Heterogeneous Hardware Parallelism
Review of the IN2P3 2016 Computing School

Vincent Lafage
Engineer, IPNO, IN2P3, CNRS, 15 rue Georges Clémenceau, 91406 ORSAY Cedex, FR
E-mail: lafage@ipno.in2p3.fr

Abstract. Parallel and hybrid Monte Carlo computation.

The Monte Carlo method is the main workhorse for computation of particle physics observables. This paper provides an overview of various HPC technologies that can be used today: multicore (OpenMP, HPX), manycore (OpenCL). The rewrite of a twenty years old Fortran 77 Monte Carlo will illustrate the various programming paradigms in use beyond language implementation. The problem of parallel random number generator will be addressed. We will give a short report of the one week school dedicated to these recent approaches, that took place in École Polytechnique in May 2016.

1. Introduction

It has been over ten years since MOORE’s law last provided us with free lunch in computing: the planned, and still observable, growth of the density of transistors per area of chip, was allowing a similar growth of the CPU clock frequencies. This density growth, kept at the constant rate of doubling every two years, has been with us for over forty five years, and while it may disappear even before we reach unmanageable quantum effects in the components at this scale, for financial rather than technical reasons, we have lost the corresponding frequency scaling around 2004. This convenient frequency growth was also correlated with an increasing power dissipation, that lead the industry to the so-called ’heat wall’. The loss of the frequency scaling means that our program have stopped getting faster by simply being ported to a faster CPU. Instead, we have to implicitly rely on the compiler know-how, or worse, explicitly by changing our code to better adapt contemporary architecture. These modern architectures have benefited of the increase in components density in many ways: with wider buses, registers and corresponding larger vector units; with deeper pipelines, better branch prediction and more complex instruction sets; with better use of dead time by sharing multiple logical thread (hyper-threading); with bigger cache memory organized in more levels. Hence the hardware speedup did not stall over these years. But we now have to know better the architecture to get the most out of it. However clever the compilers, we have, at least, to provide it with the options that enable to take advantage of the hardware. And the biggest work remains to provide hints in the code that allows the compiler to make the better use of the computer.

In order to address the recent technologies that appeared and a way to combine them on heterogeneous or hybrid hardware, we decided to prepare the 2016 IN2P3\footnote{french National Institute of Nuclear Physics and Particle Physics} school of computing.
To address various technologies within only one week of school, we needed a short enough program. All the while this program had to be of interest for the researchers of the field, and as such be mostly concentrated on particle physics. What is more, the aim was not to address niceties in parallel programming but more to concentrate on how much raw performance can result from one or the other technology. We decided to elaborate on a simple phenomenology Monte Carlo code, twenty years old. It was designed to compute the Standard Model Quantum Electro Dynamical production of three photons at the LEP collider in CERN as well as the production of the same state resulting from a generic model of new physics. This new physics was modeled with two anomalous coupling of the $Z^0$ boson to three photons: these dimensional couplings were related to a scale above which their point-like structure would in fact reveal loops of new massive particles such as magnetic monopoles.

Beyond the physics aspects, it was a rather classical code, written in Fortran 77 as was traditional in the early nineties. The initially sequential program was thoroughly profiled and checked for potential cache and branch miss.

The first part in the preparatory step was to convert the initial code in various contemporary language. A first translation was of course to Fortran 90. C++, was beyond doubt a language of choice. Ada is a less well known language in high-energy physics, but it allows easy multi-threading embedded in its high-level syntax and was chosen as well. We also had to translate to C in view of using OpenCL. In fact, this translation work allowed us to see the functional aspect of the initial code, that makes it quite modern.

The next part was to express parallelism at the algorithmic level. The particular choice of the simplest Monte Carlo algorithm lead to what is called an “embarrassingly parallel” program: a problem which can obviously be separated into a number of identical parallel sub-problems, with no, or almost no, dependency or need for communication between those parallel tasks. “Embarrassingly parallel” is a traditional but quite ironically-chosen name, as such a problem proves to be rather “delightfully parallel”.

But before going for a parallel version, purely sequential issues must be addressed.

