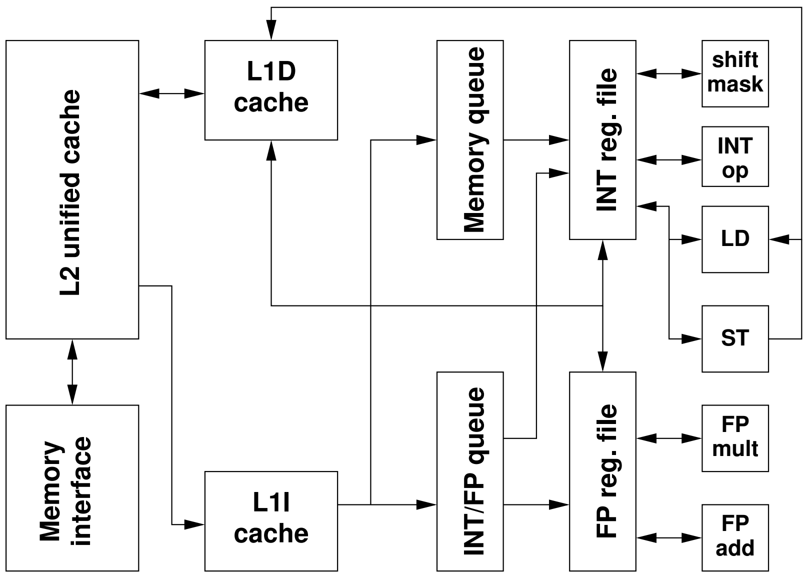


Figure 1.1: Simplified block diagram of a typical microprocessor.

1.1.1 Performance metrics and benchmarks

All those components can operate at some maximum speed called *peak performance*. Whether this limit can be reached with a specific application code depends on many factors and is one of the key topics of Section 6. Here we would like to introduce some basic performance metrics that can quantify the "speed" of a CPU. Scientific computing tends to be quite centric to floating-point data, usually with "double precision" (DP). The performance at which the FP units generate DP results for multiply and add operations is measured in *floating-point operations per second* (Flops/sec). The reason why more complicated arithmetic (divide, square root, trigonometric functions) is not counted here is that those are executed so slowly compared to add and multiply as to not contribute significantly to overall performance in most cases (see also Section 6). High performance software should thus try to avoid such operations as far as possible. At the time of writing, standard microprocessors feature a peak performance between 4 and 12 GFlops/sec.

As mentioned above, feeding arithmetic units with operands is a complicated task. The most important data paths from the programmer's point of view are those to and from the caches and main memory. The speed, or *bandwidth* of those paths is quantified in GBytes/sec. The GFlops/sec and GBytes/sec metrics usually suffice for explaining most relevant performance features of microprocessors.[1]

Fathoming the chief performance characteristics of a processor is one of the purposes of *low-level benchmarking*. A low-level benchmark is a program that tries to test some specific feature of the architecture like, e.g., peak performance or memory bandwidth. One of the most prominent examples is the *vector triad*. It comprises a nested loop, the inner level executing a combined vector multiply-add operation (see Listing 1.1). The purpose of this benchmark is to measure the performance of data transfers between memory and arithmetic units of a microprocessor. On the inner level, three *load streams* for arrays `B`, `C` and `D` and one *store stream* for `A` are active. Depending on `N`, this loop might execute in a very small time, which would be hard to measure. The outer loop thus repeats the triad `R` times so that execution time becomes large enough to be accurately measurable. In a real

[1]Please note that the "giga-" and "mega-" prefixes refer to a factor of 10^9 and 10^6, respectively, when used in conjunction with ratios like bandwidth or performance. Since recently, the prefixes "mebi-", "gibi-" etc. are frequently used to express quantities in powers of two, i.e. 1 MiB=2^{20} bytes.

Listing 1.1: Basic code fragment for the vector triad benchmark, including performance measurement.

```
double precision A(N),B(N),C(N),D(N),S,E,MFLOPS
S = get_walltime()
do j=1,R
  do i=1,N
    A(i) = B(i) + C(i) * D(i) ! 3 loads, 1 store
  enddo
  call dummy(A,B,C,D)          ! prevent loop interchange
enddo
E = get_walltime()
MFLOPS = R*N*2.d0/((E-S)*1.d6)  ! compute MFlop/sec rate
```

benchmarking situation one would choose R according to N so that the overall execution time stays roughly constant for different N.

Still the outer loop serves another purpose. In situations where N is small enough to fit into some processor cache, one would like the benchmark to reflect the performance of this cache. With R suitably chosen, startup effects become negligible and this goal is achieved.

The aim of the dummy() subroutine is to prevent the compiler from doing an obvious optimization: Without the call, the compiler might discover that the inner loop does not depend at all on the outer loop index j and drop the outer loop right away. The call to dummy(), which should reside in another compilation unit, fools the compiler into believing that the arrays may change between outer loop iterations. This effectively prevents the optimization described, and the cost for the call are negligible as long as N is not too small. Optionally, the call can be masked by an if statement whose condition is never true (a fact that must of course also be hidden from the compiler).

The MFLOPS variable is computed to be the MFlops/sec rate for the whole loop nest. Please note that the most sensible time measure in benchmarking is *wallclock time*. Any other “time” that the runtime system may provide, first and foremost the often-used CPU time, is prone to misinterpretation because there might be contributions from I/O, context switches, other processes etc. that CPU time cannot encompass. This is even more true for parallel programs (see Section 2).

Fig. 1.2 shows performance graphs for the vector triad obtained on current microprocessor and vector systems. For very small loop lengths we see poor performance no matter which type of CPU or architecture is used. On standard microprocessors, performance grows with N until some maximum is reached, followed by several sudden breakdowns. Finally, performance stays constant for very large loops. Those characteristics will be analyzed and explained in the following sections.

Vector processors (dotted line in Fig. 1.2) show very contrasting features. The low-performance region extends much farther than on microprocessors, but after saturation at some maximum level there are no breakdowns any more. We conclude that vector systems are somewhat complementary to standard CPUs in that they meet different domains of applicability. It may, however, be possible to optimize real-world code in a way that circumvents the low-performance regions. See Section 6 for details.

Low-level benchmarks are powerful tools to get information about the basic capabilities of a processor. However, they often cannot accurately predict the behavior of “real” appli-

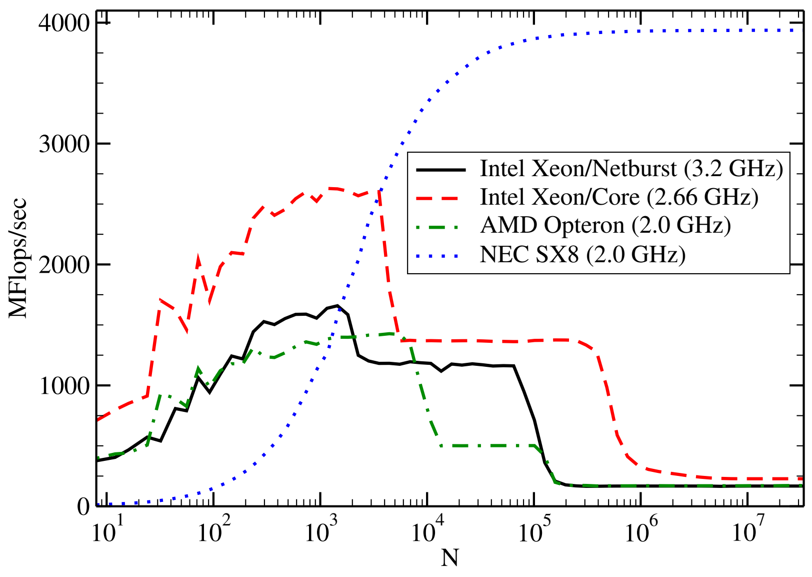


Figure 1.2: Serial vector triad performance data for different architectures. Note the entirely different performance characteristics of the vector processor (NEC SX8).

cation code. In order to decide whether some CPU or architecture is well-suited for some application (e.g., in the run-up to a procurement), the only safe way is to prepare *application benchmarks*. This means that an application code is used with input parameters that reflect as closely as possible the real requirements of production runs but lead to a runtime short enough for testing (no more than a few minutes). The decision for or against a certain architecture should always be heavily based on application benchmarking. Standard benchmark collections like the SPEC suite can only be rough guidelines.

1.1.2 Transistors galore: Moore's Law

Computer technology had been used for scientific purposes and, more specifically, for numerical calculations in physics long before the dawn of the desktop PC. For more than 30 years scientists could rely on the fact that no matter which technology was implemented to build computer chips, their "complexity" or general "capability" doubled about every 24 months. In its original form, *Moore's Law* stated that the number of components (transistors) on a chip required to hit the "sweet spot" of minimal manufacturing cost per component would increase at the indicated rate [3]. This has held true since the early 1960s despite substantial changes in manufacturing technologies that have happened over the decades. Amazingly, the growth in complexity has always roughly translated to an equivalent growth in compute performance, although the meaning of "performance" remains debatable as a processor is not the only component in a computer (see below for more discussion regarding this point).

Increasing chip transistor counts and clock speeds have enabled processor designers to implement many advanced techniques that lead to improved application performance. A multitude of concepts have been developed, including the following:

1. *Pipelined functional units*. Of all innovations that have entered computer design, pipelining is perhaps the most important one. By subdividing complex operations (like, e.g., floating point addition and multiplication) into simple components that can be executed using different functional units on the CPU, it is possible to increase instruction throughput, i.e. the number of instructions executed per clock cycle. Optimally pipelined execution leads to a throughput of one instruction per cycle. At

the time of writing, processor designs exist that feature pipelines with more than 30 stages. See the next section on page 9 for details.

2. *Superscalar architecture*. Superscalarity provides for an instruction throughput of more than one per cycle by using multiple, identical functional units concurrently. This is also called *instruction-level parallelism* (ILP). Modern microprocessors are up to six-way superscalar.

3. *Out-of-order execution*. If arguments to instructions are not available "on time", e.g. because the memory subsystem is too slow to keep up with processor speed, out-of-order execution can avoid *pipeline bubbles* by executing instructions that appear later in the instruction stream but have their parameters available. This improves instruction throughput and makes it easier for compilers to arrange machine code for optimal performance. Current out-of-order designs can keep hundreds of instructions in flight at any time, using a *reorder buffer* that stores instructions until they become eligible for execution.

4. *Larger caches*. Small, fast, on-chip memories serve as temporary data storage for data that is to be used again "soon" or that is close to data that has recently been used. This is essential due to the increasing gap between processor and memory speeds (see Section 1.2). Enlarging the cache size is always good for application performance.

5. *Advancement of instruction set design*. In the 1980s, a general move from the *CISC* to the *RISC* paradigm took place. In CISC (Complex Instruction Set Computing), a processor executes very complex, powerful instructions, requiring a large effort for decoding but keeping programs small and compact, lightening the burden on compilers. RISC (Reduced Instruction Set Computing) features a very simple instruction set that can be executed very rapidly (few clock cycles per instruction; in the extreme case each instruction takes only a single cycle). With RISC, the clock rate of microprocessors could be increased in a way that would never have been possible with CISC. Additionally, it frees up transistors for other uses. Nowadays, most computer architectures significant for scientific computing use RISC at the low level. Recently, Intel's Itanium line of processors have introduced *EPIC* (Explicitly Parallel Instruction Computing) which extends the RISC idea to incorporate information about parallelism in the instruction stream, i.e. which instructions can be executed in parallel. This reduces hardware complexity because the task of establishing instruction-level parallelism is shifted to the compiler, making out-of-order execution obsolete.

In spite of all innovations, processor vendors have recently been facing high obstacles in pushing performance limits to new levels. It becomes more and more difficult to exploit the potential of ever-increasing transistor numbers with standard, monolithic RISC processors. Consequently, there have been some attempts to simplify the designs by actually giving up some architectural complexity in favor of more straightforward ideas like larger caches, multi-core chips (see below) and even heterogeneous architectures on a single chip.

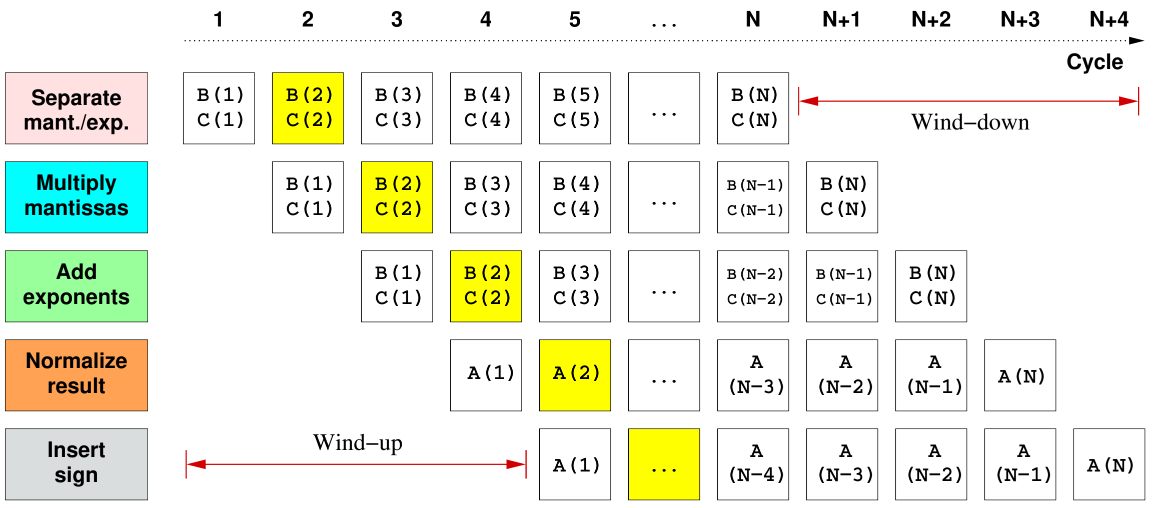


Figure 1.3: Timeline for a simplified floating-point multiplication pipeline that executes `A(:)=B(:)*C(:)`. One result is generated on each cycle after a five-cycle wind-up phase.

1.1.3 Pipelining

Pipelining in microprocessors serves the same purpose as assembly lines in manufacturing: Workers (functional units) do not have to know all details about the final product but can be highly skilled and specialized for a single task. Each worker executes the same chore over and over again *on different objects*, handing the half-finished product to the next worker in line. If it takes m different steps to finish the product, m products are continually worked on in different stages of completion. If all tasks are carefully tuned to take the same amount of time (the “time step”), all workers are continuously busy. At the end, one finished product per time step leaves the assembly line.

Complex operations like loading and storing data or performing floating-point arithmetic cannot be executed in a single cycle without excessive hardware requirements. Luckily, the assembly line concept is applicable here. The most simple setup is a “fetch–decode–execute” pipeline, in which each stage can operate independently of the others. While an instruction is being executed, another one is being decoded and a third one is being fetched from instruction (L1I) cache. These still complex tasks are usually broken down even further. The benefit of elementary subtasks is the potential for a higher clock rate as functional units can be kept simple. As an example, consider floating-point multiplication for which a possible division in to five subtasks is depicted in Fig. 1.3. For a vector product `A(:)=B(:)*C(:)`, execution begins with the first step, separation of mantissa and exponent, on elements `B(1)` and `C(1)`. The remaining four functional units are idle at this point. The intermediate result is then handed to the second stage while the first stage starts working on `B(2)` and `C(2)`. In the second cycle, only 3 out of 5 units are still idle. In the fifth cycle the pipeline has finished its so-called *wind-up* phase (in other words, the multiply pipeline has a *latency* of five cycles). From then on, all units are continuously busy, generating one result per cycle (having a pipeline *throughput* of one). When the first pipeline stage has finished working on `B(N)` and `C(N)`, the *wind-down* phase starts. Four cycles later, the loop is finished and all results have been produced.

In general, for a pipeline of depth (or latency) m, executing N independent, subsequent operations takes $N+m-1$ steps. We can thus calculate the expected speedup versus a

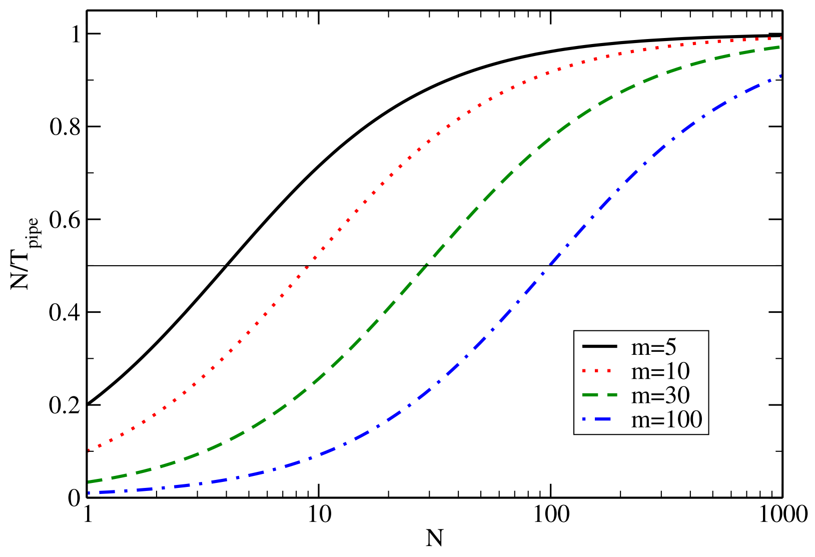


Figure 1.4: Pipeline throughput as a function of the number of independent operations. *m* is the pipeline depth.

general-purpose unit that needs *m* cycles to generate a single result,

$$\frac{T_\mathrm{seq}}{T_\mathrm{pipe}} = \frac{mN}{N+m-1} , \tag{1.1}$$

which is proportional to *m* for large *N*. The *throughput* is

$$\frac{N}{T_\mathrm{pipe}} = \frac{1}{1+\frac{m-1}{N}} , \tag{1.2}$$

approaching 1 for large *N* (see Fig. 1.4). It is evident that the deeper the pipeline the larger the number of independent operations must be to achieve reasonable throughput because of the overhead incurred by wind-up and wind-down phases.

