Comparison of Software Technologies for Vectorization and Parallelization

Sverre Jarp, Alfio Lazzaro, Andrzej Nowak, Liviu Valsan
CERN openlab, September 2012 – version 1.0
White-paper as part of the collaboration between CERN openlab and Intel SSG

Executive Summary

This paper demonstrates how modern software development methodologies can be used to give an existing sequential application a considerable performance speed-up on modern x86 server systems. Whereas, in the past, speed-up was directly linked to the increase in clock frequency when moving to a more modern system, current x86 servers present a plethora of "performance dimensions" that need to be harnessed with great care. The application we used is a real-life data analysis example in C++ analyzing High Energy Physics data. The key software methods used are OpenMP, Intel Threading Building Blocks (TBB), Intel Cilk Plus, and the auto-vectorization capability of the Intel compiler (Composer XE). Somewhat surprisingly, the Message Passing Interface (MPI) is successfully added, although our focus is on single-node rather than multi-node performance optimization. The paper underlines the importance of algorithmic redesign in order to optimize each performance dimension and links this to close control of the memory layout in a thread-safe environment. The data fitting algorithm at the heart of the application is very floating-point intensive so the paper also discusses how to ensure optimal performance of mathematical functions (in our case, the exponential function) as well as numerical correctness and reproducibility. The test runs on single-, dual-, and quad-socket servers show first of all that vectorization of the algorithm (with either auto-vectorization by the compiler or the use of Intel Cilk Plus Array Notation) gives more than a factor 2 in speed-up when the data layout in memory is properly optimized. Using coarse-grained parallelism all three approaches (OpenMP, Cilk Plus, and TBB) showed good parallel speed-up on the available CPU cores. The best one was obtained with OpenMP, but by combining Cilk Plus and TBB with MPI in order to tie processes to sockets, these two software methods nicely closed the gap and TBB came out with a slight advantage in the end. Overall, we conclude that the best implementation in terms of both ease of implementation and the resulting performance is a combination of the Intel Cilk Plus Array Notation for vectorization and a hybrid TBB and MPI approach for parallelization.

Table of Contents

Introduction

In this report we present an evaluation of several technologies used to vectorize and parallelize a maximum likelihood (ML) data analysis application [COW98]. The code has been developed by CERN openlab, and represents a prototype of the RooFit package (which is part of the ROOT software framework developed at CERN), generally used in the high energy physics (HEP) community for data analysis [ROF06].

If in the past the increase in computing performance was mainly driven by the increment of the execution units' frequency, nowadays microprocessor vendors are rather deploying transistors in making more complex units, but with limits on their frequency and the consequent power consumption. From the programmer's point of view there are two main hardware areas that are rapidly expanding: the vector register dimensions and the number of computational cores that can execute a common application. The former allows single-instruction multi-data (SIMD) executions by using *vectorization*-programming techniques; for example, with the AVX instruction set extension it is possible to compute four double-precision *vector* operations at the same execution cost as a single double-precision *scalar* operation. The number of computational cores can be considered either for the increment of cores available in a single node, *i.e.* a multi-core *shared memory* system, or for the increment of such nodes connected together within a cluster, *i.e.* a *distributed memory* system. In both cases the execution of the applications has to properly spread across the many cores by means of *parallelization*-programming techniques. These hardware considerations are valid for the conventional CPUs and for the more recent computational devices, the so-called accelerators, mainly used to offload the CPUs for intensive floating-point applications. The accelerators, such as Graphics Processing Units and Intel Many Integrated Cores (MIC), present higher number of cores (many-core) and wider vector registers with respect to CPUs for fine-grained application parallelism.

It is worth underlining that it is becoming more and more vital to design the software applications and to program taking into account the new hardware complexity in order to reach the peak performance of the systems. Several technologies are already at the disposal of the software programmers. The challenge is to introduce such technologies in existing software applications with the minimum number of changes, maximizing the performance on a large variety of systems. For this reason it is useful to compare the technologies in terms of changes required for accommodating them in existing applications and to analyze the resulting performance.

In this report we have primarily focused on the comparison of some of the vectorization and parallelization technologies in terms of programming feasibility, allowing us to improve our ML application with limited effort. We show the advantages/disadvantages that we have found for each technology. We have also looked at the performance on several x86-64 based systems commonly used by the HEP community. For vectorization we have considered the loop auto-vectorization

feature in the Intel compiler and the Intel Cilk Plus Array Notation. For parallelization in a shared memory system we have considered OpenMP, Intel Threading Building Blocks (TBB), and Intel Cilk Plus. Hybrid parallelization by using the aforementioned technologies together with Message Passing Interface (MPI) has been implemented to allow running on distributed memory systems. However, in this report we do not show results of tests of parallel execution on multiple nodes, since we have concentrated our efforts on the single node execution. Further details on the vectorization and parallelization technologies can be found in the respective documentation available on the Internet.

The report is structured as follows: a brief description of the hardware complexity, a description of the ML fit application and the strategies adopted to vectorize and parallelize it; then we describe in detail the comparisons of the technologies and finally we give the performance results when running on several x86-64 platforms.

The "seven performance dimensions" of PC servers

The complexity of microprocessors can be described in what we call the "seven performance dimensions" that are available in modern computing system designs [JAR11]. The dimensions are illustrated in Figure 1. They can split in two categories: three dimensions, namely *pipelining*, *superscalar*, and *SIMD/Vector*, together with *symmetric multithreading* (SMT) are related to performance of a single computational core unit (*intra-core* dimensions); *multi-core*, *multi-socket*, and *multiple nodes* are related to *extra-core* performance. We focus here on the x86-64 architecture, but complexity is equally present in processors based, for instance, on the SPARC architecture from Oracle, the POWER architecture from IBM, or the Itanium architecture from Intel.

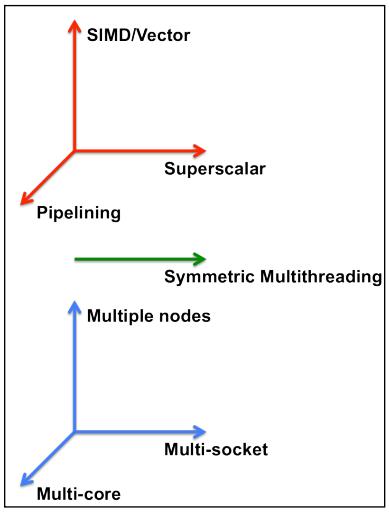

Figure 1 The seven dimensions of performance in a modern microprocessor.

In the Pentium days, from an execution point of view, there were basically only two major performance dimensions: pipelining (with frequency increases every 6 – 12 months) and to a very limited extent, superscalar execution (the so-called "U and V pipes"). The latest x86-64 systems however have six superscalar execution ports, allowing up to four independent instructions on average to be concurrently executed. As far as the programmer is concerned, superscalar execution and pipelining cannot be addressed directly. Usually the compilers and the CPU execution unit will organize the instructions in order to maximize the number of ports involved. Historically PCs with more than one Pentium processor on the motherboard were rare, so, when people needed more performance than what was available in a single system, they would split the load across multiple systems (multiple nodes dimension). Today clusters can reach several thousands of nodes, inter-connected by fast network links. The *de*

facto software standard communally used for managing the parallel execution of the applications in such clusters is MPI.