### 2. Checking for sequential performance

#### 2.1. Profiling

The usual way to access the executable run profile starts with asking the compiler for an instrumented code generation. This extra-code will of course generate some overhead at run time, but this overhead is evenly distributed. This profiling technique is a dynamical one: it requires running the executable, and then use a dedicated application (such as gprof) to look at the breakdown of computation time spent.

The resulting report looks like Table 1. One can see some effects of compiler optimization on these report, as exemplified (for the same source code) in Table 2.

#### 2.2. Monitoring cache miss and branch miss

There are further tools to dynamically investigate a code: valgrind. This program is well-known to help debug memory fault, but can do an even more powerful job of monitoring cache misses and branch misses. We ran our code in this environment, and display the results in Tables 3 and 4. One can see that cache miss is almost non-existent while there is a small, but significant, rate of branch misprediction.

### 3. Expressing parallelism

This simplest, non adaptive Monte-Carlo algorithm, rely on looping over a basic sampling routine: one first generates an event with a weight (that consists in two steps: a pseudo-random

---

2 [http://valgrind.org](http://valgrind.org)
Table 1. Flat profile from execution of an unoptimized sequential version of the code. Each line corresponds to one function of the program.
Column #1 is the percentage of the total running time used by this function.
Column #2 is a running sum of the time (s) accounted for by this function and those above it.
Column #3 is the number of seconds accounted for by this function alone.
Column #4 is the number of times this function was invoked, if it is profiled.
Column #5 is the average time spent (ns) in this function per call, if it is profiled.
Column #6 is the average time spent (ns) in this function and its descendants per call.
Column #7 is the name of the function.

| % cumulative | self  | self  | total    |
|--------------|-------|-------|----------|
| time         | seconds | seconds | calls | ns/call | ns/call | name                        |
| 12.4         | 4.44   | 4.44  | 495625330 | 8.95 | 8.95 | gsl_complex_mul [5]         |
| 11.2         | 8.47   | 4.03  | 7083219   | 568.95 | 2228.41 | computeME2 [3]              |
| 10.1         | 12.09  | 3.62  | 469162322 | 7.71 | 7.71 | gsl_complex_mul_real [6]    |
| 9.6          | 15.53  | 3.44  | 100000000 | 344.50 | 901.05 | computeSpinorProducts [4]   |
| 5.4          | 17.48  | 1.95  | 256665752 | 7.60 | 7.60 | gsl_complex_conjugate [10]  |
| 5.1          | 19.29  | 1.81  | 100000000 | 181.00 | 288.00 | generateRambo [7]           |
| 4.7          | 20.96  | 1.68  | 2000000000 | 8.38 | 8.38 | gsl_complex_negative [11]   |
| 3.7          | 22.30  | 1.33  |          |        |        |                            |
| 2.9          | 23.34  | 1.05  | 42499314  | 24.71 | 27.06 | gsl_complex_div [15]        |
| 2.7          | 24.30  | 0.95  |          |        |        |                            |
| 2.3          | 25.13  | 0.83  | 21249657  | 39.29 | 118.83 | QEDAmplitudePPP [8]         |
| 2.3          | 25.96  | 0.83  | 100000000 | 8.30 | 8.30 | gsl_complex_sub [20]        |
| 2.3          | 26.79  | 0.83  |          |        |        |                            |
| 2.1          | 27.52  | 0.73  | 21249657  | 34.59 | 114.13 | QEDAmplitudePMM [9]         |
| 1.8          | 28.16  | 0.64  | 100000000 | 64.00 | 82.52 | computeScalarProducts [22]  |
| 1.6          | 28.75  | 0.58  | 1200000000 | 4.83 | 4.83 | ocl_rng_rand48_nextState [23] |
| 0.9          | 34.55  | 0.34  | 100000000 | 34.00 | 34.00 | sortPhotons [32]            |
| 0.8          | 34.85  | 0.30  | 7083219   | 42.35 | 126.15 | anomalousAmplitudePPP [19]  |
| 0.5          | 35.04  | 0.19  | 28332876  | 6.71  | 6.71 | gsl_complex_add [33]        |