One can easily determine how large *N* must be in order to get at least *p* results per cycle ($0 < p \leq 1$):

$$p = \frac{1}{1+\frac{m-1}{N_\mathrm{c}}} \quad \Longrightarrow \quad N_\mathrm{c} = \frac{(m-1)p}{1-p} . \tag{1.3}$$

For $p = 0.5$ we arrive at $N_\mathrm{c} = m-1$. Taking into account that present-day microprocessors feature overall pipeline lengths between 10 and 35 stages, we can immediately identify a potential performance bottleneck in codes that use short, tight loops. In superscalar or even vector processors the situation becomes even worse as multiple identical pipelines operate in parallel, leaving shorter loop lengths for each pipe.

Another problem connected to pipelining arises when very complex calculations like FP divide or even transcendental functions must be executed. Those operations tend to have very long latencies (several tens of cycles for square root or divide, often more than 100 for trigonometric functions) and are only pipelined to a small level or not at all so that stalling the instruction stream becomes inevitable (this leads to so-called *pipeline bubbles*). Avoiding such functions is thus a primary goal of code optimization. This and other topics related to efficient pipelining will be covered in Section 6.

Software pipelining

Note that although a depth of five is not unrealistic for a FP multiplication pipeline, executing a “real” code involves more operations like, e.g., loads, stores, address calculations, opcode fetches etc. that must be overlapped with arithmetic. Each operand of an instruction must find its way from memory to a register, and each result must be written out, observing all possible interdependencies. It is the compiler's job to arrange instructions in a way to make efficient use of all the different pipelines. This is most crucial for in-order architectures, but also required on out-of-order processors due to the large latencies for some operations.

As mentioned above, an instruction can only be executed if its operands are available. If operands are not delivered "on time" to execution units, all the complicated pipelining mechanisms are of no use. As an example, consider a simple scaling loop:

```
do i=1,N
  A(i) = s * A(i)
enddo
```

Seemingly simple in a high-level language, this loop transforms to quite a number of assembly instructions for a RISC processor. In pseudo-code, a naive translation could look like this:

```
loop:  load A(i)
       mult A(i) = A(i) * s
       store A(i)
       branch -> loop
```

Although the multiply operation can be pipelined, the pipeline will *stall* if the load operation on `A(i)` does not provide the data on time. Similarly, the store operation can only commence if the latency for `mult` has passed and a valid result is available. Assuming a latency of four cycles for `load`, two cycles for `mult` and two cycles for `store`, it is clear that above pseudo-code formulation is extremely inefficient. It is indeed required to *interleave* different loop iterations to bridge the latencies and avoid stalls:

```
loop:  load A(i+6)
       mult A(i+2) = A(i+2) * s
       store A(i)
       branch -> loop
```

Here we assume for simplicity that the CPU can issue all four instructions of an iteration in a single cycle and that the final branch and loop variable increment comes at no cost. Interleaving of loop iterations in order to meet latency requirements is called *software pipelining*. This optimization asks for intimate knowledge about processor architecture and insight into application code on the side of compilers. Often, heuristics are applied to arrive at "optimal" code.

It is, however, not always possible to optimally software pipeline a sequence of instructions. In the presence of *dependencies*, i.e., if a loop iteration depends on the result of some other iteration, there are situations when neither the compiler nor the processor hardware can prevent pipeline stalls. For instance, if the simple scaling loop from the previous example is modified so that computing `A(i)` requires `A(i+offset)`, with `offset` being either a constant that is known at compile time or a variable:

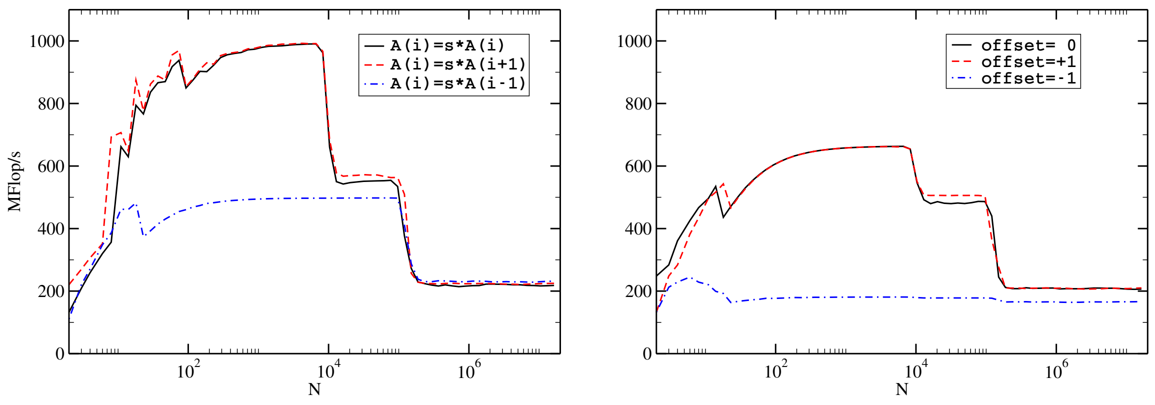


Figure 1.5: Influence of constant (left) and variable (right) offsets on the performance of a scaling loop. (AMD Opteron 2.0 GHz)

real dependency	pseudo dependency	general version
		`start=max(1,1-offset)`
`do i=2,N`	`do i=1,N-1`	`end=min(N,N-offset)`
`  A(i)=s*A(i-1)`	`  A(i)=s*A(i+1)`	`do i=start,end`
`enddo`	`enddo`	`  A(i)=s*A(i+offset)`
		`enddo`

As the loop is traversed from small to large indices, it makes a huge difference whether the offset is negative or positive. In the latter case we speak of a *pseudo dependency*, because `A(i+1)` is always available when the pipeline needs it for computing `A(i)`, i.e. there is no stall. In case of a real dependency, however, the pipelined computation of `A(i)` must stall until the result `A(i-1)` is completely finished. This causes a massive drop in performance as can be seen on the left of Fig. 1.5. The graph shows the peformance of above scaling loop in MFlops/sec versus loop length. The drop is clearly visible only in cache because of the small latencies of on-chip caches. If the loop length is so large that all data has to be fetched from memory, the impact of pipeline stalls is much less significant.

Although one might expect that it should make no difference whether the offset is known at compile time, the right graph in Fig. 1.5 shows that there is a dramatic peformance penalty for a variable offset. Obviously the compiler cannot optimally software pipeline or otherwise optimize the loop in this case. This is actually a common phenomenon, not exclusively related to software pipelining; any obstruction that hides information from the compiler can have a substantial performance impact.

There are issues with software pipelining linked to the use of caches. See below for details.

1.1.4 Superscalarity

If a processor is designed to be capable of executing more than one instruction or, more generally, producing more than one "result" per cycle, this goal is reflected in many of its design details:

- Multiple instructions can be fetched and decoded concurrently (4–6 nowadays).
- Address and other integer calculations are performed in multiple integer (add, mult, shift, mask) units (2–6).

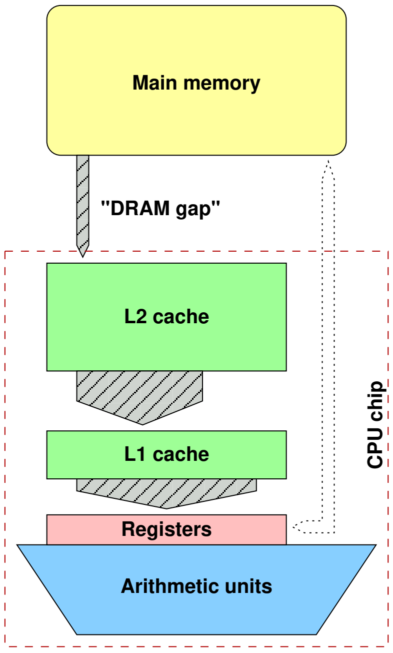
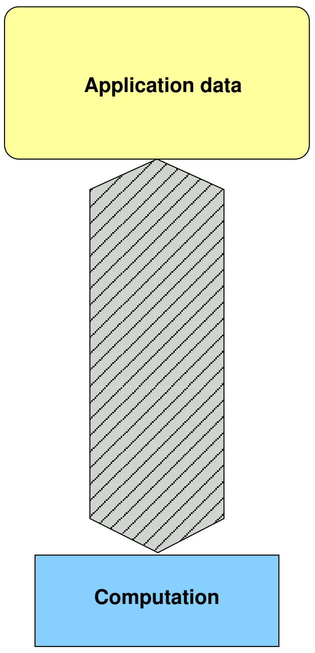


Figure 1.6: Left: simplified data-centric memory hierarchy in a cache-based microprocessor (direct access paths from registers to memory are not available on all architectures). There is usually a separate L1 cache for instructions. This model must be mapped to the data access requirements of an application (right).



- Multiple DP floating-point pipelines can run in parallel. Often there are one or two combined mult-add pipes that perform `a=b+c*d` with a throughput of one each.

- SIMD (*Single Instruction Multiple Data*) extensions are special instructions that issue identical operations on a whole array of integer or FP operands, probably in special registers. Whether SIMD will pay off on a certain code depends crucially on its recurrence structure and cache reuse. Examples are Intel's "SSE" and successors, AMD's "3dNow!" and the "AltiVec" extensions in Power and PowerPC processors. See Section 6.2.3 for details and examples.

- Caches are fast enough to sustain more than one DP load or store operation per cycle, and there are as many execution units for loads and stores available (2–4).

Out-of-order execution and compiler optimization must work together in order to fully exploit superscalarity. However, even on the most advanced architectures it is extremely hard for compiler-generated code to achieve a throughput of more than 2–3 instructions per cycle. This is why applications with very high demands for performance sometimes still resort to the use of assembly language.

1.2 Memory hierarchies

Data can be stored in a computer system in a variety of ways. As described above, CPUs feature a set of registers for instruction arguments that can be accessed without any delays. In addition there are one or more small but very fast *caches* that hold data items that have been used recently. *Main memory* is much slower but also much larger than cache. Finally, data can be stored on disk and copied to main memory as needed. This a is a complex memory hierarchy, and it is vital to understand how data transfer works between the different levels in order to identify performance bottlenecks. In the following we will concentrate on all levels from CPU to main memory (see Fig. 1.6).

1.2.1 Cache

Caches are low-capacity, high-speed memories that are nowadays usually integrated on the CPU die. The need for caches can be easily understood by the fact that data transfer rates to main memory are painfully slow compared to the CPU's arithmetic performance. At a peak performance of several GFlops/sec, *memory bandwidth*, i.e. the rate at which data can be transferred from memory to the CPU, is still stuck at a couple of GBytes/sec, which is entirely insufficient to feed all arithmetic units and keep them busy continuously (see Section 6 for a more thorough analysis). To make matters worse, in order to transfer a single data item (usually one or two DP words) from memory, an initial waiting time called *latency* occurs until bytes can actually flow. Often, latency is defined as the time it takes to transfer a zero-byte message. Memory latency is usually of the order of several hundred CPU cycles and is composed of different contributions from memory chips, the chipset and the processor. Although Moore's Law still guarantees a constant rate of improvement in chip complexity and (hopefully) performance, advances in memory performance show up at a much slower rate. The term *DRAM gap* has been coined for the increasing "distance" between CPU and memory in terms of latency and bandwidth.

Caches can alleviate the effects of the DRAM gap in many cases. Usually there are at least two *levels* of cache (see Fig. 1.6), and there are two L1 caches, one for instructions ("I-cache") and one for data. Outer cache levels are normally *unified*, storing data as well as instructions. In general, the "closer" a cache is to the CPU's registers, i.e. the higher its bandwidth and the lower its latency, the smaller it must be to keep administration overhead low. Whenever the CPU issues a read request ("load") for transferring a data item to a register, first-level cache logic checks whether this item already resides in cache. If it does, this is called a *cache hit* and the request can be satisfied immediately, with low latency. In case of a *cache miss*, however, data must be fetched from outer cache levels or, in the worst case, from main memory. If all cache entries are occupied, a hardware-implemented algorithm *evicts* old items from cache and replaces them with new data. The sequence of events for a cache miss on a write is more involved and will be described later. Instruction caches are usually of minor importance as scientific codes tend to be largely loop-based; I-cache misses are rare events.

Caches can only have a positive effect on performance if the data access pattern of an application shows some *locality of reference*. More specifically, data items that have been loaded into cache are to be used again "soon enough" to not have been evicted in the meantime. This is also called *temporal locality*. Using a simple model, we will now estimate the performance gain that can be expected from a cache that is a factor of τ faster than memory (this refers to bandwidth as well as latency; a more refined model is possible but does not lead to additional insight). Let β be the *cache reuse ratio*, i.e. the fraction of loads or stores that can be satisfied from cache because there was a recent load or store to the same address. Access time to main memory (again this includes latency and bandwidth) is denoted by T_m. In cache, access time is reduced to $T_\mathrm{c} = T_\mathrm{m}/\tau$. For some finite β, the average access time will thus be $T_\mathrm{av} = \beta T_\mathrm{c} + (1-\beta)T_\mathrm{m}$, and we calculate an access performance gain of

$$G(\tau,\beta) = \frac{T_\mathrm{m}}{T_\mathrm{av}} = \frac{\tau T_\mathrm{c}}{\beta T_\mathrm{c} + (1-\beta)\tau T_\mathrm{c}} = \frac{\tau}{\beta + \tau(1-\beta)} \;. \tag{1.4}$$

As Fig. 1.7 shows, a cache can only lead to a significant performance advantage if the hit ratio is relatively close to one.

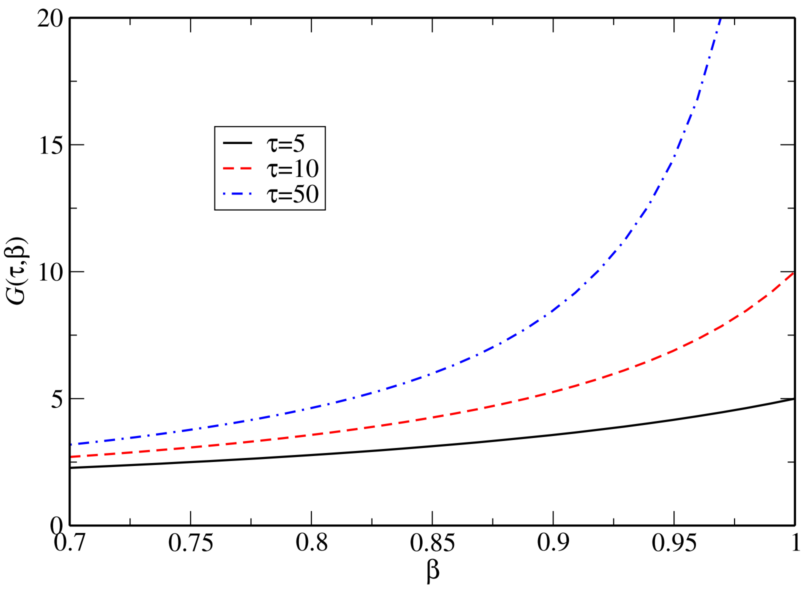


Figure 1.7: Performance gain vs. cache reuse ratio. τ parametrizes the speed advantage of cache vs. main memory.

Unfortunately, supporting temporal locality is not sufficient. Many applications show *streaming* patterns where large amounts of data are loaded to the CPU, modified and written back, without the potential of reuse “in time”. For a cache that only supports temporal locality, the reuse ratio β (see above) is zero for streaming. Each new load is expensive as an item has to be evicted from cache and replaced by the new one, incurring huge latency. In order to reduce the latency penalty for streaming, caches feature a peculiar organization into *cache lines*. All data transfers between caches and main memory happen on the cache line level. The advantage of cache lines is that the latency penalty of a cache miss occurs only on the first miss on an item belonging to a line. The line is fetched from memory as a whole; neighboring items can then be loaded from cache with much lower latency, increasing the *cache hit ratio* γ, not to be confused with the reuse ratio β. So if the application shows some *spatial locality*, i.e. if the probability of successive accesses to neighboring items is high, the latency problem can be significantly reduced. The downside of cache lines is that erratic data access patterns are not supported. On the contrary, not only does each load incur a miss and subsequent latency penalty, it also leads to the transfer of a whole cache line, polluting the memory bus with data that will probably never be used. The effective bandwidth available to the application will thus be very low. On the whole, however, the advantages of using cache lines prevail, and very few processor manufacturers have provided means of bypassing the mechanism.

