Software optimization for High Performance Computing

Gérald Monard

RFCT - Nord-Est - 2014
http://www.monard.info
2.2 Common sense optimizations ............................................ 52
  2.2.1 Do less work! ......................................................... 53
  2.2.2 Avoid expensive operations! ..................................... 55
  2.2.3 Shrink the working set! .......................................... 58
2.3 Simple measures, large impact ........................................ 59
  2.3.1 Elimination of common subexpressions ....................... 59
  2.3.2 Avoiding branches ................................................ 60
  2.3.3 Using SIMD instruction sets ..................................... 63
2.4 The role of compilers .................................................. 65
  2.4.1 General optimization options ................................... 65
  2.4.2 Inlining ............................................................. 66
  2.4.3 Aliasing ............................................................ 67
  2.4.4 Computational accuracy ......................................... 69
  2.4.5 Using compiler logs .............................................. 70

3 Parallel computers ....................................................... 71
  3.1 Taxonomy of parallel computing paradigms ....................... 72
3.2 Shared-memory computers ........................................ 73
  3.2.1 Cache coherence .............................................. 74
  3.2.2 UMA ............................................................. 75
  3.2.3 ccNUMA .......................................................... 76
3.3 Distributed-memory computers .................................... 77

4 Basics of parallelization .............................................. 80
  4.1 Why parallelize? .................................................. 80
  4.2 Parallelism ....................................................... 81
    4.2.1 Data parallelism ............................................ 81
    4.2.2 Functional parallelism .................................... 82
  4.3 Parallel scalability .............................................. 83
    4.3.1 Speedup and efficiency ................................... 84
    4.3.2 Amdahl’s Law ............................................... 85
    4.3.3 Gustafson-Barsis’s Law ................................... 86

5 Shared-memory parallel programming with OpenMP ............ 88
  5.1 Parallel execution ............................................... 89
Bibliography

• Introduction to High Performance Computing for Scientists and Engineers
  G. Hager, G. Wellein — CRC Press

• High Performance Computing
  K. Dowd, C. Severance — O’Reilly

• Parallel Programming in C with MPI and OpenMP
  M. J. Quinn — McGraw Hill

• Computer Science, An Overview
  J. G. Brookshear — Addison-Wesley
1 Modern Processors

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, CDC, NEC, Fujitsu, or Thinking Machines.

Those systems were way ahead of standard “commodity” computers in terms of performance and price.

Single-chip general-purpose 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 theoretical peak performance.

Today the situation has changed considerably. The HPC world is dominated by cost-effective, off-the-shelf systems with processors that were not primarily designed for scientific computing.
1.1 Stored-program computer architecture

When we talk about computer systems at large, we always have a certain architectural concept in mind. This concept was conceived by Turing in 1936, and first implemented in a real machine (EDVAC) in 1949 by Eckert and Mauchly.

Figure 1.1: Stored-program computer architectural concept. The “program,” which feeds the control unit, is stored in memory together with any data the arithmetic unit requires.
In a stored-program digital computer: its instructions are numbers that are stored as data in memory.

- Instructions are read and executed by a control unit;
- a separate arithmetic/logic unit is responsible for the actual computations and manipulates data stored in memory along with the instructions;
- I/O facilities enable communication with users.
- Control and arithmetic units together with the appropriate interfaces to memory and I/O are called the Central Processing Unit (CPU).

Programming a stored-program computer amounts to modifying instructions in memory, which can in principle be done by another program; a compiler is a typical example, because it translates the constructs of a high-level language like C or Fortran into instructions that can be stored in memory and then executed by a computer.
This blueprint is the basis for all mainstream computer systems today, and its inherent problems still prevail:

- Instructions and data must be continuously fed to the control and arithmetic units, so that the speed of the memory interface poses a limitation on compute performance. This is often called the von Neumann bottleneck.