Table 2. Profile from executable optimized with -O3 -march=native -funroll-loops

| % cumulative | self  | self  | total    |
|--------------|-------|-------|----------|
| time         | seconds | seconds | calls | ns/call | ns/call | name                        |
| 24.3         | 4.69   | 4.69  | 469162322 | 10.00 | 10.00 | gsl_complex_mul_real [3]    |
| 7.3          | 6.10   | 1.41  | 1000000000 | 14.10 | 14.10 | gsl_complex_sub [5]         |
| 7.3          | 7.51   | 1.41  |          |        |        | __ieee754_log_avx [7]       |
| 7.3          | 8.92   | 1.41  |          |        |        | sincos [6]                  |
| 7.1          | 10.28  | 1.36  | 200000000 | 6.80  | 6.80  | gsl_complex_negative [8]    |
| 5.6          | 11.36  | 1.08  |        |        |        | scalbn [9]                  |
| 5.0          | 12.33  | 0.97  |          |        |        | computeSpinorProducts [1]   |
| 4.4          | 13.18  | 0.85  |          |        |        | generateRambo [4]           |
| 4.3          | 14.00  | 0.82  |          |        |        | computeME2 [2]              |
| 3.2          | 14.61  | 0.61  | 120000000 | 5.08  | 5.08  | ocl_rng_rand48_getDouble [11] |
| 2.6          | 15.12  | 0.51  | 495625330 | 1.03  | 1.03  | gsl_complex_mul [13]        |
| 2.5          | 15.61  | 0.49  |          |        |        | __scalbn [14]               |
| 2.1          | 16.01  | 0.40  |          |        |        | below [15]                  |
| 1.8          | 16.36  | 0.35  | 28332876  | 12.35 | 12.35 | gsl_complex_add [17]        |
| 1.8          | 16.70  | 0.34  |          |        |        | selectEvent [19]            |
| 1.7          | 17.03  | 0.33  | 42499314  | 7.76  | 8.24  | gsl_complex_div [16]        |
| 1.3          | 18.14  | 0.26  | 269987256  | 0.96  | 0.96  | gsl_complex_abs2 [26]       |
| 1.0          | 18.33  | 0.19  | 256665752  | 0.74  | 0.74  | gsl_complex_conjugate [28]  |
| 0.8          | 18.48  | 0.15  |          |        |        | computeScalarProducts [27]  |
| 0.7          | 18.61  | 0.13  | 21249657  | 6.12  | 29.49 | QEDAmplitudePMM [10]        |
Table 3. Cache & branch misses
These listings describe the number of fetch (refs) of instruction (I) and data (D) from cache Level 1 (I1, D1) or Last Level (LLi, LLd, total in LL), as well as the number of cache-miss occurring and the corresponding rate. In the case of data, the number of read access and write access are shown separately. The number of branches occurring in the execution, and the number of branch mispredictions are also displayed.

\begin{verbatim}
time valgrind --tool=cachegrind --branch-sim=yes mc

==32423== I refs: 21,866,876,065
==32423== I1 misses: 90,912
==32423== LLi misses: 2,966
==32423== I1 miss rate: 0.00% => 4.2.10^-6
==32423== LLi miss rate: 0.00%

==32423== D refs: 10,907,321,258 (7,120,262,982 rd + 3,787,058,276 wr)
==32423== D1 misses: 17,778 ( 15,391 rd + 2,387 wr)
==32423== LLd misses: 9,880 ( 8,407 rd + 1,473 wr)
==32423== D1 miss rate: 0.0% ( 0.0% + 0.0% ) => 1.6.10^-6
==32423== LLd miss rate: 0.0% ( 0.0% + 0.0% )

==32423== LL refs: 108,690 ( 106,303 rd + 2,387 wr)
==32423== LL misses: 12,846 ( 11,373 rd + 1,473 wr)
==32423== LL miss rate: 0.0% ( 0.0% + 0.0% )

==32423== Branches: 920,763,441 ( 853,502,037 cond + 67,261,404 ind)
==32423== Mispredicts: 48,791,446 ( 42,788,533 cond + 6,002,913 ind)
==32423== Mispred rate: 5.2% ( 5.0% + 8.9% )
\end{verbatim}

