An Introduction to Performance Programming

Romain Dolbeau
CAPS entreprise
Rennes, France
Email: romain@dolbeau.org

***Abstract*—The subject of performance programming is a complex one, about which many scholarly papers have been published. Deriving from classic introductions such as Chellappa et al. [1], this article aims at providing an experience-based approach to the newcomers. Its target audience includes all those interested in writing high-performing applications, whether their background includes computer science or not. On the basis of existing codes, we will detail a path leading to performance improvements with minimal effort and in a minimal amount of time. We'll also cover good practices to ensure the correctness and efficiency of the work done.**

The first version of this paper was published in two parts: An Introduction to Performance Programming (part I) and An Introduction to Performance Programming (part II), with a French translation previously published in two parts as well: Programmer pour la performance (1ère partie) and Programmer pour la performance (2ème partie).

The only update in this PDF version are some minor typographical corrections, and the use of my personal e-mail address since the company CAPS entreprise is now defunct.

Revision 1.73, 2016/08/19 14:20:43

***Index Terms*—Code tuning, Performance**

I. Introduction

Performance is a difficult thing to obtain in programming. Not just because writing fast code is complex, but also because understanding why the code isn't fast in the first place is often far from obvious. When tasked with improving the performance of a code, developers find themselves with a well-known multi-step agenda that has to be iterated upon:

1) Identify the sections of the code that take up most of the time;
2) Analyze why these sections are so costly;
3) Improve their performance by targeting bottlenecks.

But this conventional methodology, based on Amdalh's law [2], is only a very succinct explanation - or a high-level view - of the actual processes involved. It does not express the low-level concerns of the everyday programmer. For performance isn't just about the code; it is about the entire sequence of decisions and elements that leads from the code to the actual computations, i.e.

1) The chosen algorithms;
2) The specific implementation of these algorithms in the chosen programming language;
3) The organization of the data on which the algorithms will work;
4) The choice of an optimizing compiler to convert the code from source to binary;
5) The third-party libraries on which the code relies;
6) The actual hardware on which the code will run.

Each of these decisions and elements must be taken into account to achieve the goal of optimized performance; ignoring any one can lead to a significant waste of efficiency. And while the process of creating and running codes is sequential through or within each elementary step, it is the entire sequence that has to be targeted at once. Optimizing one step for a given set of states of the other steps does not mean it will remain optimal once these other steps have changed.

This paper suggests a very pragmatic approach built on years of experience in optimizing code - an approach that tries to maximize the efficiency of the programmer's time. Rather than rewriting the code straight away, a better idea is to start by leveraging all the tools available to maximize its efficiency, and then only to consider modifying it. In this respect, the important points this paper will try to make are the following:

Reproducibility
: we want to be able to reproduce both the results of the execution of the code (fast but wrong is not a good thing), and the measured performance. This is an issue for programs whose performance is data-sensitive, such as convergence algorithms, as several data sets will have to be validated.

Maintainability
: even if ultimate performance is the objective, the code will have to be maintained in one way or another. As obvious as it sounds, from two implementations with similar performance, the easier to explain and maintain is usually the better choice.

Prioritization
: time should be invested where the best results are likely to be obtained. Manually fine-tuning a kernel is fun but very often it is also an inefficient use of time. Once a code section is fast enough for its purpose, there is no point in making it faster.

Performance
: with everything else above in mind, performance becomes a much more manageable goal.

II. Evaluation and Validation

A. Test cases

The single most important thing when working on a code is to maintain the validity of the results. The second most important thing is to reliably quantify the gain (or loss) of performance of the modified, validated version.

Validating results is always a difficult task. While some codes will have the good property of producing output files whose content depends solely on the input parameters, it is not always the case. Any code involving IEEE754-2008 [3] floating-point operations will suffer from rounding approximations, unless compiled with extremely strict conformance options that prevent any performance gain. Any code involving random numbers (such as Monte Carlo methods) will produce unreproducible results. And not all algorithms are equal with regards to numerical accuracy, a subject covered in details by Higham [4].

Accordingly, the first step is to establish a test case (or a set of test cases) that will have several good properties for validating the results:

- An execution time short enough that it can be tested on a regular basis;
- An execution time long enough that small mistakes are likely to become noticeable in the output, rather than being masked by numerical approximations;
- A good coverage of the significant portions of the computations, both in terms of amount of code and type of input data (e.g. when propagating energy in a discretized space, the first few steps are usually full of zeroes, while boundary conditions are not reached and thus not tested until many iterations have been run);
- A clear validating procedure to ensure the output is correct.

These aspects should be discussed with the people interested in running the code for production. The second and third point, in particular, are not always easy to achieve. For codes involving random numbers, test cases should be de-randomized, for instance by fixing a seed or by precomputing one or more set of values, as it is usually the easiest way to ensure reproducibility.

The second step is to establish another test case (or another set of test cases), whose purpose will be to check performance. This test case will usually be larger (and longer running) than the validating case. It should also come with a clear validating procedure, in order to trap any residual mistake that might not be noticeable with the smaller test case. And it should be reasonably representative of typical production inputs. This can be very hard to achieve for extremely long and/or extremely big codes. Unfortunately, there is no shortcut here, as representativeness is highly dependent on the type of code.

Finally, never forget that it does not make much sense to try and improve a code with remaining issues. Verifying that the code doesn't misbehave is therefore an important preliminary step. Compilers can optionally do runtime bounds checking. Tools like Valgrind will check for accesses that haven't been initialized or that reach beyond properly allocated memory. Any such issue in the code must be dealt with prior to any kind of optimization as it is bound, sooner or later, to result in unreliable behaviors.

B. Measuring performance

While measuring performance seems at first glance quite simple, it is actually a fairly complex subject as well. What should you measure? And how should you measure it? When you've come up with reasonable answers to these two questions, there remains the difficulty of ensuring the consistency of measurement from one run to another.

What to measure is logically the first decision. Eventually, something akin to the **time** command is going to be used: when the code is run in production, it is the entire execution time from beginning to end that is going to matter to the consumers of the code. It's also going to matter to the programmer, as it is the time taken by any validation run or full-scale measurement. If this metric is obviously important, others are, too. After a profiling has been done (see section II-C below), the code will be seen as a sequence of stages: the prolog or initialization phase (usually an $O(problem\ size)$), one or more computational steps of varying complexity, and the epilog (also usually an $O(problem\ size)$). The costliest of these three steps will usually be the computational part, but it is not always the case. Prologs and epilogs generally consists of I/O and data movements, which are outside the scope of this article. If they dominate the execution time, either the code is not amenable to improvements, or the test case is too small for a significant measurement.

Gustafson's law [5] helps us here, as for most codes there should be a problem size big enough that the computation part dominates the execution time. Gustafson's law is an important consideration: even if the test case used during the optimization phase doesn't offer an overall improvement (because of significant prolog and epilog times combined with Amdhal's Law), larger production cases might since the fraction of time spent in the computational phase will be higher. It is therefore important to keep track not only of the overall execution time, but also of each main phases of the code. An $\times 2$ improvement in a computation phase representing 50% of the test case runtime is an overall $\times 1.33$ improvement of the whole code, but an $\times 2$ improvement in a computation phase representing 98% of the production case runtime is an overall $\times 1.96$ improvement.

How to measure it is the second step. The problem here is that time scales on computers can vary by orders of magnitudes. On a 2.5 GHz CPU, a single CPU cycle takes 0.4 ns, or 4×10^{-10}s.

