Report on parallelization of MLfit benchmark using OpenMP and MPI

Sverre Jarp, Alfio Lazzaro, Julien Leduc, Andrzej Nowak, Liviu Valsan
CERN openlab, July 2012

Abstract

This report describes the development of an MPI parallelization support on top of the existing OpenMP parallel version of the MLfit benchmark for a hybrid evaluation on multicore and distributed computational hosts. MLfit benchmark is used at CERN openlab as a representative of data analysis applications used in the high energy physics community. The report includes the results of scalability runs obtained with several configurations and systems.

1 Introduction

This report describes the MPI parallelization for the MLfit benchmark (version 5). It also includes the results of scalability tests when running in several software configurations (such as, only OpenMP, only MPI, and a tradeoff between them) and hardware solutions (single multi-socket host, multiple hosts). Also, comparisons of the performance when running on a conventional cluster of server hosts and on a DELL microserver are presented.

Descriptions of the algorithm and its OpenMP parallelization can be found in [Jar11]. Scalability results for the version of the code based on OpenMP parallelization (version 4) are reported in [Jar12]. They are used as a reference for the new version of the application described in this report.

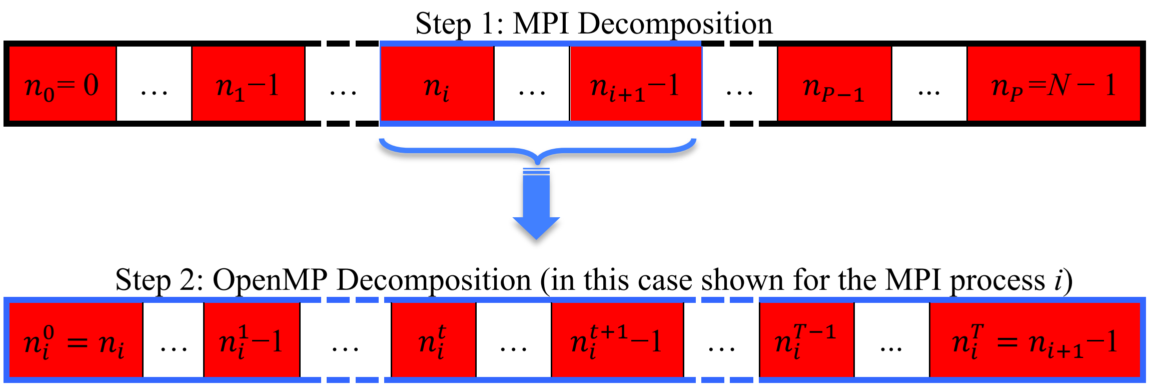
Implementing MPI parallelization on top of OpenMP parallelization allows going beyond the constraint of the parallel execution on a single host. In this respect MPI is the *de facto* standard for massive HPC parallelization on distributed hosts connected by network links. The standard does not make any basic distinction whether the MPI processes are running on a single multicore host or if they are distributed on independent hosts. In response to the rise of multicore systems, however, it is possible to consider the hybrid parallelization where each MPI process can run several OpenMP parallel threads. Therefore, it becomes possible to exploit both shared memory parallelism enforced by OpenMP and message passing parallelism between processes enforced by MPI.

The current report is organized as follows: section 2 describes the MPI implementation and section 3 reports on the tests and results when running the application with different configurations on several hardware systems.

2 MPI implementation

2.1 Algorithm description

In the parallel OpenMP implementation described in [Jar11] the input data are stored in memory in arrays of N elements. Each OpenMP thread, then, executes on a given independent subset of consecutive elements of the input N elements. The elements are statically partitioned. The partitioning is implemented in a way so that one thread can have at most one element of difference with respect to the other threads, to ensure an equal load-balancing. A reduce operation is performed in parallel on the final results, with each thread summing its own results in a different accumulator; finally, the master thread sums up all accumulators. This algorithm is modified to exploit both MPI and OpenMP in the computation. Each MPI process holds a copy of the whole input dataset. The same algorithm of the decomposition of the data elements, described before, is applied twice, for the MPI processes and then for the OpenMP threads belonging to each MPI process. A sketch of the algorithm is shown in the following picture:



