The future of commodity computing and many-core versus the interests of HEP software

2012 J. Phys.: Conf. Ser. 396 052058

(http://iopscience.iop.org/1742-6596/396/5/052058)

Journal of Physics: Conference Series **396** (2012) 052058 doi:10.1088/1742-6596/396/5/052058

The future of commodity computing and many-core versus the interests of HEP software

Sverre Jarp, Alfio Lazzaro, Andrzej Nowak

CERN openlab, Geneva, Switzerland

Sverre.Jarp@cern.ch, Alfio.Lazzaro@cern.ch, Andrzej.Nowak@cern.ch

Abstract. As the mainstream computing world has shifted from multi-core to many-core platforms, the situation for software developers has changed as well. With the numerous hardware and software options available, choices balancing programmability and performance are becoming a significant challenge. The expanding multiplicative dimensions of performance offer a growing number of possibilities that need to be assessed and addressed on several levels of abstraction. This paper reviews the major trade-offs forced upon the software domain by the changing landscape of parallel technologies – hardware and software alike. Recent developments, paradigms and techniques are considered with respect to their impact on the rather traditional HEP programming models. Other considerations addressed include aspects of efficiency and reasonably achievable targets for the parallelization of large scale HEP workloads.

1. Introduction

Technological developments of the IT space in recent years have made increasingly apparent the fact that the gap between the raw performance of commodity hardware and High Energy Physics (HEP) software as it is today is growing at an alarming rate. Already simple "back of the envelope" performance projections, also presented further, demonstrate that the current growth model, which has served HEP well for many years, does not allow to take full advantage of latest technologies, and that scalability and sustainability are threatened. Computing hardware is still growing exponentially according to Moore's Law [1], but this growth is not transparent and requires action on the programmer's side.

On the other hand, HEP computing problems are for the most part "embarrassingly parallel", meaning that parallelism is inherent in the data. Particles in analyzed collisions travel through the same detector, in the presence of the same global fields and conditions. In addition, hardware used today is homogeneous to a large extent, and where one had to worry about IBMs, Crays, Apollos and VAX machines, today there is one dominant, performing but largely unharnessed standard – C++ on x86 (and the majority running on Linux).

The historical trade-off has been that some performance would be sacrificed for substantial benefits in code manageability, organization and flow. That has been related to the adoption of C++ as the language of choice, the versatility of which allows for freeform expression while still leaving room for considerable performance. The growing complexity of the code and the frequency-power wall of the 90's have brought the "free lunch" kind of developments in performance to a standstill. More than that, the shifts in the computing landscape have only enhanced the focus on the hardware by offering improvements where they (historically) have not been expected – e.g. vectors, specialized execution

units, hardware threads etc. That has forced software engineers – traditionally liberated from such worries – to think more of the hardware, and not just in one dimension, and to expend significantly more effort on optimization.

The argument presented in this document postulates that a well-organized collaboration thinking well into the future is necessary to improve the usage efficiency of modern commodity platforms and to ensure economic scalability on future ones. Decisions taken in this collaboration would be best made when based on well justified fact, while realistic goals set when taking necessity and feasibility into account.

2. Hardware and software – a fragile relationship

Software performance is a complex equation involving code, compilers, libraries and finally the behavior of various subsystems of the hardware – such as memory, disk I/O, network and CPU. Historically, upper layers have depended on the transparent scaling and growth of the lower layers. In the particular case of microelectronics, Moore's law for transistor scaling is still in force, as demonstrated on Figure 1. The plot was constructed based on data covering over 1700 CPU models produced by Intel between 1971 and 2011, and the trend in silicon developments presented could hold even for a decade [2].