Execution times that go beyond the hour take on the order of 10^{14} cycles, of which only the most significant digits make sense. Conversely, a small function of a few thousand cycles would take on the order of 10^{-7}s, a precision greater than most system calls would ensure in practice. It is a good idea to keep these orders of magnitudes in mind when choosing the proper timer. For most human-scale measurements (tenths of seconds or more), the **gettimeofday** function is perfectly adequate. For hundreds of microseconds or less (millions of cycles or less) measurements, use **_rdtsc** (in x86-64 compilers) to get the current CPU cycle count. For time scales in between, things

are less clear-cut: **gettimeofday** having a theoretical precision of a microsecond, it should be adequate. But since **_rdtsc** returns a 64 bits integer, measuring billions of cycles is also an option.

Consistency of measurement might be the most overlooked aspect of the problem. Codes seldom run on their own on a computer; they operate in an environment that includes the operating system, various housekeeping programs running permanently or regularly, and potentially other users. Whenever possible, reliable measurements should be made on an otherwise idle machine - one where very little is running on top of the operating system. Any other code will influence the results, by consuming memory bandwidth, inter-cpu bandwidth, cache space, I/0 bandwidth, or even by simply raising the power consumption of a CPU and affecting the thermal behavior of the others.

That is not all; the operating system behavior can drastically affect the performance of the CPU. Modern machines have multiple cores, each of them with its private L1 cache (and often private L2 caches). Whether or not the operating system let the code run on the same core for the entire run will greatly affect performance, as each move from one core to another will force the code to reload the local caches. Moving from one socket to another has an even greater effect: not only the shared intra-socket cache (L3 on the Sandy Bridge architecture) will be reloaded, but NUMA effects will happen as well. To keep that under control, the Linux operating system exposes commands to pin or lock a process on a subset of cores, to specify which NUMA memory domains to use, and so forth (see for instance **numactl**). Understanding the exact topology of the machine is also advisable, using tools from e.g. the **hwloc** library. When a code has already been parallelized, many tools have the ability to automatically enforce the pinning: the **KMP_AFFINITY** environment variable in the Intel OpenMP [6] implementation, the **GOMP_CPU_AFFINITY** environment variable in the GNU OpenMP implementation, the **-binding** option in the Intel MPI [7] library, and so on. The golden rule here is that ensuring a consistent placement of processes and threads in the machine ensures reproducible measurements.

C. Profiling

Profiling is the process of evaluating the relative cost of each part of the code. Without proper profiling, there is no way to tell which part of the code should be improved. No matter how convincing is the argument that function XYZ is the most important in the code, only an objective assessment should be trusted. Intuition, past experience, alternative implementations and test on completely different hardware platforms do not represent the current status of the code on the currently available hardware.

There are two main classes of profiling: instrumentation and sampling. Instrumentation adds instructions inside the binary to dynamically measure the relative use of each function. While comparatively reliable for creating call trees (how each function calls each other) and call counts, it has the bad property of adding overhead to the code and potentially altering its behavior. Sampling takes an unmodified code, and checks its running at regular intervals to evaluate the relative importance of each part. While much less intrusive, it can sometimes miss short but important parts of the code, or have weird side-effect due to the sampling interval. Neither is strictly better than the other, so both types should be used. Tools like **gprof**, **Intel VTune** [8] or **callgrind** [9] are all useful, to the extent that all available tools should be used, not just one of them. Each has its strengths and weaknesses, and it is the combination of their results that gives a clear picture of the code's dynamic behavior.

In an ideal world, profiling should be done on nearly production-ready code, i.e. a highly optimized binary. But more often than not, optimizations obfuscate the code to the point that profiling results are not very helpful. In difficult contexts, optimizations (and inlining in particular) should be "dialed down" step by step, and the consistency of profiling results checked accordingly until meaningful numbers are obtained. For instance, if the original results indicate that the do_the_computations function does 95% of the work, then subsequent results with less inlining should show the same order of magnitude for the sum of the time periods spent in do_the_computations and its call tree (all the functions it calls, and all the functions the latter call, and so on). In case of inconsistent results, you must absolutely understand why, and identify which optimization changed the code's behavior so drastically that the profiling results change. It might become important later to understand how to improve performance.

Profiling shouldn't be viewed as a one-shot operation. Once the costlier parts have been identified and improved, profile your code all over again to get a new, up-to-date picture. Improvements in one part will make the other parts comparatively more important, but side-effect such as new cache behavior may alter their performance, for better of for worse.

III. Using the compiler

Most developers see the compiler as a necessary evil, except for those who have forsaken it completely (or so they think) for so-called interpreted languages. Obviously, these languages cannot be run directly on the hardware, and therefore a translation layer is still present. While it could be a virtual machine, for performance reason it is often a just-in-time compiler. Which, as its name implies, is still a compiler, but one that tries to be fast rather than efficient and on which the developer has very little control. For the vast majority of codes requiring all-out performance, the compiler will be offline and highly tweakable. The examples in the paragraphs below will be related to Intel's C/C++ and Fortran compilers, as they are probably the most ubiquitous in high performance computing, but most comments are applicable to other languages and other compilers as well.

A. The hardware problem

To understand why compiling the programming language into machine code via an offline compiler is so important, we first have to mention a few things about hardware considerations (a good starting point on the subject is Hennessy-Patterson [10]). A computer CPU does not execute sophisticated, high-level code. It only executes very basic instructions, of which three classes are really important to us: memory operations, computations, and control flow operations. Any computational loop (or even non-loop kernels) will be essentially a set of control flow instructions surrounding a sequence of memory loads, operations, and memory stores. Anything else will be overheads that are useless to the higher-level algorithms but eat up CPU time. The list include in particular function calls, register moves, register spilling, virtual function handling, indirect references, etc.

The entire point of a good optimizing compiler is to both eliminate as much of these overheads as possible and to produce binary code that will be as efficient as possible on the target architecture for the computational parts of the code. Any spurious instructions in the code flow will degrade performance, and much more so when the code is highly efficient. Let's take a synthetic example with a fairly naive x86-64 version of the BLAS [11] **saxpy** function, first in a basic C implementation (listing 1) then in its assembly version (listing 2).

Listing 2. Trivial `saxpy`

```
1  ..___tag_value_saxpy.1:
2          xorl      %eax, %eax
3          movslq    %edi, %rdi
4          testq     %rdi, %rdi
5          jle       ..B1.5
6  ..B1.3:
7          movsd     (%rsi,%rax,8), %xmm1
8          mulsd     %xmm0, %xmm1
9          addsd     (%rdx,%rax,8), %xmm1
10         movsd     %xmm1, (%rdx,%rax,8)
11         incq      %rax
12         cmpq      %rdi, %rax
13         jl        ..B1.3
14 ..B1.5:
15         ret
```

One might consider that the two lines with **incq** (line 11) and **cmpq** (line 12), dealing with control flow, are overheads that should be dealt with. Actually, they're inexpensive: in the context of an otherwise idle integer unit, any modern out-of-order CPU core will execute them at the same time as the useful floating-point instructions. The real killer here is the sequence starting with the load instructions (line 7) as they will take hundreds of CPU cycles if they miss the various caches, followed by the producer-consumer dependency between the multiplication (line 8) and the addition (line 9), and between the addition and the store (line 10). Obviously, the literature is full of solutions to these problems. Loop unrolling will mask remaining overheads from the control flow instructions. It will also, along with software pipelining, try to mask the producers-consumers dependencies. These are textbook considerations, and most if not all of them can be taken care of by the compiler (or indeed the BLAS library itself).