Table 4. Cache & branch misses, optimized with -O3 -march=native -funroll-loops

\begin{verbatim}
time valgrind --tool=cachegrind --branch-sim=yes mc

==20432== I refs: 8,807,755,982
==20432== I1 misses: 16,437
==20432== LLi misses: 2,828
==20432== I1 miss rate: 0.00%
==20432== LLi miss rate: 0.00%

==20432== D refs: 3,307,797,849 (1,911,804,779 rd + 1,395,993,070 wr)
==20432== D1 misses: 17,778 ( 15,391 rd + 2,387 wr)
==20432== LLd misses: 9,880 ( 8,407 rd + 1,473 wr)
==20432== D1 miss rate: 0.0% ( 0.0% + 0.0% )
==20432== LLd miss rate: 0.0% ( 0.0% + 0.0% )

==20432== LL refs: 34,087 ( 31,711 rd + 2,387 wr)
==20432== LL misses: 12,675 ( 11,204 rd + 1,471 wr)
==20432== LL miss rate: 0.0% ( 0.0% + 0.0% )

==20432== Branches: 701,657,379 ( 642,396,799 cond + 59,260,580 ind)
==20432== Mispredicts: 48,791,446 ( 42,788,533 cond + 6,002,913 ind)
==20432== Mispred rate: 5.2% ( 5.0% + 8.9% )
\end{verbatim}

number generator, such as the one in [3], and a routine to translate these to a random event, such as [6]). The program then checks if it passes cuts, computes matrix element, weights it, sums it up, and starts again. As each event is supposed to be independent, and there are no
explicit dependencies, the algorithm should lend itself to parallelization. The parallel threads only have to meet at the end of the loop to sum up their partial results, an operation known as reduction.

3.1. Random number generation

Even good generators, such as the one from Knuth [7,8], that avoid the classical problem of linear congruential generator (see [9]) are essentially sequential: you have to go through all the sequence to reach a given index, and this is the major obstacle to parallelizing the algorithm. The famous “Mersenne twister” generator [10], allows, among others, to “fast forward” efficiently through the sequence, and having a very long period, can start by generating the seeds of well separated sub-sequences, one for each thread. So doing keeps the sequential nature of the generator, but allows to have non-overlapping parallel sub-sequences. This was our starting point for this school. But since then we have been introduced to Random123, a random generator that is a direct function of the index of the sequence (see [11] in [12], as well as [13]). What is more, it is blessed with very good randomness properties, passing the BigCrush [14] test suite, and we are porting it to all future versions of our code.

4. Technologies illustrated

4.1. OpenMP

This technology has already been with us for almost twenty years. It should, in our opinion, be considered as the natural entrance for beginner and non-expert wishing to turn their old software into a multithreading package. It relies on compiler directives that are commented by default but can be turned on when parallelizing. It allows to keep the purely sequential version alive within the same code base. Parallelization is mostly an industrialization process: it is about organizing data space so that different actors can process independently, sharing only the minimal requested information. OpenMP directive allow to specify this data and process organization.

We should now discuss the scaling of the performance and what it tells us about the underlying architecture and, paradoxically, the real sequential performance. The program has been run on a 16 physical cores machine with hyper-threading, with setting the number of simultaneous thread from 1 to 32 (see Figures 1 and 2). It turns out that on 32 logical cores, the CPU usage was about 27: a result accepted with mixed feelings, as it was impressive, but yet not perfect, and with the program so cleanly parallel in principle, the cause of the default in scaling was not clear. What is more, the speedup, ideal up to 11 threads, has a plateau until 15 threads, before increasing again more and more slowly.

In order not to be fooled by the speedup, we need to know how close to the full potential of the hardware we could reach. This is now possible with the recent tool maqao [3], a Modular Assembly Quality Analyzer and Optimizer which can be used for static binary executable analysis. This reports as exemplified in Table 5 and shows that at most the sequential version was using but 13% of the computing capability of the CPU. In fact, the program did not express vectorization (see Table 5), and the vector arithmetic unit is left unsupplied. This point shed some light on our understanding of the effect of hyper-threading on a code that in principle should absolutely not benefit from it: there is no input/output in the loop and hardly any cache-miss, be it for the few data required or for the instructions. The alternate thread sharing a given physical core should then have almost no opportunity to use dead time, and so be almost useless.