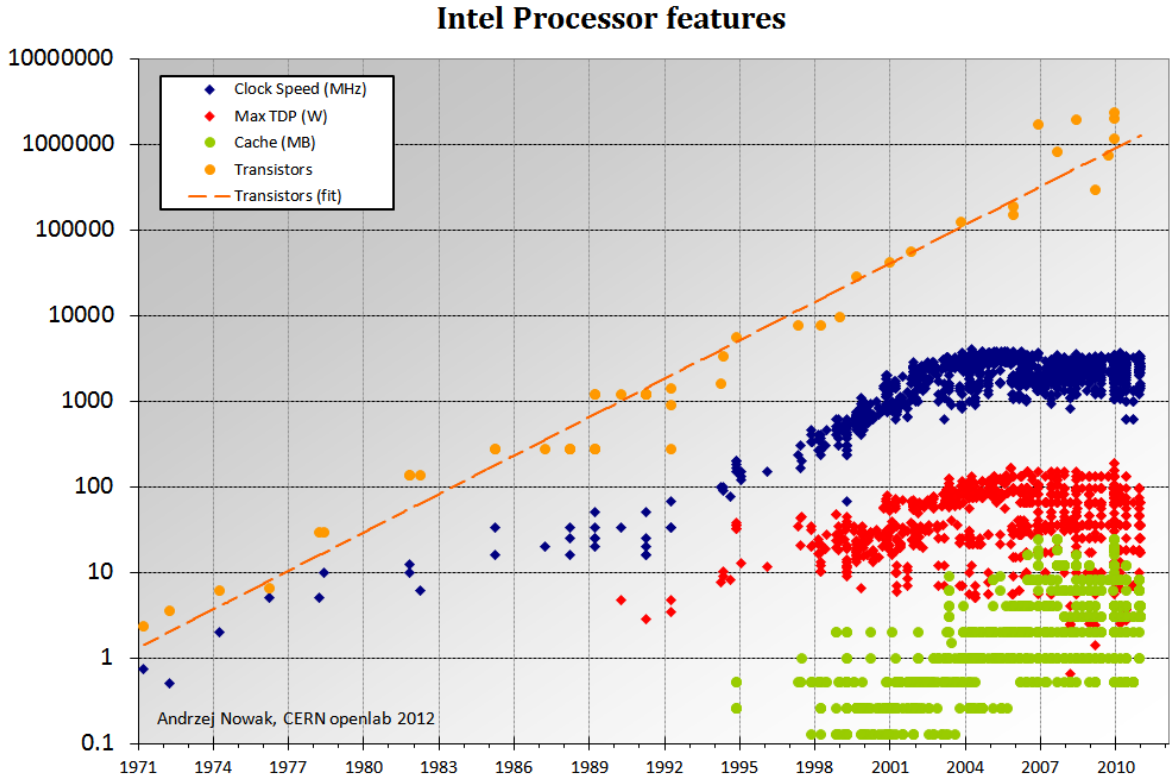

Figure 1: Moore's Law for transistor scaling

Mainly because of heat dissipation issues, savings generated by improved process technologies do not translate to major frequency increases, but are typically re-invested in other areas. It should also be noted that technologies auxiliary to the CPU such as memory or IO do not progress or scale at the same speed as raw compute power.

In order to achieve balanced HEP computing in the future, a further focus on hardware-software co-design will be needed. In particular, a deeper understanding of issues related to software running on a particular piece of hardware will bring more benefit than in the recent past.

2.1. The 7+ dimensions of performance

The overall performance of a modern microchip, especially one from the Intel Architecture family, can be represented as a product of several "performance dimensions" [3]:

- Vectors
- Instruction Pipelining
- Instruction Level Parallelism (ILP)
- Hardware threading
- Clock frequency
- Multi-core
- Multi-socket
- Multi-node

Out of all those listed above, the only dimension which did not use to involve any form of parallelism and which scaled transparently, was frequency. Today, for the reasons presented in this document, it is clear that embracing parallelism and parallel scalability on multiple levels of hardware, as opposed to only one, is the most promising way forward.

The "performance dimensions", even though interdependent, are fundamentally multiplicative. In the HEP context (but not only), a gain or loss in one of them translates in a multiplicative scaling factor with respect to overall system performance. The mapping of these categories onto a fewer number of software dimensions of parallelism is discussed further.

2.2. Where HEP software is today

This section outlines major characteristics of popular HEP software, primarily in view of the packages used by the four major experiments at CERN, with particular focus on Geant4 and ROOT and the production environment on the LHC computing grid. This is being said while keeping in mind that not all performance dimensions can be fully taken advantage of by all computing problems and implementations. Although the discussion touches on both online computing (rendering the data coming straight out of the detectors) as well as offline (processing of the pre-rendered data), the focus is on the latter.

Starting from the lowest level, currently used HEP software rarely makes use of vectors. Production versions of major HEP packages do not vectorize explicitly in any major way and make very limited use of auto-vectorization. For the most part, current code focused on programming productivity does not allow for vectorization to be applied easily. One prominent reason is that at a strategic level control flow has been given extra privileges at the expense of data flow. A major outcome, apart from the constraints on vectorization, is restricted Instruction Level Parallelism (ILP – enabled by the superscalar nature of x86 processors). As opposed to offline, there are more concentrated vectorization efforts in the online domain. In the latter case data parallelism that already exists in the detectors' electronics produces more pressure on the programming model. Those efforts aside, for the most part one double precision number per operation is used, where there could be four or eight (as defined by Intel's AVX [4] and LRBni [5] extensions, respectively).