The real point of this example is to illustrate what happens when the compiler does its job properly. While the vectorizer and the aforementioned optimizations will improve the performance of this code, any additional overhead will degrade it. The worst-case scenario is the introduction of additional, dependent load operations in the loop. Any of those (such as accessing an object before accessing a field in it, or an indirect access through a pointer to pointer) will add extra, potentially large latency to the dependency chain. On Sandy Bridge, a **mulsd** operation has a latency of 5 cycles. A **movsd** from memory can consume from 3 cycles (best case scenario, hitting the L1 cache) to 230-250 cycles (hitting the main memory of the same NUMA node on a dual-socket Sandy Bridge) to 350-370 cycles (hitting the main memory of the other NUMA node) to over 600 cycles (hitting the most remote NUMA node in a quad-socket Sandy Bridge).

That is why analyzing the assembly code is so important, for it is the binary code that will eventually be executed. One cannot reasonably hope to optimize the performance of a program on a processor without understanding how this processor works.

B. General optimization

The basic rule in exploiting a compiler is to start by letting the people who created it do the job. Compilers include a lot of optimizations. Many of them are more or less mandatory because they are cheap (in terms of compilation time) and offers large gains. Some of them are additional extras that are very often used, as they are usually worth the extra compilation time. Finally, some can be very costly, and can have wildly different effects on performance depending on the specific code involved.

The first job is therefore to choose a compiler. The easiest choice is the machine vendor's, as it is likely to be the most suited to the target hardware architecture. But third-party compilers are also of interest, precisely because of their strengths and weaknesses. If multiple compilers are available, try them all. It will help you identify bugs and approximations in the source code and, eventually, lead to the selection of one compiler over the others. This, however, will not be a definitive choice: changing parts or all of the code to improve it might help one compiler more than another, to the extent that the former second choice might become the new first choice. Also, be sure to use the latest available version, barring some fatal bug in it.

The optimization options to try first are those prefixed with **-O** and suffixed with a number (the larger the number, the larger the set of optimizations applied). All codes should work at all optimization levels. If it is not the case, either there is a bug in the compiler or there is a problem with the code.

Listing 1. C `saxpy`

```
void saxpy(int n, double alpha, double *x, double *y) {
  int i;
  for (i = 0 ; i < n ; i++) {
    y[i] = alpha * x[i] + y[i];
  }
}
```

While it is common to blame compilers for failure at high optimization levels, the culprit is generally the code itself. Quite often, the code exercises corner cases or unspecified behaviors of the language, leading to an apparent unreliable behavior on the part of the compiler. In this case, the first step is to ensure reproducibility at all general optimization levels, either by fixing the code or by proving there really is a bug in the compiler (and having the compiler's vendor fix it, of course). Each and every time a new set of options is used, the results of the code should be checked for conformance.

Then, compilers such as Intel's have inter-procedural optimizations that try to optimize the code across function boundaries rather than within each function independently. In recent versions, these optimizations are active by default inside each compilation units (aka files). The **-ipo** option enables these optimizations across compilation units. Instead of compiling each file independently (creating ".o" files), the compiler simply creates stubs and then, at link-time, it compiles everything in one go. The downside is that the entire code has to be recompiled each time anything changes. The upside is that all the information is available to the compiler, including hard-coded values (array bounds, constants...) and call trees. This allows the compiler to do a much better job: for instance, if it is aware of the number of iterations in a loop, it can unroll more efficiently. If a scalar parameter to a function is known to be a constant, the compiler can use that knowledge. For example, nice constants such as 0 and 1 are identity elements for the addition and multiplication respectively; using them as such prevents redundant computations.

Another must-try with recent Intel compilers is the set of inlining options. Normally, a compiler will try to inline a certain number of functions but will use heuristics to prevent an excessively large code size and/or compilation time. However, this often limits its ability to merge loops and/or functions, and to eliminate redundant computations and/or memory accesses. A particularly aggressive set of options is `-inline-forceinline -no-inline-factor -no-inline-max-per-compile -no-inline-max-per-routine -no-inline-max-total-size -no-inline-max-size -no-inline-min-size` which, combined with **-ipo**, tells the compiler to merge as much of the code as it can into computational functions, regardless of compilation time, its own memory usage and the size of the resulting binary. Although these options combined can be really aggressive in some contexts, many codes will actually benefit a lot from them. For instance, when applied to the Qiral QCD code [12], the improvement is superior than $\times 2$ with version 13.1.1.163 of the Intel compiler.

Those are general guidelines for the most common options; each compiler has its own set of advanced settings, some of which can be useful on some codes. As long as the results of the code remain valid, any set of options can be used. Again, if "the compiler broke the code", then there is a problem somewhere that must be tracked down. Otherwise, the problem may reappear with a different compiler, a different machine or, worse, a different data set.

C. Vectorization

The next step is to understand whether the compiler properly vectorizes the code. The word "vector" shouldn't be taken too literally here, as it usually calls back to vector systems (from the Cray 1 [13] onward). In the current era, "vector" refers to the short, fixed-length registers in the CPU and their associated operations (like the SSE and AVX instructions sets in the x86-64 world or the QPX on the IBM BlueGene/Q [14]). There have already been detailed works on the subject of vectorizers, such as Intel's own [15]. These are of course interesting for advanced users, but might be too detailed for the newcomer to vectorization.

The principle of vector instruction sets is to allow a single instruction to perform multiple operations at once, thus adding potential performance at a low cost. Analyzing why this is a good trade-off would require a long explanation; suffice it to say that nearly all contemporary CPU use such instruction sets. We will detail some interesting differences between them in section III-D below.

A good starting point to understand vectorization better is SSE2's **mulsd** mentioned in section III-A. In the Intel documentation [16], this instruction's full name is "Multiply Scalar Double-Precision Floating-Point Values". As this description implies, **mulsd** performs a multiplication on double-precision floating-point values. However, it has a counterpart called **mulpd**, or "Multiply Packed Double-Precision Floating-Point Values". Instead of a single multiplication, **mulpd** executes one per element in the so-called vector. The precise size of the vector depends on the instruction set (SSE or AVX). If we take the example of SSE where vector registers are 128 bits, then we have two double-precision value (64 bits each, as per IEEE754-2008 [3]) in each register. In other words, each instruction will compute 2 flops instead of one. If we go back to the saxpy function from listing 1, we can update the

assembly version to exploit this new instruction, as shown in listing 3.

Listing 3. Trivial `saxpy`, with SSE

```
..___tag_value_saxpy.1:
        xorl      %eax, %eax
        movslq    %edi, %rdi
        testq     %rdi, %rdi
        jle       ..B1.5
..B1.3:
        movupd    (%rsi,%rax,8), %xmm1
        mulpd     %xmm0, %xmm1
        addpd     (%rdx,%rax,8), %xmm1
        movupd    %xmm1, (%rdx,%rax,8)
        incq      %rax
        incq      %rax
        cmpq      %rdi, %rax
        jl        ..B1.3
..B1.5:
        ret
```

Obviously, this code will not work if the number of elements in the vector is not a multiple of two, as we compute two values for each iteration (note that it was hand-modified from the automatically generated code in listing 2 as the compiler vectorized code, including all the necessary checks, is not suitable for an introductory example). But it does illustrate the point that with the same number of instructions, we perform twice as much work. On such a simple loop the compiler has no issue with vectorization. It will even mention it in its output when using the **-vec-report3** option, as can be seen in listing 4.