Architectural optimizations and programming techniques may mitigate the adverse effects of this constriction, but it should be clear that it remains a most severe limiting factor.

- The architecture is inherently sequential, processing a single instruction with (possibly) a single operand or a group of operands from memory. The term SISD (Single Instruction Single Data) has been coined for this concept.
1.2 General-purpose cache-based microprocessor architecture

Figure 1.2: Simplified block diagram of a typical cache-based microprocessor (one core). Other cores on the same chip or package (socket) can share resources like caches or the memory interface. The functional blocks and data paths most relevant to performance issues in scientific computing are highlighted.
Figure 1.2 shows a very simplified block diagram of a modern cache-based general-purpose 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 the chip area.

The rest consists of administrative logic that helps to feed those units with operands.

- **CPU registers**, which are generally divided into floating-point and integer (or “general purpose”) varieties, can hold operands to be accessed by instructions with no significant delay; in some architectures, all operands for arithmetic operations must reside in registers.

  Typical CPUs nowadays have between 16 and 128 user-visible 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, prob-
ably not in the order they were issued.

- Finally, caches hold data and instructions to be (re-)used soon.

The major part of the chip area is usually occupied by caches.

1.2.1 Performance metrics and benchmarks

All the components of a CPU core can operate at some maximum speed called peak performance.

Scientific computing tends to be quite centric to floating-point data, usually with “double precision” (DP).

The performance at which the FP units generate 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 operations often share execution resources with multiply and add units, and are executed so slowly as to not contribute significantly to overall performance in practice.
High performance software should thus try to avoid such operations as far as possible.

Actually, standard commodity microprocessors are designed to deliver at most two or four double-precision floating-point results per clock cycle. With typical clock frequencies between 2 and 3 GHz, this leads to a peak arithmetic performance between 4 and 12 GFlops/sec per core.

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 performance, 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.
A “computation” or algorithm of some kind is usually defined by manipulation of data items; a concrete implementation of the algorithm must, however, run on real hardware, with limited performance on all data paths, especially those to main memory.

**Figure 1.3:** (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. (Right) The “DRAM gap” denotes the large discrepancy between main memory and cache bandwidths. This model must be mapped to the data access requirements of an application.
Fathoming the chief performance characteristics of a processor or system 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 prominent examples is the vector triad, introduced by Schönauser. It comprises a nested loop, the inner level executing a multiply-add operation on the elements of three vectors and storing the result in a fourth.
Listed 1.1: Basic code fragment for the vector triad benchmark, including performance measurement.

```
1 double precision, dimension(N) :: A, B, C, D
2 double precision :: S, E, MFLOPS
3
4 do i=1,N
5     A(i) = 0.d0; B(i) = 1.d0
6     C(i) = 2.d0; D(i) = 3.d0
7     enddo
8
9 call get_walltime(S) ! get time stamp
10 do j=1,R
11     do i=1,N
12     A(i) = B(i) + C(i) * D(i) ! 3 loads, 1 store
13     enddo
14     if(A(2).lt.0) call dummy(A, B, C, D) ! prevent loop interchange
15     enddo
16 call get_walltime(E) ! get time stamp
17 MFLOPS = R*N*2.d0/((E-S)*1.d6) ! compute MFlop/sec rate
```

The purpose of this benchmark is to measure the performance of data transfers between memory and arithmetic units of a processor. 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 practice one would choose R according to N so that the overall execution time stays roughly constant for different N.

The aim of the masked-out call to 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 possible call to dummy() fools the compiler into believing that the arrays may change between outer loop iterations. This effectively prevents the optimization described, and the additional cost is negligible because the condition is always false (which the compiler does not know).

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, also called elapsed time.

Any other “time” that the system may provide, first and foremost the much stressed CPU time, is prone to misinterpretation because there might be contributions from I/O, context switches, other processes, etc., which CPU time cannot encompass. This is even more true for parallel programs.

