Chapter 2

Basic optimization techniques for serial code

In the age of multi-1000-processor parallel computers, writing code that runs efficiently on a single CPU has grown slightly old-fashioned in some circles. The argument for this point of view is derived from the notion that it is easier to add more CPUs and boasting massive parallelism instead of investing effort into serial optimization. There is actually some plausible theory, outlined in Section 5.3.8, to support this attitude. Nevertheless there can be no doubt that single-processor optimizations are of premier importance. If a speedup of two can be achieved by some simple code changes, the user will be satisfied with much fewer CPUs in the parallel case. This frees resources for other users and projects, and puts hardware that was often acquired for considerable amounts of money to better use. If an existing parallel code is to be optimized for speed, it must be the first goal to make the single-processor run as fast as possible. This chapter summarizes basic tools and strategies for serial code profiling and optimizations. More advanced topics, especially in view of data transfer optimizations, will be covered in Chapter 3.

2.1 Scalar profiling

Gathering information about a program's behavior, specifically its use of resources, is called *profiling*. The most important "resource" in terms of high performance computing is runtime. Hence, a common profiling strategy is to find out how much time is spent in the different functions, and maybe even lines, of a code in order to identify *hot spots*, i.e., the parts of the program that require the dominant fraction of runtime. These hot spots are subsequently analyzed for possible optimization opportunities. See Section 2.1.1 for an introduction to function- and line-based profiling.

Even if hot spots have been determined, it is sometimes not clear from the start what the actual reasons for a particular performance bottleneck are, especially if the function or code block in question extends over many lines. In such a case one would like to know, e.g., whether data access to main memory or pipeline stalls limit performance. If data access is the problem, it may not be straightforward to identify which data items accessed in a complex piece of code actually cause the most delay. *Hardware performance counters* may help to resolve such issues. They are provided on all current processors and allow deep insights into the use of resources within the chip and the system. See Section 2.1.2 for more information.

One should point out that there are indeed circumstances when nothing can be done any more to accelerate a serial code any further. It is essential for the user to be able to identify the point where additional optimization efforts are useless. Section 3.1 contains guidelines for the most common cases.

2.1.1 Function- and line-based runtime profiling

In general, two technologies are used for function- and line-based profiling: Code *instrumentation* and *sampling*. Instrumentation works by letting the compiler modify each function call, inserting some code that logs the call, its caller (or the complete call stack) and probably how much time it required. Of course, this technique incurs some significant overhead, especially if the code contains many functions with short runtime. The instrumentation code will try to compensate for that, but there is always some residual uncertainty. Sampling is less invasive: the program is interrupted at periodic intervals, e.g., 10 milliseconds, and the program counter (and possibly the current call stack) is recorded. Necessarily this process is statistical by nature, but the longer the code runs, the more accurate the results will be. If the compiler has equipped the object code with appropriate information, sampling can deliver execution time information down to the source line and even machine code level. Instrumentation is necessarily limited to functions or *basic blocks* (code with one entry and one exit point with no calls or jumps in between) for efficiency reasons.

Function profiling

The most widely used profiling tool is `gprof` from the GNU binutils package. `gprof` uses both instrumentation and sampling to collect a flat function profile as well as a callgraph profile, also called a *butterfly graph*. In order to activate profiling, the code must be compiled with an appropriate option (many modern compilers can generate gprof-compliant instrumentation; for the GCC, use `-pg`) and run once. This produces a non-human-readable file `gmon.out`, to be interpreted by the `gprof` program. The flat profile contains information about execution times of all the program's functions and how often they were called:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 70.45      5.14     5.14 26074562     0.00     0.00  intersect
 26.01      7.03     1.90  4000000     0.00     0.00  shade
  3.72      7.30     0.27      100     2.71    73.03  calc_tile
```

There is one line for each function. The columns can be interpreted as follows:

% time Percentage of overall program runtime used *exclusively* by this function, i.e., not counting any of its callees.

cumulative seconds Cumulative sum of exclusive runtimes of all functions up to and including this one.

self seconds Number of seconds used by this function (exclusive). By default, the list is sorted according to this field.

calls The number of times this function was called.

self ms/call Average number of milliseconds per call that were spent in this function (exclusive).

total ms/call Average number of milliseconds per call that were spent in this function, including its callees (inclusive).

In the example above, optimization attempts would definitely start with the `intersect()` function, and `shade()` would also deserve a closer look. The corresponding exclusive percentages can hint at the maximum possible gain. If, e.g., `shade()` could be optimized to become twice as fast, the whole program would run in roughly $7.3 - 0.95 = 6.35$ seconds, i.e., about 15% faster.

Note that the outcome of a profiling run can depend crucially on the ability of the compiler to perform *function inlining*. Inlining is an optimization technique that replaces a function call by the body of the callee, reducing overhead (see Section 2.4.2 for a more thorough discussion). If inlining is allowed, the profiler output may be strongly distorted when some hot spot function gets inlined and its runtime is attributed to the caller. If the compiler/profiler combination has no support for correct profiling of inlined functions, it may be required to disallow inlining altogether. Of course, this may itself have some significant impact on program performance characteristics.

A flat profile already contains a lot of information, however it does not reveal how the runtime contribution of a certain function is composed of several different callers, which other functions (callees) are called from it, and which contribution to runtime they in turn incur. This data is provided by the *butterfly graph*, or *callgraph profile*:

```
index % time    self  children    called     name
                0.27    7.03     100/100         main [2]
[1]     99.9    0.27    7.03     100         calc_tile [1]
                1.90    5.14 4000000/4000000     shade [3]
-----------------------------------------------
                                                 <spontaneous>
[2]     99.9    0.00    7.30                 main [2]
                0.27    7.03     100/100         calc_tile [1]
-----------------------------------------------
                             5517592             shade [3]
                1.90    5.14 4000000/4000000     calc_tile [1]
[3]     96.2    1.90    5.14 4000000+5517592 shade [3]
                5.14    0.00 26074562/26074562     intersect [4]
                             5517592             shade [3]
-----------------------------------------------
                5.14    0.00 26074562/26074562     shade [3]
[4]     70.2    5.14    0.00 26074562          intersect [4]
```

Each section of the callgraph pertains to exactly one function, which is listed together with a running index (far left). The functions listed above this line are the current function's callers, whereas those listed below are its callees. Recursive calls are accounted for (see the `shade()` function). These are the meanings of the various fields:

% time The percentage of overall runtime spent in this function, including its callees (inclusive time). This should be identical to the product of the number of calls and the time per call on the flat profile.

self For each indexed function, this is exclusive execution time (identical to flat profile). For its callers (callees), it denotes the inclusive time this function (each callee) contributed to each caller (this function).

children For each indexed function, this is inclusive minus exclusive runtime, i.e., the contribution of all its callees to inclusive time. Part of this time contributes to inclusive runtime of each of the function's callers and is denoted in the respective caller rows. The callee rows in this column designate the contribution of each callee's callees to the function's inclusive runtime.

called denotes the number of times the function was called (probably split into recursive plus nonrecursive contributions, as shown in case of `shade()` above). Which fraction of the number of calls came from each caller is shown in the caller row, whereas the fraction of calls for each callee that was initiated from this function can be found in the callee rows.

There are tools that can represent the butterfly profile in a graphical way, making it possible to browse the call tree and quickly find the "critical path," i.e., the sequence of functions (from the root to some leaf) that shows dominant inclusive contributions for all its elements.

Line-based profiling

Function profiling becomes useless when the program to be analyzed contains large functions (in terms of code lines) that contribute significantly to overall runtime:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 73.21     13.47    13.47        1    13.47    18.40  MAIN__
  6.47     14.66     1.19 21993788     0.00     0.00  mvteil_
  6.36     15.83     1.17 51827551     0.00     0.00  ran1_
  6.25     16.98     1.15 35996244     0.00     0.00  gzahl_
```