Another performance dimension became more commonplace with the introduction of the Pentium Pro processor, namely the multiple-socket dimension. Currently, dual-socket and quad-socket systems are commonly available in computer centers. Nowadays, these systems follow a non-uniform memory access (NUMA) pattern when accessing memory, *i.e.* the CPU of a given socket can access directly only a portion of the total memory and indirectly, through a request to the corresponding CPUs, the rest of the memory. The direct and indirect memory accesses have different latencies. Thereby programmers must pay attention to memory allocations and memory access patterns.

In 2001 we saw the introduction of SIMD vectors in the second wave of Streaming SIMD Extensions (SSE2). In a 128-bit register, one could now perform two 64-bit (double-precision) operations or four 32-bit (single-precision) operations in parallel. Thus another performance dimension was born. In 2011 Intel introduced a new extension in its processors, Advanced Vector Extensions (AVX), which doubles the dimension of the registers. Furthermore, the Intel MIC platforms use 512-bit registers and there is a high possibility of these registers becoming available in the server CPUs space. Therefore, x86-64 CPUs can be considered as truly vector computational-capable systems. Programmers can exploit vectorization directly, using intrinsic operations for example, or indirectly by using the auto-vectorization feature offered by the compilers. In both cases a correct organization of the data in vectors is compulsory.

In 2004 we were blessed with the first multi-core processors. The cores (two or more cores on a die) were complete processing units with an entire set of execution logic, their own instruction and data caches, and so on. Modern x86-64 processors can contain up to 16 cores, although between 4 and 8 cores are normally available in computer center server processors. SMT is just a “pseudo-dimension” that was first introduced on the x86-64 architecture one year prior to multi-core, with the availability of the Intel Pentium D. It differs from multi-core since it does not provide more execution logic on the processor die; it simply allows two (or more) hardware threads to compete for the available logic and caches. Modern x86-64 CPUs have the possibility to execute two hardware threads and the speed-up depends on the applications (usually in the range between 1.2x and 1.3x). Considering dual-socket systems with 6 cores CPUs and SMT enabled, these systems can run up to 24 parallel hardware threads with shared memory. Programmers have to use parallelization technologies for designing and implementing their applications, with the consequent increase in complexity of dealing with parallel executions with respect to sequential executions (for example, due to false sharing and race conditions). Beside the aforementioned NUMA effect, it is important to consider the correct usage of the CPU cache memories, and in particular the last level cache (LLC) memory which in most cases is shared between the cores within the CPU. It can be useful to pin the threads to the cores by using affinity mask settings to avoid swapping of threads between the cores.

We can now conclude that five dimensions can be directly of interest to the programmers by using vectorization (SIMD/Vector) and parallelization (SMT, multi-core, multi-socket, and multiple nodes) technologies. We underline the fact that the dimensions are multiplicative, but that a lot of existing software was not designed to take advantage of all of them, especially the ones that naturally relate to data-level parallelism. As already mentioned, current CPUs are potentially able to execute four SIMD operations in double precision (using AVX instructions). This gives a speed-up of a factor 4x on the execution of the applications. Of course, the programmers must design their applications to exploit these vector operations. On the other hand, rapid advancements in multicore and multi-threading technologies open new challenges and it is more apparent than ever that the future of efficient computing lies in the effective utilization of parallel and many-core architectures.

Description of the ML application

The HEP community makes large use of many complex data analysis techniques, like maximum likelihood fits, neural networks, and boosted decision trees. These techniques are largely used in HEP experiments to analyze the collected data. Data samples are usually a collection of N independent *events*, an event being the measurement of a set of n *observables* $\hat{x} = (x^1 \dots x^n)$ (energies, masses, spatial and angular variables…) recorded in a brief span of time by particle physics detectors. The events can be classified in S different *species*. Each observable x^v is distributed for the given species j with a probability distribution function (PDF) $\mathcal{P}_j^v(x^v; \hat{\theta}_j^v)$ where $\hat{\theta}_j^v$ are parameters of the PDF that can be related to the prediction obtained from physics models. If the observables are uncorrelated, then the total PDF for the species j is expressed by

$$\mathcal{P}_j(\hat{x}; \hat{\theta}_j) = \prod_{v=1}^{n} \mathcal{P}_j^v(x^v; \hat{\theta}_j^v).$$

The PDFs are normalized over their observables, as function of their parameters, which implies an analytical or numerical evaluation of their integral. The *extended likelihood* function is

$$\mathcal{L} = \frac{e^{-\sum_{j=1}^{S} n_j}}{N!} \prod_{i=1}^{N} \sum_{j=1}^{S} n_j \mathcal{P}_j(\hat{x}_i; \hat{\theta}_j),$$

where n_j are the number of events belonging to each species. The maximization of this function (the maximum likelihood technique) over the given data sample allows to estimate the values and errors of the free parameters of the maximization. It is usual to minimize the equivalent function $-\ln(\mathcal{L})$, the negative log-likelihood (NLL)

$$NLL = \sum\nolimits_{j=1}^{S} n_j - \sum_{i=1}^{N} \left(\ln \sum_{j=1}^{S} n_j \mathcal{P}_j(\hat{x}_i; \hat{\theta}_j) \right),$$

that is a sum of logarithms. The terms of the sum can be graphically visualized as a tree, where the leaves are the PDFs $\mathcal{P}_j^v(x^v;\hat{\theta}_j^v)$, which are then linked to the corresponding product PDFs $\mathcal{P}_j(\hat{x}_i;\hat{\theta}_j)$, and finally to the root that is $\sum_{j=1}^{S} n_j \mathcal{P}_j(\hat{x}_i;\hat{\theta}_j)$ (sum PDF). Product and sum PDFs are denoted as composite PDFs. Therefore, the root has S child nodes, each with n children, which means that in the tree there are $S\times(n+1)+1$ nodes in total. The evaluation of the term in the sum of logarithms consists in traversing the entire tree, first evaluating the leaves and then moving up to the root. A final reduction of the logarithm results is performed and then combined with $\sum_{j=1}^{S} n_j$ to get the NLL value. A sketch of the tree is shown in Figure 2.