A useful C routine to get a wallclock time stamp like the one used in the triad benchmark above could look like in Listing 1.2. The reason for providing the function with and without a trailing underscore is that Fortran compilers usually append an underscore to subroutine names. With both versions available, linking the compiled C code to a main program in Fortran or C will always work.
Listing 1.2: A C routine for measuring wallclock time, based on the `gettimeofday()` POSIX function. Under the Windows OS, the `GetSystemTimeAsFileTime()` routine can be used in a similar way.

```c
#include <sys/time.h>

void get_walltime_(double* wcTime) {
    struct timeval tp;
    gettimeofday(&tp, NULL);
    *wcTime = (double)(tp.tv_sec + tp.tv_usec/1000000.0);
}

void get_walltime(double* wcTime) {
    get_walltime_(wcTime);
}
```
Figure 1.4: Serial vector triad performance versus loop length for several generations of Intel processor architectures (clock speed and year of introduction is indicated), and the NEC SX-8 vector processor. Note the entirely different performance characteristics of the latter.
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” application 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 or before writing a proposal for a computer time grant), 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.
1.2.2 Moore’s Law

Computer technology had been used for scientific purposes and, more specifically, for numerically demanding calculations long before the dawn of the desktop PC.

For more than thirty 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.

This trend is commonly ascribed to Moore’s Law.

Gordon Moore, co-founder of Intel Corp., postulated in 1965 that the number of components (transistors) on a chip that are required to hit the “sweet spot” of minimal manufacturing cost per component would continue to increase at the indicated rate.

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 “per-
formance” remains debatable as a processor is not the only component in a
computer.

Increasing chip transistor counts and clock speeds have enabled processor de-
signers to implement many advanced techniques that lead to improved appli-
cation 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 dif-
ferent 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. Actually, processor designs exist that feature pipelines with
more than 30 stages.
2. **Superscalar architecture.** Superscalarity provides “direct” instruction-level parallelism by enabling an instruction throughput of more than one per cycle.

This requires multiple, possibly identical functional units, which can operate currently.

Modern microprocessors are up to six-way superscalar.

3. **Data parallelism through SIMD instructions.** SIMD (Single Instruction Multiple Data) instructions issue identical operations on a whole array of integer or FP operands, usually in special registers.

They improve arithmetic peak performance without the requirement for increased superscalarity.

Examples are Intel’s “SSE” and its successors, AMD’s “3dNow!,” the “AltiVec” extensions in Power and PowerPC processors, and the “VIS” instruction set in Sun’s UltraSPARC designs.
4. **Out-of-order execution.** If arguments to instructions are not available in registers “on time,” e.g., because the memory subsystem is too slow to keep up with processor speed, out-of-order execution can avoid idle times (also called **stalls**) 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.
5. **Larger caches.** Small, fast, on-chip memories serve as temporary data storage for holding copies of 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.

Enlarging the cache size does usually not hurt application performance, but there is some tradeoff because a big cache tends to be slower than a small one.

6. **Simplified instruction set.** In the 1980s, a general move from the CISC to the RISC paradigm took place.

In a CISC (Complex Instruction Set Computer), a processor executes very complex, powerful instructions, requiring a large hardware effort for decoding but keeping programs small and compact.

This lightened the burden on programmers, and saved memory, which was a scarce resource for a long time.
A RISC (Reduced Instruction Set Computer) 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.

Although x86-based processors execute CISC machine code, they perform an internal on-the-fly translation into RISC “μ-ops.”

1.2.3 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
Figure 1.5: Timeline for a simplified floating-point multiplication pipeline that executes $A(\cdot) = B(\cdot) \times C(\cdot)$. One result is generated on each cycle after a four-cycle wind-up phase.
Complex operations like loading and storing data or performing floating-point arithmetic cannot be executed in a single cycle without excessive hardware requirements.

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 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.

The wind-up phase corresponds to the latency (or depth) of the pipeline: it is the number of cycle necessary to obtain the first result.