Here the `MAIN` function in a Fortran program requires over 73% of overall runtime but has about 1700 lines of code. If the hot spot in such functions cannot be found by simple common sense, tools for line-based profiling should be used. Many products, free and commercial, exist that can perform this task to different levels of sophistication. As an example we pick the open source profiling tool OProfile [T19], which can in some sense act as a replacement for `gprof` because it can do function-based flat and butterfly profiles as well. With OProfile, the only prerequisite the binary has to fulfill is that debug symbols must be included (usually this is accomplished by the `-g` compiler option). Any special instrumentation is not required. A profiling daemon must be started (usually with the rights of a super user), which subsequently monitors the whole computer and collects data about all running binaries. The user can later extract information about a specific binary. Among other things, this can be an annotated source listing in which each source line is accompanied by the number of sampling hits (first column) and the relative percentage of total program samples (second column):

```
                  :       DO 215 M=1,3
  4292   0.9317 :          bremsdir(M) = bremsdir(M) + FH(M)*Z12
  1462   0.3174 : 215   CONTINUE
                  :
   682   0.1481 :       U12 = U12 + GCL12 * Upot
                  :
                  :       DO 230 M=1,3
  3348   0.7268 :          F(M,I)=F(M,I)+FH(M)*Z12
  1497   0.3250 :          Fion(M)=Fion(M)+FH(M)*Z12
   501   0.1088 :230    CONTINUE
```

This kind of data has to be taken with a grain of salt, though. The compiler-generated symbol tables must be consistent so that a machine instruction's address in memory can be properly matched to the correct source line. Modern compilers can reorganize code heavily if high optimization levels are enabled. Loops can be fused or split, lines rearranged, variables optimized away, etc., so that the actual code executed may be far from resembling the original source. Furthermore, due to the strongly pipelined architecture of modern microprocessors it is usually impossible to attribute a specific moment in time to a particular source line or even machine instruction. However, looking at line-based profiling data on a loop-by-loop basis (samples integrated across the loop body) is relatively safe; in case of doubt, recompilation with a lower optimization level (and inlining disabled) may provide more insight.

Above source line profile can be easily put into a form that allows identification of hot spots. The cumulative sum over all samples versus source line number has a steep slope wherever many sampling hits are aggregated (see Figure 2.1).

2.1.2 Hardware performance counters

The first step in performance profiling is concerned with pinpointing the hot spots in terms of runtime, i.e., clock ticks. But when it comes to identifying the actual reason for a code to be slow, or if one merely wants to know by which resource it is limited, clock ticks are insufficient. Luckily, modern processors feature a small number of *performance counters* (often far less than ten), which are special on-chip registers that get incremented each time a certain event occurs. Among the usually several hundred events that can be monitored, there are a few that are most useful for profiling:

- *Number of bus transactions, i.e., cache line transfers.* Events like "cache misses" are commonly used instead of bus transactions, however one should

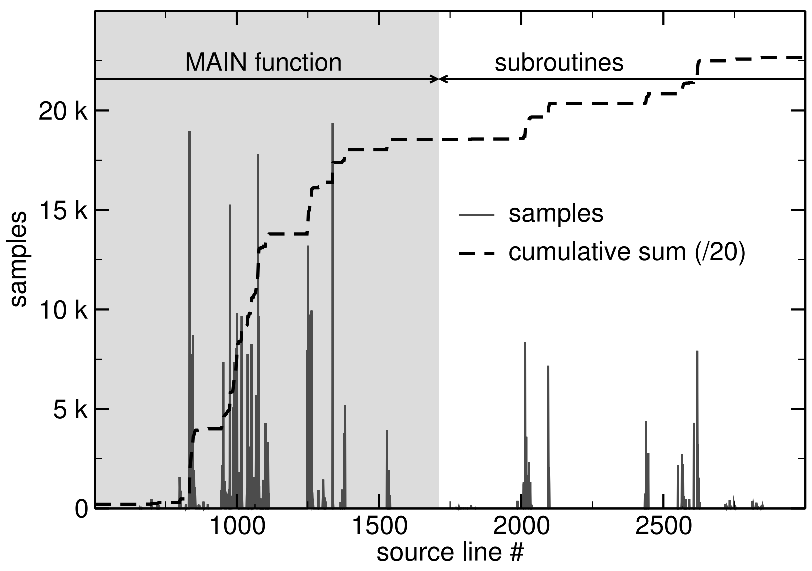

Figure 2.1: Sampling histogram (solid) with number of samples vs. source code line number. Dashed line: Cumulative sum over samples. Important hot spots (about 50% of overall time) are around line 1000 of the `MAIN` function, which encompasses over 1700 lines of code.

be aware that prefetching mechanisms (in hardware or software) can interfere with the number of cache misses counted. In that respect, bus transactions are often the safer way to account for the actual data volume transferred over the memory bus. If the memory bandwidth used by a processor is close to the theoretical maximum (or better, close to what standard low-level bandwidth benchmarks like STREAM [W119, 134] can achieve; see Section 3.1), there is no point in trying to optimize further for better bus utilization. The number can also be used for checking the correctness of some theoretical model one may have developed for estimating the data transfer requirements of an application (see Section 3.1 for an example of such a model).

- *Number of loads and stores.* Together with bus transactions, this can give an indication as to how efficiently cache lines are used for computation. If, e.g., the number of DP loads and stores per cache line is less than its length in DP words, this may signal strided memory access. One should however take into account how the data is used; if, for some reason, the processor pipelines are stalled most of the time, or if many arithmetic operations on the data are performed out of registers, the strided access may not actually be a performance bottleneck.

- *Number of floating-point operations.* The importance of this very popular metric is often overrated. As will be shown in Chapter 3, data transfer is the dominant performance-limiting factor in scientific code. Even so, if the number of floating-point operations per clock cycle is somewhere near the theoretical maximum (either given by the CPU's peak performance, or less if there is an asymmetry between MULT and ADD operations), standard code optimization is unlikely to achieve any improvement and algorithmic changes are probably in order.