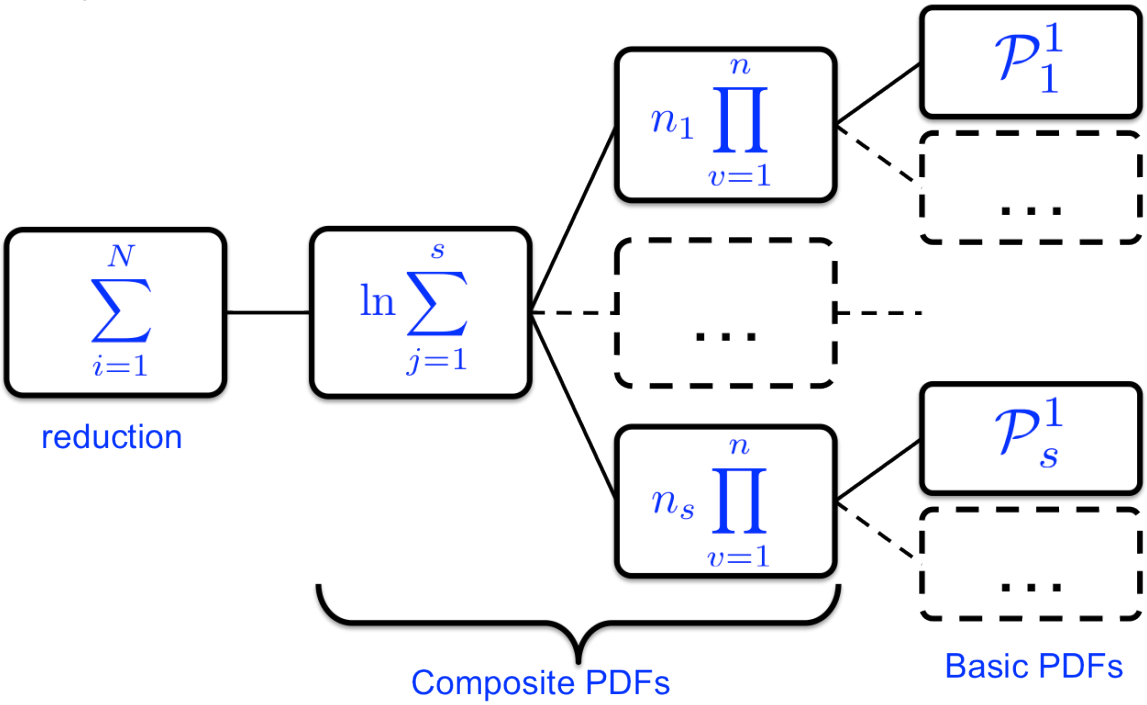

Figure 2 Sketch of the NLL evaluation tree.

The time spent for the NLL evaluation depends on the number of events and the complexity of the PDFs. The search for the minimum for NLL can be carried out numerically [MIN72]. The whole procedure of minimization requires several evaluations of the NLL, which themselves require the calculation of the corresponding PDFs for each observable and each event of the data sample. Hence, it becomes important to speed-up the evaluation to have fast data analyses.

The common software used in HEP community for the evaluation of the NLL is RooFit, which is part of the general data analysis framework ROOT. The code is implemented in C++ and all floating-point operations are performed in double precision. Currently RooFit implements an algorithm for the NLL evaluation that is based on a single *external* loop over the events, where each iteration evaluates the entire tree for a given event and then accumulates the values for the reduction operation. The values of the observables are organized in memory as arrays (an array for each observable), each array composed of N double precision numbers that are read-only during the process of minimization. Each PDF has a common interface with an overloaded

virtual method that returns the value of the PDF for the specific values of the observables. Given this algorithm design, the implementation cannot take full advantage of data vectorization and other code optimizations (like function inlining). The parallelization is implemented by splitting the loop over the events in different concurrent blocks, with a reduction operation to get the final sum. It is based on *fork* calls, without shared memory objects to avoid race conditions. Therefore, the side effect of the implementation of this algorithm is that there is a proportional increase of the memory footprint with respect to the number of parallel threads.

CERN openlab has developed a prototype that makes use of a redesigned and optimized algorithm with respect to the algorithm used in RooFit. For each NLL evaluation the tree is traversed sequentially only once. The algorithm starts evaluating the basic PDFs, belonging to a given product PDF, looping over all values of their observables, and storing the results in arrays (an array for each PDF). Then it does the evaluation of the corresponding product PDF, looping and combining the arrays of results of the daughter PDFs in a new array. It repeats the procedure for all product PDFs. After that it loops again and combines the arrays of results of the product PDFs to get a new array of results for the sum PDF (final results). So in total there are $S\times(n+1)+1$ loops, *i.e.* a loop for each node of the tree, instead of the single external loop of the original RooFit algorithm. Eventually, the algorithm calculates the logarithm of the final results and their sum (reduction). The main change to the implementation is that now the virtual method for the evaluation of each PDF is called only once per NLL evaluation and it runs internally the loop over all events, which is implemented as a `for` loop. It returns the corresponding array of results, each one composed by N double precision numbers. These arrays are allocated in memory at the beginning of the minimization process and deallocated at the end. It is important to note that the N loop iterations are independent. The loops access consecutive elements of the arrays of observables and results, allowing coalescing of memory accesses and data vectorization, with a significant speed-up in the execution. The drawback of this new algorithm with respect to the original RooFit is that it has to manage several arrays of results, so a correct use of cache memories becomes important.

Vectorization approaches

The Intel compiler was able to auto-vectorize the loops in the original application after we have added the `#pragma ivdep` directive before each loop. Note that the functions inside the loops use transcendental operations. The Intel compiler uses the Short Vector Math Library (SVML) library to vectorize in this case. As an alternative to `#pragma ivdep` directive, we can use the Intel Cilk Plus Array Notation for replacing the loops, which gives a very clear way to express loop vectorization.

Parallelization approaches

The easiest way to introduce parallelization is at level of the $S\times(n+1)+1$ `for` loops. We have used two different technologies:

1. OpenMP, via the `#pragma omp parallel for` directive applied before the `for` loop.
2. Intel Cilk Plus, via the `_Cilk_for` keyword which replaces the `for` keyword.

This solution is defined as *fine-grained* parallelism. Since it requires explicitly `for` loops, it cannot use the vector syntax based on Intel Cilk Plus Array Notation. The arrays of data and results are shared among the threads, so that there is a negligible increment in the memory footprint of the application when running in parallel. Furthermore, race conditions can easily be avoided since the parallel regions are confined to the loop iterations. Also the loop that computes the reduction has been parallelized. The reproducibility of the results of the reduction is compulsory for achieving a stable behavior during the minimization procedure, *i.e.* stable results of the ML fits. For this purpose a specific algorithm was implemented. It preserves the order of the operations for a given number of threads and it reduces the rounding problem due to non-associative floating point arithmetic, using the double-double compensation algorithm 2Sum [SUM01]. To use this algorithm a new type was defined, which overloads the sum operator. In the case of OpenMP it is not possible to use the `reduction` clause because it does not allow the use of custom type variables. Therefore we implemented our own parallel block-wise reduction algorithm. Intel Cilk Plus provides a special template class (`cilk::reducer_opadd<>`) for the reduction that also works with custom types and gives reproducibility results. Consequently, the Intel Cilk Plus implementation becomes easier than in OpenMP.