The throughput is the number of results available per cycle.

When the first pipeline stage has finished working on the last item, the wind-down phase starts.
1.2.4 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 (3–6 nowadays).
- Address and other integer calculations are performed in multiple integer (add, mult, shift, mask) units (2–6). This is closely related to the previous point, because feeding those units requires code execution.
- Multiple floating-point pipelines can run in parallel. Often there are one or two combined multiply-add pipes that perform $a = b + c \times d$ with a throughput of one each.
- Caches are fast enough to sustain more than one load or store operation per cycle, and the number of available execution units for loads and stores reflects that (2–4).
1.2.5 SIMD

Figure 1.8: Example for SIMD: Single precision FP addition of two SIMD registers (x,y), each having a length of 128 bits. Four SP flops are executed in a single instruction.
1.3 Memory hierarchies

Data can be stored in a computer system in many different ways. CPUs have a set of registers, which can be accessed without delay. In addition there are one or more small but very fast caches holding copies of recently used data items. 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.
1.3.1 Cache

Caches are low-capacity, high-speed memories that are commonly integrated on the CPU die.

Usually there are at least two levels of cache, called L1 and L2, respectively. L1 is normally split into two parts, one for instructions (“I-cache,” “L1I”) and one for data (“L1D”).

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.

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 a cache are to be used again “soon enough” to not have been evicted in the meantime. This is also called temporal locality.
1.3.2 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.

Figure 1.12: Timing diagram on the influence of cache misses and subsequent latency penalties for a vector norm loop. The penalty occurs on each new miss.
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 asynchronously.
Figure 1.13: Computation and data transfer can be overlapped much better with prefetching. In this example, two outstanding prefetches are required to hide latency completely.
1.4 Multicoreprocessors

In recent years it has become increasingly clear that, although Moore’s Law is still valid and will probably be at least for the next decade, standard micro-processors 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 power-performance dilemma in the form of multicore designs.
**Figure 1.15:** Dual-core processor chip with separate L1, L2, and L3 caches (Intel “Montecito”). Each core constitutes its own cache group on all levels.

**Figure 1.16:** Quad-core processor chip, consisting of two dual-cores. Each dual-core has shared L2 and separate L1 caches (Intel “Harpertown”). There are two dual-core L2 groups.
**Figure 1.17:** Hexa-core processor chip with separate L1 caches, shared L2 caches for pairs of cores and a shared L3 cache for all cores (Intel “Dunnington”). L2 groups are dual-cores, and the L3 group is the whole chip.

**Figure 1.18:** Quad-core processor chip with separate L1 and L2 and a shared L3 cache (AMD “Shanghai” and Intel “Nehalem”). There are four single-core L2 groups, and the L3 group is the whole chip. A built-in memory interface allows to attach memory and other sockets directly without a chipset.
2 Basic optimization techniques for serial code

2.1 Scalar profiling

Function- and line-based runtime 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.
The most widely used profiling tool is *gprof* from the GNU binutils package. *gprof* collects 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:

<table>
<thead>
<tr>
<th></th>
<th>% cumulative</th>
<th>self</th>
<th>seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>70.45</td>
<td>5.14</td>
<td>5.14</td>
<td>26074562</td>
<td>0.00</td>
<td>0.00</td>
<td>intersect</td>
</tr>
<tr>
<td>2</td>
<td>26.01</td>
<td>7.03</td>
<td>1.90</td>
<td>4000000</td>
<td>0.00</td>
<td>0.00</td>
<td>shade</td>
</tr>
<tr>
<td>3</td>
<td>3.72</td>
<td>7.30</td>
<td>0.27</td>
<td>100</td>
<td>2.71</td>
<td>73.03</td>
<td>calc_tile</td>
</tr>
</tbody>
</table>
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).
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.