- *Mispredicted branches.* This counter is incremented when the CPU has predicted the outcome of a conditional branch and the prediction has proved to be wrong. Depending on the architecture, the penalty for a mispredicted branch can be tens of cycles (see Section 2.3.2 below). In general, scientific codes tend to be loop-based so that branches are well predictable. However, "pointer chasing" and computed branches increase the probability of mispredictions.

- *Pipeline stalls.* Dependencies between operations running in different stages of the processor pipeline (see Section 1.2.3) can lead to cycles during which not all stages are busy, so-called *stalls* or *bubbles*. Often bubbles cannot be avoided, e.g., when performance is limited by memory bandwidth and the arithmetic units spend their time waiting for data. Hence, it is quite difficult to identify the point where there are "too many" bubbles. Stall cycle analysis is especially important on in-order architectures (like, e.g., Intel IA64) because bubbles cannot be filled by the hardware if there are no provisions for executing instructions that appear later in the instruction stream but have their operands available.

- *Number of instructions executed.* Together with clock cycles, this can be a guideline for judging how effectively the superscalar hardware with its multiple execution units is utilized. Experience shows that it is quite hard for compiler-generated code to reach more than 2–3 instructions per cycle, even in tight inner loops with good pipelining properties.

There are essentially two ways to use hardware counters in profiling. In order to get a quick overview of the performance properties of an application, a simple tool can measure overall counts from start to finish and probably calculate some *derived metrics* like "instructions per cycle" or "cache misses per load or store." A typical output of such a tool could look like this if run with some application code (these examples were compiled from output generated by the `lipfpm` tool, which is delivered with some SGI Altix systems):

```
CPU Cycles........................................ 8721026107
Retired Instructions.............................. 21036052778
Average number of retired instructions per cycle........ 2.398151
L2 Misses......................................... 101822
Bus Memory Transactions........................... 54413
Average MB/s requested by L2...................... 2.241689
Average Bus Bandwidth (MB/s)...................... 1.197943
Retired Loads..................................... 694058538
Retired Stores.................................... 199529719
Retired FP Operations............................. 7134186664
Average MFLOP/s................................... 1225.702566
```

```
Full Pipe Bubbles in Main Pipe......................... 3565110974
Percent stall/bubble cycles............................ 40.642963
```

Note that the number of performance counters is usually quite small (between 2 and 4). Using a large number of metrics like in the example above may require running the application multiple times or, if the profiling tool supports it, *multiplexing* between different sets of metrics by, e.g., switching to another set in regular intervals (like 100 ms). The latter introduces a statistical error into the data. This error should be closely watched, especially if the counts involved are small or if the application runs only for a very short time.

In the example above the large number of retired instructions per cycle indicates that the hardware is well utilized. So do the (very small) required bandwidths from the caches and main memory and the relation between retired load/store instructions to L2 cache misses. However, there are pipeline bubbles in 40% of all CPU cycles. It is hard to tell without some reference whether this is a large or a small value. For comparison, this is the profile of a vector triad code (large vector length) on the same architecture as above:

```
CPU Cycles............................................. 28526301346
Retired Instructions................................... 15720706664
Average number of retired instructions per cycle........ 0.551095
L2 Misses.............................................. 605101189
Bus Memory Transactions................................ 751366092
Average MB/s requested by L2........................... 4058.535901
Average Bus Bandwidth (MB/s)........................... 5028.015243
Retired Loads.......................................... 3756854692
Retired Stores......................................... 2472009027
Retired FP Operations.................................. 4800014764
Average MFLOP/s........................................ 252.399428
Full Pipe Bubbles in Main Pipe......................... 25550004147
Percent stall/bubble cycles............................ 89.566481
```

The bandwidth requirements, the low number of instructions per cycle, and the relation between loads/stores and cache misses indicate a memory-bound situation. In contrast to the previous case, the percentage of stalled cycles is more than doubled. Only an elaborate stall cycle analysis, based on more detailed metrics, would be able to reveal the origin of those bubbles.

Although it can provide vital information, collecting "global" hardware counter data may be too simplistic in some cases. If, e.g., the application profile contains many phases with vastly different performance properties (e.g., cache-bound vs. memory-bound, etc.), integrated counter data may lead to false conclusions. Restricting counter increments to specific parts of code execution can help to break down the counter profile and get more specific data. Most simple tools provide a small library with an API that allows at least enabling and disabling the counters under program control. An open-source tool that can do this is, e.g., contained in the LIKWID [T20, W120] suite. It is compatible with most current x86-based processors.

A even more advanced way to use hardware performance counters (that is, e.g., supported by OProfile, but also by other tools like Intel VTune [T21]) is to use sampling to attribute the events they accumulate to functions or lines in the application code, much in the same way as in line-based profiling. Instead of taking a snapshot of the instruction pointer or call stack at regular intervals, an overflow value is defined for each counter (or, more exactly, each metric). When the counter reaches this value, an interrupt is generated and an IP or call stack sample is taken. Naturally, samples generated for a particular metric accumulate at places where the counter was incremented most often, allowing the same considerations as above not for the whole program but on a function and even code line basis. It should, however, be clear that a correct interpretation of results from counting hardware events requires a considerable amount of experience.

2.1.3 Manual instrumentation

If the overheads subjected to the application by standard compiler-based instrumentation are too large, or if only certain parts of the code should be profiled in order to get a less complex view on performance properties, manual instrumentation may be considered. The programmer inserts calls to a a wallclock timing routine like `gettimeofday()` (see Listing 1.2 for a convenient wrapper function) or, if hardware counter information is required, a profiling library like PAPI [T22] into the program. Some libraries also allow to start and stop the standard profiling mechanisms as described in Sections 2.1.1 and 2.1.2 under program control [T20]. This can be very interesting in C++ where standard profiles are often very cluttered due to the use of templates and operator overloading.

The results returned by timing routines should be interpreted with some care. The most frequent mistake with code timings occurs when the time periods to be measured are in the same order of magnitude as the timer resolution, i.e., the minimum possible interval that can be resolved.

2.2 Common sense optimizations

Very simple code changes can often lead to a significant performance boost. The most important "common sense" guidelines regarding the avoidance of performance pitfalls are summarized in the following sections. Some of those hints may seem trivial, but experience shows that many scientific codes can be improved by the simplest of measures.