The MPI decomposition determines the index limits n_i of elements of the input data arrays so that the MPI processes with rank $i = 0,1, \ldots (P-1)$, where P is the number of MPI processes involved, execute on the elements in the range $[n_i, n_{i+1} - 1]$. Then the OpenMP decomposition runs for each MPI process for determining the subsequent index limits n_i^t for the OpenMP threads $t = 0,1, \ldots (T-1)$, where T is the number of OpenMP threads. Therefore, the OpenMP thread t of the MPI process i runs on the elements of the input data arrays with indices in the range $[n_i^t, n_i^{t+1} - 1]$. Also, the reduce operation is performed in two steps. The first step consists of performing the reduction for the OpenMP threads of each MPI process as already described in the OpenMP-only 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 used for OpenMP, based on double-double compensation algorithm [Jar11], is used for the reduction of the MPI partial results. Then, after each likelihood evaluation, all MPI processes will proceed to execute the same part of code (*e.g.* the minimization in a maximum likelihood fit), 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 any exchange of other values in the remaining part of the application, *e.g.* the values of the parameters of the probability density functions during the minimization in a maximum likelihood fit. A check after each likelihood evaluation is executed to ensure that there were no errors during the evaluation. Each process sends an integer that can be zero in case of error or the number of analyzed events otherwise, respectively. 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.

To conclude, the application runs exactly the same as in the OpenMP-only implementation within one MPI process, except for the different start and end values of the indices for the input data elements. The only real difference from the OpenMP-only case is that each worker now owns a different sum accumulator relative only to its subset of data. In order for the final result to be computed, a second parallel reduction must be performed thus summing up all the MPI partial results owned by each worker. 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.

2.2 Implementation design

A first design requirement for the inclusion of MPI support in the MLfit application is the possibility to compile without MPI support without losing functionality, *i.e.* switching back to the OpenMP-only parallelization. This is achieved by using preprocessor macros. In particular all calls to MPI APIs are decorated with special disabling macros, in a way so that it is possible to completely disable MPI, removing all the function calls and replacing them with default operations valid for OpenMP-only execution [Car11].

A second design requirement is the possibility to *encapsulate* explicit MPI calls inside more advanced functions, *i.e.* all MPI calls inside the MLfit application must be called through special wrapper functions. This allows to decouple the MLfit code from the MPI direct calls, in particular reducing the required number of changes in the MLfit initial code. A singleton class has been implemented for that. The class provides some static public methods for the MPI initialization and finalization, request of the MPI rank and number of processes, reduce operation, and timing. The singleton design pattern is particularly effective since the MPI initialization, which is placed in the default constructor of the class, is called only once, the first time the MPI support is requested in the MLfit, *i.e.* there is no need to explicitly initialize MPI in the MLfit code. A specific test to check if the MPI was initialized is made at the beginning of the execution of each public method. MPI finalization is placed in the destructor of the class, which is automatically called at the end of the execution of the application.

Timing of the MPI application is a delicate matter since it requires a common time reference between multiple processes that can potentially run on different network connected systems. An explicit synchronization between all MPI processes (by means of an MPI `Barrier` operation) is made before taking time references. Then, in the implementation described in this report, only the MPI process with rank 0 (master process) takes care of providing the timing of the application.

Finally, the implementation takes care of correct printing to standard output for the entire application. In particular it provides two possibilities: all MPI processes can print (in this case a label that reports the MPI rank is put at the beginning of each line) or only an MPI process with a given rank can print, the outputs from the other processes being discarded. Furthermore the implementation provides a thread-safe mechanism in the case of output from several OpenMP threads, which is based on critical regions and independent buffers for each thread. In detail, the implementation intercepts all calls to the C++ `std::cout` stream, redirecting its default `std::streambuf` buffer to a new buffer by using the `std::cout.rdbuf` method. Then, methods `overflow` and `sync` of this new buffer are overloaded, so that they can properly handle the print operation. The real internal buffers are implemented as `std::string` per each OpenMP thread. To summarize, the implementation allows printing to standard output without modifying the original application source code. The default for the MLfit application is that only the master MPI process can print to standard output.