Instruction Pipelining usually cannot be addressed directly, and as far as the programmer is concerned, it is a part of ILP analysis. Presently, ILP in the x86 domain can be as high as 4, meaning that an x86 CPU can execute up to 4 instructions in a single cycle. In HEP this number usually

oscillates between 0.65 and 1.1 (cycles-per-instruction[1] [CPI] of 0.9-1.5), meaning that executing only one instruction per cycle on average is closer to the "high end"[2]. Particular issues related to low ILP and a low amount of bytes processed per floating point operation (FLOP) are caused by the nature of C++ sourced code, which implies an indirect and dynamic runtime nature of software. Earlier studies [6] found that prominent representatives of HEP code can have as little as 5 to 6 instructions between jumps and 35-70 instructions between calls, with very deep call stacks. Such "jumpy" behavior heavily impacts microarchitectural efficiency, in part because it complicates branch prediction, reduces out-of-order execution possibilities and drastically reduces prefetching opportunities – thus increasing risks of costly instruction and data cache misses. Tuning the performance of such a fragmented profile, where the top consumers barely account for a double-digit percentage of clock cycles, is a complicated ordeal. Finally, the compiler used and its options usually have a rather significant impact on ILP.

As far as hardware threading is concerned, it is often advantageous, but not always. The set of benchmarks usually run at CERN openlab shows a consistent 25% improvement on representative CPU-bound C++ code [7]. In production, SMT (Hyper Threading) remains switched off.

In the multi-core and multi-socket domains, HEP is doing quite well, using all available cores in a system for processing. The implied cost is a pre-negotiated and fixed amount of memory made available per process. Such an approach simplifies production issues and assures steady, independent scaling of many jobs, but will not work for memory-constrained accelerators (such as Intel MIC) and might not scale as the gap between the memory and the CPU grows. It should also be noted that the increasing price performance of memory creates more favorable conditions for SMT.

Finally, thanks to the successful application of the Grid concept, multi-node operation is completely transparent to the programmer. Particularly in offline, HEP does not typically demand HPC-like setups with numerous nodes equipped with fast communication hardware, which enables a cost-effective and safe approach of up to per-node scheduling through queues. Workloads such as CFD or QCD are a rare exception to the average case and can often be approached by applying proven HPC methods. Increasing demand in the online domain, however, and growing distances between sockets, might prompt a reconsideration of the drivers of the behaviors described in the two previous paragraphs.

Table 1 and Table 2 summarize the points made above and – in a very rough "back of the envelope" calculation – contrast production values for HEP code with values typically seen in somewhat optimized industry benchmarks and production environments. Table 1 presents values achieved for each relevant "performance dimension", while each cell in Table 2 represents a multiplicative factor (progressing from the left to the right) illustrating maximum, industry and HEP hardware usage efficiency. The maximum is derived as the theoretical performance of the most potent platform from the standard server segment, ignoring the (small) dependency of performance dimensions on each other. The industry values represent optimized jobs (e.g. from the SPEC2006 benchmark[3]) on representative modern platforms, while the HEP row represents values as they stand for HEP today based on recent production environments at CERN. The values presented in the tables that follow have only an indirect relation to a "value for money" factor. Furthermore, they encompass both the domain of influence of the programmer as well as that of the planners.

[1] The cycles per instruction figure is commonly used as an approximate measure of microarchitectural efficiency. As any ratio, it is dependent on both the numerator and denominator.

[2] Keeping in mind the CPU will actually retire the instructions in "bursts" of several per cycle, followed by several cycles of whitespace.

[3] Optimized SPEC2006 jobs have a CPI of ~0.7; the HEPSPEC06 average is 1.22;

Table 1: Limits on performance dimension scaling; ceiling, industry, HEP

	SIMD	ILP	HW THREADS	CORES	SOCKETS
MAX	4	4	1.35	8	4
INDUSTRY	2.5	1.43	1.25	8	2
HEP	1	0.80[4]	1	6	2

In the case of the industry, an optimal AVX vectorization factor of 2.5x is assumed, in lieu of the hard to reach 4x, and a CPI of 0.7. The SMT benefit is established at 25%, while for HEP it is 0% since SMT remains switched off. A typical modern server platform has 2 sockets with 8 "Sandy Bridge" cores each, while in HEP the previous generation is the prevailing one – or even the one before it ("Westmere" with 6 cores or "Nehalem" with 4).

Table 2: Limits on performance dimension scaling (multiplied); ceiling, industry, HEP

	SIMD	ILP	HW THREADS	CORES	SOCKETS
MAX	4	16	21.6	172.8	691.2
INDUSTRY	2.5	3.57	4.46	35.71	71.43
HEP	1	0.80	0.80	4.80	9.60

It becomes apparent that an optimized industry application on an industry standard platform has an order of magnitude performance gap to the theoretical raw maximum within reasonably established limits. There certainly are many good reasons for the differences, but the distance already is worrying. In the case of HEP, however, there is another significant factor – almost an order of magnitude – of difference from a representative application, and close to two orders of magnitude from the raw hardware performance of an ideal (but still economical) server platform – again, for a variety of reasons. That might suggest that HEP is only using a single digit percentage of the available raw power of a modern platform that could have been purchased, while letting the remaining high double-digit percentage go to waste.