2.2.1 Do less work!

In all but the rarest of cases, rearranging the code such that less work than before is being done will improve performance. A very common example is a loop that checks a number of objects to have a certain property, but all that matters in the end is that *any* object has the property at all:

```
logical :: FLAG
FLAG = .false.
do i=1,N
  if(complex_func(A(i)) < THRESHOLD) then
    FLAG = .true.
  endif
enddo
```

If `complex_func()` has no side effects, the only information that gets communicated to the outside of the loop is the value of `FLAG`. In this case, depending on the probability for the conditional to be true, much computational effort can be saved by leaving the loop as soon as `FLAG` changes state:

```
logical :: FLAG
FLAG = .false.
do i=1,N
  if(complex_func(A(i)) < THRESHOLD) then
    FLAG = .true.
    exit
  endif
enddo
```

2.2.2 Avoid expensive operations!

Sometimes, implementing an algorithm is done in a thoroughly “one-to-one” way, translating formulae to code without any reference to performance issues. While this is actually good (performance optimization always bears the slight danger of changing numerics, if not results), in a second step all those operations should be eliminated that can be substituted by “cheaper” alternatives. Prominent examples for such “strong” operations are trigonometric functions or exponentiation. Bear in mind that an expression like `x**2.0` is often not optimized by the compiler to become `x*x` but left as it stands, resulting in the evaluation of an exponential and a logarithm. The corresponding optimization is called *strength reduction*. Apart from the simple case described above, strong operations sometimes appear with a limited set of fixed arguments. This is an example from a simulation code for nonequilibrium spin systems:

```
integer :: iL,iR,iU,iO,iS,iN
double precision :: edelz,tt
...                                ! load spin orientations
edelz = iL+iR+iU+iO+iS+iN          ! loop kernel
BF    = 0.5d0*(1.d0+TANH(edelz/tt))
```

The last two lines are executed in a loop that accounts for nearly the whole runtime of the application. The integer variables store spin orientations (up or down, i.e., -1 or $+1$, respectively), so the `edelz` variable only takes integer values in the range $\{-6,\ldots,+6\}$. The `tanh()` function is one of those operations that take vast amounts of time (at least tens of cycles), even if implemented in hardware. In the case described, however, it is easy to eliminate the `tanh()` call completely by *tabulating* the function over the range of arguments required, assuming that `tt` does not change its value so that the table does only have to be set up once:

```
double precision, dimension(-6:6) :: tanh_table
integer :: iL,iR,iU,iO,iS,iN
double precision :: tt
...
do i=-6,6                          ! do this once
  tanh_table(i) = 0.5d0*(1.d0+TANH(dble(i)/tt))
enddo
...
BF = tanh_table(iL+iR+iU+iO+iS+iN) ! loop kernel
```

The table look-up is performed at virtually no cost compared to the `tanh()` evaluation since the table will be available in L1 cache at access latencies of a few CPU cycles. Due to the small size of the table and its frequent use it will fit into L1 cache and stay there in the course of the calculation.

2.2.3 Shrink the working set!

The *working set* of a code is the amount of memory it uses (i.e., actually touches) in the course of a calculation, or at least during a significant part of overall runtime. In general, shrinking the working set by whatever means is a good thing because it raises the probability for cache hits. If and how this can be achieved and whether it pays off performancewise depends heavily on the algorithm and its implementation, of course. In the above example, the original code used standard four-byte integers to store the spin orientations. The working set was thus much larger than the L2 cache of any processor. By changing the array definitions to use `integer(kind=1)` for the spin variables, the working set could be reduced by nearly a factor of four, and became comparable to cache size.

Consider, however, that not all microprocessors can handle "small" types efficiently. Using byte-size integers for instance could result in very ineffective code that actually works on larger word sizes but extracts the byte-sized data by mask and shift operations. On the other hand, if SIMD instructions can be employed, it may become quite efficient to revert to simpler data types (see Section 2.3.3 for details).

2.3 Simple measures, large impact

2.3.1 Elimination of common subexpressions

Common subexpression elimination is an optimization that is often considered a task for compilers. Basically one tries to save time by precalculating parts of complex expressions and assigning them to temporary variables before a code construct starts that uses those parts multiple times. In case of loops, this optimization is also called *loop invariant code motion*:

```
! inefficient                    tmp=s+r*sin(x)
do i=1,N                  →      do i=1,N
  A(i)=A(i)+s+r*sin(x)             A(i)=A(i)+tmp
enddo                            enddo
```

A lot of compute time can be saved by this optimization, especially where "strong" operations (like `sin()`) are involved. Although it may happen that subexpressions are obstructed by other code and not easily recognizable, compilers are in principle able to detect this situation. They will, however, often refrain from pulling the subexpression out of the loop if this required employing associativity rules (see Section 2.4.4 for more information about compiler optimizations and reordering of arithmetic expressions). In practice, a good strategy is to help the compiler by eliminating common subexpressions by hand.

2.3.2 Avoiding branches

"Tight" loops, i.e., loops that have few operations in them, are typical candidates for software pipelining (see Section 1.2.3), loop unrolling, and other optimization techniques (see below). If for some reason compiler optimization fails or is inefficient, performance will suffer. This can easily happen if the loop body contains conditional branches:

```
do j=1,N
  do i=1,N
    if(i.ge.j) then
      sign=1.d0
    else if(i.lt.j) then
      sign=-1.d0
    else
      sign=0.d0
    endif
    C(j) = C(j) + sign * A(i,j) * B(i)
  enddo
enddo
```

In this multiplication of a matrix with a vector, the upper and lower triangular parts get different signs and the diagonal is ignored. The `if` statement serves to decide about which factor to use. Each time a corresponding conditional branch is encountered by the processor, some *branch prediction* logic tries to guess the most probable outcome of the test before the result is actually available, based on statistical methods. The instructions along the chosen path are then fetched, decoded, and generally fed into the pipeline. If the anticipation turns out to be false (this is called a *mispredicted branch* or *branch miss*), the pipeline has to be *flushed* back to the position of the branch, implying many lost cycles. Furthermore, the compiler refrains from doing advanced optimizations like unrolling or SIMD vectorization (see the following section). Fortunately, the loop nest can be transformed so that all `if` statements vanish:

```
do j=1,N
  do i=j+1,N
    C(j) = C(j) + A(i,j) * B(i)
  enddo
enddo
do j=1,N
  do i=1,j-1
    C(j) = C(j) - A(i,j) * B(i)
  enddo
enddo
```

By using two different variants of the inner loop, the conditional has effectively been moved outside. One should add that there is more optimization potential in this loop nest. Please consider Chapter 3 for more information on optimizing data access.

2.3.3 Using SIMD instruction sets

Although vector processors also use SIMD instructions and the use of SIMD in microprocessors is often termed “vectorization,” it is more similar to the multitrack property of modern vector systems. Generally speaking, a “vectorizable” loop in this context will run faster if more operations can be performed with a single instruction, i.e., the size of the data type should be as small as possible. Switching from DP to SP data could result in up to a twofold speedup (as is the case for the SIMD capabilities of x86-type CPUs [V104, V105]), with the additional benefit that more items fit into the cache.