The scheduling of iterations in the OpenMP parallelization is statically partitioned for all loops, *i.e.* each thread executes a fixed number of iterations. The static partitioning is implemented in such a way that one thread can have maximum one iteration of difference with respect to the other threads, to ensure an equally balanced workload. The same technique is applied to the loop that performs the reduction operation. In the case of Intel Cilk Plus the dynamic scheduler is used for all loops, leaving for the runtime system to decide the grain size of the loops. Both OpenMP and Intel Cilk Plus allow running the application when removing completely any parallel-related code for scalability tests by either setting the environment variables (`OMP_NUM_THREADS` and `CILK_NWORKERS`, respectively) or the compiler flag `-cilk-serialize` in the case of Intel Cilk Plus.

Although *fine-grained* parallelism has the big advantage of easy implementation in a thread-unsafe application, implementations show some limitations that reduce the overall performance:

- For each NLL evaluation, $S\times(n+1)+1$ independent parallel regions have to be considered. This leads to a larger overhead than necessary, which drastically reduces the scalability.
- $S\times(n+1)+1$ arrays of results and n arrays of observables have to be managed, each array composed of N double precision values. The amount of data to manage becomes considerable in the case of complex models and large data samples, so it is crucial to have an optimal organization of data inside the cache memories. Tests have proved that there is a significant penalty to the scalability due to LLC load misses. An analysis of the problem shows that

the culprits are the loops of the composite PDFs, which have to combine arrays of results with just a simple operation.

To remove the potential overhead due to fine-grained parallelism, the entire NLL evaluation was redesigned using a different pattern: there is only one parallel region for each evaluation, and this region will start at the root of the tree. This solution is defined as *coarse-grained* parallelism. In the case of the OpenMP implementation, the parallelization starts at the root level via a `#pragma omp parallel` directive. The partitioning of the loop iterations is done as before, but now each thread executes the entire evaluation from the root to the leaves, including the reduction, within its own partition only. The Intel Cilk Plus implementation based on `_Cilk_for` keyword cannot be accommodated for the coarse-grained parallelism. Therefore, we have implemented a new algorithm based on `_Cilk_spawn` and `_Cilk_sync` keywords. The algorithm splits the events in blocks, each block being executed in parallel by a `_Cilk_spawn` call. The user decides the block dimension B. Then this value is used to determine the number of blocks $n_B = N/B$, where B is adjusted so that the minimum value of n_B is equal to the number of available Intel Cilk Plus workers (which is also the default behavior when the user does not provide a block dimension). The NLL evaluation is executed for the events in the blocks, including the reduction. Each block is dynamically executed by an Intel Cilk Plus task. A sketch of the necessary code is the following:

```
int nBlocks = (userBlockDim==0) ? __cilkrts_get_nworkers() :
              std::max(__cilkrts_get_nworkers(),
                       int(double(nEvents)/userBlockDim+0.5));

int blockDim = nEvents/nBlocks;

// Result of the reduction
cilk::reducer_opadd<ValueAndError_t> result;

for (int iBlock = 0; iBlock<nBlocks-1; iBlock++) {
  // RunNLL runs the NLL evaluation for the
  // events with indices [iBlock*blockDim, (iBlock+1)*blockDim[ and
  // it accumulates the result of the reduction
  _Cilk_spawn RunNLL(iBlock*blockDim, blockDim, result);
}

// Take care of the remaining events
RunNLL((nBlocks-1)*blockDim, nEvents-(nBlocks-1)*blockDim, result);

_Cilk_sync;
```

This algorithm also reduces the load on memory by splitting the data domain into blocks so that the entire procedure of evaluation is done block by block (block splitting optimization). The optimization directly targets cache misses, since it increases locality and thereby increases cache efficiency. For this reason it was also added to the OpenMP implementation. In this case the procedure of decomposition applies to the events executed by each thread following the following order of execution: start of the parallel region, decomposition of the events for the threads,

each thread splits the execution of its events into blocks, static execution of the block by each thread. Clearly the application will benefit from systems with a bigger LLC size. In the case of OpenMP it is also beneficial to use a scattered affinity topology that maximizes the cache memory available per thread, *i.e.* threads are bound to cores of CPUs on different sockets before filling the cores of a given CPU. For example, running with 4 threads on the dual-socket systems means 2 threads per CPU (instead of 4 threads on the same CPU). We set the affinity mask by using the Intel environment variable `KMP_AFFINITY`.

We have also implemented an Intel TBB version of the algorithm for the coarse-grained parallelism. A sketch of the implemented code is the following:

```
struct RunTBB {
  ValueAndError_t result; // local result of the reduction
  NLL&            nll;    // internal pointer to the NLL object

  RunTBB(NLL& _nll) : nll(_nll), result(0) { }
  RunTBB(RunTBB &other, tbb::split) : nll(other.nll), result(0) { }

  void operator()(const tbb::blocked_range<size_t>& range) {

    // RunNLL runs the NLL evaluation for the
    // events with indices [range.begin(), range.end()[ and
    // it accumulates the result of the reduction
    nll.RunNLL(range.begin(), range.size(), result);
  }

  void join(RunTBB& other) {
    result += other.result;
  }
};

// Call to TBB parallelization
tbb::parallel_deterministic_reduce(TBBRange(0, nEvents, blockDim),
                                   nll);
```

The beauty of this parallelization technology is that Intel TBB provides automatic block splitting. Each block is dynamically executed by an Intel TBB task and there is a deterministic reduction (like in Intel Cilk Plus). The Intel TBB implementation is very concise.

Although coarse-grained parallelism improves performance in terms of parallel scalability, it can have problematic consequences in the case of a complex C++ code like RooFit. Indeed, the parallel region covers a larger portion of the execution, so it is crucial not to modify member variables of the object the method is running on, or global variables, without carefully assuring that race conditions are avoided.

The last implementation includes the possibility to simultaneously exploit MPI together with the other parallelization technologies. Each MPI process holds a copy of the whole input dataset. Then two decompositions of the events are considered:

1. The same algorithm of decomposition of data elements described for OpenMP threads is applied for the MPI processes, *i.e.* each process manages a fixed number of events during the minimization process.
2. The events belonging to each MPI process are partitioned and analyzed following the specific technology OpenMP, Intel Cilk Plus or Intel TBB.