Assuming a streaming application working on DP floating point data on a CPU with a cache line length of $L_c = 16$ words, spatial locality fixes the hit ratio at $\gamma = (16-1)/16 = 0.94$, a seemingly large value. Still it is clear that performance is governed by main memory bandwidth and latency — the code is *memory-bound*. In order for an application to be truly *cache-bound*, i.e. decouple from main memory so that performance is not governed by bandwidth or latency any more, γ must be large enough that the time it takes to process in-cache data becomes larger than the time for reloading it. If and when this happens depends of course on the details of the operations performed.

By now we can interpret the performance data for cache-based architectures on the vector triad in Fig. 1.2. At very small loop lengths, the processor pipeline is too long to be efficient. Wind-up and wind-down phases dominate and performance is poor. With growing `N` this effect becomes negligible, and as long as all four arrays fit into the innermost cache, performance saturates at a high value that is set by cache bandwidth and the ability of the CPU to issue load and store instructions. Increasing `N` a little more gives rise to a sharp drop in performance because the innermost cache is not large enough to hold all data. Second-level cache has usually larger latency but similar bandwidth to L1 so that the penalty is larger than expected. However, streaming data from L2 has the disadvantage that L1 now has to provide data for registers as well as continuously reload and evict cache lines from/to L2, which puts a strain on the L1 cache's bandwidth limits. This is why performance is usually hard to predict on all but the innermost cache level and main memory. For each cache level another performance drop is observed with rising `N`, until finally even the large outer cache is too small and all data has to be streamed from main memory. The size of the different caches is directly related to the locations of the bandwidth breakdowns. Section 6 will describe how to predict performance for simple loops from basic parameters like cache or memory bandwidths and the data demands of the application.

Storing data is a little more involved than reading. In presence of caches, if data to be written out already resides in cache, a *write hit* occurs. There are several possibilities for handling this case, but usually outermost caches work with a *write-back* strategy: The cache line is modified in cache and written to memory as a whole when evicted. On a *write miss*, however, cache-memory consistency dictates that the cache line in question must first be transferred from memory to cache before it can be modified. This is called *read for ownership* (RFO) and leads to the situation that a data write stream from CPU to memory uses the bus twice, once for all the cache line RFOs and once for evicting modified lines (the data transfer requirement for the triad benchmark code is increased by 25 % due to RFOs). Consequently, streaming applications do not usually profit from write-back caches and there is often a wish for avoiding RFO transactions. Some architectures provide this option, and there are generally two different strategies:

- *Nontemporal stores*. These are special store instructions that bypass all cache levels and write directly to memory. Cache does not get "polluted" by store streams that do not exhibit temporal locality anyway. In order to prevent excessive latencies, there is usually a *write combine buffer* of sorts that bundles a number of successive stores.

- *Cache line zero*. Again, special instructions serve to "zero out" a cache line and mark it as modified without a prior read. The data is written to memory when evicted. In comparison to nontemporal stores, this technique uses up cache space for the store stream. On the other hand it does not slow down store operations in cache-bound situations.

Both can be applied by the compiler and hinted at by the programmer by means of directives. In very simple cases compilers are able to apply those instructions automatically in their optimization stages, but one must take care to not slow down a cache-bound code by using nontemporal stores, rendering it effectively memory-bound.

1.2.2 Cache mapping

So far we have implicitly assumed that there is no restriction on which cache line can be associated with which memory locations. A cache design that follows this rule is called *fully associative*. Unfortunately it is quite hard to build large, fast and fully associative caches because of large bookkeeping overhead: For each cache line the cache logic must

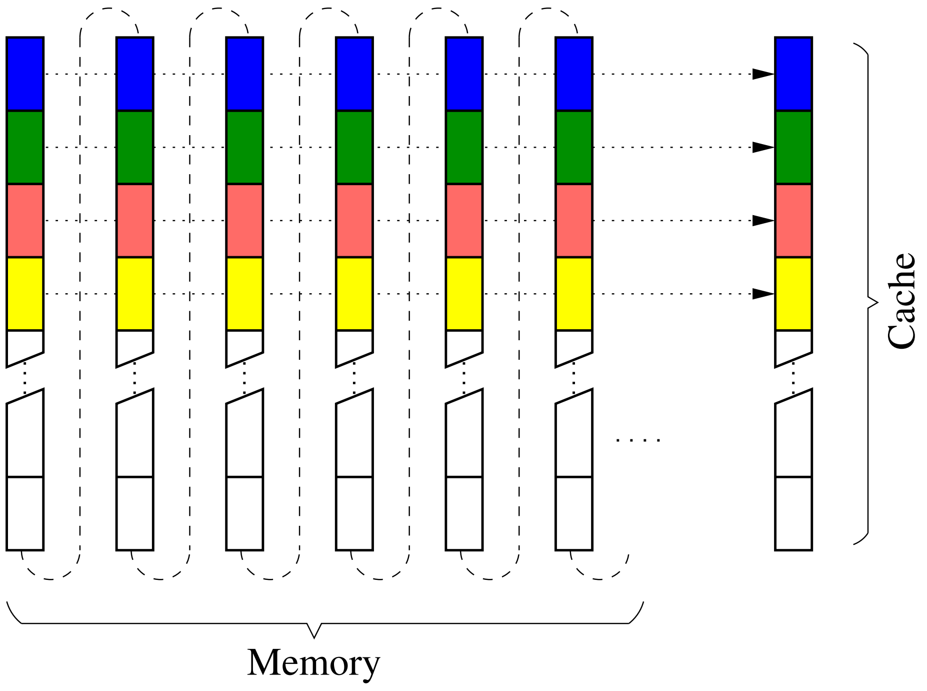


Figure 1.8: In a direct-mapped cache, memory locations which lie a multiple of the cache size apart are mapped to the same cache line (shaded boxes).

store its location in the CPU's address space, and each memory access must be checked against the list of all those addresses. Furthermore, the decision which cache line to replace next if the cache is full is made by some algorithm implemented in hardware. Usually, there is a *least recently used* (LRU) strategy that makes sure only the "oldest" items are evicted.

The most straightforward simplification of this expensive scheme consists in a *direct-mapped cache* which maps the full cache size repeatedly into memory (see Fig. 1.8). Memory locations that lie a multiple of the cache size apart are always mapped to the same cache line, and the cache line that corresponds to some address can be obtained very quickly by masking out the most significant bits. Moreover, an algorithm to select which cache line to evict is pointless. No hardware and no clock cycles need to be spent for it.

The downside of a direct-mapped cache is that it is disposed toward *cache thrashing*, which means that cache lines are loaded into and evicted from cache in rapid succession. This happens when an application uses many memory locations that get mapped to the same cache line. A simple example would be a "strided" triad code for DP data:

```
do i=1,N,CACHE_SIZE/8
  A(i) = B(i) + C(i) * D(i)
enddo
```

By using the cache size in units of DP words as a stride, successive loop iterations hit the same cache line so that *every* memory access generates a cache miss. This is different from a situation where the stride is equal to the line length; in that case, there is still some (albeit small) N for which the cache reuse is 100 %. Here, the reuse fraction is exactly zero no matter how small N may be.

To keep administrative overhead low and still reduce the danger of cache thrashing, a *set-associative cache* is divided into *m* direct-mapped caches equal in size, so-called *ways*.

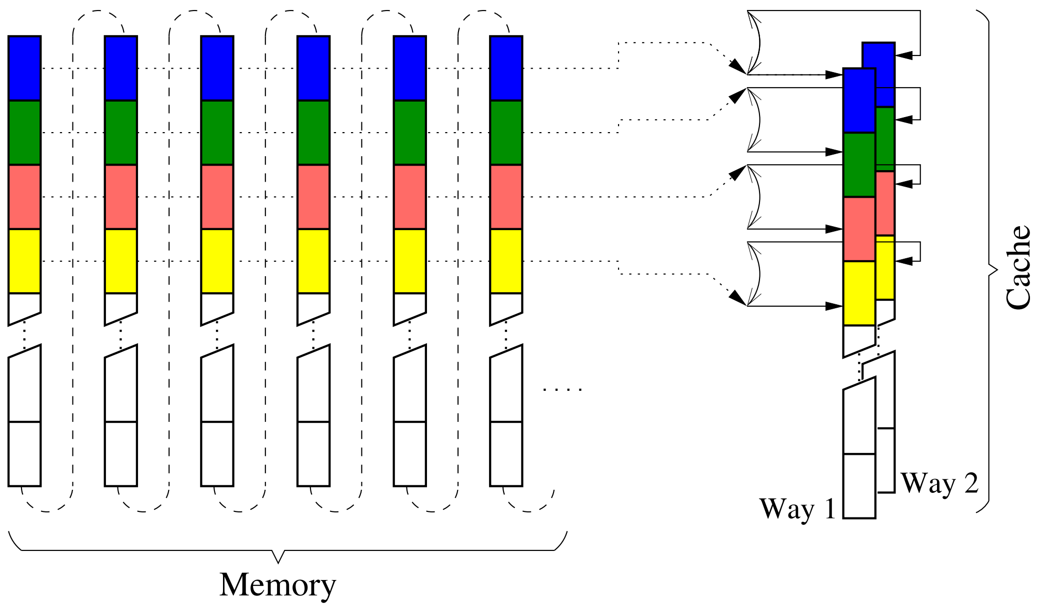


Figure 1.9: In an m-way set-associative cache, memory locations which are located a multiple of $\frac{1}{m}$th of the cache size apart can be mapped to either of m cache lines (here shown for $m = 2$).

The number of ways m is the number of different cache lines a memory address can be mapped to (see Fig. 1.9 for an example of a two-way set-associative cache). On each memory access, the hardware merely has to determine which way the data resides in or, in the case of a miss, which of the m possible cache lines should be evicted.

For each cache level the tradeoff between low latency and prevention of thrashing must be considered by processor designers. Innermost (L1) caches tend to be less set-associative than outer cache levels. Nowadays, set-associativity varies between two- and 16-way. Still, the *effective cache size*, i.e. the part of the cache that is actually useful for exploiting spatial and temporal locality in an application code could be quite small, depending on the number of data streams, their strides and mutual offsets. See Section 6 for examples.

1.2.3 Prefetch

Although exploiting spatial locality by the introduction of cache lines improves cache efficiency a lot, there is still the problem of latency on the first miss. Fig. 1.10 visualizes the situation for a simple vector norm kernel:

```
do i=1,N
  S = S + A(i)*A(i)
enddo
```

There is only one load stream in this code. Assuming a cache line length of four elements, three loads can be satisfied from cache before another miss occurs. The long latency leads to long phases of inactivity on the memory bus.

Making the lines very long will help, but will also slow down applications with erratic access patterns even more. As a compromise one has arrived at typical cache line lengths between 64 and 128 bytes (8–16 DP words). This is by far not big enough to get around latency, and streaming applications would suffer not only from insufficient bandwidth but

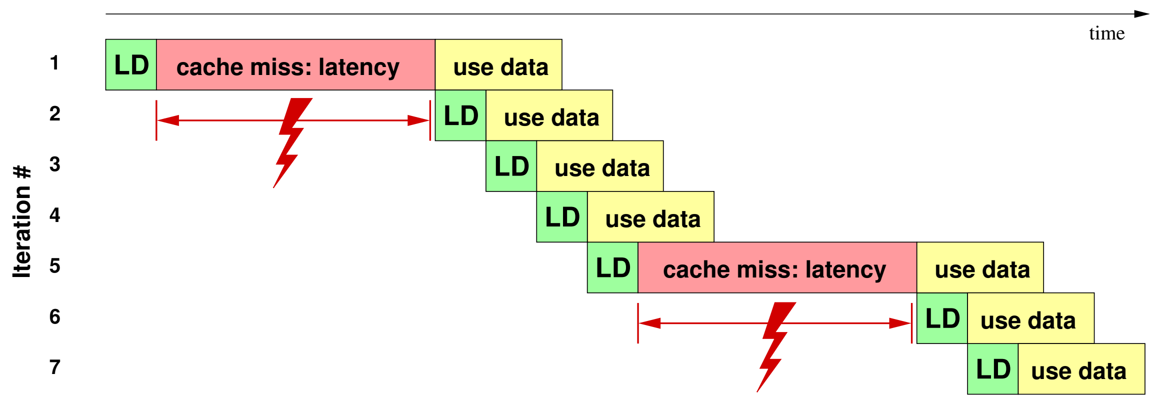


Figure 1.10: Timing diagram on the influence of cache misses and subsequent latency penalties for a vector norm loop. The penalty occurs on each new miss.

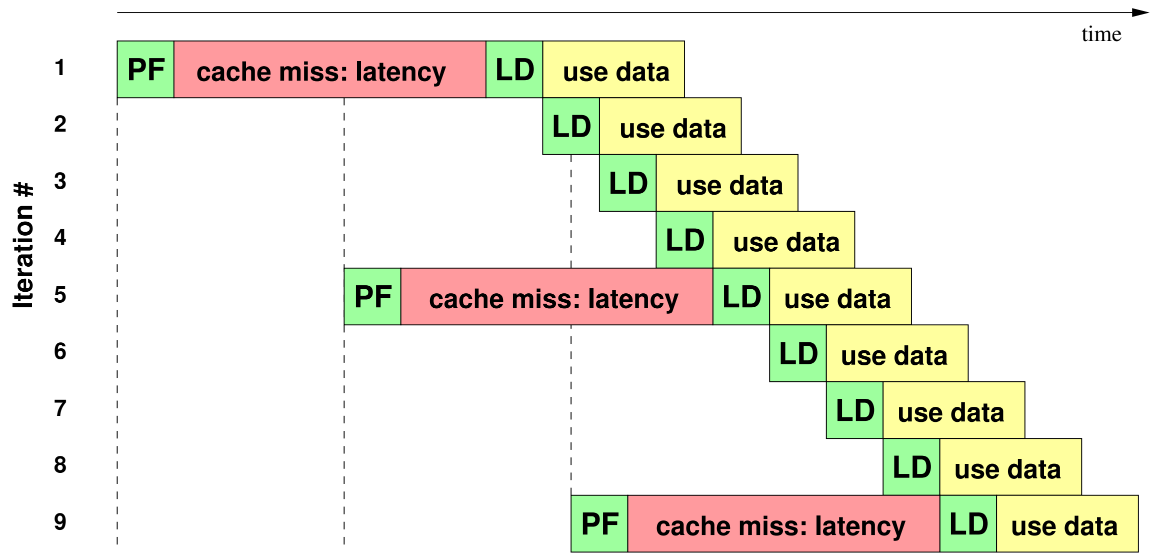


Figure 1.11: Computation and data transfer can be overlapped much better with prefetching. In this example, two outstanding prefetches are required to hide latency completely.

also from low memory bus utilization. Assuming a typical commodity system with a memory latency of 100 ns and a bandwidth of 4 GBytes/sec, a single 128-byte cache line transfer takes 32 ns, so 75 % of the potential bus bandwidth is unused. Obviously, latency has an even more severe impact on performance than bandwidth.

The latency problem can be solved in many cases, however, by *prefetching*. Prefetching supplies the cache with data ahead of the actual requirements of an application. The compiler can do this by interleaving special instructions with the software pipelined instruction stream that "touch" cache lines early enough to give the hardware time to load them into cache (see Fig. 1.11) asynchronously. This assumes there is the potential of asynchronous memory operations, a prerequisite that is to some extent true for current architectures. As an alternative, some processors feature a *hardware prefetcher* that can detect regular access patterns and tries to read ahead application data, keeping up the continuous data stream and hence serving the same purpose as prefetch instructions. Whichever strategy is used, it must be emphasized that prefetching requires resources that are limited by design. The memory subsystem must be able to sustain a certain number of *outstanding prefetch* *operations*, i.e. pending prefetch requests, or else the memory pipeline will stall and latency cannot be hidden completely. We can estimate the number of outstanding prefetches required for hiding the latency completely: If T_l is the latency and B is the bandwidth, the transfer of a whole line of length L_c DP words takes a time of

$$T = T_\mathrm{l} + \frac{8L_\mathrm{c}}{B}\ . \tag{1.5}$$

One prefetch operation must be initiated per cache line transfer, and the number of cache lines that can be transferred during time T is the number of prefetches P that the processor must be able to sustain (see Fig. 1.11):

$$P = \frac{T}{8L_\mathrm{c}/B} \tag{1.6}$$

As an example, for a cache line length of 128 bytes (16 DP words), $B = 6.4$ GBytes/sec and $T_\mathrm{l} = 140$ ns we get $P = 160/20 = 8$ outstanding prefetches. If this requirement cannot be met, latency will not be hidden completely and the full memory bandwidth will not be utilized. On the other hand, an application that executes so many floating-point operations on the cache line data that they cannot be hidden behind the transfer will not be limited by bandwidth and put less strain on the memory subsystem (see Sect. 7.1 for appropriate performance models). In such a case, fewer outstanding prefetches will suffice.