Certainly, preferring SIMD instructions over scalar ones is no guarantee for a performance improvement. If the code is strongly limited by memory bandwidth, no SIMD technique can bridge this gap. Register-to-register operations will be greatly accelerated, but this will only lengthen the time the registers wait for new data from the memory subsystem.

In Figure 1.8, a single precision ADD instruction was depicted that might be used in an array addition loop:

```
real, dimension(1:N) :: r, x, y
do i=1, N
  r(i) = x(i) + y(i)
enddo
```

All iterations in this loop are independent, there is no branch in the loop body, and the arrays are accessed with a stride of one. However, the use of SIMD requires some rearrangement of a loop kernel like the one above to be applicable: A number of iterations equal to the SIMD register size has to be executed as a single “chunk” without any branches in between. This is actually a well-known optimization that can pay off even without SIMD and is called *loop unrolling* (see Section 3.5 for more details outside the SIMD context). Since the overall number of iterations is generally not a multiple of the register size, some remainder loop is left to execute in scalar mode. In pseudocode, and ignoring software pipelining (see Section 1.2.3), this could look like the following:

```
! vectorized part
rest = mod(N,4)
do i=1,N-rest,4
  load R1 = [x(i),x(i+1),x(i+2),x(i+3)]
  load R2 = [y(i),y(i+1),y(i+2),y(i+3)]
  ! "packed" addition (4 SP flops)
  R3 = ADD(R1,R2)
  store [r(i),r(i+1),r(i+2),r(i+3)] = R3
enddo
! remainder loop
do i=N-rest+1,N
  r(i) = x(i) + y(i)
enddo
```

`R1`, `R2`, and `R3` denote 128-bit SIMD registers here. In an optimal situation all this is carried out by the compiler automatically. Compiler directives can be used to give hints as to where vectorization is safe and/or beneficial.

The SIMD load and store instructions suggested in this example might need some special care. Some SIMD instruction sets distinguish between *aligned* and *unaligned* data. For example, in the x86 (Intel/AMD) case, the "packed" SSE load and store instructions exist in aligned and unaligned flavors [V107, O54]. If an aligned load or store is used on a memory address that is not a multiple of 16, an exception occurs. In cases where the compiler knows nothing about the alignment of arrays used in a vectorized loop and cannot otherwise influence it, unaligned (or a sequence of scalar) loads and stores must be used, incurring some performance penalty. The programmer can force the compiler to assume optimal alignment, but this is dangerous if one cannot make absolutely sure that the assumption is justified. On some architectures alignment issues can be decisive; every effort must then be made to align all loads and stores to the appropriate address boundaries.

A loop with a true dependency as discussed in Section 1.2.3 cannot be SIMD-vectorized in this way (there is a twist to this, however; see Problem 2.2):

```
do i=2,N
  A(i)=s*A(i-1)
enddo
```

The compiler will revert to scalar operations here, which means that only the lowest operand in the SIMD registers is used (on x86 architectures).

Note that there are no fixed guidelines for when a loop qualifies as vectorized. One (maybe the weakest) possible definition is that all arithmetic within the loop is executed using the full width of SIMD registers. Even so, the load and store instructions could still be scalar; compilers tend to report such loops as "vectorized" as well. On x86 processors with SSE support, the lower and higher 64 bits of a register can be moved independently. The vector addition loop above could thus look as follows in double precision:

```
1  rest = mod(N,2)
2  do i=1,N-rest,2
3    ! scalar loads
4    load R1.low = x(i)
5    load R1.high = x(i+1)
6    load R2.low = y(i)
7    load R2.high = y(i+1)
8    ! "packed" addition (2 DP flops)
9    R3 = ADD(R1,R2)
10   ! scalar stores
11   store r(i) = R3.low
12   store r(i+1) = R3.high
13 enddo
14 ! remainder "loop"
15 if(rest.eq.1) r(N) = x(N) + y(N)
```

This version will not give the best performance if the operands reside in a cache. Although the actual arithmetic operations (line 9) are SIMD-parallel, all loads and stores are scalar. Lacking extensive compiler reports, the only option to identify such a failure is manual inspection of the generated assembly code. If the compiler cannot be convinced to properly vectorize a loop even with additional command line options or source code directives, a typical "last resort" before using assembly language altogether is to employ *compiler intrinsics*. Intrinsics are constructs that resemble assembly instructions so closely that they can usually be translated 1:1 by the compiler. However, the user is relieved from the burden of keeping track of individual registers, because the compiler provides special data types that map to SIMD operands. Intrinsics are not only useful for vectorization but can be beneficial in all cases where high-level language constructs cannot be optimally mapped to some CPU functionality. Unfortunately, intrinsics are usually not compatible across compilers even on the same architecture [V112].

Finally, it must be stressed that in contrast to real vector processors, RISC systems will not always benefit from vectorization. If a memory-bound code can be optimized for heavy data reuse from registers or cache (see Chapter 3 for examples), the potential gains are so huge that it may be acceptable to give up vectorizability along the way.

2.4 The role of compilers

Most high-performance codes benefit, to varying degrees, from employing compiler-based optimizations. Every modern compiler has command line switches that allow a (more or less) fine-grained tuning of the available optimization options. Sometimes it is even worthwhile trying a different compiler just to check whether there is more performance potential. One should be aware that the compiler has the extremely complex job of mapping source code written in a high-level language to machine code, thereby utilizing the processor's internal resources as well as possible. Some of the optimizations described in this and the next chapter can be applied by the compiler itself in simple situations. However, there is no guarantee that this is actually the case and the programmer should at least be aware of the basic strategies for automatic optimization and potential stumbling blocks that prevent the latter from being applied. It must be understood that compilers can be surprisingly smart and stupid at the same time. A common statement in discussions about compiler capabilities is "The compiler should be able to figure that out." This is often enough a false assumption.

Ref. [C91] provides a comprehensive overview on optimization capabilities of several current C/C++ compilers, together with useful hints and guidelines for manual optimization.

2.4.1 General optimization options

Every compiler offers a collection of standard optimization options (`-O0`, `-O1`,`...`). What kinds of optimizations are employed at which level is by no means standardized and often (but not always) documented in the manuals. However, all compilers refrain from most optimizations at level `-O0`, which is hence the correct choice for analyzing the code with a debugger. At higher levels, optimizing compilers mix up source lines, detect and eliminate "redundant" variables, rearrange arithmetic expressions, etc., so that any debugger has a hard time giving the user a consistent view on code and data.

Unfortunately, some problems seem to appear only with higher optimization levels. This might indicate a defect in the compiler, however it is also possible that a typical bug like an array bounds violation (reading or writing beyond the boundaries of an array) is "harmless" at `-O0` because data is arranged differently than at `-O3`. Such bugs are notoriously hard to spot, and sometimes even the popular "printf debugging" does not help because it interferes with the optimizer.