However, most loops are not that simple, and this is where understanding the compiler is important. A modern compiler will inform you about its successes and failures, but what these actually mean is not always obvious. Let's take a very concrete example with the Hydro [17] code. Hydro is a mini-application built from RAMSES [18][19][20]. It is used to study large-scale structures and galaxy formation. This code includes several variants, each of them designed to take advantage of a certain kind of hardware: a standard C version using OpenMP [6] directives and MPI [7] calls; a version using OpenACC directives to exploit accelerators; an OpenCL for the same purpose, etc. What we are going to look at is the **riemann** function, which exists in two radically different versions: the original in the OpenACC version of the code and an updated and vectorization-friendly variant in the C version. Compiling the original, untweaked version results in what is shown in listing 5. The loop nest of interest is the one referenced at line 102, 107 and 138 (see its structure in listing 6). The compiler output shows that it failed to vectorize the innermost loop at line 138, because it couldn't be certain that all iterations were data-independent from each other. The compiler is absolutely right: this innermost loop is a convergence loop, and each iteration is predicated on the results of all the preceding iterations. Therefore, it cannot be easily vectorized. As for the other two loops, the compiler doesn't even try to vectorize them since it could not process the innermost loop, a requirement for the current version of the vectorizer.

There is an easy way to force the compiler to vectorize the code as it is: if we replace the upper bound of the innermost loop (**Hniter_riemann**) by a hardwired value like 10, then the compiler can fully unroll the innermost loop, making the **narray** loop the new innermost loop. In practice, the compiler still fails as it still can't figure out that the two external loop are fully parallel, but a directive exists to help it out: **#pragma ivdep**. Using it on both these loops allows vectorization of the code (with slightly modified line numbers), as show in listing 7. However, this does not really help: the code is now wrong, not to mention that hardwiring a value is not an acceptable practice. To be able to properly exploit the hardware in this critical function, the code needs to be modified so that the compiler can vectorize it with the original semantic. For this particular code, the author decided to switch the convergence loop (the problematic one at line 138) and the **narray** loop at line 107. Of course, this implies a complete reorganization of the loops (and the use of additional temporary arrays). The new structure of the code is shown in listing 8, while the result of compiling it is given in listing 9. Note that the presence of an extra "SIMD" in the message is an artefact of the directives required to tell the compiler that the loop iterations are data-independent.

There is a lot of reasons for the vectorizer to fail, and this paper is not the place to explain them all. If some compiler output messages are self-explanatory, unfortunately some aren't. That being said, a few basic rules can help with identifying and fixing issues.

- The Intel compiler only tries to vectorize the innermost loop. Small innermost loops can and should be unrolled (automatically, by a pragma or manually) if the number of iterations is known. If it is not known, the situation is more difficult. It might be possible to invert the loop with a larger, parallel outer loop and the addition of temporary arrays.
- Except for some types of reduction (arithmetic accumulation in a scalar variable), the Intel compiler vectorizes parallel loops. If the compiler complains that there is a vector dependence, check that there really isn't, and use the appropriate directives to help with the compilation.
- Conditionals in the loop nest might prevent vectorization, and make it less efficient anyway. It's often a good idea to move iteration-independent conditions out of the loop nest.

While the compiler will tell which loops have been vectorized, it is not the only condition for successfully using vector instructions sets. Compilers will often generate multiple variants of a loop depending on various parameters (number of iterations, alignment of data arrays...) and there is no guarantee the vector version will always be picked up at

Listing 4. Compiling saxpy

```
$ icc -O3 -vec-report3 -S trivial_saxpy.c
trivial_saxpy.c(4): (col. 3) remark: LOOP WAS VECTORIZED.
```

Listing 5. Compiling the OpenACC riemann

```
HydroC/oaccHydroC_2DMPI/Src$ icc -std=c99 -O3 -vec-report2 -Wno-unknown-pragmas -S riemann.c
riemann.c(51): (col. 3) remark: LOOP WAS VECTORIZED.
riemann.c(138): (col. 14) remark: loop was not vectorized: existence of vector dependence.
riemann.c(107): (col. 11) remark: loop was not vectorized: not inner loop.
riemann.c(102): (col. 7) remark: loop was not vectorized: not inner loop.
riemann.c(298): (col. 11) remark: loop was not vectorized: existence of vector dependence.
riemann.c(296): (col. 9) remark: loop was not vectorized: not inner loop.
riemann.c(291): (col. 7) remark: loop was not vectorized: not inner loop.
```

Listing 6. Structure of the OpenACC riemann

```
for (int s = 0; s < slices; s++) { // line 102
  for (int i = 0; i < narray; i++) { // line 107
    // some code here
    for (iter = 0; iter < Hniter_riemann; iter++) { // line 138
      // some code here
    }
    // some code here
  }
}
```

Listing 7. Vectorizing the OpenACC riemann

```
HydroC/oaccHydroC_2DMPI/Src$ icc -std=c99 -O3 -vec-report2 -Wno-unknown-pragmas -S riemann.c
riemann.c(51): (col. 3) remark: LOOP WAS VECTORIZED.
riemann.c(109): (col. 11) remark: LOOP WAS VECTORIZED.
riemann.c(103): (col. 7) remark: loop was not vectorized: not inner loop.
riemann.c(300): (col. 11) remark: loop was not vectorized: existence of vector dependence.
riemann.c(298): (col. 9) remark: loop was not vectorized: not inner loop.
riemann.c(293): (col. 7) remark: loop was not vectorized: not inner loop.
```

Listing 8. Structure of the C riemann

```
for (int s = 0; s < slices; s++) { // line 123
  for (int i = 0; i < narray; i++) { // line 148
    // some code here
  }
  for (iter = 0; iter < Hniter_riemann; iter++) { // line 171
    for (int i = 0; i < narray; i++) { // line 179
      // some code here
    }
  }
  for (int i = 0; i < narray; i++) { // line 206
    // some code here
  }
}
```

Listing 9. Vectorizing the C riemann

```
HydroC/HydroC99_2DMpi/Src$ icc -std=c99 -O3 -vec-report2 -Wno-unknown-pragmas -S riemann.c
riemann.c(65): (col. 3) remark: LOOP WAS VECTORIZED.
riemann.c(148): (col. 5) remark: SIMD LOOP WAS VECTORIZED.
riemann.c(179): (col. 7) remark: SIMD LOOP WAS VECTORIZED.
riemann.c(171): (col. 5) remark: loop was not vectorized: not inner loop.
riemann.c(206): (col. 5) remark: SIMD LOOP WAS VECTORIZED.
riemann.c(123): (col. 3) remark: loop was not vectorized: not inner loop.
riemann.c(300): (col. 2) remark: SIMD LOOP WAS VECTORIZED.
riemann.c(296): (col. 7) remark: loop was not vectorized: not inner loop.
riemann.c(295): (col. 5) remark: loop was not vectorized: not inner loop.
```

runtime. Hardware counters inside the CPU can be used to validate the results during execution. Tools such as VTune [8] or Likwid [21] can be very useful to validate that the vectorizer has been successfully exploited.

D. Target machine

We mentioned it in previous sections, there are two main vector instruction sets for the x86-64 platform: SSE and AVX. SSE was originally meant for single-precision floating-point and integer data. Double-precision floating-point was added in SSE2, and subsequent extensions added new instructions for various purposes. They all have in common a 128-bit register size, which is enough to hold four single-precision values, or two double-precision values, in the **XMM** registers.

The AVX instructions set was introduced using a completely new instructions encoding scheme (called VEX). While this does not really concern most programmers, AVX's main selling point was the doubling of the register width to 256 bits, for eight single-precision or four double-precision values. In the original AVX, only floating-point values can be worked on in the 256 bits **YMM** registers. The lower half of these **YMM** registers is aliased with the **XMM** registers. As you can guess, going from SSE to AVX instructions in the same code causes a small performance penalty.