Applications with heavy demands on bandwidth can easily overstrain the prefetch mechanism. A second processor core using a shared path to memory can sometimes provide for the missing prefetches, yielding a slight bandwidth boost (see Sect. 1.3 for more information on multi-core design). In general, if streaming-style main memory access is unavoidable, a good programming guideline is to try to establish long continuous data streams.

Finally, a note of caution is in order. Figs. 1.10 and 1.11 stress the role of prefetching for hiding latency, but the effects of bandwidth limitations are ignored. It should be clear that prefetching cannot enhance available memory bandwidth, although the transfer time for a single cache line is dominated by latency.

1.3 Multi-core processors

In recent years it has become increasingly clear that, although Moore's Law is still valid and will be at least for the next decade, standard microprocessors are starting to hit the "heat barrier": Switching and leakage power of several-hundred-million-transistor chips are so large that cooling becomes a primary engineering effort and a commercial concern. On the other hand, the necessity of an ever-increasing clock frequency is driven by the insight that architectural advances and growing cache sizes alone will not be sufficient to keep up the one-to-one correspondence of Moore's Law with application performance.

Processor vendors are looking for a way out of this dilemma in the form of *multi-core* designs. The technical motivation behind multi-core is based on the observation that power dissipation of modern CPUs is proportional to the third power of clock frequency f_c (actually it is linear in f_c and quadratic in supply voltage V_cc, but a decrease in f_c allows for a proportional decrease in V_cc). Lowering f_c and thus V_cc can therefore dramatically reduce power dissipation. Assuming that a single core with clock frequency f_c has a performance of p and a power dissipation of W, some relative change in performance $\varepsilon_p = \Delta p/p$ will

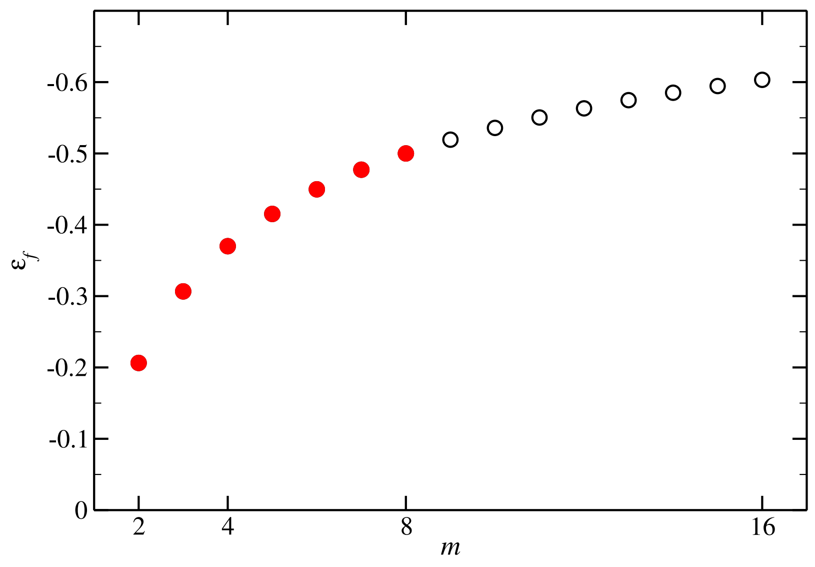


Figure 1.12: Relative frequency reduction required to keep a given power envelope versus number of cores on a multi-core chip. The filled dots represent available technology at the time of writing.

emerge for a relative clock change of $\varepsilon_f = \Delta f_c / f_c$. All other things being equal, $|\varepsilon_f|$ is an upper limit for $|\varepsilon_p|$, which in turn will depend on the applications considered. Power dissipation is

$$W + \Delta W = (1+\varepsilon_f)^3 W \; . \tag{1.7}$$

Reducing clock frequency opens the possibility to place more than one CPU core on the same die while keeping the same *power envelope* as before. For m cores, this condition is expressed as

$$(1+\varepsilon_f)^3 m = 1 \quad \Longrightarrow \quad \varepsilon_f = m^{-1/3} - 1 \tag{1.8}$$

Fig. 1.12 shows the required relative frequency reduction with respect to the number of cores. The overall performance of the multi-core chip,

$$p_m = (1+\varepsilon_p) p m \; , \tag{1.9}$$

should at least match the single-core performance so that

$$\varepsilon_p > \frac{1}{m} - 1 \tag{1.10}$$

is a limit on the performance penalty for a relative clock frequency reduction of ε_f that should be observed for multi-core to stay useful.

Of course it is not easily possible to grow the CPU die by a factor of m with a given manufacturing technology. Hence the most simple way to multi-core is to place separate CPU dies in a common package. At some point advances in manufacturing technology, i.e. smaller structure lengths, will then enable the integration of more cores on a die. Additionally, some compromises regarding the single-core performance of a multi-core chip with respect to the previous generation will be made so that the number of transistors per core will go down as will the clock frequency. Some manufacturers have even adopted a more radical approach by designing new, much simpler cores, albeit at the cost of introducing new programming paradigms.

Finally, the over-optimistic assumption (1.9) that m cores show m times the performance of a single core will only be valid in the rarest of cases. Nevertheless, multi-core has by now been adopted by all major processor manufacturers. There are, however, significant differences in how the cores in a package can be arranged to get good performance. Caches can be shared or exclusive to each core, the memory interface can be on- or off-chip, fast data paths between the cores' caches may or may not exist, etc.

The most important conclusion one must draw from the multi-core transition is the absolute demand for parallel programming. As the single core performance will at best stagnate over the years, getting more speed for free through Moore's law just by waiting for the new CPU generation does not work any more. The following section outlines the principles and limitations of parallel programming. More details on dual- and multi-core designs will be revealed in the section on shared-memory programming on page 43.

In order to avoid any misinterpretation we will always use the terms "core", "CPU" and "processor" synonymously.

2 Parallel computing

We speak of *parallel computing* whenever a number of processors (cores) solve a problem in a cooperative way. All modern supercomputer architectures depend heavily on parallelism, and the number of CPUs in large-scale supercomputers increases steadily. A common measure for supercomputer "speed" has been established by the Top500 list [4] that is published twice a year and ranks parallel computers based on their performance in the LINPACK benchmark that solves a dense system of linear equations of unspecified size. Although LINPACK is not generally accepted as a good metric because it covers only a single architectural aspect (peak performance), the list can still serve as an important indicator for trends in supercomputing. The main tendency is clearly visible from a comparison of processor number distributions in Top500 systems (see Fig. 2.1): Top of the line HPC systems do not rely on Moore's Law alone for performance but *parallelism* becomes more important every year. This trend will accelerate even more by the advent of multi-core processors — the June 2006 list contains only very few dual-core systems (see also Section 1.3).

2.1 Basic principles of parallelism

Parallelization is the process of formulating a problem in a way that lends itself to concurrent execution by several "execution units" of some kind. This is not only a common problem in computing but also in many other areas like manufacturing, traffic flow and even business processes. Ideally, the execution units (workers, assembly lines, border crossings, CPUs,...) are initially given some amount of work to do which they execute in exactly the same amount of time. Therefore, using N workers, a problem that takes a time T to be solved sequentially will now take only T/N. We call this a *speedup* of N.

Of course, reality is not perfect and some concessions will have to be made. Not all workers might execute at the same speed (see Fig. 2.2), and the tasks might not be easily partitionable into N equal chunks. 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 (Fig. 2.5). Finally, the parallel workflow may require some communication between workers, adding some overhead that would not be present in the serial case (Fig. 2.6). All these effects can impose limits on speedup. How well a task can be parallelized is usually quantified by some *scalability* metric.

In the following sections we will first identify some of the more common strategies of parallelization, independent of the hardware and software that is at the programmer's disposal, and then investigate parallelism on a theoretical level: Simple performance models will be derived that allow insight into the most prominent limiting factors for scalability.

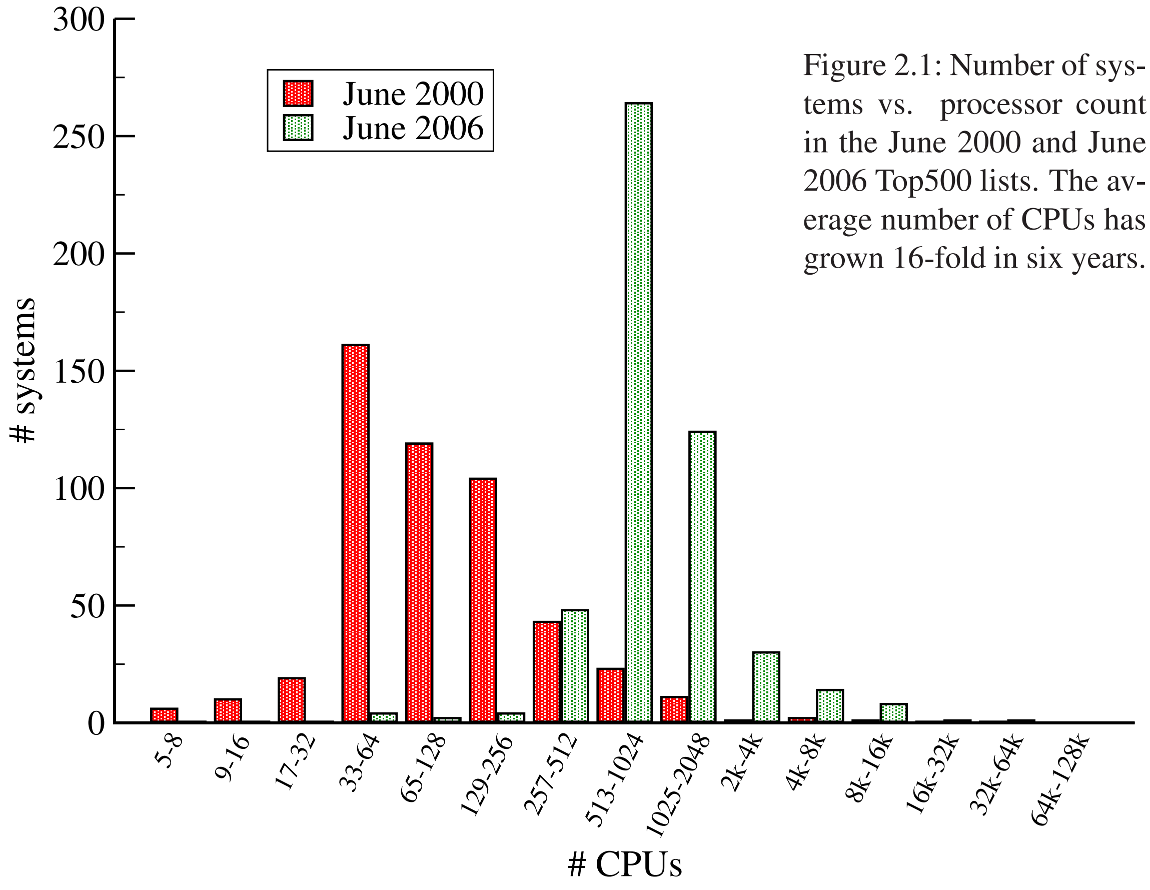


Figure 2.1: Number of systems vs. processor count in the June 2000 and June 2006 Top500 lists. The average number of CPUs has grown 16-fold in six years.

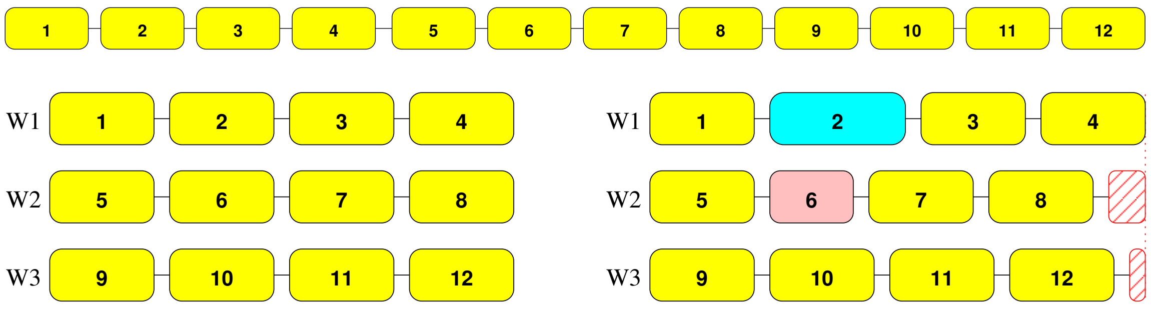


Figure 2.2: Parallelizing a sequence of tasks (top) using three workers (W1...W3). Left bottom: perfect speedup. Right bottom: some tasks executed by different workers at different speeds lead to *load imbalance*. Hatched regions indicate unused resources.

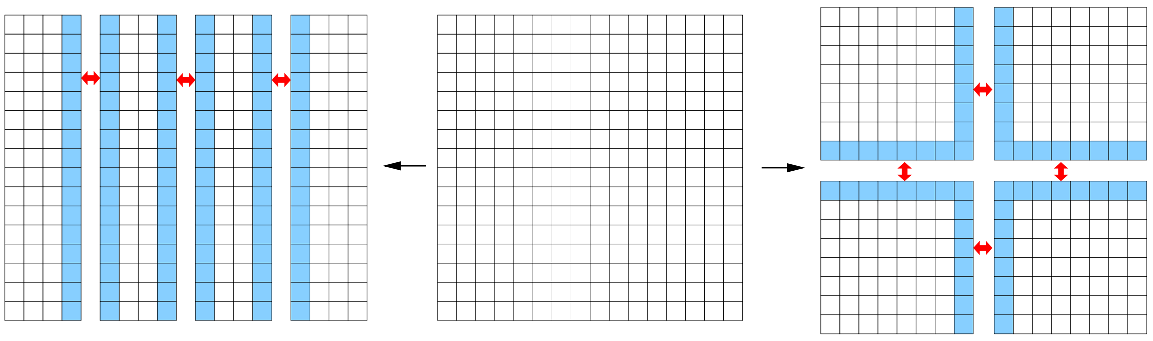


Figure 2.3: Domain decomposition of a two-dimensional simulation with next-neighbor interactions. Cutting into stripes (left) is simple but incurs more communication than optimal decomposition (right). Shaded cells participate in network communication.

2.2 Parallelization strategies

2.2.1 Data parallelism

Many simulations in science and engineering 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. Such grids are not necessarily cartesian but 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 simulation code that updates physical variables on a $n \times n$ grid. Domain decomposition subdivides the computational domain into N subdomains. How exactly this is to be done is the choice of the programmer, but some guidelines should be observed (see Fig. 2.3). First, the computational effort should be equal for all domains to avoid load imbalance (see Fig. 2.2 right). Second, depending on the locality properties of the algorithm, it may be necessary to *communicate* data across domain boundaries. E. g., next-neighbor relations require communication of a single data layer. The data volume to be considered in this case is proportional to the overall area of the domain cuts. Comparing the two alternatives in Fig. 2.3, one arrives at a communication cost of $n(N-1)$ for stripe domains, whereas an optimal decomposition into square subdomains leads to a cost of $2n(\sqrt{N}-1)$. Hence for large N the optimal decomposition has an advantage in communication cost of $2/\sqrt{N}$. Whether this difference is significant or not in reality depends on the problem size and other factors, of course.

Note that this calculation 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. E. g., to get the first derivative of some quantity with respect to the coordinates, only a next neighbor relation has to be implemented and the communication layers in Fig. 2.3 have a width of one. For higher-order derivatives this changes significantly, and if there is some long-ranged interaction like a Coulomb potential, the layers 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

Listing 2.1: Straightforward implementation of the Jacobi algorithm

```
do it=1,itmax
  dphimax=0.d0
  do k=1,kmax-1
    do i=1,imax-1
      dphi=(phi(i+1,k,t0)+phi(i-1,k,t0)-2.d0*phi(i,k,t0))*dy2 &
            +(phi(i,k+1,t0)+phi(i,k-1,t0)-2.d0*phi(i,k,t0))*dx2
      dphi=dphi*dt
      dphimax=max(dphimax,abs(dphi))
      phi(i,k,t1)=phi(i,k,t0)+dphi
    enddo
  enddo
  ! swap arrays
  i = t0 ; t0=t1 ; t1=i
  ! required precision reached?
  if(dphimax.lt.eps) exit
enddo
```

more slowly than volume if the problem size increases with N constant. Therefore one can alleviate communication bottlenecks just by choosing a larger problem size. The expected effects of strong and weak scaling with optimal domain decomposition in three dimensions will be discussed below.