3 Tests and scalability results

3.1 Technical Setup

Four systems are used in the tests:

A. Intel Sandy Bridge-EP system (Intel(R) Xeon(R) CPU E5-2680)
   a. Dual-socket, 16 cores @ 2.70GHz
   b. Cache size per CPU: 20480KB
   c. Memory size: 64GB

B. Intel Westmere-EX system (Intel(R) Xeon(R) CPU E7-4870)
   a. Quad-socket, 40 cores @ 2.40GHz
   b. Cache size per CPU: 30720KB
   c. Memory size: 128GB

C. Intel Westmere-EP system (Intel(R) Xeon(R) CPU X5650)
   a. Dual-socket, 12 cores @ 2.67GHz
   b. Cache size per CPU: 12288KB
   c. Memory size: 48GB

D. Intel Sandy Bridge system (Intel(R) Xeon(R) CPU E3-1280)
   a. Single-socket, 4 cores @ 3.50GHz
   b. Cache size per CPU: 8192KB
   c. Memory size: 8GB

All systems have Turbo mode disabled. The Westmere system has Linux version: Scientific Linux CERN SLC release 5.7, based on Red Hat Enterprise Linux release 5.7, while the Sandy Bridge system has release 6.2. Code is compiled with Intel ICC v12.1.0 and Intel MPI v4.0.3. Vectorization is based on AVX for the Sandy Bridge systems (by using the compiler flag `-mavx`) and SSE for the Westmere systems (`-msse3`), respectively. Concerning multi-node systems, tests are performed on 2 Westmere-EP servers (system C in the list) and a microserver system (DELL PowerEdge C5220) equipped with 4 hosts (system D). The network connection is based on standard 1Gb Ethernet copper links, one link per each host, connected on the same switch to limit latency.

OpenMP threads of the MPI processes are bound to cores of CPUs on different sockets before filling the cores of a given CPU. This allows to maximize the available cache memory per each thread. For the same reason, the processes' topology used in the multi-node tests maximizes the number of hosts involved. The systems are SMTenabled, which means that the hardware threading feature is activated and used during the tests. Thus, if there are no more physical cores available, the jobs are pinned to hardware threads by requiring 2 threads per core.

3.2 Comparison of MLfit version 4 and 5

The first set of tests is a comparison of the performance (wall-clock time) obtained when running version 5 of the code with respect to version 4 (OpenMP-only implementation). This can be used for validating the new version. These initial tests have been executed only on the Sandy Bridge-EP system (system A). The same likelihood model described in [Jar12] is used. Also the dimensions of the blocks and the number of OpenMP threads are the same. The input data sample is composed of 1,000,000 events. These characteristics are used in all tests presented in the current report. The number of OpenMP threads shown refers to the total number of threads obtained by the product of the number of MPI processes per node and the number of OpenMP threads per each MPI process. Two independent verifications have been performed:

1. Compiling version 5 without MPI support
2. Compiling version 5 with MPI support and running with a single MPI process.

In both cases the performance results are consistent (within statistical errors) with the results obtained when running version 4. That proves that version 5 without MPI support or when it is requesting only one MPI process, is just like version 4 based on OpenMP-only parallelization.

3.3 Single-host performance results

Tests have also been executed on systems A and B. The goal is to compare the performance results obtained when running a single MPI process with several OpenMP threads ((1 MPI)×(# OpenMP)) with respect to several MPI processes each one with a given number of OpenMP threads ((# MPI)×(# OpenMP)). A side effect of the latter configuration is that all input data are replicated for each MPI process, so the memory footprint increases. The topology for the affinity of the threads maximizes the cache memory per each thread, *e.g.* 4 MPI processes with 2 OpenMP threads each on a dual-socket system will run with 2 MPI processes per socket, *i.e.* 4 OpenMP threads per socket.

Results show that the configurations (# MPI)×(# OpenMP) gives between 1% and 2% better performance with respect to the configuration (1 MPI)×(# OpenMP) for a given total number of threads. In particular, the best configuration is reached when considering an MPI process per socket, so that the corresponding OpenMP threads run on that socket. Results of the comparison for this configuration with respect to the configuration (1 MPI)×(# OpenMP) are:

1. +1.2% for the dual-socket system (A), *i.e.* (2 MPI)×(# OpenMP)
2. +1.9% for the quad-socket system (B), *i.e.* (4 MPI)×(# OpenMP)

The improvement is due to better access to the input data, replicated per each MPI process, the application being NUMA-aware only for the arrays of generated results.