One of the most overlooked feature of the new AVX instruction set is the VEX.128 subset of instructions. These instructions only work on the lower 128-bit part of the **YMM** registers - the part aliased with **XMM** registers. While they may seem quite redundant with the SSE instructions, they actually offer a very important upgrade: whereas SSE instructions have mostly two operands, AVX instructions have three. listing 10 is an excerpt from the Intel documentation [16] in which the SSE version stores its result in source operand **xmm1**, while the VEX.128 version destroys neither of its source operands. Whenever both sources are reused, this saves a register-register **movapd** instruction. This may not look like a lot, in particular in numerical codes where one of the operands will be one-use only most of the time. But the VEX.128 instructions are not limited to the floating-point instructions; many integer instructions have been re-implemented as well. They cannot access the upper 128 bits of the 256 bits **YMM** registers, but they do have access to the new three-operands format.

A code that benefits greatly from this is the Keccak algorithm by Bertoni et al. [22], the winner of the NIST competition to create SHA-3. The "SIMD instructions and tree hashing" implementation [23] uses integer SSE instructions to implement two 64-bit instances of the algorithm simultaneously. The implementation is done using C intrinsics rather than assembly. The instructions in use are mostly logical xor's, logical or's and shifts (used to implement the rotation of a 64 bit value). A brief extract implementing a rotation is shown in listing 11 for SSE and listing 12 for AVX. Extra copies to preserve the input values are required by the algorithm, which reuses the input data several time. Statistics on the partially unrolled loop show that the number of data movement instructions has gone from 396 (35% of the 1132 instructions) to 111 (13% of the 837 instructions). All the remaining moves are memory accesses, none of them being register-register. Speed measurement shows an improvement of more than 20% for short hashes. As the only cost for high-level language or intrinsics-based code is to use the **-mavx** options (or similar), switching from SSE to AVX can offer a nice improvement for a very small cost.

Listing 11. Keccak rotation in SSE

```
movdqa    %xmm6, %xmm8
movdqa    %xmm6, %xmm14
psllq     $1, %xmm8
psrlq     $63, %xmm14
por       %xmm14, %xmm8
```

Listing 12. Keccak rotation in AVX

```
vpsllq    $1, %xmm0, %xmm14
vpsrlq    $63, %xmm0, %xmm15
vpor      %xmm15, %xmm14, %xmm14
```

IV. Data layouts

As previously mentioned, a minimal understanding of the hardware is required to exploit it. Efficiently utilizing the

Listing 10. `mulpd` from Intel documentation

```
66 0F 59 /r                    RM  V/V SSE2  Multiply packed double-precision floating-point
MULPD xmm1, xmm2/m128                        values in xmm2/m128 by xmm1.
VEX.NDS.128.66.0F.WIG 59 /r RVM V/V AVX   Multiply packed double-precision floating-point
VMULPD xmm1,xmm2, xmm3/m128                  values from xmm3/mem to xmm2 and stores
                                             result in xmm1.

(...)
MULPD (128-bit Legacy SSE version)
DEST[63:0] = DEST[63:0] * SRC[63:0]
DEST[127:64] = DEST[127:64] * SRC[127:64]
DEST[VLMAX-1:128] (Unmodified)
VMULPD (VEX.128 encoded version)
DEST[63:0] = SRC1[63:0] * SRC2[63:0]
DEST[127:64] = SRC1[127:64] * SRC2[127:64]
DEST[VLMAX-1:128]  0
```

computational resources of the CPU is important, but since the term "memory wall" was coined by Wulf and McKee [24], it has been known that memory accesses were the limiting factor for many kinds of codes. The expression "bandwidth-bound algorithm" is dreaded by many of those tasked with improving performance, as it means that the available bandwidth from the main memory to the CPU cores is the limiting factor. But even without changing the algorithm, it is sometimes possible to drastically improve performance.

One of the main issues with memory bandwidth is the amount wasted by poor data layouts. This issue is not new, and was already targeted by papers such as the work of Truong et al. [25]. But it still seems to be largely unknown to most developers. We first have to go back to computer architecture (and mention again Hennessy & Patterson [10]). The main technique for a CPU to avoid the hundreds of cycles required for data to arrive from main memory are caches, tiny but fast memories close to the execution cores. While their existence is known, it seems that most programmers are unaware of their behavior and write code that greatly impedes their efficiency.

A. Arrays of Structures vs. Structures of Arrays

Caches do not retain data at single element granularity, e.g. a single double-precision value. While this would help with temporal locality (the fact that a recently used element might soon be reused), it would do nothing for spatial locality (the fact that it's likely the next useful element will be a memory neighbor to a recently used element). To gain such spatial locality, caches have a granularity of a cache line, a contiguous amount of memory. Common sizes are 32 or 64 bytes of well-aligned memory (i.e. the address of the first byte is an integer multiple of the cache line size). Whenever a code requires an element from memory, the entire cache line is loaded from the main memory and retained in the cache. If a neighboring element from the same cache line is subsequently needed, the access will be a cache hit, and therefore very fast. This is a well known mechanism, taught in most computer science classes.

But the implications are not always well understood. If a cache line is 64 bytes, then every memory transaction will involve the whole 64 bytes, no matter how few bytes are actually needed by the code. If only a single double-precision element (8 bytes) is needed from each cache line, 87.5% of the memory bandwidth is wasted on unused bytes. Therefore, it is very important to ensure that as few elements as possible in each loaded cache line are not used. Unfortunately, some extremely common programming techniques goes against that principle. The most obvious offenders are structures (and of course objects, which are generally structures with associated functions).

While there is a lot to be said in favor of structures, that is not our subject; therefore we'll look at the downside, the way they can sometimes waste memory bandwidth. The content of a well-defined structure is a lot of information related to an abstract concept in the code - it could be a particle used in fluid simulation, the current state of a point in a discretized space, or even an entire car. All the occurrences of that concept will be allocated in what is described as an Array of Structures. Subsequently, functions in the code will go through all occurrences in a loop to process the data, such as this:

```
foreach particle in charged_particles
  update_velocity(particle, electric_field)
```

Presumably, the function **update_velocity** will change the speed of the charged particle in the electric field. But while the speed might be defined with only a few values (e.g. the current velocity in **X**, **Y** and **Z**), the structure itself is likely to contain much more information that's irrelevant to that particular step of computations. But as structures are contiguous in memory, much of that information will be loaded as they belong to the same cache line. Let's assume each particle in our example is defined by 16 double-precision values, i.e. 128 bytes. The structure would occupy 2 full cache lines all by itself. Let's also assume all 3 velocities are in the same half of the structure, i.e. in the same cache line. Each time we need to

update a velocity, the CPU will:

1) Load the cache line (64 bytes) containing the three velocities (24 bytes);
2) Perform any required computations, and update the value in the cache;
3) Eventually, when space in the cache is needed, the whole cache line (64 bytes) will be flushed to memory.

62.5% of the memory bandwidth required to load and store the particle is wasted by unnecessary data. If the three velocities were spanning both halves of the structure, then the bandwidth requirement would double for the same amount of useful data - wasting 81.25%.