Another notable observation is that, like many other domains of computing, HEP is doing well on the rather high level of abstraction such as multi-node and multi-core processing, but hardware usage efficiency drops considerably on lower levels closer to the silicon – such as vectors and ILP.

2.3. Hardware trends and their implications

Given the figures presented above, it is prudent to consider prevailing industry trends in PC hardware development.

Vectorization clearly is on the rise, with the vector width in x86 doubled to 256 bits in Intel Sandy Bridge and AMD Bulldozer processors, and quadrupled to 512 bits in the Intel Aubrey Isle (Knights or "MIC") family of processors. Furthermore, Intel has announced that the current incarnation of AVX would scale without too much trouble to 512 bits (or even further, if needed), and further relevant vector extension instructions such as AVX2 and AVX3 are in the pipeline for future processors. Vectorization is increasingly important to achieve full performance – but, more importantly, a vectorized program can achieve a higher sustained throughput rate. While today's hardware might not yet fully support a 1-cycle 256-bit throughput for all operations, industry trends suggest upcoming implementations. Vectorization also implicitly defines a data-centric design for maximizing efficiency, while major HEP toolkits like ROOT and Geant4 are still rather control-flow oriented. Difficulties with vectorization in the HEP community, despite the embarrassingly parallel nature of the data, are amplified by past experiences which have often not been positive.

[4] Value based on CPU-bound performance figures for major LHC frameworks

Even if additional execution units are being added to new generations of CPUs, from the practical point of view the changes are either cosmetic or they only give an advantage when using advanced features of the CPU. These are (inadvertently) often ignored by HEP software, the bottlenecks of which are located in other locations of the microarchitecture. It is therefore recommended to focus on improving current sub-par ILP figures through software based changes - in order to have a hope of once exploiting these advanced features.

ILP issues can be addressed through a variety of techniques, the most prominent being performance counter based software tuning and compiler tuning. While relatively complex, the former approach seems to be one of the few good solutions to such problems. Effort is being invested both in the industry [8][9] as well as in research organizations [10] to improve the accessibility to low level performance monitoring information and enhance its correspondence with advice directly useful to the developer in a high level language.

As far as Hardware Threading is concerned, it is receiving moderate improvements over time (e.g. extended buffers) and should work well for typical C++ code. There are therefore few good reasons to ignore it, memory issues notwithstanding. The Intel "MIC" family of accelerators features novel 4-way hardware threading which – while unlikely to appear in server chips anytime soon – illustrates the concept more vividly than the standard 2-way version and could potentially offer further improvements in performance. It should be noted that on many workloads improved ILP creates some canceling effects on Hardware Threading – that is the performance of the two can be inversely related[5].

In the case of multi-core, growth depends on the segment. Core counts per chip and per machine in the desktop space grow arithmetically [11], while the enterprise space is still closer to a geometric increase. The popular EP segment is usually positioned somewhere between the two and it will likely experience a strong sustained arithmetic growth for the next several years, which will slow down further on. What is clear, however, is that parallelism can be expected to expand noticeably in dimensions other than the number of cores.

Socket count is another area in which one could expect some moderate growth. Already today AMD is positioning their cost-effective 4-socket designs against Intel's predominantly dual socket EP segment – the segment from which the majority of HEP production machines are sourced. Intel answered by producing cheaper versions of multi-socket platforms, most notably the 4-socket family of Sandy Bridge based server processors. It is not unreasonable to expect that average system-wide socket count could grow in several years, as demand from customers increases. However, sustained growth is limited by cache coherency, IO and RAM access issues – as is the case with large core counts per package.

X86 based systems will also see incursions into the hybrid world, primarily in terms of multiple socket hybrid systems or (later) single socket ones. The concept of microservers has seen some preliminary examinations, as has the idea of hybrid chips with larger and smaller cores mixed in a single package. In the latter case, "big" cores would resemble more today's server chips, while "smaller" cores would be either application specific acceleration units or data-driven compute engines. Such an architecture is already entering x86-based platforms through the "back door", in the form of graphics chips on Intel Sandy Bridge and Ivy Bridge processors, and has long been a feature of ARM based systems. Lastly, accelerated systems equipped with accelerators such as Intel "MIC" or GPUs will required a hybrid approach for best performance and cost efficiency, where code scales in many dimensions simultaneously.

2.4. Where HEP software will be tomorrow

[5] Hardware threading, as implemented in x86, relies on "whitespace" in the execution of one thread so that another thread can take advantage of the unused resources. Higher execution efficiency of one thread can mean less opportunity for another to use the CPU's resources.

Considering the trends outlined above, it is now possible to make a rudimentary projection of how HEP applications would perform on future hardware in several years' time. Assuming the current course of action, convincing prototypes will not be produced in a short period of time.