As an example we will consider a simple *Jacobi* method for solving the diffusion equation for a scalar function $T(\vec{r},t)$,

$$\frac{\partial T}{\partial t} = \Delta T \ , \tag{2.1}$$

on a rectangular lattice subject to Dirichlet boundary conditions. The differential operators are discretized using finite differences (we restrict ourselves to two dimensions with no loss of generality):

$$\begin{aligned}\frac{\delta T(x_i,y_i)}{\delta t} &= \frac{T(x_{i+1},y_i)+T(x_{i-1},y_i)-2T(x_i,y_i)}{(\delta x)^2}\\ &\quad + \frac{T(x_i,y_{i-1})+T(x_i,y_{i+1})-2T(x_i,y_i)}{(\delta y)^2} \ .\end{aligned} \tag{2.2}$$

In each time step, a correction δT to T at coordinate (x_i,y_i) is calculated by (2.2) using the “old” values from the four next neighbor points. Of course, the updated T values must be written to a second array. After all points have been updated (a “sweep”), the algorithm is repeated. Listing 2.1 shows a possible implementation that uses a simple convergence criterion in order to solve for the steady state.

This code can be easily parallelized using domain decomposition. If, e. g., the grid is divided into strips along the x direction (index k in Listing 2.1), each worker performs a single sweep on its local strip. Subsequently, all boundary values needed for the next sweep must be communicated to the neighboring domains. On some computer systems this communication process is transparent to the program because all processors have access to a common, shared address space (see Chapter 4). In general, however, this cannot be assumed and some extra grid points, so-called *halo* or *ghost layers*, are used to store the

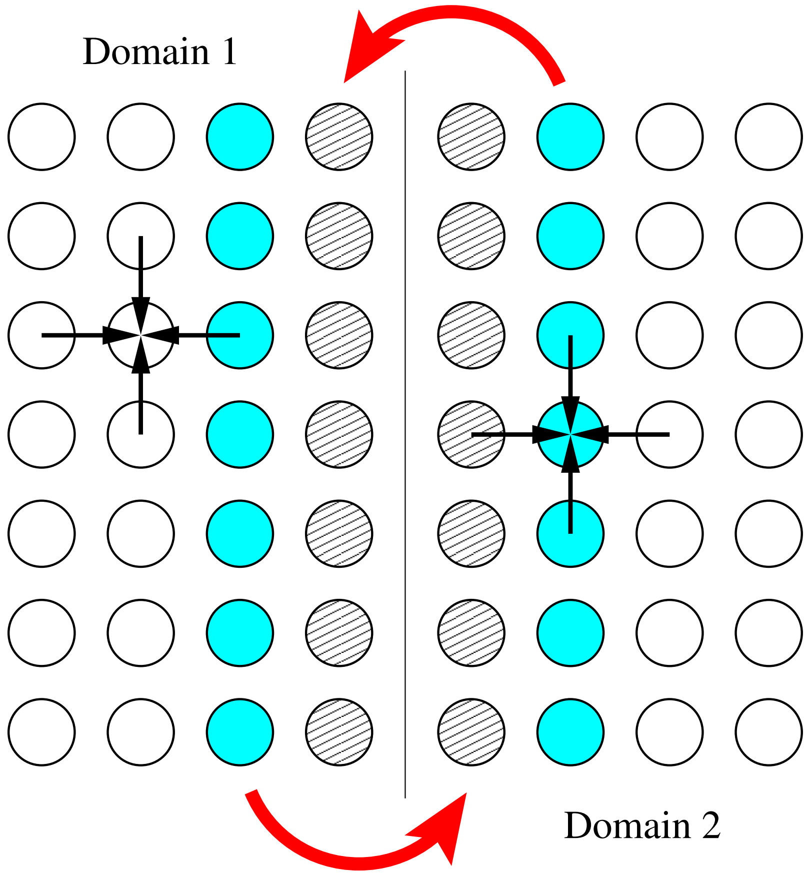


Figure 2.4: Using halo ("ghost") layers for communication across domain boundaries when solving boundary value problems. After the local updates in each domain, the boundary layers (shaded) are copied to the halo of the neighboring domain (hatched).

boundary data (see Fig. 2.4). After the exchange, each domain is ready for the next sweep. The whole process is completely equivalent to purely serial execution.

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. In Chapter 7, some optimizations are demonstrated that will significantly improve scalar performance.

2.2.2 Functional parallelism

Sometimes, solving a complete problem can be split into more or less disjoint subtasks that may have to be executed in some specific order, each one potentially using results of the previous one as input or being completely unrelated up to some point. The tasks can be worked on in parallel, using appropriate amounts of resources so that load imbalance is kept under control. Pipelining hardware in processor cores (see Sect. 1.1.3) is a prominent example for functional, or *task* parallelism.

Functional decomposition

Going back to the Jacobi algorithm from the previous section, a typical fine-grained scenario could, for each domain, assign some resources to communication and others to computational work. While the computational resources update local lattice points, communication can be performed in the background.

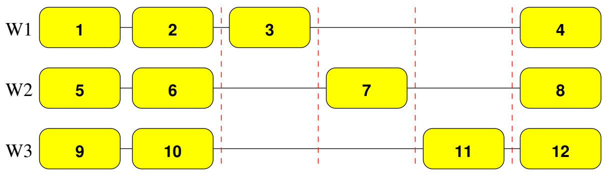


Figure 2.5: Parallelization in presence of a bottleneck that effectively serializes part of the concurrent execution. Tasks 3, 7 and 11 cannot overlap across the dashed “barriers”.

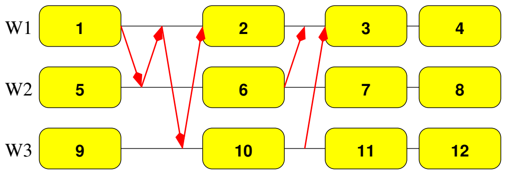


Figure 2.6: Communication processes (arrows represent messages) limit scalability if they cannot be overlapped with each other or with calculation.

Task queueing

On the application level, functional parallelism might be implemented as a work queue that holds tasks to be completed which get processed as resources become available. Mutual dependencies are implicit: Whenever a task is enqueued, it is clear that all its dependencies are fulfilled.

2.3 Performance models for parallel scalability

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 (non-parallelizable) and p is the perfectly parallelizable fraction. The 1-CPU (serial) runtime for this case,

$$T_{\mathrm{f}}^{\mathrm{s}} = s+p \;, \tag{2.3}$$

is thus normalized to one. Solving the same problem on N CPUs will require a runtime of

$$T_{\mathrm{f}}^{\mathrm{p}} = s+\frac{p}{N} \;. \tag{2.4}$$

This is called *strong scaling* because the amount of work stays constant no matter how many CPUs 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 problem as

$$T_{\mathrm{v}}^{\mathrm{s}} = s+pN^{\alpha} \;. \tag{2.5}$$

Consequently, the parallel runtime is

$$T_{\mathrm{v}}^{\mathrm{p}} = s+pN^{\alpha-1} \;. \tag{2.6}$$

The term *weak scaling* has been coined for this approach.

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

2.3.1 Scalability limitations

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 will define "performance" as "work over time", unless otherwise noted. Serial performance for fixed problem size (work) $s+p$ is thus

$$P_{\mathrm{f}}^{\mathrm{s}} = \frac{s+p}{T_{\mathrm{f}}^{\mathrm{s}}} = 1 \; , \tag{2.7}$$

as expected. Parallel performance is in this case

$$P_{\mathrm{f}}^{\mathrm{p}} = \frac{s+p}{T_{\mathrm{f}}^{\mathrm{p}}(N)} = \frac{1}{s+\frac{1-s}{N}} \; , \tag{2.8}$$

and application speedup ("scalability") is

$$S_{\mathrm{f}} = \frac{P_{\mathrm{f}}^{\mathrm{p}}}{P_{\mathrm{f}}^{\mathrm{s}}} = \frac{1}{s+\frac{1-s}{N}} \qquad \text{“Amdahl's Law”} \tag{2.9}$$

We have derived the well-known *Amdahl Law* which limits application speedup for large N to $1/s$. It 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

$$P_{\mathrm{f}}^{\mathrm{s}p} = \frac{p}{T_{\mathrm{f}}^{\mathrm{s}}} = p \; , \tag{2.10}$$

and parallel performance is

$$P_{\mathrm{f}}^{\mathrm{p}p} = \frac{p}{T_{\mathrm{f}}^{\mathrm{p}}(N)} = \frac{1-s}{s+\frac{1-s}{N}} \; . \tag{2.11}$$

Calculation of application speedup finally yields

$$S_{\mathrm{f}}^{p} = \frac{P_{\mathrm{f}}^{\mathrm{p}p}}{P_{\mathrm{f}}^{\mathrm{s}p}} = \frac{1}{s+\frac{1-s}{N}} \; , \tag{2.12}$$

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

$$P_{\mathrm{v}}^{\mathrm{s}} = \frac{s+p}{T_{\mathrm{f}}^{\mathrm{s}}} = 1 \; , \tag{2.13}$$

as $N=1$. Based on (2.5) and (2.6), Parallel performance (work over time) is

$$P_{\mathrm{v}}^{\mathrm{p}} = \frac{s+pN^{\alpha}}{T_{\mathrm{v}}^{\mathrm{p}}(N)} = \frac{s+(1-s)N^{\alpha}}{s+(1-s)N^{\alpha-1}} = S_{\mathrm{v}} \; , \tag{2.14}$$

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

$$S_\mathrm{v} \xrightarrow{N\gg 1} \frac{s+(1-s)N^\alpha}{s} = 1 + \frac{p}{s}N^\alpha \ , \tag{2.15}$$

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 $\alpha = 1$, (2.14) simplifies to

$$S_\mathrm{v}(\alpha = 1) = s + (1-s)N \ , \quad \text{“Gustafson's Law”} \tag{2.16}$$

and speedup is linear in N, even for small N. This is called *Gustafson's Law*. 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

$$P_\mathrm{v}^{\mathrm{s}p} = p \tag{2.17}$$

and parallel performance is

$$P_\mathrm{v}^{\mathrm{p}p} = \frac{pN^\alpha}{T_\mathrm{v}^\mathrm{p}(N)} = \frac{(1-s)N^\alpha}{s+(1-s)N^{\alpha-1}} \ , \tag{2.18}$$

which leads to an application speedup of

$$S_\mathrm{v}^p = \frac{P_\mathrm{v}^{\mathrm{p}p}}{P_\mathrm{v}^{\mathrm{s}p}} = \frac{N^\alpha}{s+(1-s)N^{\alpha-1}} \ . \tag{2.19}$$

Not surprisingly, speedup and performance are again not identical and differ by a factor of p. The important fact is that, in contrast to (2.16), for $\alpha = 1$ application speedup becomes purely linear in N *with no constant term*. So even though the overall work to be done (serial and parallel part) has not changed, scalability as defined in (2.19) 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).

2.3.2 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 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

$$\varepsilon = \frac{\text{performance on } N \text{ CPUs}}{N\times \text{ performance on one CPU}} = \frac{\text{speedup}}{N}. \tag{2.20}$$

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

$$\varepsilon = \frac{S_\mathrm{v}}{N} = \frac{sN^{-\alpha}+(1-s)}{sN^{1-\alpha}+(1-s)} \ . \tag{2.21}$$

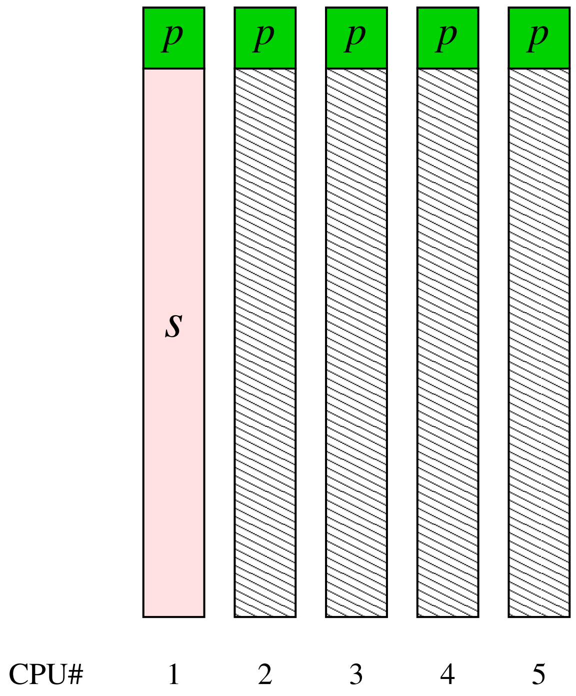


Figure 2.7: Weak scaling with an inappropriate definition of "work" that includes only the parallelizable part. Although "work over time" scales perfectly with CPU count, i. e. $\varepsilon_p = 1$, most of the resources (hatched boxes) are unused because $s \gg p$.

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,

$$\varepsilon_p = \frac{S_\mathrm{v}^p}{N} = \frac{N^{\alpha-1}}{s + (1-s)N^{\alpha-1}} , \tag{2.22}$$

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 Fig. 2.7). 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.

2.3.3 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, we will include a simple communication model. For simplicity we presuppose that communication cannot be overlapped with computation (see Fig. 2.6), 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 (2.6):

$$T_{\mathrm{v}}^{\mathrm{pc}} = s + pN^{\alpha-1} + c_\alpha(N) \ . \tag{2.23}$$

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

$$S_{\mathrm{v}}^{\mathrm{c}} = \frac{s + pN^\alpha}{T_{\mathrm{v}}^{\mathrm{pc}}(N)} = \frac{s + (1-s)N^\alpha}{s + (1-s)N^{\alpha-1} + c_\alpha(N)} \ . \tag{2.24}$$

The functional dependence $c_\alpha(N)$ can have a variety of forms; the dependency on α is sometimes functional, sometimes conceptual. Furthermore we assume that the amount of communication is the same for all workers. A few special cases are described below:

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

$$S_{\mathrm{v}}^{\mathrm{c}} = \frac{1}{s + \frac{1-s}{N} + (\kappa+\lambda)N} \xrightarrow{N\gg 1} \frac{1}{(\kappa+\lambda)N} \ , \tag{2.25}$$

  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.

- $\alpha = 0$, *non-blocking network:* If the communication network can sustain $N/2$ concurrent messages with no collisions, $c_\alpha(N) = \kappa + \lambda$ and

$$S_{\mathrm{v}}^{\mathrm{c}} = \frac{1}{s + \frac{1-s}{N} + \kappa + \lambda} \xrightarrow{N\gg 1} \frac{1}{s + \kappa + \lambda} \ . \tag{2.26}$$

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

- $\alpha = 0$, *non-blocking network, 3D domain decomposition:* 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:

$$S_{\mathrm{v}}^{\mathrm{c}} = \frac{1}{s + \frac{1-s}{N} + \kappa N^{-\beta} + \lambda} \xrightarrow{N\gg 1} \frac{1}{s + \lambda} \ . \tag{2.27}$$

  This arises, e.g., when domain decomposition (see page 25) is employed on a computational domain along all coordinate axes. In this case $\beta = 2/3$.

- $\alpha = 1$, *non-blocking network, 3D domain decomposition:* 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):

$$S_{\mathrm{v}}^{\mathrm{c}} = \frac{s + pN}{s + p + \kappa + \lambda} \xrightarrow{N\gg 1} \frac{s + (1-s)N}{1 + \kappa + \lambda} \ . \tag{2.28}$$

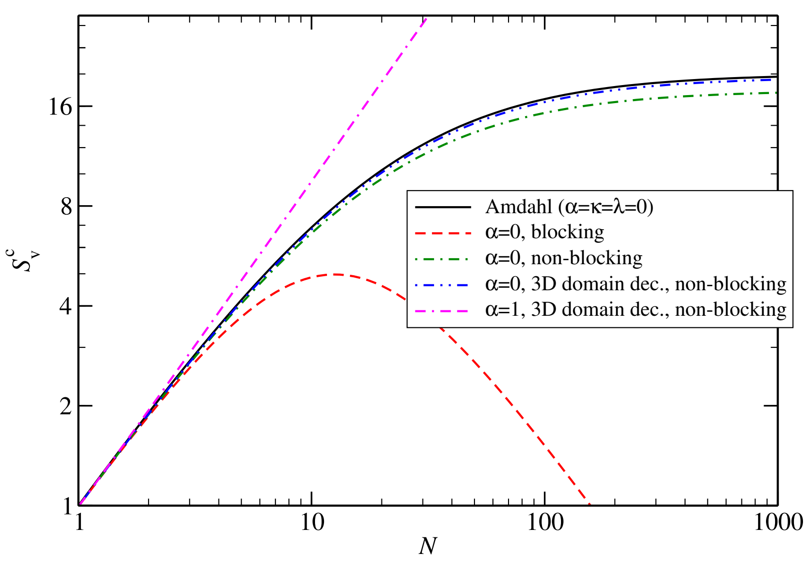


Figure 2.8: Predicted parallel scalability for different models at $s = 0.05$. In general, $\kappa = 0.005$ and $\lambda = 0.001$ except for the Amdahl case which is shown for reference.