Note that each MPI process allocates the results array for its corresponding number of events. Also, the reduce operation is performed in two steps. The first step consists of performing the reduction for each MPI process as already described in the non-MPI implementation. In this way each MPI process holds a partial result of the reduction. The second step consists of broadcasting all partial results to all MPI processes, so that each MPI process will have all partial results. The MPI function `Allgather` is used for this operation. Then a second reduction is executed on the MPI partial results to get the final results on all MPI processes. Note that the same algorithm for the reduction based on double-double compensation algorithm is used for the reduction of the MPI partial results. Then, after each NLL evaluation, all MPI processes will proceed to execute the same part of code for the minimization, so that at the very end of the application each MPI process will have the same final results. This implementation choice allows limiting the number of MPI communications with respect to a configuration where only an MPI process drives the evaluation since it does not require the exchange of any other values in the remaining part of the application, *e.g.* the values of the parameters of the PDFs during the minimization. A check after each NLL evaluation is executed to ensure that there were no errors during the evaluation. Each process sends an integer value that can be the number of analyzed events or zero in case of an error. Then the MPI function `Allreduce` is called to sum up all integers and the result is compared with the total number of events. The application stops if the comparison fails. It must be underlined that the `Allgather` and `Allreduce` calls are the only communication functions for each evaluation of the likelihood function, with a small number of results to be moved between MPI processes. Hence, a negligible overhead due to MPI communications is expected.

To conclude, we have 5 implementations that can be classified as follows:

- Fine-grained parallelism (w/o block splitting, vectorization based on `#pragma ivdep` only)
  - OpenMP
  - Intel Cilk Plus
- Coarse-grained parallelism (w/ block splitting, vectorization based either on `#pragma ivdep` or Intel Cilk Plus Array Notation)
  - OpenMP (static scheduling of the blocks)
  - Intel Cilk Plus (dynamic scheduling of the blocks)
  - Intel TBB (dynamic scheduling of the blocks)

All these implementations can be executed with MPI. We have made a single executable with some command line parameter options for setting the implementation and block size to use:

- `-e`: use coarse-grained parallelism (by default is fine-grained parallelism).

- `-a <int>`: specifies the technology to use. Values are: 0 for OpenMP, 1 for Intel TBB, 2 for Intel Cilk Plus.
- `-n <int>`: set the number of events. The events are read from an external file.
- `-b <int>`: set the block size.

The results are reproducible, independently of the number of threads/tasks and the implementation used. Therefore it is possible to compare their execution performance.

Tests

Benchmark configuration

The likelihood function definition was taken from the data analysis performed at the *BaBar* experiment [BBR09]. Thus, this is an example of a real-world application in use by the HEP community. There are 3 observables and 5 species. In total there are 21 PDFs: 7 Gaussians, 5 polynomials, 3 Argus functions, combined by 5 PDFs for multiplication and one for addition. All PDFs have an analytical integral. Input data was composed of 1,000,000 entries per 3 observables, for a total of about 23MB. Results are stored in 21 arrays of 1,000,000 values, *i.e.* about 160MB. When the events were organized in blocks, a heuristic approach was followed to decide their dimensions, which depends on the number of parallel threads. They can be put in relation with the cache size available on the system, so the dimensions decrease accordingly with the number of threads. In particular the block dimensions allow having the fastest execution for a given execution. The MINUIT2 package was used to minimize the NLL function [MIN72]. The number of necessary NLL evaluations to find the minimum for the set of initial parameters used in our tests was 5890. It is worth underlining that the application is floating-point intensive; in particular the execution of the exponential function takes about 60% of the total execution time.

The execution times reported are the average over three consecutive runs of the application. The reference sequential execution time was taken when running the OpenMP implementation with coarse-grained parallelism and a single thread (`OMP_NUM_THREADS=1`). We found that the best block size for this configuration is 10,000 events for all systems used. First of all we compared the reference sequential execution times obtained, respectively, with the times obtained after running the AVX vectorization version based on the loop auto-vectorization feature of the Intel compiler and the Intel Cilk Plus Array Notation. Then we looked at the speed-up given by vectorization with respect to a non-vectorized version of the code. We also compared the sequential execution times of several different OpenMP and Intel Cilk Plus implementations, with vectorization based on the loop auto-vectorization feature of the Intel compiler and AVX instructions. Finally we computed the scalability results where the speed-up values were obtained from the ratio between the reference sequential execution time and the corresponding parallel execution times with a given number of threads, both with vectorization based on the loop auto-vectorization feature of the Intel compiler and AVX instructions. The fraction of

sequential execution time that corresponds to code that can be parallelized is 99.75%. Note that the Intel Cilk Plus and Intel TBB implementations manage automatically the maximum number of threads that can run on the systems. We compared their performance with the OpenMP implementations executed with the numbers of software threads matching the number of hardware threads. Consequently, the block sizes used in Intel Cilk Plus and Intel TBB implementations are the same used in the OpenMP implementations executed with the maximum number of threads.

Technical setup

Three systems were used in the tests with the following configurations:

A. Single-socket system
   - Intel Ivy Bridge Xeon E3-1265L @ 2.5GHz CPU
   - 4 cores with SMT-enabled: 8 hardware threads
   - 8MB L3 cache size
   - 8GB RAM DDR3 1333MHz

B. Dual-socket system
   - Intel Sandy Bridge Xeon E5-2680 @ 2.7GHz CPU
   - 2x8 cores with SMT-enabled: 32 hardware threads
   - 20MB L3 cache size per CPU
   - 64GB RAM DDR3 1333MHz

C. Quad-socket system
   - Intel Sandy Bridge E5-4650 @ 2.7GHz CPU
   - 4x8 cores with SMT-enabled: 64 hardware threads
   - 20MB L3 cache size per CPU
   - 256GB RAM DDR3 1333MHz

All systems are able to execute SSE and AVX vector instructions. They were SMT-enabled, which means that the hardware threading feature was activated and used during the tests. When executing the OpenMP implementation an affinity mask setting was used for pinning threads to physical cores. If there were no more physical cores available, the jobs were pinned to hardware threads, requiring 2 threads per CPU core. Turbo mode was disabled in all systems because we were interested in scalability tests.

The systems were running 64-bit Scientific Linux CERN 6.2 (SLC6), based on Red Hat Enterprise Linux 6 (Server). The default SLC 6 Linux kernel (version 2.6.32-220.1.el6) was used for all the measurements. The code was compiled with ICC v13.0.0 (20120316), using the following standard flags:

```
-O2 –m64 -fPIC -funroll-loops -finline-functions -ip -vec-report1 –tbb -w1 -Wall -openmp -openmp-report2
```

We have used Intel MPI v4.0.3.

Performance results

The results are presented in corresponding sections per each system. The vectorization tests and the comparison of the sequential executions were executed only for the single-socket system.

Single-socket Ivy Bridge system

The reference sequential execution time was 1888 seconds when running a non-vectorized version of the code, obtained by using the `-no-vec` compiler flag. The reference sequential execution of the application with the AVX vectorization based on the loop auto-vectorization feature of the Intel compiler was 1.8% (standard deviation is 0.2%) faster than the corresponding execution of the code vectorized with the Intel Cilk Plus Array Notation.

We compiled the code in three different configurations by using the flags:

1. `-no-vec`
2. `-msse3`
3. `-mavx`

The average speed-up results obtained through a comparison of the reference sequential execution times of such configurations are shown in Table 1. We found that the speed-up results did not significantly depend on vectorization technology and the number of threads (standard deviation is 0.03x). A description of the vectorization performance can be found in Ref. [JAR12].