2.4.2 Inlining

Inlining tries to save overhead by inserting the complete code of a function or subroutine at the place where it is called. Each function call uses up resources because arguments have to be passed, either in registers or via the stack (depending on the number of parameters and the calling conventions used). While the *scope* of the former function (local variables, etc.) must be established anyway, inlining does remove the necessity to push arguments onto the stack and enables the compiler to use registers as it deems necessary (and not according to some calling convention), thereby reducing *register pressure*. Register pressure occurs if the CPU does not have enough registers to hold all the required operands inside a complex computation or loop body (see also Section 2.4.5 for more information on register usage). And finally, inlining a function allows the compiler to view a larger portion of code and probably employ optimizations that would otherwise not be possible. The programmer should never rely on the compiler to optimize inlined code perfectly, though; in performance-critical situations (like tight loop kernels), obfuscating the compiler's view on the "real" code is usually counterproductive.

Whether the call overhead impacts performance depends on how much time is spent in the function body itself; naturally, frequently called small functions bear the highest speedup potential if inlined. In many C++ codes, inlining is absolutely essential to get good performance because overloaded operators for simple types tend to be small functions, and temporary copies can be avoided if an inlined function returns an object (see Section 2.5 for details on C++ optimization).

Compilers usually have various options to control the extent of automatic inlining, e.g., how large (in terms of the number of lines) a subroutine may be to become an inlining candidate, etc. Note that the c99 and C++ `inline` keyword is only a hint to the compiler. A compiler log (if available, see Section 2.4.6) should be consulted to see whether a function was really inlined.

On the downside, inlining a function in multiple places can enlarge the object code considerably, which may lead to problems with L1 instruction cache capacity. If the instructions belonging to a loop cannot be fetched from L1I cache, they compete with data transfers to and from outer-level cache or main memory, and the latency for fetching instructions becomes larger. Thus one should be cautious about altering the compiler's inlining heuristics, and carefully check the effectiveness of manual interventions.

2.4.3 Aliasing

The compiler, guided by the rules of the programming language and its interpretation of the source, must make certain assumptions that may limit its ability to generate optimal machine code. The typical example arises with pointer (or reference) formal parameters in the C (and C++) language:

```
void scale_shift(double *a, double *b, double s, int n) {
  for(int i=1; i<n; ++i)
    a[i] = s*b[i-1];
}
```

Assuming that the memory regions pointed to by `a` and `b` do not overlap, i.e., the ranges [`a`,`a+n−1`] and [`b`,`b+n−1`] are disjoint, the loads and stores in the loop can be arranged in any order. The compiler can apply any software pipelining scheme it considers appropriate, or it could unroll the loop and group loads and stores in blocks, as shown in the following pseudocode (we ignore the remainder loop):

```
loop:
  load R1 = b(i+1)
  load R2 = b(i+2)
  R1 = MULT(s,R1)
  R2 = MULT(s,R2)
  store a(i) = R1
  store a(i+1) = R2
  i = i + 2
  branch -> loop
```

In this form, the loop could easily be SIMD-vectorized as well (see Section 2.3.3).

However, the C and C++ standards allow for arbitrary *aliasing* of pointers. It must thus be assumed that the memory regions pointed to by `a` and `b` do overlap. For instance, if `a==b`, the loop is identical to the "real dependency" Fortran example on page 12; loads and stores must be executed in the same order in which they appear in the code:

```
loop:
  load R1 = b(i+1)
  R1 = MULT(s,R1)
  store a(i) = R1
  load R2 = b(i+2)
  R2 = MULT(s,R2)
  store a(i+1) = R2
  i = i + 2
  branch -> loop
```

Lacking any further information, the compiler must generate machine instructions according to this scheme. Among other things, SIMD vectorization is ruled out. The processor hardware allows reordering of loads and stores within certain limits [V104, V105], but this can of course never alter the program's semantics.

Argument aliasing is forbidden by the Fortran standard, and this is one of the main reasons why Fortran programs tend to be faster than equivalent C programs. All C/C++ compilers have command line options to control the level of aliasing the compiler is allowed to assume (e.g., `-fno-fnalias` for the Intel compiler and `-fargument-noalias` for the GCC specify that no two pointer arguments for any function ever point to the same location). If the compiler is told that argument aliasing does not occur, it can in principle apply the same optimizations as in equivalent Fortran code. Of course, the programmer should not "lie" in this case, as calling a function with aliased arguments will then probably produce wrong results.

2.4.4 Computational accuracy

As already mentioned in Section 2.3.1, compilers sometimes refrain from rearranging arithmetic expressions if this required applying associativity rules, except with very aggressive optimizations turned on. The reason for this is the infamous nonassociativity of FP operations [135]: `(a+b)+c` is, in general, not identical to `a+(b+c)` if `a`, `b`, and `c` are finite-precision floating-point numbers. If accuracy is to be maintained compared to nonoptimized code, associativity rules must not be used and it is left to the programmer to decide whether it is safe to regroup expressions by hand. Modern compilers have command line options that limit rearrangement of arithmetic expressions even at high optimization levels.

Note also that *denormals*, i.e., floating-point numbers that are smaller than the smallest representable number with a nonzero lead digit, can have a significant impact on computational performance. If possible, and if the slight loss in accuracy is tolerable, such numbers should be treated as ("flushed to") zero by the hardware.

Listing 2.1: Compiler log for a software pipelined triad loop. "Peak" indicates the maximum possible execution rate for the respective operation type on this architecture (MIPS R14000).

```
#<swps> 16383 estimated iterations before pipelining
#<swps>     4 unrollings before pipelining
#<swps>    20 cycles per 4 iterations
#<swps>     8 flops        ( 20% of peak) (madds count as 2)
#<swps>     4 flops        ( 10% of peak) (madds count as 1)
#<swps>     4 madds        ( 20% of peak)
#<swps>    16 mem refs     ( 80% of peak)
#<swps>     5 integer ops  ( 12% of peak)
#<swps>    25 instructions ( 31% of peak)
#<swps>     2 short trip threshold
#<swps>    13 integer registers used.
#<swps>    17 float registers used.
```

2.4.5 Register optimizations

It is one of the most vital, but also most complex tasks of the compiler to care about register usage. The compiler tries to put operands that are used "most often" into registers and keep them there as long as possible, given that it is safe to do so. If, e.g., a variable's address is taken, its value might be manipulated elsewhere in the program via the address. In this case the compiler may decide to write a variable back to memory right after any change on it.

Inlining (see Section 2.4.2) will help with register optimizations since the optimizer can probably keep values in registers that would otherwise have to be written to memory before the function call and read back afterwards. On the downside, loop bodies with lots of variables and many arithmetic expressions (which can easily occur after inlining) are hard for the compiler to optimize because it is likely that there are too few registers to hold all operands at the same time. As mentioned earlier, the number of integer and floating-point registers in any processor is strictly limited. Today, typical numbers range from 8 to 128, the latter being a gross exception, however. If there is a register shortage, variables have to be *spilled*, i.e., written to memory, for later use. If the code's performance is determined by arithmetic operations, register spill can hamper performance quite a bit. In such cases it may even be worthwhile splitting a loop in two to reduce register pressure.