Fig. 2.8 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. 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.

3 Distributed-memory computing

After covering the principles and limitations of parallelization we will now turn to the concrete architectures that are at the programmer's disposal to implement a parallel algorithm on. Two primary paradigms have emerged, and each features a dominant and standardized programming model: *Distributed-memory* and *shared-memory* systems. In this section we will be concerned with the former while the next section covers the latter.

Fig. 3.1 shows a simplified block diagram, or *programming model*, of a distributed-memory parallel computer. Each processor P (with its own local cache C) is connected to *exclusive* local memory, i.e. no other CPU has direct access to it. Although many parallel machines today, first and foremost the popular PC clusters, consist of a number of shared-memory *compute nodes* with two or more CPUs for price/performance reasons, the programmer's view does not reflect that (it is even possible to use distributed-memory programs on machines that feature shared memory only). Each node comprises at least one network interface (NI) that mediates the connection to a *communication network*. On each CPU runs a serial process that can communicate with other processes on other CPUs by means of the network. In the simplest case one could use standard switched Ethernet, but a number of more advanced technologies have emerged that can easily have ten times the bandwidth and 1/10 th of the latency of Gbit Ethernet. As shown in the section on performance models, the exact layout and "speed" of the network has considerable impact on application performance. The most favourable design consists of a non-blocking "wirespeed" network that can switch $N/2$ connections between its N participants without any bottlenecks. Although readily available for small systems with tens to a few hundred nodes, non-blocking switch fabrics become vastly expensive on very large installations and some compromises are usually made, i.e. there will be a bottleneck if all nodes want to communicate concurrently.

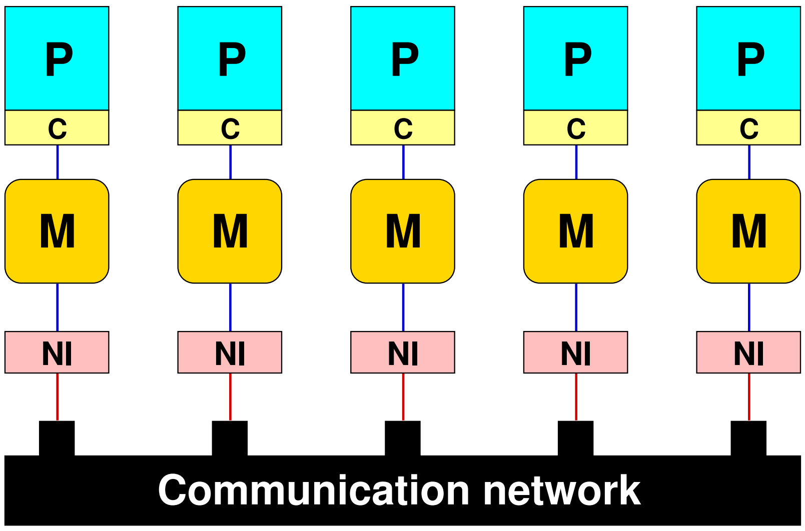


Figure 3.1: Simplified programmer's view, or programming model, of a distributed-memory parallel computer.

3.1 Message Passing

As mentioned above, distributed-memory parallel programming requires the use of explicit *message passing* (MP), i.e. communication between processes. This is surely the most tedious and complicated but also the most flexible parallelization method. Nowadays there is an established standard for message passing called MPI (Message Passing Interface) that is supported by all vendors [5]. MPI conforms to the following rules:

- The same program runs on all processes (Single Program Multiple Data, or SPMD). This is no restriction compared to the more general MPMD (Multiple Program Multiple Data) model as all processes taking part in a parallel calculation can be distinguished by a unique identifier called *rank* (see below).
- The program is written in a sequential language like Fortran, C or C++. Data exchange, i.e. sending and receiving of messages, is done via calls to an appropriate library.
- All variables in a process are local to this process. There is no concept of shared memory.

One should add that message passing is not the only possible programming paradigm for distributed-memory machines. Specialized languages like High Performance Fortran (HPF), Unified Parallel C (UPC) etc. have been created with support for distributed-memory parallelization built in, but they have not developed a broad user community and it is as yet unclear whether those approaches can match the efficiency of MPI.

In a message passing program, messages move data between processes. A message can be as simple as a single item (like a DP word) or even a complicated structure, perhaps scattered all over the address space. For a message to be transmitted in an orderly manner, some parameters have to be fixed in advance:

- Which processor is sending the message?
- Where is the data on the sending processor?
- What kind of data is being sent?
- How much data is there?
- Which process/es is/are going to receive the message?
- Where should the data be left on the receiving process(es)?
- How much data are the receiving processes prepared to accept?

As we will see, all MPI calls that actually transfer data have to specify those parameters in some way. MPI is a very broad standard with (in its latest version) over 500 library routines. Fortunately, most applications merely require less than ten of those to work.

Listing 3.1: A very simple, fully functional “Hello World” MPI program.

```
 1  program mpitest
 2  use MPI
 3
 4  integer rank, size, ierror
 5
 6  call MPI_Init(ierror)
 7  call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
 8  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
 9
10  write(*,*) 'Hello World, I am ',rank,' of ',size
11
12  call MPI_Finalize(ierror)
13
14  end
```

3.2 A brief glance on MPI

In order to compile and link MPI programs, compilers and linkers need options that specify where include files and libraries can be found. As there is considerable variation in those locations across installations, most MPI implementations provide compiler wrapper scripts (often called `mpicc`, `mpif77`, etc.) that supply the required options automatically but otherwise behave like “normal” compilers. Note that the way that MPI programs should be compiled and started is not fixed by the standard, so please consult system documentation by all means.

Listing 3.1 shows a simple “Hello World” type MPI program in Fortran 90. In line 2, the `MPI` module is loaded which provides required globals and definitions (in Fortran 77 and C/C++ one would use the preprocessor to read in the `mpif.h` or `mpi.h` header files, respectively). All MPI calls take an `INTENT(OUT)` argument, here called `ierror`, that transports information about the success of the MPI operation to the user code (in C/C++, the return code is used for that). As failure resiliency is not built into the MPI standard today and checkpoint/restart features are usually implemented by the user code anyway, the error code is rarely checked at all.

The first call in every MPI code should go to `MPI_Init` and initializes the parallel environment (line 6). In C/C++, `&argc` and `&argv` are passed to `MPI_Init` so that the library can evaluate and remove any additional command line arguments that may have been added by the MPI startup process. After initialization, MPI has set up a so-called *communicator*, called `MPI_COMM_WORLD`. A communicator defines a group of MPI processes that can be referred to by a communicator *handle*. The `MPI_COMM_WORLD` handle describes all processes that have been started as part of the parallel program. If required, other communicators can be defined as subsets of `MPI_COMM_WORLD`. Nearly all MPI calls require a communicator as an argument.

The calls to `MPI_Comm_size` and `MPI_Comm_rank` in lines 7 and 8 serve to determine the number of processes (`size`) in the parallel program and the unique identifier (the *rank*) of the calling process, respectively. The ranks in a communicator, in this case `MPI_COMM_WORLD`, are numbered starting from zero up to $N-1$. In line 12, the parallel program is shut down by a call to `MPI_Finalize`. Note that no MPI process except rank 0 is guaranteed to execute any code beyond `MPI_Finalize`.

In order to compile and run the source code in Listing 3.1, a "common" implementation would require the following steps:

```
$ mpif90 -O3 -o hello.exe hello.F90
$ mpirun -np 4 ./hello.exe
```

This would compile the code and start it with four processes. Be aware that processors may have to be allocated from some batch system before parallel programs can be launched. How MPI processes are mapped to actual processors is entirely up to the implementation. The output of this program could look like the following:

```
Hello World, I am 3 of 4
Hello World, I am 0 of 4
Hello World, I am 2 of 4
Hello World, I am 1 of 4
```

Although the `stdout` and `stderr` streams of MPI programs are usually redirected to the terminal where the program was started, the order in which outputs from different ranks will arrive there is undefined.

This example did not contain any real communication apart from starting and stopping processes. An MPI message is defined as an array of elements of a particular MPI datatype. Datatypes can either be basic types (corresponding to the standard types that every programming language knows) or *derived types* that must be defined by appropriate MPI calls. The reason why MPI needs to know the data types of messages is that it supports heterogeneous environments where it may be necessary to do on-the-fly data conversions. For some message transfer to take place, the data types on sender and receiver sides must match. If there is exactly one sender and one receiver we speak of *point-to-point communication*. Both ends are identified uniquely by their ranks. Each message can carry an additional integer label, the so-called *tag* that may be used to identify the type of a message, as a sequence number or any other accompanying information. In Listing 3.2 we show an MPI program fragment that computes an integral over some function `f(x)` in parallel. Each MPI process gets assigned a subinterval of the integration domain (lines 9 and 10), and some other function can then perform the actual integration (line 12). After that each process holds its own partial result, which should be added to get the final integral. This is done at rank 0, who executes a loop over all ranks from 1 to `size` − 1, receiving the local integral from each rank in turn via `MPI_Recv` and accumulating the result in `res`. Each rank apart from 0 has to call `MPI_Send` to transmit the data. Hence there are `size` − 1 send and `size` − 1 matching receive operations. The data types on both sides are specified to be `MPI_DOUBLE_PRECISION`, which corresponds to the usual `double precision` type in Fortran (be aware that MPI types are named differently in C/C++ than in Fortran). The message tag is not used here, so we set it to 0 because identical tags are required for message matching as well.

While all parameters are necessarily fixed on `MPI_Send`, there is some more variability on the receiver side. `MPI_Recv` allows *wildcards* so that the source rank and the tag do not have to be specified. Using `MPI_ANY_SOURCE` as source rank and `MPI_ANY_TAG` as tag will match any message, from any source, with any tag as long as the other matching criteria like datatype and communicator are met (this would have been possible in the integration example without further code changes). After `MPI_Recv` has returned to the user code, Listing 3.2: Program fragment for parallel integration in MPI.

```
   integer stat(MPI_STATUS_SIZE)
   call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
! integration limits
   a=0.d0
   b=2.d0
   res=0.d0
! limits for "me"
   mya=a+rank*(b-a)/size
   myb=mya+(b-a)/size
! integrate f(x) over my own chunk - actual work
   psum = integrate(mya,myb)
! rank 0 collects partial results
   if(rank.eq.0) then
      res=psum
      do i=1,size-1
         call MPI_Recv(tmp, &    ! receive buffer
                       1,   &    ! array length
                                 ! datatype
                       MPI_DOUBLE_PRECISION,&
                       i,   &    ! rank of source
                       0,   &    ! tag (additional label)
                                 ! communicator
                       MPI_COMM_WORLD,&
                       stat,&    ! status array (msg info)
                       ierror)
         res=res+tmp
      enddo
      write (*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
   else
      call MPI_Send(psum,   &    ! send buffer
                    1,      &    ! array length
                    MPI_DOUBLE_PRECISION,&
                    0,      &    ! rank of destination
                    0,      &    ! tag
                    MPI_COMM_WORLD,ierror)
   endif
```

the `status` array can be used to extract the missing pieces of information, i.e. the actual source rank and message tag, and also the length of the message as the array size specified in `MPI_Recv` is only an upper limit.

The accumulation of partial results as shown above is an example for a *reduction* operation, performed on all processes in the communicator. MPI has mechanisms that make reductions much simpler and in most cases more efficient than looping over all ranks and collecting results. As reduction is a procedure that all ranks in a communicator participate in, it belongs to the so-called *collective communication* operations in MPI. Collective communication, as opposed to point-to-point communication, requires that every rank calls the same routine, so it is impossible for a message sent via point-to-point to match a receive that was initiated using a collective call. The whole `if`...`else`...`endif` construct (apart from printing the result) in Listing 3.2 could have been written as a single call:

```
call MPI_Reduce(psum,   & ! send buffer
                res,    & ! receive buffer
                1,      & ! array length
                MPI_DOUBLE_PRECISION,&
                MPI_SUM,& ! type of operation
                0,      & ! root (accumulate res there)
                MPI_COMM_WORLD,ierror)
```

Most collective routines define a “root” rank at which some general data source or sink is located. Although rank 0 is a natural choice for “root”, it is in no way different from other ranks.

There are collective routines not only for reduction but also for barriers (each process stops at the barrier until all others have reached the barrier as well), broadcasts (the root rank transmits some data to everybody else), scatter/gather (data is distributed from root to all others or collected at root from everybody else), and complex combinations of those. Generally speaking, it is a good idea to prefer collectives over point-to-point constructs that “emulate” the same semantics. Good MPI implementations are optimized for data flow on collective operations and also have some knowledge about network topology built in.

All MPI functionalities described so far have the property that the call returns to the user program only after the message transfer has progressed far enough so that the send/receive buffer can be used without problems. I.e., received data has arrived completely and sent data has left the buffer so that it can be safely modified without inadvertently changing the message. In MPI terminology, this is called *blocking communication*. Although collective operations are always blocking, point-to-point communication can be performed with *non-blocking* calls as well. A non-blocking point-to-point call merely initiates a message transmission and returns very quickly to the user code. In an efficient implementation, waiting for data to arrive and the actual data transfer occur in the background, leaving resources free for computation. In other words, non-blocking MPI is a way in which computation and communication may be overlapped. As long as the transfer has not finished (which can be checked by suitable MPI calls), the message buffer must not be used. Non-blocking and blocking MPI calls are mutually compatible, i.e. a message sent via a blocking send can be matched by a non-blocking receive. Table 3.1 gives a rough overview of available communication modes in MPI.

	Point-to-point	**Collective**

Blocking	`MPI_Send(buf,...)`
	`MPI_Ssend(buf,...)`
	`MPI_Bsend(buf,...)`
	`MPI_Recv(buf,...)`
	(buf can be used after call returns)
		`MPI_Barrier(...)`
		`MPI_Bcast(...)`
		`MPI_Reduce(...)`
		(all processes in communicator must call)

Non-blocking	`MPI_Isend(buf,...)`
	`MPI_Irecv(buf,...)`
	(buf can not be used or modified after call returns; check for completion with `MPI_Wait(...)`/`MPI_Test(...)`)
		N/A

Table 3.1: Non-exhaustive overview on MPI's communication modes.

3.3 Basic performance characteristics of networks

As mentioned before, there are various options for the choice of a network in a distributed-memory computer. The simplest and cheapest solution to date is Gbit Ethernet, which will suffice for many throughput applications but is far too slow — in terms of bandwidth and latency — for parallel code with any need for fast communication. Assuming that the total transfer time for a message of size N [bytes] is composed of latency and streaming parts,

$$T = T_\mathrm{l} + \frac{N}{B} \tag{3.1}$$

and B being the maximum network bandwidth in MBytes/sec, the effective bandwidth is

$$B_\mathrm{eff} = \frac{N}{T_\mathrm{l} + \frac{N}{B}} \; . \tag{3.2}$$

In Fig. 3.2, the model parameters in (3.2) are fitted to real data obtained on a Gbit Ethernet network. Obviously this simple model is able to describe the gross features well.

For the measurement of effective bandwidth the *PingPong* benchmark is frequently used. The basic code sends a message of size N [bytes] once back and forth between two nodes:

```
S = get_walltime()
if(rank.eq.0) then
  call MPI_Send(buf,N,MPI_BYTE,1,0,...)
  call MPI_Recv(buf,N,MPI_BYTE,1,0,...)
else
  call MPI_Recv(buf,N,MPI_BYTE,0,0,...)
  call MPI_Send(buf,N,MPI_BYTE,0,0,...)
endif
E = get_walltime()
MBYTES  = 2*N/(E-S)/1.d6    ! MByte/sec rate
TIME    = (E-S)/2*1.d6      ! transfer time in microsecs
                            ! for single message
```

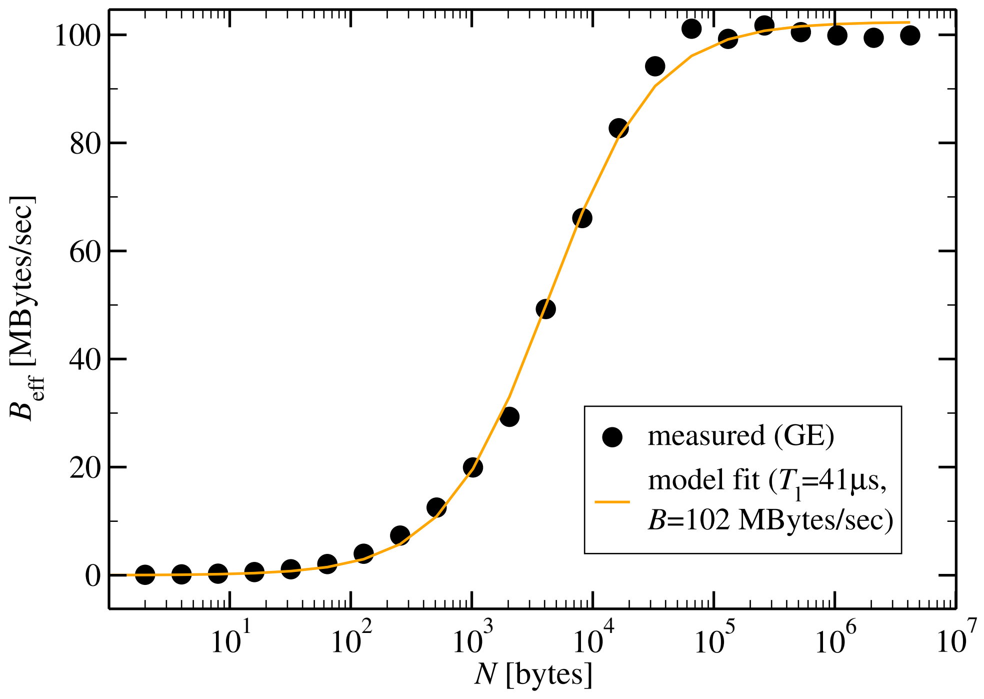


Figure 3.2: Fit of the model for effective bandwidth (3.2) to data measured on a Gbit Ethernet network.