3.4 Multi-host performance results

Tests of scalability are executed using the cluster composed by the two systems of type C. For comparison they are also executed on the system B using the configuration (4 MPI)×(# OpenMP) (see section 3.3 for an explanation of the configuration). In these tests we are interested to the scalability of the application, *i.e.*

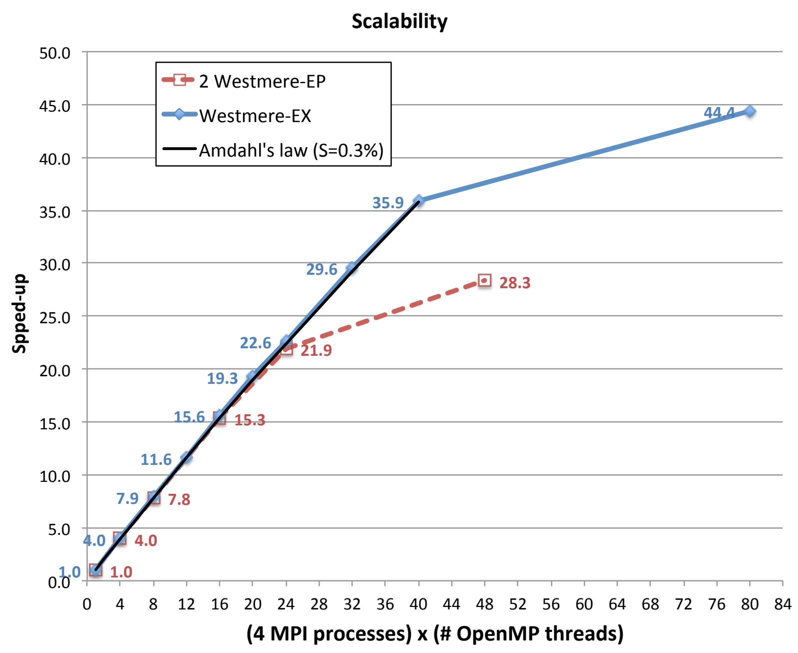


Figure 1 Speed-up results when running on the two Westmere-EP systems and the Westmere-EX system. Note that 48 (80) threads case for Westmere-EP (Westmere-EX) plot engages SMT.

strong scaling. We look at the speed-up, defined as the ratio between the execution time spent by the application running in sequential, *i.e.* (1 MPI)×(1 OpenMP), and the execution time spent when running in parallel with a certain number of MPI processes and OpenMP threads. The fraction of code that it is parallelized is 99.7%. The sequential execution time is about 1200 seconds when running on the system C. Taking in account the considerations described in section 3.3 for sockets and MPI processes, the topology used for the tests on the two systems of type C is (4 MPI)×(# OpenMP), where 2 MPI processes are launched on each system and then their corresponding OpenMP threads are pinned within the sockets of the system. Plots of application scalability are shown in Figure 1. The results when running on the two Westmere-EP systems are compatible with the results obtained from running on the single Westmere-EX system. A slight drop of the performance is seen for the Westmere-EP results when running large number of total threads. This is due to the smaller size of the L3 cache memory that limits the scalability on the Westmere-EP systems. For instance the performance when running in total 6 OpenMP threads for each of the two hosts is about 3.2% better than running the corresponding 12 OpenMP threads on a single host, *i.e.* with all cores engaged. The same behavior is not present on Westmere-EX, where the scalability is close to the expectation, since the system has a bigger L3 cache per CPU and it is a quad-socket system. The breakdown per function call of the execution time when running with the configuration (4 MPI)×(12 OpenMP) on the Westmere-EP systems, *i.e.* full load, is shown in Figure 2. Ignoring what are the specific functions related to the red bars, the important point from this plot is that the workload is balanced across the OpenMP threads and MPI processes for the most time-consuming functions. Then only the master thread of each MPI process runs specific functions, which are represented by the bars at the right of the plot: loading data (blue), MPI `Allgather` function and reduce operation (yellow), MPI `Allreduce` function (gray), remaining functions (black). All MPI functions in this configuration affect of 1.4% of the total execution time. Therefore it is possible to conclude that MPI overhead can be considered small in these tests.