One solution for this issue is called structures of arrays. As the name implies, the idea is to inverse the relationship by first allocating all atoms of data in arrays, and then grouping them into a structure. Instead of having an array of particles, each with its velocities, the code would use three arrays of velocities. Each array would be contained as a single velocity in **X**, **Y** or **Z** for all particles. The result for the function **update_velocity** above is that we would need three cache lines instead of one: one for each of the **X**, **Y** and **Z** velocities. But that is not a bad thing. The first particle would require 192 bytes of bandwidth, loading 8 elements of each array. However, the next 7 particles would require no bandwidth at all - their data had been prefetched by the first particle, thanks to spatial locality. The average bandwidth per particle is therefore an optimal 24 bytes per particle, with a greatly reduced latency because of locality. If that function was purely bandwidth-bound, we would have gained a factor of $\times 2.66$ by reorganizing data in a cache-friendly manner.

B. Multi-dimensional array vs. array of pointers

This is an issue that doesn't exist in the Fortran language, where multi-dimensional arrays are the norm. But in the C language family the issue is pervasive. A lot of code utilizes not multi-dimensional arrays but array of pointers, adding a useless intermediate load. Listing 13 illustrates this form of inefficient code. The data is stored behind a pointer to a pointer (hence the two stars). The body of the code looks good to the untrained eye: it has a nice pair of square brackets to access the element of the two dimensional data, as is done in Fortran. But the performance will be suboptimal. The real meaning of the code includes not one, but two chained loads to access the data. The machine must first evaluate **A[i]**, itself a pointer to double. Then this pointer is used to retrieve **A[i][j]**, the data itself. This is effectively an indirect access.

An efficient way can be exactly the same in the body, by utilizing a properly typed pointer to the data. This is illustrated in listingg 14. By specifying the dimensions of the array in the parameter list, it becomes possible to use the clean multi- dimensional notation of the C language - which unfortunately looks exactly the same as a pointer-to-pointer double dereference. This new version will simply compute the linear access **i * n + j**, and do a single lookup in memory to access the data. Of course, the data allocation is also different: instead of first allocating the array of the pointer, followed by a loop allocating all the pointers to double, a single allocation of the entire data set is used. The exact same allocation that would be used for the ugly, explicitly linearized version illustrated in listing 15 that most programmers justifiably try to avoid.

Listing 13. Example of array of pointers

```
void matrix_sum_aop(int n, double** A,
                                double** B) {
  for (int i = 0 ; i < n ; i++) {
    for (int j = 0 ; j < n ; j++) {
      A[i][j] += B[i][j];
    }
  }
}
```

Listing 14. Example of multi-dimensional arrays

```
void matrix_sum_mda(int n, double A[n][n],
                           double B[n][n]) {
  for (int i = 0 ; i < n ; i++) {
    for (int j = 0 ; j < n ; j++) {
      A[i][j] += B[i][j];
    }
  }
}
```

Listing 15. Example of explicitly linearized arrays

```
void matrix_sum_linear(int n, double* A,
                              double* B) {
  for (int i = 0 ; i < n ; i++) {
    for (int j = 0 ; j < n ; j++) {
      A[i * n + j] += B[i * n + j];
    }
  }
}
```

V. Algorithms

The word algorithm can cover different types of problems depending upon the audience. Computer scientists will think in terms of basic programming techniques as explained by Aho et al. [26]. Other scientists in need of high performance computing will likely think more in terms of numerical analysis as introduced by Atkinson [27]. In either cases, the idea is to achieve the desired results with minimal complexity, that is, with a minimal increase in work when augmenting the problem size. Complexity is usually taught in most computer science courses, and numerous books have been written on the subject such as Arora et al [28].

It is usually quite difficult for a computer scientist to replace a numerical algorithm chosen to solve a particular problem by a "better" one. This requires understanding what the code is

trying to achieve in computer terms but also in physical (or chemical, astronomical...) terms as well. It also entails understanding the numerical stability, approximation, the boundary conditions, and whatever other requirements might exist for solving the underlying problem. This is something that must be done with involvement someone from the scientific field. More often than not, the constraints of numerical algorithms are such that they unfortunately cannot be changed.

What remains to be studied is the implementation of such algorithms, which quite often are built on top of other algorithms - those of the computer science variety. Matrix inversion, matrix multiplication, finding a maximum value, fast Fourier transforms and so on are the building blocks of many numerical algorithms. They are quite amenable to improvements, substitution and well-studied optimizations.

One of the primary tasks of the optimizer in a numerical code will be to identify these types of conventional algorithms, and make sure the implementation used is the best one for the code. This is usually not a lot of work, as vendor-supplied libraries will include most of them. For instance, the Intel MKL [29] library (accessible in the recent Intel compiler by simply calling the **-mkl** option to the compiler and the linker) includes full implementations of BLAS [11] or LAPACK [30], fast Fourier transforms including a FFTW3 [31] compatible interface, and so on. One important aspect when using such libraries are their domain of efficiency: extremely small problem sizes are not well suited to the overhead of a specialized library. For instance, large matrix multiplications should make use of an optimized function such as **dgemm**. Small constant size matrix multiplications (such as 3×3 matrices used in Euclidean space rotation) should be kept as small hand-written functions, potentially amenable to aggressive inlining. In a similar way, whereas large FFT can take advantage of FFTW3 itself or its MKL counterpart, small FFT of a given size can sometimes be readily implemented with the FFTW codelet generator described in [32]. A more work-intensive optimization is the specialization of algorithms for constant input data. For instance, if the small 3×3 matrix mentioned earlier is a rotation around the main axis of a 3D space, the presence of zeroes and one in the matrix can be hard-coded in the function, removing multiple useless operations.

VI. Parallel codes

Parallelization is one of the richest and most complex subjects of computer science, and is out of the scope of this paper. Many introductory books have been written on the subject among which for instance [33][34][35][36]. This section deals with running parallel code on contemporary machines without falling into common pitfalls.

A. NUMA

All recent multi-socket x86-64 systems are NUMA (Non-Uniform Memory Access), i.e. the latency of a load instruction depends on the address accessed by the load. The common implementation in all AMD Opterons and all Intel Xeons since the 55xx (i.e. Nehalem-EP family) is to have one or more memory controller(s) in each socket, and to connect each socket via a cache coherent dedicated network. In AMD systems the links are HyperTransport, while for Xeons they are QPI (QuickPath Interconnect). Whenever a CPU must access a memory address located in a memory chip connected to another socket, the request and result have to go through the network, adding additional latencies. Such access is therefore slower than accessing a memory address located in a locally connected memory chip. This effect is illustrated in figure 1, which plots the average latency of memory access when stepping through an array of varying size on a dual socket Xeon X5680 Westmere-EP. The first three horizontal plateaus are measurements where the array fits in one of the three levels of caches, while the much higher levels at over 12 MiB measure the time to access the main memory. The four lines represent the four possible relations between the computation and the memory: 0/0 and 1/1 represent access to locally attached memory, while 0/1 and 1/0 represent access to the other socket's memory. Going from about 65 ns to about 106 ns represents an approximately 63% increase in latency when accessing remote memory.

B. NUMA on Linux

To avoid these kinds of performance issues, it is important to understand how memory is allocated (virtually and physically) by the operating system, to ensure memory pages (the concept of paging is among those covered by Tanenbaum [37]) are physically located close to the CPU that will use them. The Linux page size will be 4 KiB or 2 MiB; the last one is more efficient for TLB [10] but is not yet perfectly supported. This page size will be the granularity at which the memory can be placed in the available physical memories.

In Linux, this placement is made by specifying in which "NUMA node" the physical page should reside. The default behavior is only partially satisfactory. At the time the code requires allocation of memory, virtual space is reserved but no physical page is allocated. The first time an address inside the page is written, the page will be allocated on the same NUMA node that generated the write (if that memory is full, the system will fall back to allocating on other nodes). This is a very important aspect. If for one reason or another the memory is "touched" (written to) by only one thread, then all physical pages will live in the NUMA node where the thread was executing at the time. Of course, if the advice from section II-B was followed, then the thread has been properly "pinned" on the CPU and will not move. This means than any subsequent access by this same thread will benefit from the minimal latency. That's the good aspect of the behavior.