In Table 3 below, it is assumed that vector width will grow to 512 bits in the not so distant future. With the arrival of new vector instructions, it is assumed that practical vectorization efficiency will increase by 20% (2.5x over 4 increased to 6x over 8). Since there are very limited efforts to vectorize in HEP, the factor assumed is still 1. ILP improvements in the industry could be expected due to upcoming extended execution units, of which HEP would likely not make use because of bottlenecks located elsewhere. The very approximate improvement factor for the industry is estimated at 10%. Hardware threading could give HEP a 25% benefit if it is switched on – as soon as the memory equation becomes attractive. Whole node scheduling could certainly help in this respect (one out of many) and give non-negligible compute savings. Finally, the mainstream is expected to switch to 4-socket platforms, but there is no clear assurance such would be the strategy for HEP. In this calculation, however, the factor remained at 2 since the change is uncertain and in any case HEP would be able to follow it with the usual delays – provided that would be the economical option. In this respect, it is also possible that the most economical platform will cease to be the one with the most cores per package. Hence the difference of 10 and 12 cores between the "ceiling" and a "typical" value.

Table 3: Expected limits on performance dimension scaling; ceiling, industry, HEP

	SIMD	**ILP**	**HW THREADS**	**CORES**	**SOCKETS**
MAX	8	4	1.35	12	4
INDUSTRY	6	1.57	1.25	10	2
HEP	1	0.80	1.25	8	2

Table 4: Expected limits on performance dimension scaling (multiplied); ceiling, industry, HEP

	SIMD	**ILP**	**HW THREADS**	**CORES**	**SOCKETS**
MAX	8	32	43.2	518.4	2073.6
INDUSTRY	6	9.43	11.79	117.86	235.71
HEP	1	0.8	1	8	16

Between now and the extrapolated future point, there are considerable increases in all cases, but the HEP case still benefits from only a moderate gain which is not accounting for the existing gap. In fact, the gap would widen, as the projected improvements in the ceiling case and the industry case are two times higher than those expected in the case of HEP. These numbers are one more argument for a concerted and focused change of course, which is required to guarantee HEP software sustainability and adequate performance, and which would correspond predictably to computing related expenditures. Planning for such improvements will need to consider not only extrapolations of current values, but also an understanding of the trends and an expectation of change. Above all, planning for scalable parallelism on several planes needs to be an integral element of any development strategy. This applies both to programmers, who for the most part did not yet produce software scalable in multiple performance dimensions, but also to resource planners who have control over purchasing policies and understand very well acquisition opportunities and constraints.

Current core or vector width increases might or might not be sustainable – but an anticipation of such changes and a sense of the variable size of the multiplicative dimensions discussed should already give improved scalability results. Particular tradeoffs related to such decisions are discussed in the section that follows.

Summing up the above, data-centric design does not promise to be the panacea but certainly deserves a careful examination. Through improving data flow HEP can hope to successfully exploit advanced features of upcoming hardware and improve scalability on multiple fronts.

3. Realistic considerations for parallelism in HEP

This section outlines some basic, yet realistic considerations for parallelism in HEP, from the point of view of available parallelization technologies. Fundamental questions address the necessity, feasibility and realistic perspectives of parallelization of a given component or piece of software. Given the looming shifts in hardware, certain tradeoffs might need to be revisited, and any choice will need to account for prospective changes in both hardware and software technologies. A holistic view is instrumental in the effort to achieve good results [12].

Summarizing the hardware dimensions from the point of view of computational efficiency, one can identify three distinct software levels: vectors, potential instruction level parallelism (ILP) and software threads. Vectors map directly onto SIMD hardware, while ILP maps onto microarchitectural usage efficiency. Since any major OS should be nearly transparent while managing software threads, hardware threading, many-core and multi-socket parallelism are all covered by threads native to the operating system. These can be implemented in a variety of ways. Finally, the last layer which is multi-node, can also be handled by a variety of HPC technologies, but is of less relevance in HEP and is not discussed.

There are numerous options for parallelism emerging, and many are particularly interesting because they span more than one dimension of software parallelism. However, few of them have the stability and user base guaranteeing a safe future. On the other hand, well established industry standards often feature robust support and active communities while lacking adaptation to modern hardware – for example work on the OpenMP and MPI standards progress relatively slowly. Despite of this shortcoming, solutions that have seen or are seeing major adoption seem to offer a safer bet than emerging, untested solutions because of their maturity and an implicit assurance of longevity. Another key factor, and often a prominent differentiator, is the parallelism model implemented by a particular parallel framework, library or API. The relationship between the model and the implementation does not have to be unidirectional, especially considering that HEP processing schemes are rich in conceptual parallelism at various levels of granularity. The relationship of the two with the underlying architecture, however, does not leave much room for negotiation and more often it is it that forces certain decisions on the software, rather the other way around.