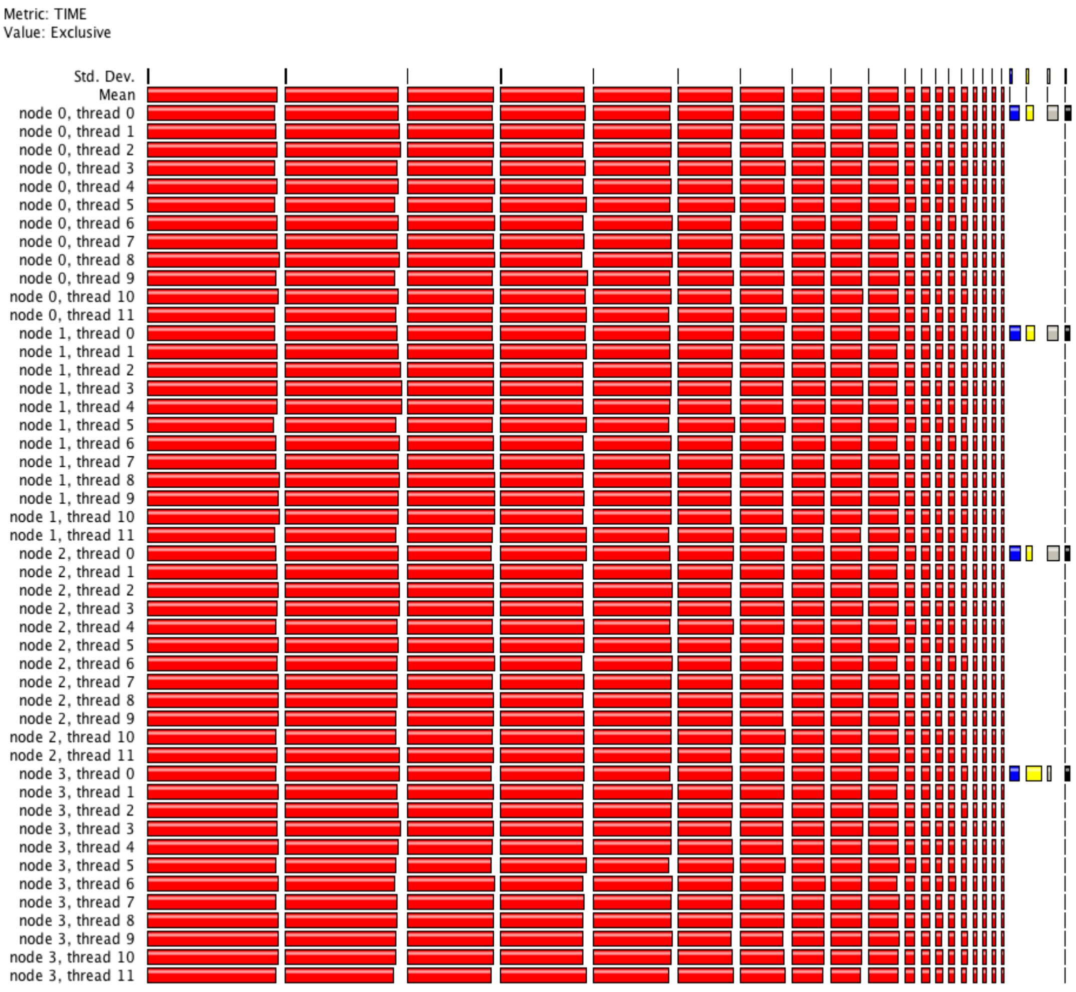


Figure 2 Breakdown per function call of the execution time when running with the configuration (4 MPI)×(12 OpenMP) on the two Westmere-EP systems (bar length represent exclusive time). Note that node 0 and 1 (2 and 3) refers to the MPI processes running on the same system.

The last tests presented in this report are executed on the DELL PowerEdge C5220 microserver with 4 hosts of type D, *i.e.* 16 cores in total. For comparison the tests were also executed on system A, which has the same number of total cores. Note that the comparison is made between a 4-core CPU (E3 family) and an 8-core CPU (E5 family). In particular the total L3 cache size available on the former is much smaller than on the latter (8MB versus 20MB, respectively). The same consideration applies to the L3 cache size per each core (2MB versus 2.5MB). This introduces some performance penalty. Indeed, already when running sequentially, the performance of the E3 CPU is 1.2% lower (frequency scaled) than when running on the E5 CPU, and it increases to 6.2% when running 4 threads in total. The configurations used in these tests are (4 MPI)×(# OpenMP) for the microserver hosts, *i.e.* an MPI process per host, and (2 MPI)×(# OpenMP) for system A, respectively. Hence, the number of MPI communications remains the same during the tests, being increased is only the number of OpenMP threads. Plots of scalability are shown in Figure 3. The drop in performance of the microserver results becomes considerable when a high number of