One of the consequences is that in a two-socket system (two NUMA nodes), only half the memory will be used by the single-thread process (unless consuming a large amount of memory). It also means that only half the memory bandwidth from the entire system can be exploited, leaving the second memory controller in the second socket unused. Another consequence is that if the code subsequently becomes multi-threaded (for instance through the use of OpenMP directives),

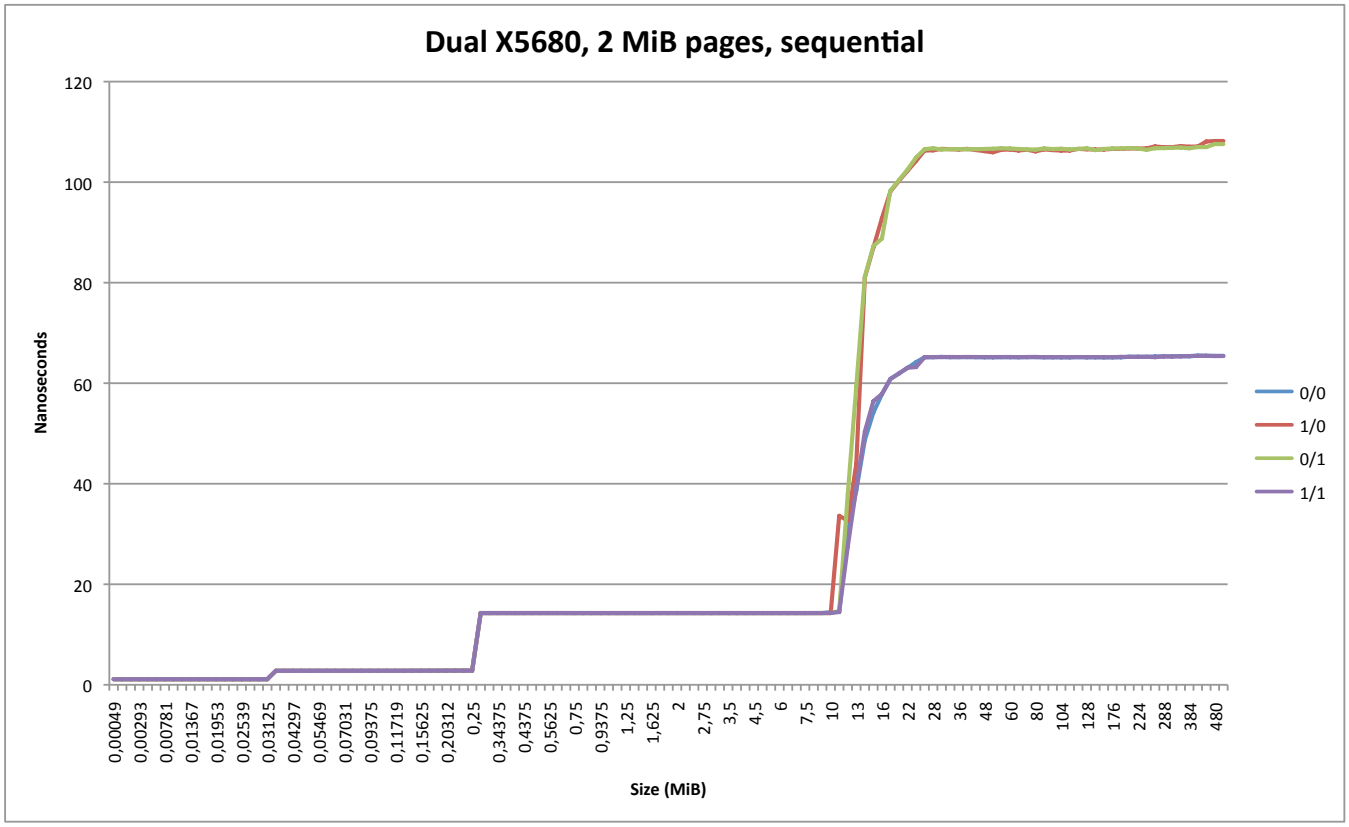

Fig. 1. Latency to walk through an array of varying size, from the lat_mem_rd benchmark

then half the threads will exclusively use remote memory, and performance will suffer accordingly.

Depending on the kind of workload, there are some simple heuristics to limit the nefarious effects of NUMA:

Single process, single thread
: the default behavior will only use the local NUMA node as explained. This might be a good thing (it optimizes the latency) or a bad thing (it cuts the available memory bandwidth in half). Some code might benefit from interleaving pages on all nodes, either by prefixing the command with **numactl –interleave=all**, or by replacing the allocation function such as **malloc** by a NUMA-specific function such as **numa_alloc_interleaved**.

Multiple processes, each single thread
: this is the norm for MPI-based code that does not include multithreading. The default behavior of Linux (provided the process have been properly "pinned" to their CPU) is excellent, as all processes will have low latency memory, yet the multiplicity of processes will ensure the use of all memory controllers and available bandwidth.

Single process, multiple threads
: this is the norm for OpenMP-based code. The default behavior is usually quite bad; more often than not, the code will cause the allocation of physical pages on the master thread's node, despite parallel sections running on all CPUs. Reading data from a file, receiving data from the network, explicit non-parallel zeroing or initialization of the allocated memory will all lead to this single-node allocation. Forcing interleaving as above usually helps (by utilizing all available bandwidth), but is not optimal (it forces the averaging of latency rather than minimizing it); it still should be tried first, as it is very easy to implement. The ideal solution is to be able to place each page on the node whose threads will make the most use of it, but that is not a simple task.

Multiple processes, each multiple threads
: although it might seem the most complicated case, it usually isn't: the typical placement of one process in each socket, with as many threads as there are cores in the socket leads to an optimal placement by the default behavior. No matter which thread allocates the memory, all the threads of the same process will see optimal latency.

C. False sharing

False sharing is a long-standing problem in a cache coherent multi-core machine (it is described as "widely believed to be a serious problem" by Bolosky and Scott [38]). Modern CPUs generally use invalidation-based coherency protocol (relevant details can be found in Culler et al. [39]), whereas every write

to a memory address requires the corresponding cache line to be present in the local CPU cache with exclusive write access-that is, no other cache holds a valid copy.

"True sharing" occurs when a single data element is used simultaneously by more than one CPU. It requires careful synchronization to ensure respect of the semantic, and incurs a performance penalty as each update requires the invalidation of the other CPU's copy. Back-and-forth can become very costly but is unavoidable if required by the algorithm (unless, of course, one changes the algorithm).

"False sharing" occurs when two different data elements are used by two different CPUs, but happen to reside in the same cache line. There is no need for synchronization. However, because the granularity of invalidation is the whole cache line, each data update by a CPU will invalidate the other CPU's copy. This requires costly back-and-forth of the cache line. This can greatly affect performance and is completely unnecessary: if the two data elements lived in different cache lines, no invalidation would occur for either, thus ensuring maximum performance.

For very small data structures (such as a per-thread counter), this can be a huge problem as each update will lead to an unnecessary invalidation. For instance, listing 16 describes a structure of two elements: **neg** and **pos**. Without the intermediate padding array **pad**, those two values would share a cache line, and when used by two different threads, would cause a major performance issue. The padding ensures that they are in different cache lines, thus avoiding the problem. A synthetic benchmark using this structure running two threads on two different sockets in a dual X5680 system would see a increase in time of 50% as opposed to when the padding was not in use. Hardware counters show a very large number of cache line replacement in the modified state when padding is not used, versus a negligible number when it is (cache coherency and the modified state are well explained by Culler et al. [39]).