	`-no-vec`	`-msse3`
`-msse3`	1.82x	
`-mavx`	2.23x	1.23x

Table 1 Average speed-up results for different vectorization configurations (given by the indicated compiler flags) on the single-socket system. The results are obtained from the ratio between execution times for the configurations in the columns and the corresponding configurations in the rows.

The comparison of the reference sequential execution time with respect to the sequential execution time of the other implementations is the following:

- OpenMP implementation, fine-grained parallelism: 13.3% slower.
- Intel Cilk Plus implementation, fine-grained parallelism: 13.7% slower.
- Intel Cilk Plus implementation, coarse-grained parallelism: 0.3% faster.

We used `OMP_NUM_THREADS=1` in the case of the OpenMP implementation, while we used the Intel compiler flag `-cilk-serialize` for the Intel Cilk Plus implementation. The sequential execution times of the OpenMP and Intel Cilk Plus implementations based on fine-grained parallelism were consistent one with each other and slower than the reference sequential OpenMP execution (13.5% in average). This was expected because they do not use the block splitting optimization. As expected, the Intel Cilk Plus implementation based on coarse-grained parallelism

gave a consistent performance boost with respect to the reference sequential OpenMP execution, which is based on coarse-grained parallelism too.

# Threads	*Block size (# events)*
1	10,000
2	5,000
4 and 8	1,000

Table 2 Block dimensions used in the tests performed on the single-socket system. Note that the 8 threads case uses SMT.

The block sizes used in the scalability tests are reported in Table 2. The scalability results for the OpenMP implementation based on coarse-grained parallelism are shown in Figure 3. We can clearly see that this implementation scaled badly when running with 4 threads, *i.e.* the number of cores available on the system (speed-up is 2.87x against 3.97x expected, with a standard deviation of 0.01x). Analysis of the problem showed that it was a consequence of the small L3 cache size and the static partitioning of the events over the threads. We found also that setting the affinity allowed to stabilize the execution times (the standard deviations were at least twice larger without setting the affinity).

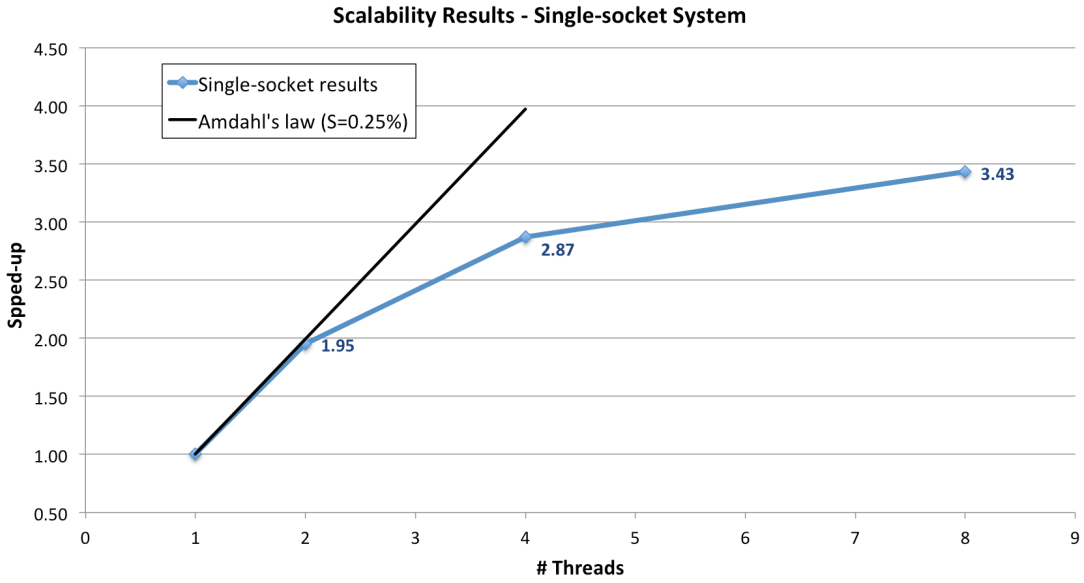

Figure 3 Scalability results for the OpenMP implementation based on coarse-grained parallelism running on the single-socket system. The solid black line without markers is the theoretical speed-up obtained by Amdahl's law with a parallel fraction of 99.75%. The 8 threads result uses SMT.

Finally we compared the performance of the different implementations when the system was fully loaded. In Figure 4 we show the results of the speed-up obtained with respect to the reference sequential execution time. The numbers draw two interesting conclusions:

- The implementations based on coarse-grained parallelism scaled better than those based on fine-grained parallelism (being almost twice faster). This was an expected consequence of the parallel implementation runtime overhead and the block splitting optimization.
- Among the implementations based on coarse-grained parallelism, both Intel Cilk Plus and Intel TBB implementations gave better performance results than the OpenMP implementation. As aforementioned, the OpenMP implementation suffered from the small L3 cache size and the static partitioning of the events over the threads, while the other two implementations used dynamic partitioning of the events in tasks. The Intel TBB implementation gave the best performance, slightly better (4%) than the corresponding Intel Cilk Plus performance.

We also tried the hybrid MPI implementations, using 2 MPI processes, but we did not find any benefit in performance. Note that in this case the number of OpenMP threads was accordingly reduced by a factor 2 so that the total number of running threads was equal to the number of MPI processes times the number of OpenMP threads. This procedure is automatic for the Intel Cilk Plus and Intel TBB implementations, whose runtime systems are able to recognize the Intel MPI runtime system underneath.