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.
Butterfly graph, or callgraph profile, reveals 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:

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called/100</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99.9</td>
<td>0.27</td>
<td>7.03</td>
<td>100</td>
<td>main [2]</td>
</tr>
<tr>
<td>2</td>
<td>0.27</td>
<td>7.03</td>
<td></td>
<td>100/100</td>
<td>calc_tile [1]</td>
</tr>
<tr>
<td>3</td>
<td>1.90</td>
<td>5.14</td>
<td>40000000/40000000</td>
<td>shade [3]</td>
<td></td>
</tr>
</tbody>
</table>

<spontaneous>

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called/100</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>99.9</td>
<td>0.00</td>
<td>7.30</td>
<td>100</td>
<td>main [2]</td>
</tr>
<tr>
<td>3</td>
<td>0.27</td>
<td>7.03</td>
<td></td>
<td>100/100</td>
<td>calc_tile [1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called/100</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>96.2</td>
<td>1.90</td>
<td>5.14</td>
<td>40000000/40000000</td>
<td>shade [3]</td>
</tr>
<tr>
<td>4</td>
<td>5.14</td>
<td>0.00</td>
<td>26074562/26074562</td>
<td>intersect [4]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5517592</td>
<td></td>
<td></td>
<td></td>
<td>shade [3]</td>
</tr>
<tr>
<td>6</td>
<td>1.90</td>
<td>5.14</td>
<td>40000000+5517592</td>
<td>shade [3]</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5.14</td>
<td>0.00</td>
<td>26074562/26074562</td>
<td>shade [3]</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5517592</td>
<td></td>
<td></td>
<td></td>
<td>shade [3]</td>
</tr>
<tr>
<td>9</td>
<td>70.2</td>
<td>5.14</td>
<td>0.00</td>
<td>26074562</td>
<td>intersect [4]</td>
</tr>
</tbody>
</table>
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 func-
tion’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.
gprof provides mostly a function-based runtime profiling. This kind of profiling becomes useless when the program to be analyzed contains large functions (in terms of code lines) that contribute significantly to overall runtime. Line-based profiling provide a way to identify hot spot within such functions.

2.1.1 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 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.

- **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.

- **Number of floating-point operations.** The importance of this very popular metric is often overrated. Data transfer is most usually the dominant performance-limiting factor in scientific code.
• **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.

• **Pipeline stalls.** Dependencies between operations running in different stages of the processor pipeline 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.

• **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.
2.2 Common sense optimizations

Very simple code changes can often lead to a significant performance boost. 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:

```fortran
1 logical :: FLAG
2 FLAG = .false.
3 do i=1,N
4    if(complex_func(A(i)) < THRESHOLD) then
5       FLAG = .true.
6    endif
7  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:

```plaintext
1  logical :: FLAG
2  FLAG = .false.
3  do i=1,N
4      if(complex_func(A(i)) < THRESHOLD) then
5         FLAG = .true.
6       exit
7     endif
8  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, 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.
This is an example from a simulation code for nonequilibrium spin systems:

```fortran
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 \texttt{tanh()} call completely by tabulating the function over the range of arguments required, assuming that \texttt{tt} does not change its value so that the table does only have to be set up once:

\begin{verbatim}
1 double precision, dimension(-6:6) :: tanh_table
2 integer :: iL,iR,iU,iO,iS,iN
3 double precision :: tt
4 ...
5 do i=-6,6 ! do this once
6    tanh_table(i) = 0.5d0*(1.d0+TANH(dble(i)/tt))
7 enddo
8 ...
9 BF = tanh_table(iL+iR+iU+iO+iS+iN) ! loop kernel
\end{verbatim}

The table look-up is performed at virtually no cost compared to the \texttt{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.

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 \texttt{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.
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.

In case of loops, this optimization is also called loop invariant code motion:

```plaintext
1 ! inefficient
2 do i=1,N
3   A(i)=A(i)+s+r*sin(x)
4 enddo

->

1 tmp=s+r*sin(x)
2 do i=1,N
3   A(i)=A(i)+tmp
4 enddo
```

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, loop unrolling, and other optimization techniques. If for some reason compiler optimization fails or is inefficient, performance will suffer. This can easily happen if the loop body contains conditional branches:

```plaintext
1  do  j=1,N
2    do  i=1,N
3       if(i.ge.j) then
4          sign=1.d0
5      else if(i.lt.j) then
6          sign=-1.d0
7       else
8          sign=0.d0
9      endif
10     C(j) = C(j) + sign * A(i,j) * B(i)
11   enddo
12 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 mis-predicted 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.
Fortunately, the loop nest can be transformed so that all if statements vanish:

```fortran
1     do  j=1,N
2       do  i=j+1,N
3           C(j) = C(j) + A(i,j) * B(i)
4       enddo
5     enddo
6     do  j=1,N
7       do  i=1,j-1
8           C(j) = C(j) - A(i,j) * B(i)
9       enddo
10    enddo
```
2.3.3 Using SIMD instruction sets

The use of SIMD in microprocessors is often termed “vectorization”.

Generally speaking, a “vectorizable” loop in this context will run faster if more operations can be performed with a single instruction,

```fortran
1 real, dimension(1:N) :: r, x, y
2 do i=1, N
3   r(i) = x(i) + y(i)
4 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.
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 this could look like the following:

```plaintext
! 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
endo
! remainder loop
do i=N-rest+1,N
   r(i) = x(i) + y(i)
endo
```

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.
2.4 The role of compilers

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 stan-
dardized 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 elimi-
nate “redundant” variables, rearrange arithmetic expressions, etc., so that any
debugger has a hard time giving the user a consistent view on code and data.
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).
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:

```c
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.
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 can overlap.

Lacking any further information, the compiler must generate machine instructions according to this scheme. Among other things, SIMD vectorization is ruled out.

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.
2.4.4 Computational accuracy

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.
2.4.5 Using compiler logs

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: 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).

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

We speak of parallel computing whenever a number of “compute elements” (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, which is published twice a year and ranks parallel computers based on their performance in the LINPACK benchmark.

LINPACK solves a dense system of linear equations of unspecified size. It is not generally accepted as a good metric because it covers only a single architectural aspect (peak performance).
3.1 Taxonomy of parallel computing paradigms

A widely used taxonomy for describing the amount of concurrent control and data streams present in a parallel architecture was proposed by Flynn.

The dominating concepts today are the SIMD and MIMD variants:

**SIMD** *Single Instruction, Multiple Data*. A single instruction stream, either on a single processor (core) or on multiple compute elements, provides parallelism by operating on multiple data streams concurrently.

Examples are vector processors, the SIMD capabilities of modern superscalar microprocessors, and Graphics Processing Units (GPUs).

**MIMD** *Multiple Instruction, Multiple Data*. Multiple instruction streams on multiple processors (cores) operate on different data items concurrently. The shared-memory and distributed-memory parallel computers are typical examples for the MIMD paradigm.
3.2 Shared-memory computers

A shared-memory parallel computer is a system in which a number of CPUs work on a common, shared physical address space.

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 in terms of main memory access:

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

  Today, single multicore processor chips are “UMA machines.”

- **On cache-coherent Nonuniform 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, 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).

### 3.2.1 Cache coherence

Cache coherence mechanisms are required in all cache-based multiprocessor systems, whether they are of the UMA or the ccNUMA kind. This is because copies of the same cache line could potentially reside in several CPU caches. Cache coherence protocols ensure a consistent view of memory under all circumstances.
3.2.2 UMA

The simplest implementation of a UMA system is a dual-core processor, in which two CPUs on one chip share a single path to memory.