As the speedup is quietly saturating (at about 17.6), but above the number of physical cores (16), we can see that some ten percents of hyper-threading occurs, and points to some dead time

3 http://www.maqao.org/
Figure 1. Multi-thread speedup

Figure 2. Efficiency
Table 5. Excerpt of maqao output example
This program diagnoses various types of inefficiencies directly in the executable code of one of the functions in the program: here, the function bppp computing one of the matrix element. Diagnosis is reported in paragraphs, suggesting improvements for each problem. In this case, the report exhibits issues with a lack of vectorization of the code.

```bash
$ maqao.intel64 cqa mc fct-body=bppp
Section 1: Function: bppp
-------------
13\% of peak computational performance is used (2.13 out of 16.00 FLOP per cycle (5.97 GFLOPS @ 2.80GHz))

Code clean check
----------------
Detected a slowdown caused by scalar integer instructions (typically used for address computation).
By removing them, you can lower the cost of an iteration from 23.00 to 17.00 cycles (1.35x speedup).

Vectorization status
-------------------
Your function is probably not vectorized (store and arithmetical SSE/AVX instructions are used in scalar mode and, for others, at least one is in vector mode).
Only 26\% of vector length is used.

Vectorization
-------------
Your function is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could benefit from full vectorization.
By fully vectorizing your function, you can lower the cost of an iteration from 23.00 to 5.75 cycles (4.00x speedup).
Since your execution units are vector units, only a fully vectorized function can use their full power.

Two propositions:
- Try another compiler or update/tune your current one:
- Make array accesses unit-stride.
  * If your function streams arrays of structures (AoS), try to use structures of arrays instead (SoA):
    do i a(i)x = b(i)x (slow, non stride 1) => do i a%x(i) = b%x(i) (fast, stride 1)

FMA
---
Presence of both ADD/SUB and MUL operations.
- Try to change order in which elements are evaluated (using parentheses) in arithmetic expressions containing both ADD/SUB and MUL operations to enable your compiler to generate FMA instructions wherever possible.

in the sequential version. It would be interesting to track the excess of CPU usage appearing above 12 threads.

As far as precision is concerned, runs with 1 to 32 threads lead to the same result within a relative standard deviation of $3.8 \times 10^{-9}$, which reflects different rounding errors depending on summation order. This is negligible in a multi-core approach, but with a much larger number of threads, as on GPU, it might be interesting to implement a parallel KAHAN summation algorithm.

4.2. Ada tasking
Amongst the first attempts at writing a parallel version of the Monte Carlo code, Ada was chosen for its ability for concurrent programming. The sequential version did not suffer more than 20 percent speed penalty compared to the Fortran version. But without a good parallel pseudo-random number generator, one task was dedicated to producing random number that would be used by other tasks, limiting the gullibility. The next step will use the Random123 generator.
4.3. The Go programming language

A sequential version of the Monte Carlo code was written in Google’s Go language [clang.org] achieving the same level of performance as Fortran. After implementing the Random123 generator, the concurrent aspect will be investigated.

4.4. MPI

Message Passing Interface (MPI) is among the oldest (1994) common parallel, and more specifically distributed memory, technologies. It allows for a large and uniform scalability (as one can always extend the number of node, as long as the problem can cope with this extension). It relies on the conventional communicating sequential processes (CSP) [17] model. It has been used both independently and as one component of “hybridization” in the GPU approach of the school: MPI+OpenCL.

4.5. GPU with OpenCL

General-Purpose computing on Graphics Processing Units (GPGPU) are a set of frameworks to accelerate computation by exploiting the massive vectorization of Graphics Processing Units. For instance CUDA, OpenACC and OpenCL. These last two are programming standards.

OpenCL [18] is a programming language based on C99, extended to facilitate use of parallelism with vector types and operations, synchronization, and functions that allows to write both CPU programs and GPU executed kernels. OpenCL can also run kernels on CPU. It has been used both independently and as one component of “hybridization” in the GPU approach of the school: MPI+OpenCL.