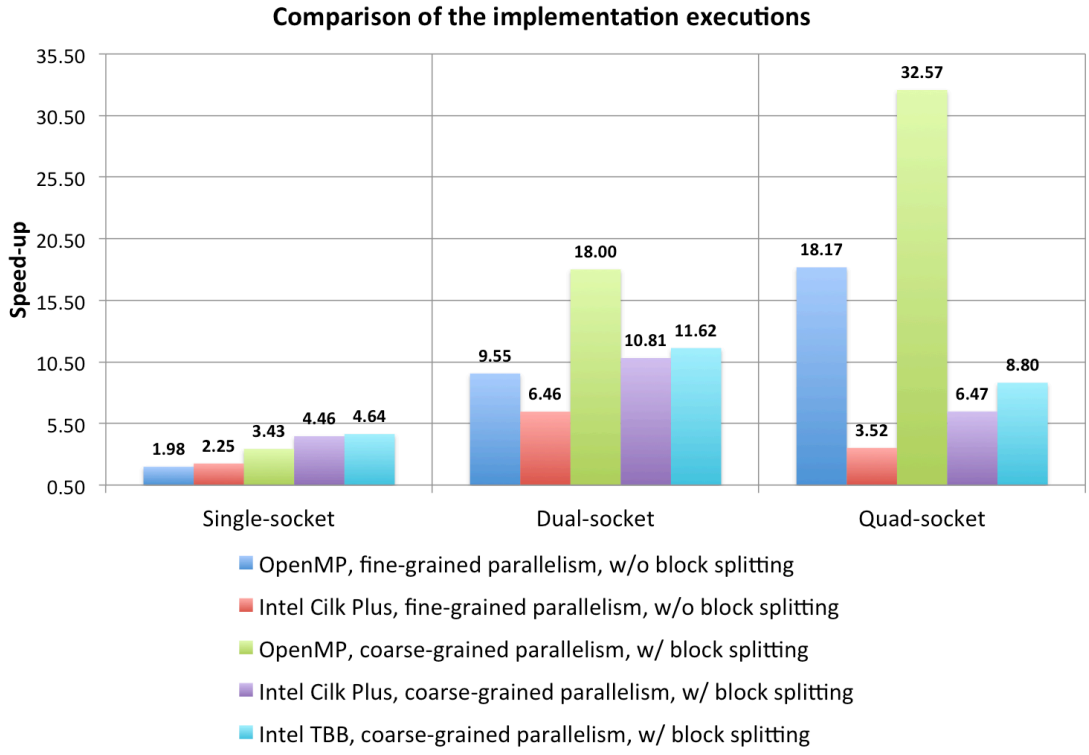

Figure 4 Comparison of several implementations for the three analyzed systems. The speed-up values were obtained running the implementations with full-loaded systems with respect to reference sequential execution time. Socket-affinity and hybrid MPI parallelization were not used in the tests.

Dual-socket Sandy Bridge-EP system

The block sizes used in the scalability tests are reported in Table 3.

# Threads	*Block size (# events)*
1 and 2	10,000
4	5,000
8	2,000
16 and 32	1,000

Table 3 Block dimensions used in the tests on the dual-socket system. Note that the 32 threads case uses SMT.

The scalability results for the OpenMP implementation based on coarse-grained parallelism are shown in Figure 5. We can clearly see that the scalability was very close to the theoretical expectation, with some degradation when the system was fully loaded (mainly due to the OpenMP runtime overhead when a high number of threads were involved). We found that setting the affinity was crucial to stabilize the execution times, which otherwise suffered large variations (in average 20%, but in some runs the execution time was twice longer).

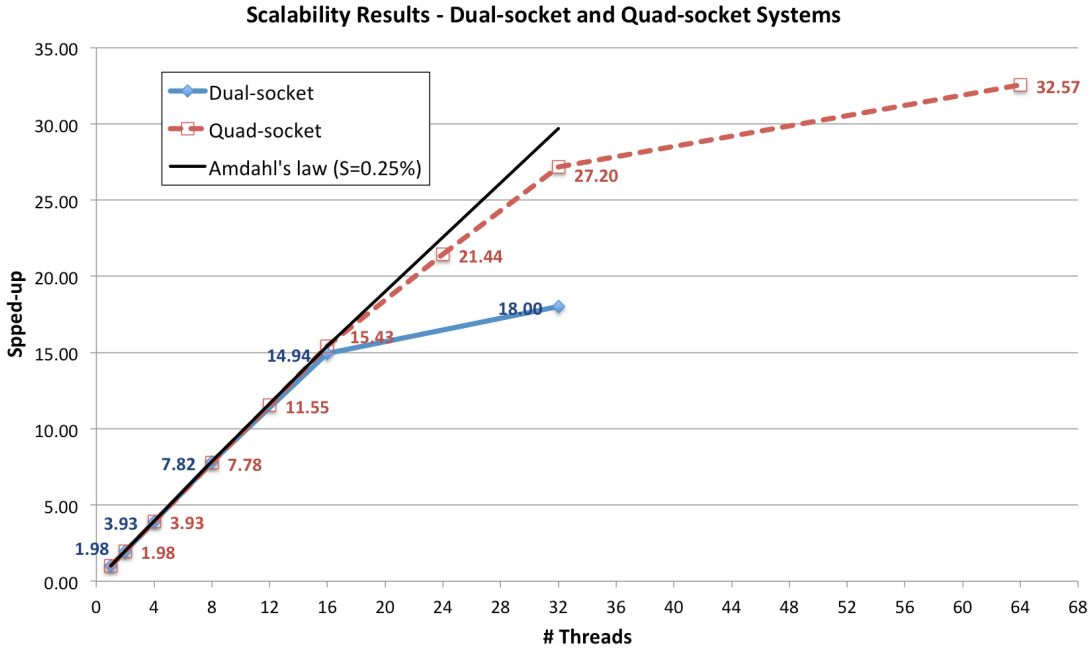

Figure 5 Scalability results for the OpenMP implementation based on coarse-grained parallelism running on the dual-socket and quad-socket systems. Data labels on the left (right) of the markers are for the dual-socket (quad-socket) system. The solid black line without markers is the theoretical speed-up obtained by Amdahl's law with a parallel fraction of 99.75%. The 32 threads result for the dual-socket system and the 64 threads result for the quad-socket system, respectively, use SMT.

The results for the performance comparison of the different implementations when the system was fully loaded are shown in Figure 4. Contrary to the results obtained

on the single-socket system, in the dual-socket case the Intel Cilk Plus and Intel TBB implementations based on the coarse-grained parallelism are much slower than the corresponding OpenMP implementation. We found that this is intrinsically due to the execution on a multi-socket system (probably a NUMA effect). Indeed, it turned out that using the command:

```
numactl --interleave=all <app> <options>
```

so that the threads were bound into the cores of a given socket, the difference became smaller (see Figure 6). This can be considered an affinity at the level of the different sockets (socket-affinity). However, performance improved a lot when running the hybrid MPI parallelization, using a number of MPI processes equal to the number of sockets. These results are shown in the same Figure 6. We found that this is mainly due to two effects:

- Intel MPI runtime system automatically does socket-affinity when running with Intel Cilk Plus or Intel TBB.
- The multi-processes parallelization based on MPI reduces the parallel implementation runtime overhead, especially when a high number of threads are involved (the side effect being the increase in the total memory used).

We noticed also that for the dual-socket system, which has enough L3 cache, there was a small benefit coming from the dynamic partitioning of the events in tasks used in the Intel Cilk Plus and Intel TBB implementations. The Intel TBB implementation based on coarse-grained parallelism gave the best performance, like in the single-socket case, which was very close to the corresponding OpenMP implementation, while the corresponding Intel Cilk Plus implementation was 6% slower. Eventually we also tried the hybrid MPI implementations with 4 MPI processes, but they did not give any benefit in performance.

Quad-socket Sandy Bridge-EP system

The block sizes used in the scalability tests are reported in Table 4. Most of the conclusions previously described for the dual-socket system holds also for the quad-socket system. Therefore, here we report only the differences.