**Figure 4.3:** A UMA system with two single-core CPUs that share a common frontside bus (FSB).

**Figure 4.4:** A UMA system in which the FSBs of two dual-core chips are connected separately to the chipset.
3.2.3 ccNUMA

In ccNUMA, a **locality domain** (LD) is a set of processor cores together with locally connected memory. This memory can be accessed in the most efficient way, i.e., without resorting to a network of any kind.

Multiple LDs are linked via a **coherent interconnect**, which allows transparent access from any processor to any other processor’s memory.

In this sense, a locality domain can be seen as a UMA “building block.” The whole system is still of the shared-memory kind, and runs a single OS instance.

![ccNUMA Diagram](image)

**Figure 4.5:** A ccNUMA system with two locality domains (one per socket) and eight cores.
3.3 Distributed-memory computers

**Figure 4.7:** Simplified programmer’s view, or “programming model,” of a distributed-memory parallel computer: Separate processes run on processors (P), communicating via interfaces (NI) over some network. No process can access another process’ memory (M) directly, although processors may reside in shared memory.
In a distributed-memory parallel computer, each processor \( P \) is connected to exclusive local memory, i.e., no other CPU has direct access to it.

Nowadays there are actually no distributed-memory systems any more that implement such a layout. In this respect, the sketch is to be seen as a programming model only.

For price/performance reasons all parallel machines today consist of a number of shared-memory “compute nodes” with two or more CPUs; the “distributed-memory programmer’s” view does not reflect that. It is even possible (and quite common) to use distributed-memory programming on pure shared-memory machines.
Each node comprises at least one network interface (NI) that mediates the connection to a communication network.

A serial process runs on each CPU that can communicate with other processes on other CPUs by means of the network.

**Figure 4.8:** Typical hybrid system with shared-memory nodes (ccNUMA type). Two-socket building blocks represent the price vs. performance “sweet spot” and are thus found in many commodity clusters.
As there is no remote memory access on distributed-memory machines, a computational problem has to be solved cooperatively by sending messages back and forth between processes.

4 Basics of parallelization

4.1 Why parallelize?

One can write parallel programs for two quite distinct reasons:

- A single core may be too slow to perform the required task(s) in a “tolerable” amount of time.
- The memory requirements cannot be met by the amount of main memory which is available on a single system, because larger problems (with higher resolution, more physics, more particles, etc.) need to be solved.
4.2 Parallelism

Writing a parallel program must always start by identifying the parallelism inherent in the algorithm at hand.

Different variants of parallelism induce different methods of parallelization.

4.2.1 Data parallelism

Many problems in scientific computing involve processing of large quantities of data stored on a computer.

If this manipulation can be performed in parallel, i.e., by multiple processors working on different parts of the data, we speak of data parallelism.

Examples:
- loop parallelism
- parallelism by domain decomposition
4.2.2 Functional parallelism

Sometimes the solution of a “big” numerical problem can be split into more or less disparate subtasks, which work together by data exchange and synchronization.

In this case, the subtasks execute completely different code on different data items.
4.3 Parallel scalability

How well a task can be parallelized is usually quantified by some scalability metric. Using such metrics, one can answer questions like:

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

We design and implement parallel programs in the hope that they will run faster than their sequential counterparts.

**Speedup** is the ratio between sequential execution time and parallel execution time:

\[
\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}
\]

The **efficiency** of a parallel program is a measure of processor utilization efficiency is defined as the speedup divided by the number of processors used:

\[
\text{Efficiency} = \frac{\text{Sequential execution time}}{\text{Processors used} \times \text{Parallel execution time}}
\]
4.3.2 Amdahl’s Law

Let $\psi(n, p)$ denote the speedup achieved by solving a problem of size $n$ with $p$ processors, $\sigma(n)$ denote the inherently sequential (serial) portion of the computation, $\varphi(n)$ denote the portion of the computation that can be executed in parallel, and $\kappa(n, p)$ denote the time required for parallel overhead.