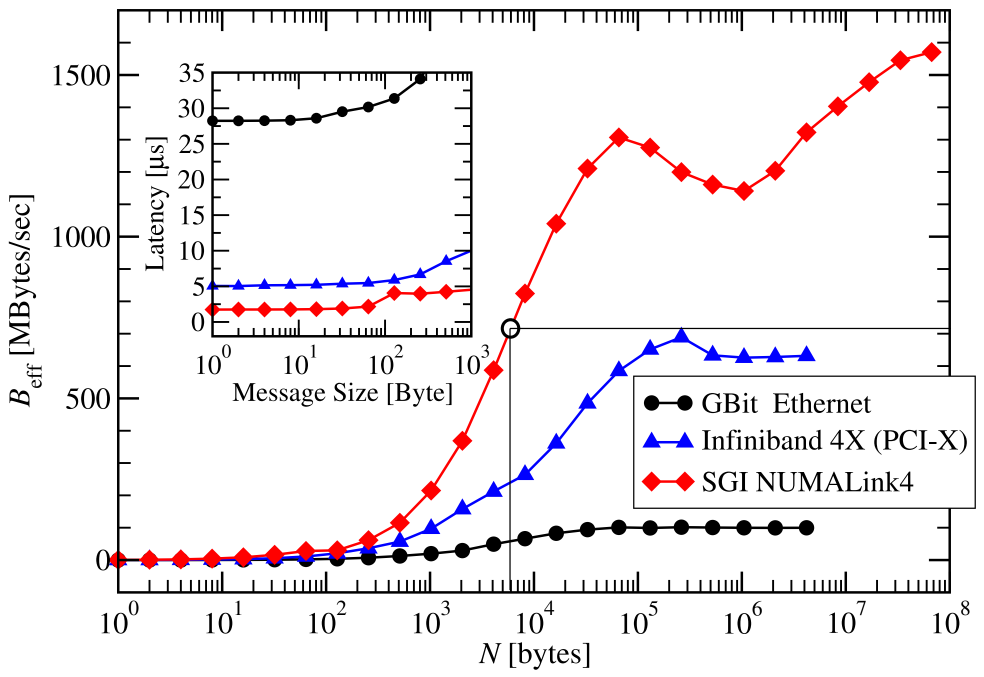


Figure 3.3: Result of the PingPong benchmark for three different networks. The $N_{1/2}$ point is marked for the NumaLink4 data. Inset: Latencies can be deduced by extrapolating to zero message length.

Bandwidth in MBytes/sec is then reported for different N (see Fig. 3.3). Common to all interconnects, we observe very low bandwidth for small message sizes as expected from the model (3.2). Latency can be measured directly by taking the $N = 0$ limit of transfer time (inset in Fig. 3.3). The reasons for latency can be diverse:

- All data transmission protocols have some overhead in the form of administrative data like message headers etc.
- Some protocols (like, e.g., TCP/IP as used over Ethernet) define minimum message sizes, so even if the application sends a single byte, a small "frame" of $N > 1$ bytes is transmitted.
- Initiating a message transfer is a complicated process that involves multiple software layers, depending on the complexity of the protocol. Each software layer adds to latency.
- Standard PC hardware as frequently used in clusters is not optimized towards low-latency I/O.

In fact, high-performance networks try to improve latency by reducing the influence of all of the above. Lightweight protocols, optimized drivers and communication devices directly attached to processor buses are all used by vendors to provide low MPI latency.

For large messages, effective bandwidth saturates at some maximum value. Structures like local minima etc. frequently occur but are very dependent on hardware and software implementations (e.g., the MPI library could decide to switch to a different buffering algorithm beyond some message size). Although saturation bandwidth can be quite high (there are systems where achievable MPI bandwidth is comparable to the local memory bandwidth of the processor), many applications work in a region on the bandwidth graph where latency effects still play a dominant role. To quantify this problem, the $N_{1/2}$ value is often reported. This is the message size at which $B_\mathrm{eff} = B/2$ (see Fig. 3.3). In the model (3.2), $N_{1/2} = BT_\mathrm{l}$. From this point of view it makes sense to ask whether an increase in maximum network bandwidth by a factor of β is really beneficial for all messages. At message size N, the improvement in effective bandwidth is

$$\frac{B_\mathrm{eff}(\beta B, T_\mathrm{l})}{B_\mathrm{eff}(B, T_\mathrm{l})} = \frac{1 + N/N_{1/2}}{1 + N/\beta N_{1/2}} \,, \tag{3.3}$$

so that for $N = N_{1/2}$ and $\beta = 2$ the gain is only 33 %. In case of a reduction of latency by a factor of β, the result is the same. Hence it is desirable to improve on both latency and bandwidth to make an interconnect more efficient for all applications.

Please note that the simple PingPong algorithm described above cannot pinpoint saturation effects: If the network fabric is not completely non-blocking and all nodes transmit or receive data (as is often the case with collective MPI operations), aggregated bandwidth, i.e. the sum over all effective bandwidths for all point-to-point connections, is lower than the theoretical limit. This can severely throttle the performance of applications on large CPU numbers as well as overall throughput of the machine.

4 Shared-memory computing

A *shared-memory parallel computer* is a system in which a number of CPUs work on a common, shared physical address space. This is fundamentally different from the distributed-memory paradigm as described in the previous section. Although transparent to the programmer as far as functionality is concerned, there are two varieties of shared-memory systems that have very different performance characteristics:

- *Uniform Memory Access* (UMA) systems feature a “flat” memory model: Memory bandwidth and latency are the same for all processors and all memory locations. This is also called *symmetric multiprocessing* (SMP).

- On *cache-coherent Non-Uniform Memory Access* (ccNUMA) machines, memory is *physically distributed* but *logically shared*. The physical layout of such systems is quite similar to the distributed-memory case (Fig. 3.1), but network logic makes the aggregated memory of the whole system appear as one single address space. Due to the distributed nature, memory access performance varies depending on which CPU accesses which parts of memory (“local” vs. “remote” access).

With multiple CPUs, copies of the same cache line may reside in different caches, probably in modified state. So for both above varieties, *cache coherence protocols* must guarantee consistency between cached data and data in memory at all times. Details about UMA, ccNUMA and cache coherence mechanisms are provided in the following sections.

4.1 UMA

The simplest implementation of a UMA system is a dual-core processor in which two CPUs share a single path to memory. Technical details vary among vendors, and it is very common in high performance computing to use more than one chip in a compute node (be they single-core or multi-core), which adds to diversity. In Figs. 4.1 and 4.2, two typical representatives of UMA systems used in HPC are shown.

In Fig. 4.1 two (single-core) processors, each in its own socket, communicate and access memory over a common bus, the so-called *frontside bus* (FSB). All arbitration protocols required to make this work are already built into the CPUs. The chipset (often termed “northbridge”) is responsible for driving the memory modules and connects to other parts of the node like I/O subsystems.

In Fig. 4.2, two dual-core chips connect to the chipset, each with its own FSB. The chipset plays an important role in enforcing cache coherence and also mediates the connection to memory. In principle, a system like this could be designed so that the bandwidth from chipset to memory matches the aggregated bandwidth of the frontside buses. Each dual-core chip features a separate L1 on each CPU but a shared L2 cache for both. The advantage of a shared cache is that, to an extent limited by cache size, data exchange between cores can be done there and does not have to resort to the slow frontside bus. Of course, a shared cache should also meet the bandwidth requirements of all connected cores, which

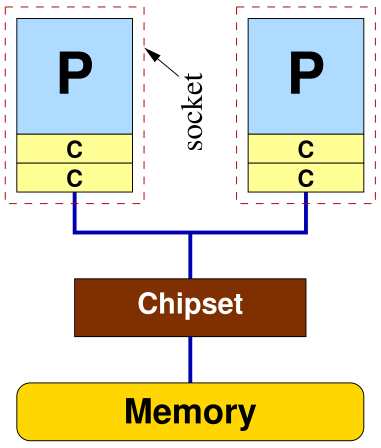


Figure 4.1: A UMA system with two single-core CPUs that share a common frontside bus (FSB).

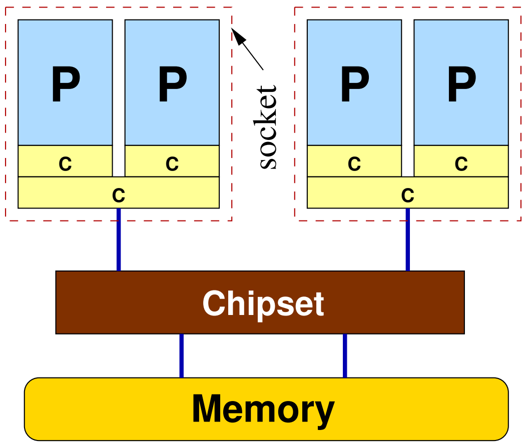


Figure 4.2: A UMA system in which the FSBs of two dual-core chips are connected separately to the chipset.

might not be the case. Due to the shared caches and FSB connections this kind of node is, while still a UMA system, quite sensitive to the exact placement of processes or threads on its cores. For instance, with only two processes it may be desirable to keep ("pin") them on separate sockets if the memory bandwidth requirements are high. On the other hand, processes communicating a lot via shared memory may show more performance when placed on the same socket because of the shared L2 cache. Operating systems as well as some modern compilers usually have tools or library functions for observing and implementing thread or process pinning.

The general problem of UMA systems is that bandwidth bottlenecks are bound to occur when the number of sockets, or FSBs, is larger than a certain limit. In very simple designs like the one in Fig. 4.1, a common *memory bus* is used that can only transfer data to one CPU at a time (this is also the case for all multi-core chips available today).

In order to maintain scalability of memory bandwidth with CPU number, non-blocking *crossbar switches* can be built that establish point-to-point connections between FSBs and memory modules (similar to the chipset in Fig. 4.2). Due to the very large aggregated bandwidths those become very expensive for a larger number of sockets. At the time of writing, the largest UMA systems with scalable bandwidth (i.e. memory bandwidth matches the aggregated FSBs bandwidths of all processors in the node) have eight CPUs. This problem can only be solved by giving up on the UMA principle.

4.2 ccNUMA

In ccNUMA, a *locality domain* (LD) is a set of processor cores together with locally connected memory which can be accessed in the most efficient way, i.e. without resorting to a network of any kind. Although the ccNUMA principle provides scalable bandwidth for very large processor counts — systems with up to 1024 CPUs in a single address space with a single OS instance are available today —, it is also found in inexpensive small two- or four-socket AMD Opteron nodes frequently used for HPC clustering (see Fig. 4.3). In this example two locality domains, i.e. dual-core chips with separate caches and a common interface to local memory, are linked using a special high-speed connection called

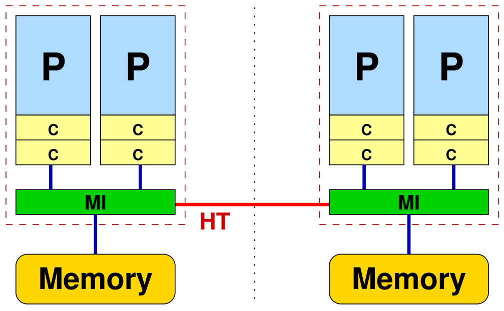


Figure 4.3: HyperTransport-based ccNUMA system with two locality domains (one per socket) and four cores.

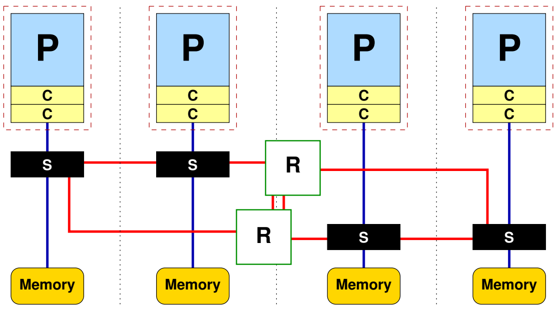


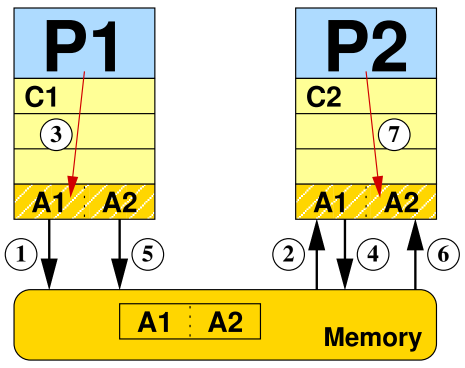
Figure 4.4: ccNUMA system with routed NUMALink network and four locality domains.

HyperTransport (HT). Apart from the minor peculiarity that the sockets can drive memory directly, making a northbridge obsolete, this system differs substantially from networked UMA designs in that the HT link can mediate direct *coherent* access from one processor to another processor's memory. From the programmer's point of view this mechanism is transparent: All the required protocols are handled by the HT hardware.

In Fig. 4.4 another approach to ccNUMA is shown that is flexible enough to scale to large machines and used in SGI Altix systems. Each processor socket connects to a communication interface (S) that provides memory access as well as connectivity to the proprietary *NUMALink* (NL) network. The NL network relies on routers (R) to switch connections for non-local access. As with HT, the NL hardware allows for transparent access to the whole address space of the machine from all CPUs. Although shown here only with four sockets, multi-level router fabrics can be built that scale up to hundreds of CPUs. It must, however, be noted that each piece of hardware inserted into a data connection (communication interfaces, routers) add to latency, making access characteristics very inhomogeneous across the system. Furthermore, as is the case with networks for distributed-memory computers, providing wire-equivalent speed, non-blocking bandwidth in large systems is extremely expensive.

In all ccNUMA designs, network connections must have bandwidth and latency characteristics that are at least the same order of magnitude as for local memory. Although this is the case for all contemporary systems, even a penalty factor of two for non-local transfers can badly hurt application performance if access can not be restricted inside locality domains. This *locality problem* is the first of two obstacles to take with high performance software on ccNUMA. It occurs even if there is only one serial program running on a ccNUMA machine. The second problem is potential *congestion* if two processors from different locality domains access memory in the same locality domain, fighting for memory bandwidth. Even if the network is non-blocking and its performance matches the bandwidth and latency of local access, congestion can occur. Both problems can be solved by carefully observing the data access patterns of an application and restricting data access of each processor to its own locality domain. Section 9 will elaborate on this topic.

In inexpensive ccNUMA systems I/O interfaces are often connected to a single LD. Although I/O transfers are usually slow compared to memory bandwidth, there are, e.g., high-speed network interconnects that feature multi-GB bandwidths between compute nodes. If data arrives at the "wrong" locality domain, written by an I/O driver that has positioned its buffer space disregarding any ccNUMA constraints, it should be copied to



1. C1 requests exclusive CL ownership
2. set CL in C2 to state I
3. CL has state E in C1 → modify A1 in C1 and set to state M
4. C2 requests exclusive CL ownership
5. evict CL from C1 and set to state I
6. load CL to C2 and set to state E
7. modify A2 in C2 and set to state M in C2

Figure 4.5: Two processors P1, P2 modify the two parts A1, A2 of the same cache line in caches C1 and C2. The MESI coherence protocol ensures consistency between cache and memory.

its optimal destination, reducing effective bandwidth by a factor of four (three if RFOs can be avoided, see page 16). In this case even the most expensive interconnect hardware is wasted. In truly scalable ccNUMA designs this problem is circumvented by distributing I/O connections across the whole machine and using ccNUMA-aware drivers.

4.3 Cache coherence

Cache coherence mechanisms are required in all cache-based multiprocessor systems, UMA as well as ccNUMA. This is because potentially copies of the same cache line could reside in several CPU caches. If, e.g., one of those gets modified and evicted to memory, the other caches' contents reflect outdated data. Cache coherence protocols ensure a consistent view of memory under all circumstances.