3.1. Key tradeoffs

Any strategic decision will require a revision of tradeoffs. Programmers will have to be committed to those for considerable amounts of time, as it is today. One fundamental question is whether the physicist programmer should know anything about parallelism – either way the decision will have benefits and drawbacks. While the continuation of the current conceptually single threaded model is tempting, it may prove to be difficult to implement without expending significant effort and may turn out to lack scalability in the further future.

Secondly, there is the established question of how much coding flexibility should be given to the user of a framework, toolkit or API. Unfortunately, support for more flexibility conflicts with expectations of high performance from the software or hardware and there is little promise for a change in that respect. The software needs to do a lot of guessing at runtime which significantly slows down processing, while the hardware is left in a position where no major predictions about code execution can be made and thus severe penalties such as cache misses or resource stalls are introduced. Heavy, indiscriminate use of OS memory, customary in HEP software, incurs even further pressure on the system. A suggestion could be made that there could be cases where this tradeoff could be reassessed more in favor of performance.

The two major issues discussed above, as well as the non-negotiable nature of the hardware and the fact that HEP software is less and less efficient on every lower level of abstraction, prompt doubts as

to how efficient a single person can be in spanning all hardware, software and algorithmic levels of any setup. While it is common to become a domain expert, a software engineer or a hardware specialist, being able to span all three or even two of those areas is a challenging task, except for some rare individuals. Since the HEP community often relies on contributions from top domain experts, their work could be supported through synergistic collaborations with groups more firmly rooted in hardware and software – without sacrificing the rapid development benefits of small workgroups.

3.2. Secondary tradeoffs

Another prominent tradeoff is the choice of the programming language. Nowadays, C++ is considered to be the de facto standard and, as outlined above, this choice has its drawbacks. Some of them could be worked around by exposing less of the language to the user, or by exposing a more rigid interface (e.g. API). Simpler languages like C are usually not robust enough to support a large software project, while dynamic languages such as Java or Python offer even less control over performance than C++, depending heavily on the implementation of the virtual machine.

Secondly, the choice of the hardware architecture underlying the software is best made consciously and with reasonable perspectives for a prolonged, supported lifetime with a wide ecosystem. Similar considerations apply when thinking of software technologies employed – whether they're proprietary or free and whether they're open or closed. Community support and activity have historically been good indicators, provided that peaks of interest in the "hype cycle" (a well-known technological pitfall, Figure 2) are filtered out.

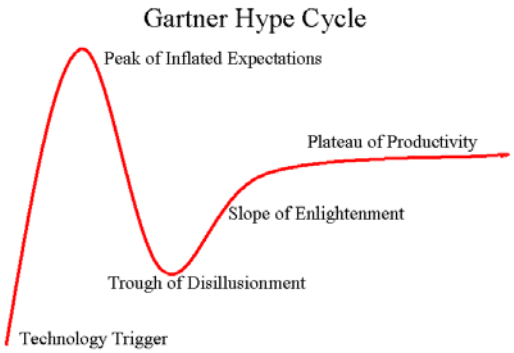

Figure 2: The "Hype Cycle" as defined by Gartner [13]

Third, there is the tradeoff of homogeneity. Homogeneous solutions have their advantages, but they restrict the opportunities for using non-standard techniques, and in particular accelerators. From a software point of view, this particularly applies to the concept of architecture-specific libraries. Companies like Intel produce special versions of their math and vector libraries, specifically optimized for various families of their processors. This allows to extract the best performance on any platform, but at the cost of a loss in homogeneity and complications during development and validation.

Furthermore, considering the rise of accelerators, two side-tradeoffs emerge: that of threaded vs. data parallel code and the related tradeoff of programming kernels vs. control-heavy code.

3.3. Considerations for parallelism in HEP software

This section addresses various software options for parallelism, grouped by the level on which they are relevant.

Starting from the lowest levels again, the responsibility for vectorization has historically been in the hands of the developer. Data parallelism can be expressed through dedicated assembly code or through intrinsics which interface native assembly instructions to C-like calls. Today there are several technologies built into compilers which assist with automatic vectorization – both GCC [14] and the

Intel Compiler [15] support this option. If the code is predictable and repetitive enough, a smart compiler will realize its chance and vectorize it - provided appropriate switches are enabled. This feature is easy to use, but can be hard to control and master. Distinctively, it does not require too much effort from the programmer and can still give good results – speedups of 2x are not unheard of. Furthermore, compiler dependent technologies such as Intel CEAN[6] allow for a more expressive use of data parallelism through specific notation in syntax. There lies the potential to autovectorize efficiently while maintaining good overall programmability, albeit at the price of tying oneself even more closely to the performance of a particular compiler. Finally, vectorization in higher level languages still remains to develop. Languages like Python, Java or even JavaScript have experimental vectorized implementations [16], but expressing data parallelism implicitly through a virtual machine can be quite challenging – and the results will rarely be stellar.