Listing 16. Example of false sharing

```
typedef struct {
  int neg;
#ifdef NOCONFLICT
  int pad[15];
#endif
  int pos;
} counters;
```

For large arrays that are updated in parallel by multiple threads, the data distribution among threads will be an important factor. If using a round-robin distribution (i.e. thread number n out of N total threads updates all elements of indices m such as $n \equiv m \pmod N$), then false sharing will occur on most updates - not good. But if a block distribution was used, where each thread accesses $1/N$ of the elements in a single continuous block, then false sharing only occurs on cache lines at the boundaries of each block - at most one cache line will have shared data between two threads. Not only will the number of unnecessary invalidations be small relative to the total number of accesses, but they will usually be masked by the fact that one thread will update at the beginning of the computation, and the other one at the end, thus minimizing the effect. And to ensure conflict-free accesses, the block size that each thread handles can be rounded to a integer multiple of the cache line size, at the cost of a slightly less efficient load balancing between threads.

VII. Conclusion

This paper introduces a few key points for the newcomer to keep in mind when trying to improve the performance of code. To summarize, it is important to be able to justify the validity of the work done, in terms of reliability, speed and the choice made when improving the code. Other points are to properly exploit the tools at hand before modifying the code, to understand the relation between the data structures and the performance of the code, avoid re-inventing the wheel and exploit pre-existing high-performance implementations of algorithms, and finally run parallel code in a manner adapted to the underlying hardware.

Hopefully, this will help programmers and scientists alike to understand some key aspects of performance and obtain higher performing code without an excessive amount of work. Happy (performance) programming!

References

[1] S. Chellappa, F. Franchetti, and M. Püschel, "How to write fast numerical code: A small introduction," in Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE), ser. Lecture Notes in Computer Science, vol. 5235. Springer, 2008, pp. 196–259.

[2] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in Proceedings of the April 18-20, 1967, spring joint computer conference, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[3] IEEE Task P754, IEEE 754-2008, Standard for Floating-Point Arithmetic, Aug. 2008. [Online]. Available: http://ieeexplore.ieee.org/servlet/opac?punumber=4610933;http://en.wikipedia.org/wiki/IEEE_754-2008

[4] N. J. Higham, Accuracy and Stability of Numerical Algorithms, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[5] J. L. Gustafson, "Reevaluating amdahl's law," Commun. ACM, vol. 31, no. 5, pp. 532–533, May 1988. [Online]. Available: http://doi.acm.org/10.1145/42411.42415

[6] OpenMP Architecture Review Board, "Openmp application program interface," Specification, 2011. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[7] T. M. Forum, "Mpi: A message passing interface," 1993.

[8] Intel Corporation. Intel®vtune™amplifier xe 2013. [Online]. Available: http://software.intel.com/en-us/intel-vtune-amplifier-xe

[9] Callgrind: a call-graph generating cache and branch prediction profiler. [Online]. Available: http://valgrind.org/docs/manual/cl-manual.html

[10] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach. San Mateo, CA: Morgan Kaufmann, 1990.

[11] National Science Foundation and Department of Energy. BLAS. http://www.netlib.org/blas/. [Online]. Available: http://www.netlib.org/blas/

[12] D. Barthou, G. Grosdidier, M. Kruse, O. Pène, and C. Tadonki, "QIRAL: A High Level Language for Lattice QCD Code Generation," CoRR, vol. abs/1208.4035, 2012.

[13] R. M. Russell, "The cray-1 computer system," Commun. ACM, vol. 21, no. 1, pp. 63–72, Jan. 1978. [Online]. Available: http://doi.acm.org/10.1145/359327.359336

[14] I. B. G. team, "Design of the ibm blue gene/q compute chip," IBM Journal of Research and Development, vol. 57, no. 1/2, pp. 1:1–1:13, 2013.

[15] A. J. C. Bik, Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance. Intel Press, 2004.

[16] Intel Corporation, Intel®64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z, March 2013, no. 325383-046.

[17] P.-F. Lavallée, G. Colin de Verdière, P. Wautelet, D. Lecas, and J.-M. Dupays, "Hydro," https://github.com/HydroBench/Hydro/tree/master/HydroC, 2012.

[18] R. Teyssier, "Cosmological hydrodynamics with adaptive mesh refinement: a new high resolution code called RAMSES," Astron. Astrophys., vol. 385, pp. 337–364, Nov. 2001. [Online]. Available: http://dx.doi.org/10.1051/0004-6361:20011817

[19] R. Teyssier, S. Fromang, and E. Dormy, "Kinematic dynamos using constrained transport with high order godunov schemes and adaptive mesh refinement," J. Comput. Phys., vol. 218, no. 1, pp. 44–67, Oct. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.jcp.2006.01.042

[20] S. Fromang, P. Hennebelle, and R. Teyssier, "A high order godunov scheme with constrained transport and adaptive mesh refinement for astrophysical magnetohydrodynamics," Astronomy and Astrophysics, vol. 457, no. 2, pp. 371–384, 2006.

[21] J. Treibig. likwid: Lightweight performance tools. [Online]. Available: http://code.google.com/p/likwid/

[22] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "The KECCAK reference," January 2011, http://keccak.noekeon.org/.

[23] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "KECCAK implementation overview," May 2012, http://keccak.noekeon.org/.

[24] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," SIGARCH Comput. Archit. News, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[25] D. Truong, F. Bodin, and A. Seznec, "Improving cache behavior of dynamically allocated data structures," in Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on, 1998, pp. 322–329.

[26] A. V. Aho and J. E. Hopcroft, The Design and Analysis of Computer Algorithms, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1974.

[27] K. E. Atkinson, An Introduction to Numerical Analysis. New York: Wiley, 1978.

[28] S. Arora and B. Barak, Computational Complexity: A Modern Approach, 1st ed. New York, NY, USA: Cambridge University Press, 2009.

[29] Intel Corporation. Intel®math kernel library 11.0. [Online]. Available: http://software.intel.com/en-us/intel-mkl

[30] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "Lapack: a portable linear algebra library for high-performance computers," in Proceedings of the 1990 ACM/IEEE conference on Supercomputing, ser. Supercomputing '90. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 2–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=110382.110385

[31] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," Proceedings of the IEEE, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[32] M. Frigo, "A fast Fourier transform compiler," in Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation, vol. 34, no. 5. ACM, May 1999, pp. 169–180.

[33] M. Herlihy and N. Shavit, The Art of Multiprocessor Programming. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[34] T. Mattson, B. Sanders, and B. Massingill, Patterns for parallel programming, 1st ed. Addison-Wesley Professional, 2004.

[35] B. Chapman, G. Jost, and R. v. d. Pas, Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press, 2007.

[36] W. Gropp, E. Lusk, and A. Skjellum, Using MPI (2nd ed.): portable parallel programming with the message-passing interface. Cambridge, MA, USA: MIT Press, 1999.

[37] A. S. Tanenbaum, Modern operating systems. Englewood Cliffs, N.J.: Prentice Hall, 1992.

[38] W. J. Bolosky and M. L. Scott, "False sharing and its effect on shared memory performance," in USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4, ser. Sedms'93. Berkeley, CA, USA: USENIX Association, 1993, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1295480.1295483

[39] D. E. Culler, A. Gupta, and J. P. Singh, Parallel Computer Architecture: A Hardware/Software Approach, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.