# Threads	*Block size (# events)*
1, 2 and 4	10,000
8	5,000
12	3,000
16 and 24	2,000
32 and 64	1,000

Table 4 Block dimensions used in the tests on the quad-socket system. Note that the 64 threads case uses SMT.

The scalability results for the OpenMP implementation based on coarse-grained parallelism are shown in Figure 5. We found that the OpenMP runtime overhead significantly impaired the scalability for a high number of threads.

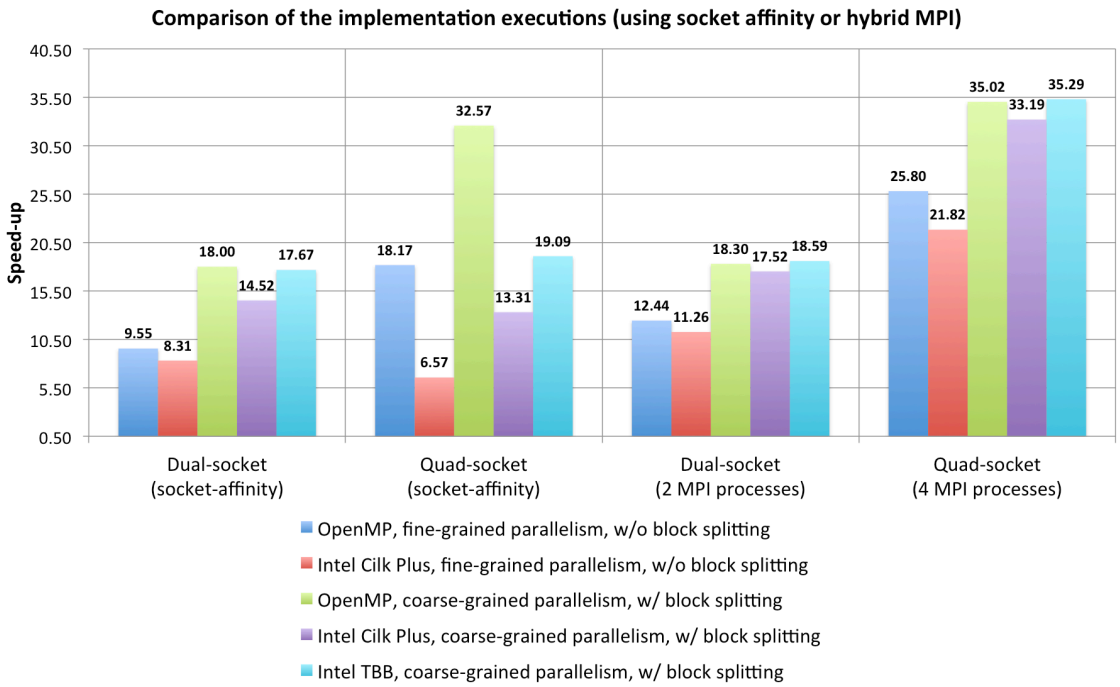

Figure 6 Comparison of several implementations for the dual-socket and quad-socket systems. The speed-up values were obtained running the implementations with full-loaded systems with respect to reference sequential execution time. Socket-affinity and hybrid MPI parallelization were used in the tests.

The results of the performance comparison of the different implementations when the system was fully loaded, with and without socket-affinity and hybrid MPI parallelization, respectively, are shown in Figure 4 and Figure 6. As naively expected, the performance degradations for the Intel Cilk Plus and Intel TBB implementations were bigger than in the dual-socket case, since the execution spread over four sockets. Therefore, it is crucial for this system to use socket-affinity. Moreover, the hybrid MPI implementations reduced dramatically the parallel implementation runtime overhead due to the lower number of threads involved. Eventually we also tried the hybrid MPI implementations with 8 MPI processes, but it did not give any benefit in performance.

Conclusions

We have described in this report the vectorization and parallelization of the ML fit application developed by CERN openlab. We have compared the different technologies used from an ease of use and performance perspective. The purpose was to study the most productive methods to improve the performance of an existing application on contemporary platforms.

First of all we found that for getting good performance it was crucial to organize data in vectors aligned in memory and perform the computation on such vectors by means of loops that can be vectorized. The vectorization was achieved by using the auto-

vectorization of the Intel compiler, adding the `#pragma ivdep` directive before each loop, or using the Intel Cilk Plus Array Notation for replacing the loops. We found that the latter gives a very clear way to express loop vectorization, although the execution of the application was about 2% slower than the version based on `#pragma ivdep`. Overall performance boost was about 2.2x when using AVX vector registers with respect to a non-vectorized version of the application.

Concerning the parallelization we found that it is very beneficial to invest effort in making the code thread-safe, to achieve the execution of the largest possible parallel regions (coarse-grained parallelism). This in turn reduces the overhead of the parallel technology runtime, improving scalability for large number of parallel threads. We found that other important points needed to achieve better performance were the management of the data inside cache memories, affinity of threads into the cores of the systems and scheduling of the threads runtime execution. Furthermore, specifically for our application it was also important to assure the reproducibility of results across different parallel executions and across executions with different number of threads. Among the three different implementations based, respectively, on OpenMP, Intel Cilk Plus, and Intel TBB, the last one gave the best results in terms of performance with the minimum number of changes required to our C++ application source code. When comparing the performance on systems with different number of sockets and high number of cores, we found that the hybrid MPI implementation improved the performance when combined with Intel Cilk Plus and Intel TBB parallel implementations, especially by using a number of MPI processes equal to the number of sockets. The application scaled very close to the theoretical expectation, also when using SMT, reaching a speed-up of about 35x on 32 SMT-enabled cores.

To conclude, the best implementation in terms of feasibility and performance for our ML fit application was the one based on Intel Cilk Plus Array Notation for vectorization and hybrid Intel TBB with MPI for parallelization.

References

COW98	G. Cowan: *Statistical Data Analysis*, Clarendon Press, Oxford (1998)
ROF06	W. Verkerke and D. Kirkby: *The RooFit Toolkit for Data Modeling*, proceedings of PHYSTAT05, Imperial College Press (2006)
JAR11	S. Jarp *et al.*: *How to harness the performance potential of current multi-core processors*, *J. Phys.: Conf. Ser.* **331** 012003 (2011)
MIN72	F. James: *MINUIT - Function Minimization and Error Analysis*, CERN Program Library Long Writeup D506 (1972). Also see the webpage http://seal.web.cern.ch/seal/MathLibs/Minuit2/html/
SUM01	P. Kornerup *et al.*: *On the Computation of Correctly-Rounded Sums*, IEEE Transactions on Computers 61 3 (2012)
BBR09	B. Aubert *et al.*: B *meson decays to charmless meson pairs containing η or η' mesons*, Phys. Rev. D80, 112002 (2009)
JAR12	S. Jarp *et al.*: *Evaluation of the Intel Sandy Bridge-EP server processor*, CERN openlab report, CERN-IT-Note-2012-005 (2012)