Some processors with hardware support for spilling like, e.g., Intel's Itanium2, feature hardware performance counter metrics, which allow direct identification of register spill.

2.4.6 Using compiler logs

The previous sections have pointed out that the compiler is a crucial component in writing efficient code. It is very easy to hide important information from the compiler, forcing it to give up optimization at an early stage. In order to make the decisions of the compiler's "intelligence" available to the user, many compilers offer options to generate *annotated source code* listings or at least *logs* that describe in some detail what optimizations were performed. Listing 2.1 shows an example for a compiler annotation regarding a standard vector triad loop as in Listing 1.1, for the (now outdated) MIPS R14000 processor. This CPU was four-way superscalar, with the ability to execute one load or store, two integer, one FP add and one FP multiply operation per cycle (the latter two in the form of a fused multiply-add [“`madd`”] instruction). Assuming that all data is available from the inner level cache, the compiler can calculate the minimum number of cycles required to execute one loop iteration (line 3). Percentages of Peak, i.e., the maximum possible throughput for every type of operation, are indicated in lines 4–9.

Additionally, information about register usage and spill (lines 11 and 12), unrolling factors and software pipelining (line 2, see Sections 1.2.3 and 3.5), use of SIMD instructions (see Section 2.3.3), and the compiler's assumptions about loop length (line 1) are valuable for judging the quality of generated machine code. Unfortunately, not all compilers have the ability to write such comprehensive code annotations and users are often left with guesswork.

Certainly there is always the option of manually inspecting the generated assembly code. All compilers provide command line options to output an assembly listing instead of a linkable object file. However, matching this listing with the original source code and analyzing the effectiveness of the instruction sequences requires a considerable amount of experience [O55]. After all there *is* a reason for people not writing programs in assembly language all the time.

2.5 C++ optimizations

There is a host of literature dealing with how to write efficient C++ code [C92, C93, C94, C95], and it is not our ambition to supersede it here. We also deliberately omit standard techniques like reference counting, copy-on-write, smart pointers, etc. In this section we will rather point out, in our experience, the most common performance bugs and misconceptions in C++ programs, with a focus on low-level loops.

One of the ineradicable illusions about C++ is that the compiler should be able to see through all the abstractions and obfuscations an “advanced” C++ program contains. First and foremost, C++ should be seen as a language that enables *complexity management.* The features one has grown fond of in this concept, like operator overloading, object orientation, automatic construction/destruction, etc., are however mostly unsuitable for efficient low-level code.

2.5.1 Temporaries

C++ fosters an “implicit” programming style where automatic mechanisms hide complexity from the programmer. A frequent problem occurs with expressions containing chains of overloaded operators. As an example, assume there is a `vec3d` class, which represents a vector in three-dimensional space. Overloaded arithmetic operators then allow expressive coding:

```
class vec3d {
  double x,y,z;
  friend vec3d operator*(double, const vec3d&);
public:
  vec3d(double _x=0.0, double _y=0.0, double _z=0.0) : // 4 ctors
        x(_x),y(_y),z(_z) {}
  vec3d(const vec3d &other);
  vec3d operator=(const vec3d &other);
  vec3d operator+(const vec3d &other) {
    vec3d tmp;
    tmp.x = x + other.x;
    tmp.y = y + other.y;
    tmp.z = z + other.z;
  }
  vec3d operator*(const vec3d &other);
  ...
};

vec3d operator*(double s, const vec3d& v) {
  vec3d tmp(s*v.x,s*v,y,s*v.z);
}
```

Here we show only the implementation of the `vec3d::operator+` method and the friend function for multiplication by a scalar. Other useful functions are defined in a similar way. Note that copy constructors and assignment are shown for reference as prototypes, but are implicitly defined because shallow copy and assignment are sufficient for this simple class.

The following code fragment shall serve as an instructive example of what really goes on behind the scenes when a class is used:

```
vec3d a,b(2,2),c(3);
double x=1.0,y=2.0;

a = x*b + y*c;
```

In this example the following steps will occur (roughly) in this order:

1. Constructors for `a`, `b`, `c`, and `d` are called (the default constructor is implemented via default arguments to the parameterized constructor)

2. `operator*(x, b)` is called

3. The `vec3d` constructor is called to initialize `tmp` in `operator*(double s, const vec3d& v)` (here we have already chosen the more efficient three-parameter constructor instead of the default constructor followed by assignment from another temporary)

4. Since `tmp` must be destroyed once `operator*(double, const vec3d&)` returns, `vec3d`'s copy

constructor is invoked to make a temporary copy of the result, to be used as the first argument in the vector addition

5. `operator*(y, c)` is called

6. The `vec3d` constructor is called to initialize `tmp` in `operator*(double s, const vec3d& v)`

7. Since `tmp` must be destroyed once `operator*(double, const vec3d&)` returns, `vec3d`'s copy constructor is invoked to make a temporary copy of the result, to be used as the second argument in the vector addition

8. `vec3d::operator+(const vec3d&)` is called in the first temporary object with the second as a parameter

9. `vec3d`'s default constructor is called to make `tmp` in `vec3d::operator+`

10. `vec3d`'s copy constructor is invoked to make a temporary copy of the summation's result

11. `vec3d`'s assignment operator is called in `a` with the temporary result as its argument

Although the compiler may eliminate the local `tmp` objects by the so-called *return value optimization* [C92] using the required implicit temporary directly instead of `tmp`, it is striking how much code gets executed for this seemingly simple expression (a debugger can help a lot with getting more insight here). A straightforward optimization, at the price of some readability, is to use compound computational/assignment operators like `operator+=`:

```
a  = y*c;
a += x*b;
```

Two temporaries are still required here to transport the results from `operator*(double, const vec3d&)` back into the main function, but they are used in an assignment and `vec3d::operator+=` right away without the need for a third temporary. The benefit is even more noticeable with longer operator chains.

However, even if a lot of compute time is spent handling temporaries, calling copy constructors, etc., this fact is not necessarily evident from a standard function profile like the ones shown in Section 2.1.1. C++ compilers are, necessarily, quite good at function inlining. Much of the implicit "magic" going on could thus be summarized as, e.g., exclusive runtime of the function invoking a complex expression. Disabling inlining, although generally advised against, might help to get more insight in this situation, but it will distort the results considerably.

Despite aggressive inlining the compiler will most probably not generate "optimal" code, which would roughly look like this:

```
a.x = x*b.x + y*c.x;
a.y = x*b.y + y*c.y;
a.z = x*b.z + y*c.z;
```

Expression templates [C96, C97] are an advanced programming technique that can supposedly lift many of the performance problems incurred by temporaries, and actually produce code like this from high-level expressions.