Good instruction level parallelism can be achieved by compiling good quality code with good compilers used with the optimal options. Special optimized math libraries (like the SVML or Intel MKL) can offer optimized execution paths for popular functions – sometimes tweaked down to a single cycle. Since a big portion of HEP software is built on popular toolkits such as ROOT and Geant4, contributions within those packages will be a well-placed investment.

In the case of multi-threading, there are plenty of well-established options. On the low level, there are OpenCL[7], OpenMP[8] and OpenACC[9]. All of those deal primarily with data parallelism through multiple threads, as opposed to just vectors. OpenCL carries the promise of transparently running on a variety of hardware (as did the Intel Array Building Blocks project[10]), but falls short of performance targets unless laborious optimization is applied. In practice, this model has shown to be demanding and restrictive on the programmer [17][18]. OpenMP allows for flexible loop scaling across multiple cores. While the runtime is at times hard to control and could introduce some overhead, it is an established option that can work well with modern code and compilers. Its main appeal is simplicity – one line of code can be sufficient to fully parallelize a loop. This trait can also be a drawback, in particular when one would be looking to enhance the parallel processing model with task processing. Finally, OpenACC promises to bridge the gap between accelerators and traditional computing through OpenMP-like pragma directives. The project is still in its infancy and has yet to demonstrate its value.

Vendor-specific APIs like NVIDIA CUDA[11] and the equivalent offerings from AMD deserve particular attention. Such programming models have been developed to work only with products of these companies, but have proved popular enough to draw the attention of the HPC community. Unfortunately, these models are usually closed, operate on non-standard extensions to languages and – as in the case of OpenCL – often require significant remodeling and adaptations of code into kernels. In the case of HEP, which runs its code on x86, such efforts spent on optimizing x86 compatible code for multi-threading and vectorization would likely pay off more in the long run. It has been shown that mythical 1000x speedups usually are not presented objectively and are produced through unfair comparisons of heavily optimized GPU code and unoptimized, single-threaded CPU code. Such speedups are often reduced to 3x or 5x when compared with a fully optimized workload for a standard processor [17][18][19], and even that comes at a price of significant tradeoffs ranging from code manageability and modularity to IEEE floating point compliance.

On a higher level there are technologies such as the well-established pthreads or Boost threads. These have proven to be performing but rather laborious to use, so the attention of the community has

[6] http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011/compiler_c/optaps/common/optaps_par_cean_prog.htm
[7] http://www.khronos.org/opencl/
[8] http://openmp.org/wp/
[9] http://www.openacc-standard.org/
[10] http://software.intel.com/en-us/articles/intel-array-building-blocks/
[11] http://www.nvidia.com/object/cuda_home_new.html

turned to more robust wrappers, such as Intel Threading Building Blocks[12] (TBB) or Cilk+[13]. The open-source TBB aims to be to parallelism what STL is to serial C++ code by providing containers and convenience functions suitable for parallel programming. It seems to be a choice closest to the current development directions in HEP, as it is rooted firmly in the C++ philosophy albeit with a fix on concurrency. The premise of the MIT-born Cilk is that the developer specifies explicitly which portions of the program are safe to be parallelized. The standard has yet to mature, in particular where it interfaces with C. Overall, a higher level toolkit, library or API for parallelism can reduce the programming workload but can also leave its mark on performance.

The notable technologies enumerated above can also be combined to produce hybrid setups, covering more (or all) levels of software parallelism. In particular, the combination of auto-vectorization, OpenMP and MPI has shown to be noteworthy, but is still based on C traditions [20].

In practice, not all available technologies or options show to be what they promise [18]. It is possible to uncover unpleasant surprises over the course of full scale tests with real world data and real world processing tasks – performance drops of 2x from the reference implementation are not uncommon. The parallel runtime usually acts as a middle-man and has its own set of issues which need to be understood. Careful, impartial and practical experiments of scale have shown to be very useful in establishing the worthiness of a particular solution, or its superiority over another.

In essence, we recommend the following three characteristics as rough guidelines for metrics when assessing parallelization technologies in a well-designed and understood test:

- Balance of performance and programmability
- Maturity and longevity
- Openness and community support

3.4. Closing Considerations

One of the final considerations is how universal can a parallel framework be. Already choosing a particular parallelization technology (and, implicitly, a parallel model) is restricting the field within which future decisions can be taken. Local requirements confronted with a global direction always play a critical role in the adoption of rejection of that solution. Secondly, the price to pay for global, universal usefulness could be high and could eclipse potential gains on other fronts. Third, devising a universal group of requirements for major HEP experiments would prove quite problematic even today – which would be even more difficult if taking into account the partially specified requirements of future experiments and upgrades.