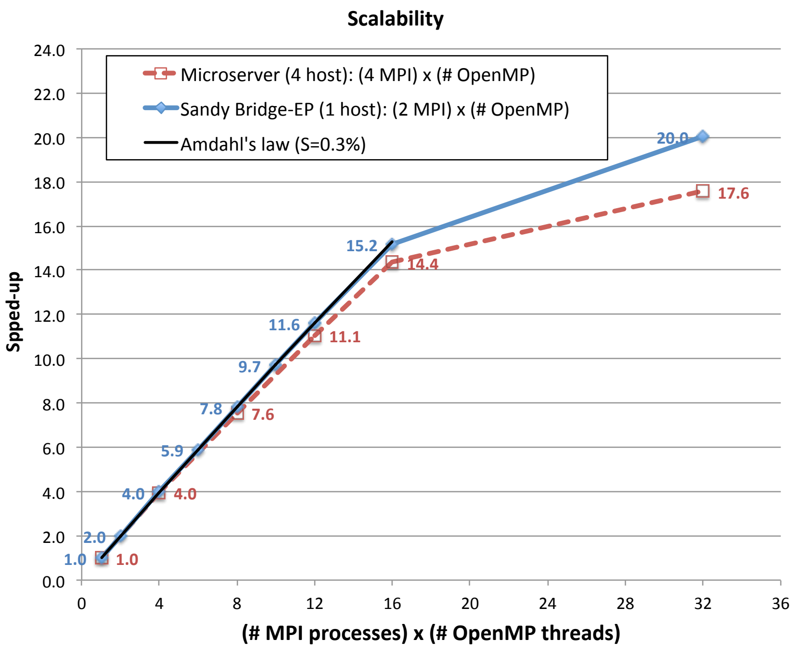


Figure 3 Speed-up results when running on the microserver system with 4 hosts and the Sandy Bridge-EP system. Note that 32 threads case uses SMT.

total threads are involved: 4.5% per 12 threads, 5.5% per 16 threads, and 13.6% per 32 SMT threads. The effect for 12 and 16 threads, which are the cases when no SMT threads are engaged, can be directly correlated to the penalty due to the smaller cache size on the E3 CPU. However, this effect does not fully explain the drop in performance when using SMT. In this case it is the SMT contribution itself that has less impact in the E3 based system with respect to the E5 dual-socket system: +22.2% and +31.2%, respectively. Running on a single host of the microserver with 4 and 8 OpenMP threads the SMT contributes for +25.0%, which is in agreement with the results when all 4 hosts are used (taking in account also the higher number of threads). The MPI functions contributions for the configurations (# MPI)×(8 OpenMP) with respect to the corresponding total execution time are:

- 2 MPI processes: 1.6% (total execution time 90.9 seconds)
- 3 MPI processes: 2.6% (total execution time 61.5 seconds)
- 4 MPI processes: 3.4% (total execution time 46.9 seconds)

Therefore the exclusive execution time for the MPI functions is 1.5-1.6 seconds, with a small dependency on the number of nodes involved.

4 Conclusion

The scalability results of the version 5 of the MLfit benchmark are overall satisfactory. The MPI parallelization already helps when running in a single host, when combined with the OpenMP parallelization requiring an MPI process per socket. The performance when running on a small cluster, composed by two hosts, is excellent, very close to the theoretical expectation. Finally, the possibility to use a system based on microserver can be a suitable solution with respect to a multi-socket server (dual or quad). Frequency scaled, the single Sandy Bridge-EP server system with 16 cores (2 E5 CPUs) is 16% faster than 4 hosts equipped with 4-core E3 CPU. The main limitation comes from the smaller L3 cache size of the 4-core E3 CPU, and a small penalty is found to be due to MPI communications. It is worth to consider that:

- One DELL PowerEdge C5220 microserver system can host up to 12 nodes. A test with 5 hosts shows that the microserver system is 6.6% faster than the single dual-socket system (frequency scaled).
- The 4-core E3 CPUs are available with higher clock frequency respect to the 8-core E5 CPUs. The comparison of performance without frequency scaling shows that the microserver with 4 hosts at 3.5GHz is 12.1% faster than the single dual-socket system with CPUs at 2.7GHz.

Therefore either using more hosts or higher frequency CPUs can easily compensate for the small drop in performance reported in the tests.

References

Jar11	S. Jarp *et al.*, *Evaluation of Likelihood Functions for Data Analysis on Graphics Processing Units*, ipdpsw, pp. 1349--1358, 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, 2011. EPRINT: CERN-IT-2011-010
Jar12	S. Jarp *et al.*, *Evaluation of the Intel Sandy Bridge-EP server processor*, 2012. EPRINT: CERN-IT-Note-2012-005
Car11	R. Caravita, *Implementation and test of MLFit application using OpenMP and MPI parallel technologies*, 2011. CERN openlab Summer Student report (see CERN openlab webpage)