The expression for the speedup is:

$$\psi(n, p) = \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)}$$

Since $\kappa(n, p) > 0$,

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

Let $f$ denote the sequential portion of the computation:

$$f = \frac{\sigma(n)}{\sigma(n) + \varphi(n)}$$
Then

\[ \psi(n, p) \leq \frac{1}{f + (1 - f)/p} \]

Amdahl’s Law: **The maximum speedup** \( \psi \) **achievable by a parallel computer** with \( p \) processors performing the computation **is** \( \psi \leq \frac{1}{f + (1 - f)/p} \)

### 4.3.3 Gustafson-Barsis’s Law

Amdahl’s law assumes that minimizing execution time is the focus of parallel computing.

It treats the problem size as a constant and demonstrates how increasing processors can reduce time.

What happens if we treat time as a constant and let the problem size increase with the number of processors?

Let \( s \) denote the fraction of time spent in the *parallel* computation performing inherently sequential operations.
The fraction of time spent in the parallel computation performing parallel operations is what remains, or \((1 - s)\). Mathematically,

\[
    s = \frac{\sigma(n)}{\sigma(n) + \phi(n)/p}
\]

\[
    (1 - s) = \frac{\phi(n)/p}{\sigma(n) + \phi(n)/p}
\]

Therefore

\[
    \psi(n, p) \leq s + (1 - s)p
\]

\[
    \psi(n, p) \leq p + (1 - p)s
\]

Gustafson-Barsis’s Law: Given a parallel program solving a problem of size \(n\) using \(p\) processors, let \(s\) denote the fraction of total execution time spent in serial code. The maximum speedup \(\psi\) achievable by this program is \(\psi \leq p + (1 - p)s\)
5 Shared-memory parallel programming with OpenMP

Shared memory opens the possibility to have immediate access to all data from all processors without explicit communication.

OpenMP is a set of compiler directives that a non-OpenMP-capable compiler would just regard as comments and ignore.

Hence, a well-written parallel OpenMP program is also a valid serial program. The central entity in an OpenMP program is not a process but a thread.

Threads are also called “lightweight processes” because several of them can share a common address space and mutually access data.

Spawning a thread is much less costly than forking a new process, because threads share everything but instruction pointer (the address of the next instruction to be executed), stack pointer and register state.

Each thread can, by means of its local stack pointer, also have “private” variables.
5.1 Parallel execution

In any OpenMP program, a single thread, the master thread, runs immediately after startup.

Truly parallel execution happens inside parallel regions, of which an arbitrary number can exist in a program.

Between two parallel regions, no thread except the master thread executes any code. This is also called the “fork-join model”.

Inside a parallel region, a team of threads executes instruction streams concurrently.

The number of threads in a team may vary among parallel regions.
Figure 6.1: Model for OpenMP thread operations: The master thread “forks” team of threads, which work on shared memory in a parallel region. After the parallel region, the threads are “joined,” i.e., terminated or put to sleep, until the next parallel region starts. The number of running threads may vary among parallel regions.
Parallel regions in Fortran are initiated by !$OMP PARALLEL and ended by !$OMP END PARALLEL directives, respectively.

The !$OMP string is a so-called sentinel, which starts an OpenMP directive (in C/C++, #pragma omp is used instead).

Inside a parallel region, each thread carries a unique identifier, its thread ID, which runs from zero to the number of threads minus one, and can be obtained by the omp_get_thread_num() API function.

There is an implicit barrier at the end of the parallel region. Only the master thread continues.
PROGRAM HELLO
INTEGER VAR1, VAR2, VAR3

Serial Code
.
.
.

Beginning of parallel section. Fork a team of threads.
Specify variable scoping

!$OMP PARALLEL PRIVATE(VAR1,VAR2) SHARED(VAR3)

Parallel section executed by all threads.
.
.
.
All threads join master thread and disband

!$OMP END PARALLEL

Resume Serial Code
.


23 .
24 .
25 END