Finally, there is the question of timescales. Several long experiments, in which openlab was involved, have shown that proper parallelization and optimization take significant amounts of time, even for very dedicated individuals. Such were the cases of ROOFIT [17][18][20] and the multi-threaded Geant4 prototype [21], for example. While on the way, the hardware changes, the software being parallelized changes as well, as does the toolkit or technology used to implement parallelism. The initial phase of re-modeling or re-programming is followed by a lengthy (but necessary) phase of optimization and verification, regardless of whether the target is a major processor or an accelerator. For representative pieces of software, each of those phases will take well over a year. Since the complexity of such operations is usually quite high, it has proven to work best when prototypes were actually useful and meaningful foundations for the software that they emulate.

4. Further recommendations and conclusions

Overall, the questions of parallelism have shifted from “whether to take advantage of it?” to “how to take advantage of it?” and “who would do it?”. In HEP, this shift translates to an assignment of responsibility for understanding the hardware and the opportunities that come with it. An engagement

[12] http://threadingbuildingblocks.org/

[13] http://software.intel.com/en-us/articles/intel-cilk-plus/

with external parties (including software authors and hardware vendors) can be beneficial for all sides on various fronts.

Whether many-core is "the" revolution or there is something else waiting is unclear. It is hard to plan for unknowns, but easier to plan for changes. Therefore, expecting a particular number of cores at a particular point in time might be both difficult and potentially dangerous. Planning for this number (and others) to change, however, does not introduce such peculiar difficulties and yet enables future scalability.

Given the arguments presented above, in view of the widening gap between the hardware and the software, and in view of future developments in HEP, we recommend to introduce or maintain the implementation of the following:

- introduce a systematized R&D program focused on parallelism with deliverables
- restate the real needs of the HEP community starting with a tabula rasa
- setting clear, realistic, contextualized goals for development in line with the real needs of the HEP community
- devising better metrics for performance and taxing for violations
- implementing a scalability process focused on rapid response
- promoting joint work across stakeholders to share the load
- a careful embrace of emerging technology
- a conscious consideration of where any performance gains should be reinvested (e.g. reduced cost, improved raw performance, accuracy etc.)

Balancing the tradeoffs mentioned earlier would give some cost estimates to each of the items above. A universal solution to upcoming computing problems in HEP might not be possible, but a universal direction in the effort to reach common goals certainly is.

References

[1] Moore G 1975 Progress in digital integrated electronics (IEEE)
[2] Held J, Bautista J and Koehl S 2006 *From a Few Cores to Many: A Tera-scale Computing Research Overview* (Intel, online)
[3] Jarp S 2008 Hunting for Performance in 7 dimensions *CERN openlab Summer Student Lectures* (CERN)
[4] Lomont C 2011 *Introduction to Intel Advanced Vector Extensions* (Intel, online)
[5] Abrash M 2009 *A First Look at the Larrabee New Instructions (LRBni)* (Dr. Dobbs, online)
[6] Levinthal D 2010 *Performance Analysis and SW optimization lab for CERN* (CERN)
[7] Jarp S, Lazzaro A, Leduc J and Nowak A 2010 *Evaluating the Scalability of HEP Software and Multi-core Hardware* (CHEP 2010)
[8] Intel 2011 *Intel VTune Amplifier XE* (Intel, online)
[9] Levinthal D 2011 *Generic Cycle Accounting GOODA* (CScADS 2011)
[10] Nowak A 2011 *Understanding Performance Tuning* (CERN openlab)
[11] Intel 2012 *Intel ARK* (Intel, online)
[12] Jarp S et al 2010 *How to harness the performance potential of current Multi-Core CPUs and GPUs* (CHEP 2010)
[13] Gartner Inc *Hype Cycle Research Methodology* (Gartner, online)
[14] Hauth T 2012 *Writing Autovectorizable Code* (CERN, online)
[15] Bik A 2004 *The Software Vectorization Handbook* (Intel Press, ISBN 0974364924)
[16] Intel 2011 *River Trail prototype at github* (Intel, online)
[17] Lazzaro A and Pantaleo F 2010 *Maximum Likelihood Fits on GPUs* (CHEP 2010)
[18] Lazzaro A and Sneen Lindal Y 2011 *Evaluation of likelihood functions on CPU and GPU devices* (ACAT 2011)

[19] Lee V et al 2010 *Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU* (ISCA 2010)
[20] Lazzaro A et al 2012 *Hybrid parallelization of maximum likelihood fitting with MPI an OpenMP* (CERN openlab)
[21] Apostolakis J, Cooperman G and Dong X 2010 *Multithreaded Geant4: semi-automatic transformation into scalable thread-parallel software* (EUROPAR 2010)