The performance was tested on the GridCL platform [19] built of four node servers (each with a dual processor Intel E52650 at 2 GHz, 16 cores, 64 GB of RAM, with AVX vectorization on four doubles, and hyperthreading 32 threads) connected with Infiniband. Each node is equipped with two GPU boards: two nodes with NVidia K20 and the two others with Intel Xeon Phi. Another standalone server is equipped with six NVidia Titan GPU boards. Table 6 details the speedup observed when comparing various configuration and shows an impressive factor 147 when using simultaneously the six NVidia Titan GPU boards.

Table 6. Measurement of run times speed-up compared to the sequential C++

The line “One Device” displays the speedup when only one GPU board (or CPU set of cores) is used, the line “All Devices” when all GPU boards of the same kind are used (across Infiniband where applicable).

|                      | C++ 1-core | OpenCL 16-cores | OpenCL NVidia GPU | OpenCL Titan GPU | OpenCL Xeon Phi GPU |
|----------------------|------------|-----------------|-------------------|------------------|---------------------|
| One Device            | 1          | 24.7            | 22.1              | 26.7             | 16.4                |
| All Devices           | 1          | 63.0            | 146.7             | 47.5             |                     |

4.6. HPX libraries [20]

HPX (High Performance ParalleX) is a C++ library implementing the ParalleX execution model proposed to go beyond CSP model [17].

[17] https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner
[18] http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf
[19] https://www.khronos.org/opencl/
[20] http://stellar-group.org/libraries/hpx/
The goal of the ParalleX model of computation is to address the key challenges of efficiency, scalability, sustained performance, and power consumption with respect to the limitations of conventional programming practices (e.g., MPI), by enabling a new computing dynamic through the application of message-driven computation in a global address space context with lightweight synchronization.

5. Other presented technologies

5.1. DSL: Domain Specialized Languages

Although we did not implement such a solution for our simple Monte Carlo with DSL, the problem we met in parallelizing helps us understanding the approach. Often a program gives a sequential solution to a problem relying on nests of loops, that in turn get across the specific architecture and / or memory model of the parallel computing environment. A complete rewriting of the code then seems unavoidable, and we then understand that implementing a solution has been losing the generality of the problem: we then call for a more abstract expression of our problem. To fill the gap of abstraction between our mathematical vision of the problem and the actual code, we need a higher-level language (closer tho the mathematical language) and a corresponding translator to our fast computing language (Fortran, C++) that will transfer formula to vectorized expression or correctly nested parallel loops in a much faster and more versatile way than we can code “by hand”. We can see this as a dedicated code generator.

The Loopy system (coded in Python) is such a generator and was presented by its author [21].

5.2. OpenCL for FPGA

The general idea of using a dedicated hardware acceleration involves a high cost. Field Programmable Gate Array (FPGA) aims at combining the speed of dedicated hardware with the flexibility of programming. But in practice, “compiling a hardware solution” is lengthy and delicate and requires a range of expertise. Building a dedicated hardware allows to exploit both larger and/or more adequate vector computing units with way deeper computation pipeline.

The idea here is to generate the specific architecture directly from an OpenCL code. However faster it is, the development cycle is still affected by large compilation/synthesizing time (order of many hour to one day).

6. Conclusion

Sixty persons applied to attend the school for thirty places. The goal of this school was to exemplify ways to turn a well-tested sequential problem into one exploiting most of the performance of contemporary computers. And it succeeded in achieving impressive performance such as speedup of 147 on a multi-GPU nodes. Hybrid programming such as MPI+OpenCL was also up to the task for large scalability.

A next edition would certainly emphasize other approaches such as CUDA Fortran [22] or the promising OpenACC [23] that take the same approach “by compiler directives” as OpenMP to turn a sequential application in one using GPU. Another environment of interest, Kokkos, implements a programming model in standard C++ for writing performance portable applications [24, 25].

As the goal is to help users move from the sequential to the parallel paradigm, tools like kremlin (see [26, 27]) should be investigated.

References

[1] Chamont D L, Grasseau G L and Lafage V I 2016 École Informatique IN2P3 « Parallélisme sur Matériel Hétérogène » URL https://indico.in2p3.fr/event/13126/

[2] Baillargeon M, Boudjema F, Chopin E and Lafage V 1996 Z. Phys. C71 431–442 (Preprint hep-ph/9506396) doi:10.1007/BF02907001