Fig. 4.5 shows an example on two processors P1 and P2 with respective caches C1 and C2. Each cache line holds two items. Two neighboring items A1 and A2 in memory belong to the same cache line and are modified by P1 and P2, respectively. Without cache coherence, each cache would read the line from memory, A1 would get modified in C1, A2 would get modified in C2 and some time later both modified copies of the cache line would have to be evicted. As all memory traffic is handled in chunks of cache line size, there is no way to determine the correct values of A1 and A2 in memory.

Under control of cache coherence logic this discrepancy can be avoided. As an example we pick the MESI protocol, which draws its name from the four possible states a cache line can take:

M *modified:* The cache line has been modified in this cache, and it resides in no other cache than this one. Only upon eviction will memory reflect the most current state.

E *exclusive:* The cache line has been read from memory but not (yet) modified. However, it resides in no other cache.

S *shared:* The cache line has been read from memory but not (yet) modified. There may be other copies in other caches of the machine.

I *invalid:* The cache line does not reflect any sensible data. Under normal circumstances this happens if the cache line was in shared state and another processor has requested exclusive ownership. A cache miss occurs if and only if the chosen line is invalid.

The order of events is depicted in Fig. 4.5. The question arises how a cache line in state M is notified when it should be evicted because another cache needs to read the most current data. Similarly, cache lines in state S or E must be invalidated if another cache requests exclusive ownership. In small systems a *bus snoop* is used to achieve this: Whenever notification of other caches seems in order, the originating cache *broadcasts* the corresponding cache line address through the system, and all caches "snoop" the bus and react accordingly. While simple to implement, this method has the crucial drawback that address broadcasts pollute the system buses and reduce available bandwidth for "useful" memory accesses. A separate network for coherence traffic can alleviate this effect but is not always practicable.

A better alternative, usually applied in larger ccNUMA machines, is a *directory-based* protocol where bus logic like chipsets or memory interfaces keep track of the location and state of each cache line in the system. This uses up some small part of main memory (usually far less than 10 %), but the advantage is that state changes of cache lines are transmitted only to those caches that actually require them. This greatly reduces coherence traffic through the system. Today even workstation chipsets implement "snoop filters" that serve the same purpose.

Coherence traffic can severely hurt application performance if the same cache line is written to frequently by different processors (*false sharing*). In Section 8.1.2 on page 85 we will give hints for avoiding false sharing in user code.

4.4 Short introduction to shared-memory programming with OpenMP

As mentioned before, programming shared-memory systems can be done in an entirely "distributed-memory" fashion, i.e. the processes making up an MPI program can run happily on a UMA or ccNUMA machine, not knowing that the underlying hardware provides more efficient means of communication. In fact, even on large "constellation" clusters (systems where the number of nodes is smaller than the number of processors per node), the dominant parallelization method is often MPI due to its efficiency and flexibility. After all an MPI code can run on shared- as well as distributed-memory systems, and efficient MPI implementations transparently use shared memory for communication if available.

However, MPI is not only the most flexible but also the most tedious way of parallelization. Shared memory opens the possibility to have immediate access to all data from all processors without explicit message passing. The established standard in this field is OpenMP [6]. OpenMP is a set of *compiler directives* that a non-OpenMP-capable compiler would just regard as comments and ignore. Thus, a well-written parallel OpenMP program is also a valid serial program (of course it is possible to write OpenMP code that will not run sequentially, but this is not the intention of the method). In contrast to MPI, the central entity in OpenMP 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, Listing 4.1: A simple program for numerical integration of a function $f(x)$ in OpenMP

```
 1   pi=0.d0
 2   w=1.d0/n
 3 !$OMP PARALLEL PRIVATE(x,sum)
 4   sum=0.d0
 5 !$OMP DO SCHEDULE(STATIC)
 6   do i=1,n
 7     x=w*(i-0.5d0)
 8     sum=sum+f(x)
 9   enddo
10 !$OMP END DO
11 !$OMP CRITICAL
12   pi=pi+w*sum
13 !$OMP END CRITICAL
14 !$OMP END PARALLEL
```

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 as 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: Thread-private data is for convenience, not for protection.

4.4.1 OpenMP worksharing and data scoping

It is indeed possible to use operating system threads (POSIX threads) directly, but this option is rarely used with numerical software. OpenMP is a layer that adapts the raw OS thread interface to make it more usable with the typical loop structures that numerical software tends to show. As an example, consider a parallel version of a simple integration program (Listing 4.1). This is valid serial code, but equipping it with the comment lines starting with the sequence `!$OMP` (called a *sentinel*) and using an OpenMP-capable compiler makes it shared-memory parallel. The `PARALLEL` directive instructs the compiler to start a *parallel region* (see Fig. 4.6). A *team of threads* is spawned that executes identical copies of everything up to `END PARALLEL` (the actual number of threads is unknown at compile time as it is set by an environment variable). By default, all variables which were present in the program before the parallel region are *shared* among all threads. However, that would include `x` and `sum` of which we later need *private* versions for each thread. OpenMP provides a way to make existing variables private by means of the `PRIVATE` clause. If, in the above example, any thread in a parallel region writes to `sum` (see line 4), it will update its own private copy, leaving the other threads' untouched. Therefore, before the loop starts each thread's copy of `sum` is set to zero.

In order to *share* some amount of work between threads and actually reduce wallclock time, *work sharing directives* can be applied. This is done in line 5 using the `DO` directive with the optional `SCHEDULE` clause. The `DO` directive is always related to the immediately following loop (line 6) and generates code that distributes the loop iterations among the team of threads (please note that the loop counter variable is automatically made private).

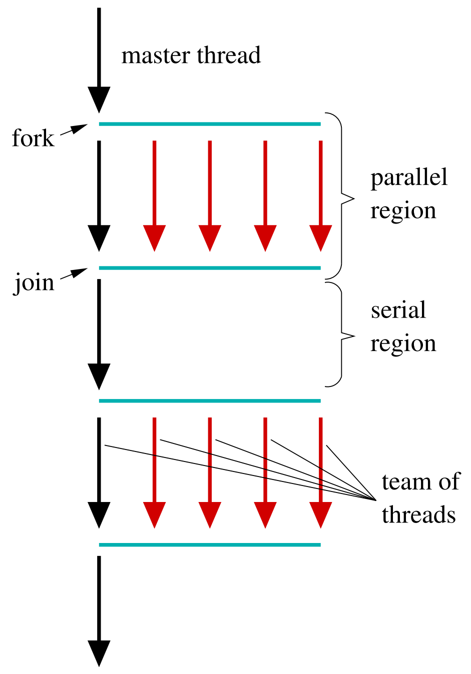


Figure 4.6: Model for OpenMP thread operations: The master thread "forks" a thread team that work on shared memory in a parallel region. After the parallel region, the threads are "joined" or put to sleep until the next parallel region starts.

4.4.2 Loop scheduling

How this is done is controlled by the argument of `SCHEDULE`. The simplest possibility is `STATIC` which divides the loop in 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 *chunk size* like in "`STATIC,1`" that dictates that chunks of size 1 should be distributed across threads in a round-robin manner.

There are alternatives to static schedule for other types of workload (see Fig. 4.7). *Dynamic* scheduling assigns a chunk of work, defined by the chunk size, to the next thread that has finished its chunk. This allows for a very flexible distribution which is usually not reproduced from run to run. Threads that get assigned to "easier" chunks will end up completing less of them, and load imbalance is greatly reduced. The downside is that dynamic scheduling generates significant overhead if the chunks are too small in terms of execution time. This is why it is often desirable to use a moderately large chunk size 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 chunk size is always proportional to the remaining number of iterations divided by the number of threads. The smallest chunk size 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 which are limited by memory bandwidth may suffer from insufficient access locality on ccNUMA systems (see Sect. 4.2 for an introduction to ccNUMA architecture and Chapter 9 for ccNUMA-specific optimization methods). The static schedule is thus the only choice under such circumstances.

For debugging and profiling purposes, OpenMP provides a facility to determine the loop scheduling at runtime. If the scheduling clause in the code specifies "`RUNTIME`", the loop is

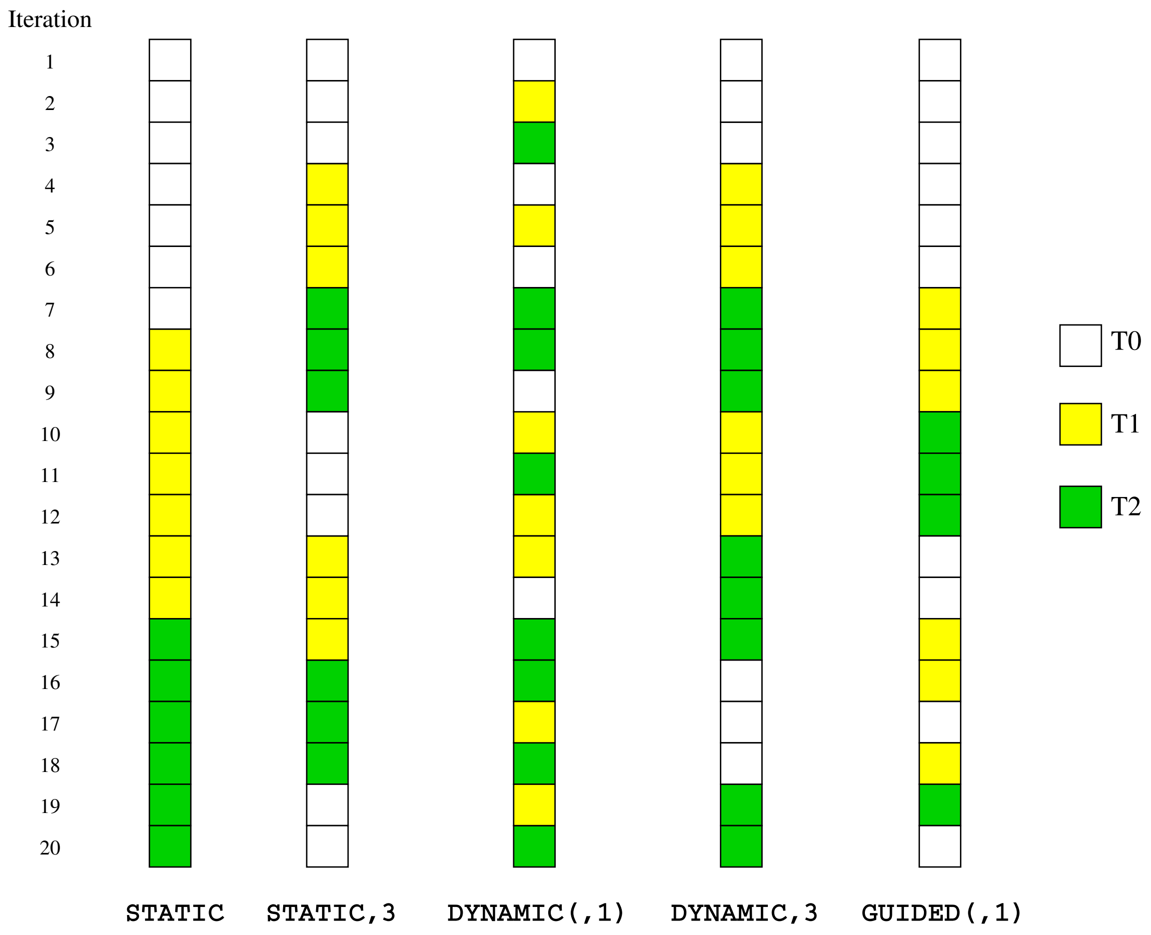


Figure 4.7: Loop schedules in OpenMP. The example loop has 20 iterations and is executed by 3 threads (T0, T1, T2). Default chunk size for `DYNAMIC` and `GUIDED` is one.

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.

4.4.3 Protecting shared data

The parallelized loop in Listing 4.1 computes a partial sum in each thread's private `sum` variable. To get the final result, all the partial sums must be accumulated in the global `pi` variable (line 12), but `pi` is shared so that uncontrolled updates would lead to a *race condition*, i.e. the exact order and timing of operations will influence the result. In OpenMP, *critical sections* solve this problem by making sure that at most one thread at a time executes some piece of code. In the example, the `CRITICAL` and `END CRITICAL` directives bracket the update to `pi` so that a correct result emerges at all times.

Critical sections hold the danger of *deadlocks* when used inappropriately. A deadlock arises when one or more threads 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. OpenMP provides two solutions for this problem:

- A critical section may be given a *name* that distinguishes it from others. The name is specified in parentheses after the `CRITICAL` directive:

  ```
  !$OMP PARALLEL DO PRIVATE(x)
    do i=1,N
      x=sin(2*PI*x/N)
  !$OMP CRITICAL (psum)
      sum=sum+func(x)
  !$OMP END CRITICAL (psum)
    enddo
  !$OMP END PARALLEL DO
    ...
    SUBROUTINE func(v)
    double precision v
  !$OMP CRITICAL (prand)
    v=v+random_func()
  !$OMP END CRITICAL (prand)
    END SUBROUTINE func
  ```

  Without the names on the two different critical sections in this code would deadlock.

- There are OpenMP API functions (see below) 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.

Whenever there are different shared resources in a program that must be protected from concurrent access each for its own but are otherwise unconnected, named critical sections or OpenMP locks should be used both for correctness and performance reasons.

Listing 4.2: Fortran sentinels and conditional compilation with OpenMP

```
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
```

4.4.4 Miscellaneous

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. Conditional compilation however is supported by the preprocessor symbol `_OPENMP` which is defined only if OpenMP is available and (in Fortran) the special sentinel `!$` that acts as a comment only if OpenMP is not enabled (see Listing 4.2). Here we also see a part of OpenMP that is not concerned with directives. The `use omp_lib` declaration loads the OpenMP API function prototypes (in C/C++, `#include <omp.h>` serves the same purpose). The `omp_get_thread_num()` function determines the *thread ID*, a number between zero and the number of threads minus one, while `omp_get_num_threads()` returns the number of threads in the current team. So if the general disposition of OpenMP towards loop-based code is not what the programmer wants, one can easily switch to an MPI-like style where thread ID determines the tasks of each thread.

In above example the second API call (line 8) is located in a `SINGLE` region, which means that it will be executed by exactly one thread, namely the one that reaches the `SINGLE` directive first. 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 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 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.

There is an important reason for serializing the `write` statements in line 10. As a rule, Listing 4.3: C/C++ example with reduction clause for adding noise to the elements of an array and calculating its vector norm. `rand()` is not thread-safe so it must be protected by a critical region.

```
1    double r,s;
2  #pragma omp parallel for private(r) reduction(+:s)
3    for(i=0; i<N; ++i) {
4  #pragma omp critical
5      {
6        r = rand();             // not thread-safe
7      }
8      a[i] += func(r/RAND_MAX); // func() is thread-safe
9      s = s + a[i] * a[i];      // calculate norm
10   }
```

I/O operations and general OS functionality, but also common library functions should be serialized because they are usually not *thread-safe*, i.e. calling them in parallel regions from different threads at the same time may lead to errors. A prominent example is the `rand()` function from the C library as it uses a static variable to store its hidden state (the seed). Although local variables in functions are private to the calling thread, static data is shared by definition. This is also true for Fortran variables with a SAVE attribute.

One should note that the OpenMP standard gives no hints as to how threads are to be distributed among the processors, let alone observe locality constraints. Usually the OS makes a good choice regarding placement of threads, but sometimes (especially on multi-core architectures and ccNUMA systems) it makes sense to use OS-level tools, compiler support or library functions to explicitly pin threads to cores. See Section 9 on page 89 for details.

So far, all OpenMP examples were concerned with the Fortran bindings. Of course there is also a C/C++ interface that has the same functionality. The C/C++ sentinel is called `#pragma omp`, and the only way to do conditional compilation is to use the `_OPENMP` symbol. Loop parallelization only works for "canonical" `for` loops that have standard integer-type loop counters (i.e., no STL-style iterator loops) and is done via `#pragma omp for`. All directives that act on code regions apply to compound statements and an explicit ending directive is not required.

The example in Listing 4.3 shows a C code that adds some random noise to the elements of an array `a[]` and calculates its vector norm. As mentioned before, `rand()` is not thread-safe and must be protected with a critical region. The function `func()`, however, is assumed to be thread-safe as it only uses automatic (stack) variables and can thus be called safely from a parallel region (line 8). Another peculiarity in this example is the fusion of the `parallel` and `for` directives to `parallel for`, which allows for more compact code. Finally, the reduction operation is not performed using critical updates as in the integration example. Instead, an OpenMP `reduction` clause is used (end of line 2) that automatically initializes the summation variable `s` with a sensible starting value, makes it private and accumulates the partial results to it.

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. This limit can often be adjusted by the user or the administrators. However, in a threaded program there are as many stacks as there are threads, and the way the thread-local stacks get their limit set is not standardized at all. Please consult OS and compiler documentation as to how thread-local stacks are limited. Stack overflows are a frequent source of problems with OpenMP programs.

Running an OpenMP program is as simple as starting the executable binary just like in the serial case. The number of threads to be used is determined by an environment variable called `OMP_NUM_TREADS`. There may be other means to influence the way the program is running, e.g. OS scheduling of threads, pinning, getting debug output etc., but those are not standardized.