It should nonetheless be clear that it is not the purpose of C++ inlining to produce the optimal code, but to rectify the most severe performance penalties incurred by the language specification. Loop kernels bound by memory or even cache bandwidth, or arithmetic throughput, are best written either in C (or C style) or Fortran. See Section 2.5.3 for details.

2.5.2 Dynamic memory management

Another common bottleneck in C++ codes is frequent allocation and deallocation. There was no dynamic memory involved in the simple 3D vector class example above, so there was no problem with abundant (de)allocations. Had we chosen to use a general vector-like class with variable size, the performance implications of temporaries would have been even more severe, because construction and destruction of each temporary would have called `malloc()` and `free()`, respectively. Since the standard library functions are not optimized for minimal overhead, this can seriously harm overall performance. This is why C++ programmers go to great lengths trying to reduce the impact of allocation and deallocation [C98].

Avoiding temporaries is of course one of the key measures here (see the previous section), but two other strategies are worth noting: *Lazy construction* and *static construction*. These two seem somewhat contrary, but both have useful applications.

Lazy construction

For C programmers who adopted C++ as a "second language" it is natural to collect object declarations at the top of a function instead of moving each declaration to the place where it is needed. The former is required by C, and there is no performance problem with it as long as only basic data types are used. An expensive constructor should be avoided as far as possible, however:

```
1 void f(double threshold, int length) {
2   std::vector<double> v(length);
3   if(rand() > threshold*RAND_MAX) {
4     v = obtain_data(length);
5     std::sort(v.begin(), v.end());
6     process_data(v);
7   }
8 }
```

In line 2, construction of `v` is done unconditionally although the probability that it is really needed might be low (depending on `threshold`). A better solution is to defer construction until this decision has been made:

```
1  void f(double threshold, int length) {
2    if(rand() > threshold*RAND_MAX) {
3      std::vector<double> v(obtain_data(length));
4      std::sort(v.begin(), v.end());
5      process_data(v);
6    }
7  }
```

As a positive side effect we now call the copy constructor of `std::vector<>` (line 3) instead of the `int` constructor followed by an assignment.

Static construction

Moving the construction of an object to the *outside* of a loop or block, or making it `static` altogether, may even be faster than lazy construction if the object is used often. In the example above, if the array length is constant and `threshold` is usually close to 1, static allocation will make sure that construction overhead is negligible since it only has to be paid once:

```
1  const int length=1000;
2
3  void f(double threshold) {
4    static std::vector<double> v(length);
5    if(rand() > threshold*RAND_MAX) {
6      v = obtain_data(length);
7      std::sort(v.begin(), v.end());
8      process_data(v);
9    }
10 }
```

The vector object is instantiated only once in line 4, and there is no subsequent allocation overhead. With a variable length there is the chance that memory would have to be re-allocated upon assignment, incurring the same cost as a normal constructor (see also Problem 2.4). In general, if assignment is faster (on average) than (re-)allocation, static construction will be faster.

Note that special care has to be taken of static data in shared-memory parallel programs; see Section 6.1.4 for details.

2.5.3 Loop kernels and iterators

The runtime of scientific applications tends to be dominated by loops or loop nests, and the compiler's ability to optimize those loops is pivotal for getting good code performance. Operator overloading, convenient as it may be, hinders good loop optimization. In the following example, the template function `sprod<>()` is responsible for carrying out a scalar product over two vectors:

```
1  using namespace std;
2
3  template<class T> T sprod(const vector<T> &a, const vector<T> &b) {
4    T result=T(0);
5    int s = a.size();
6    for(int i=0; i<s; ++i)    // not SIMD vectorized
7      result += a[i] * b[i];
8    return result;
9  }
```

In line 7, `const T& vector<T>::operator[]` is called twice to obtain the current entries from `a` and `b`. STL may define this operator in the following way (adapted from the GNU ISO C++ library source):

```
const T& operator[](size_t __n) const
    { return *(this->_M_impl._M_start + __n); }
```

Although this looks simple enough to be inlined efficiently, current compilers refuse to apply SIMD vectorization to the summation loop above. A single layer of abstraction, in this case an overloaded index operator, can thus prevent the creation of optimal loop code (and we are not even referring to more complex, high-level loop transformations like those described in Chapter 3). However, using iterators for array access, vectorization is not a problem:

```
template<class T> T sprod(const vector<T> &a, const vector<T> &b) {
  typename vector<T>::const_iterator ia=a.begin(),ib=b.begin();
  T result=T(0);
  int s = a.size();
  for(int i=0; i<s; ++i)    // SIMD vectorized
    result += ia[i] * ib[i];
  return result;
}
```

Because `vector<T>::const_iterator` is `const T*`, the compiler sees normal C code. The use of iterators instead of methods for data access can be a powerful optimization method in C++. If possible, low-level loops should even reside in separate compilation units (and written in C or Fortran), and iterators be passed as pointers. This ensures minimal interference with the compiler's view on the high-level C++ code.

The `std::vector<>` template is a particularly rewarding case because its iterators are implemented as standard (C) pointers, but it is also the most frequently used container. More complex containers have more complex iterator classes, and those may not be easily convertible to raw pointers. In cases where it is possible to represent data in a "segmented" structure with multiple `vector<>`-like components (a matrix being the standard example), the use of *segmented iterators* still enables fast low-level algorithms. See [C99, C100] for details.

Problems

For solutions see page 288 *ff.*

2.1 *The perils of branching.* Consider this benchmark code for a stride-one triad "with a twist":

```
do i=1,N
  if(C(i)<0.d0) then
    A(i) = B(i) - C(i) * D(i)
  else
    A(i) = B(i) + C(i) * D(i)
  endif
enddo
```

What performance impact do you expect from the conditional compared to the standard vector triad if array `C` is initialized with (a) positive values only (b) negative values only (c) random values between −1 and 1 for loop lengths that fit in L1 cache, L2 cache, and memory, respectively?

2.2 *SIMD despite recursion?* In Section 1.2.3 we have studied the influence of loop-carried dependencies on pipelining using the following loop kernel:

```
start=max(1,1-offset)
end=min(N,N-offset)
do i=start,end
  A(i)=s*A(i+offset)
enddo
```

If `A` is an array of single precision floating-point numbers, for which values of `offset` is SIMD vectorization as shown in Figure 1.8 possible?

2.3 *Lazy construction on the stack.* If we had used a standard C-style `double` array instead of a `std::vector<double>` for the lazy construction example in Section 2.5.2, would it make a difference where it was declared?

2.4 *Fast assignment.* In the static construction example in Section 2.5.2 we stated that the benefit of a static `std::vector<>` object can only be seen with a constant vector length, because assignment leads to re-allocation if the length can change. Is this really true?