Implementation and test of MLFit application using OpenMP and MPI parallel technlogies

Ruggero Caravita

CERN openlab

8 August 2011

oTN-2011-01 openlab Summer Student Report

CERN openlab

Implementation and test of MLFit application using OpenMP and MPI parallel technologies

Ruggero Caravita
Alfio Lazzaro

12 August 2011
Version 1
Distribution: **Public**

Abstract

The aim of the present document is to describe the openlab summer student project consisting in adding the support of the MPI parallel library to the MLFit prototype. The project includes both code implementation and test, aiming to benchmark the application in multiple scenarios (such as only OpenMP, only MPI and a tradeoff between them) and hardware solutions.

1 Introduction

Briefly, MLFit prototype is a simplified version of the RooFit package (a crucial part of the official ROOT library used in likelihood-based data analysis) aimed to be an easy to compile but sketched implementation preserving the same programming interface of RooFit without all the physics-related details. In this document I will present only a summarized overview of the simplified algorithm used in MLFit; an exhaustive description on maximum-likelihood optimization techniques and the RooFit fitting algorithm from which MLFit is taken can be found in (1).

Significance improvements in performance of the MLFit can be achieved with a well-designed parallel implementation which allows work splitting on modern CPUs and/or GPUs. Currently, many versions with different parallel technologies have been (or are being) developed, i.e. OpenMP, CUDA, OpenCL, and tested on many hardware platform, from NVIDIA and ATI GPUs to Intel Xeon and MIC.

In the present work I will present the MPI version I developed on the top of the existing OpenMP implementation, in order to exploit both shared memory parallelism enforced by OpenMP and message passing parallelism between workers enforced by MPI. I will present also the test results obtained running the application on many hardware platforms, from a multi-socket Xeon server to Intel SCC research microprocessor.

For more details about the technique exploited for parallelization in such pre-existing version of MLFit with OpenMP, see (2).

The algorithm used in MLFit and its parallel implementation are briefly explained in the next introductive paragraphs. In the second section I discuss the details about the MPI implementation in MLFit with particular care to the software design to fit all the mandatory usability requirements. In the third and fourth sections it is discussed how to print to standard output in a parallel application with no code modification from the user and how to measure time. In the fifth section the scalability tests with OpenMP and MPI are shown, together with a performance comparison in the hybrid scenario (both OpenMP and MPI).

1.1 Brief overview on the algorithm

The aim of the RooFit library is to perform fitting procedures, i.e. adjusting PDF parameters in order to best reproduce the numerical data from the experiments, given a set of probability distribution functions from theoretical models. From now on, let's refer to probability distribution functions as PDFs and to an experiment numerical data as event.

The well known technique used by RooFit to perform fitting is called *maximum likelihood estimation*: without going into the mathematical details (see (3) for a detailed description), at the present level what should be taken into account is that there exists a function

$$NLL(\theta) = -\sum_{i}^{N} \log\left(PDF_{\theta}(\mathbf{x}_i)\right) \qquad (0.0)$$

that should minimized with respect to the θ parameters in the least time possible, where N is the number of events. The computational time depends on the number of events and complexity of the PDF.

The goal of the prototype is to speed up at the best the PDF evaluation without changing library interface, in order to import the same evaluation technique in the production RooFit package.

A simple sketch of the algorithm used in the sequential single-threaded prototype to benchmark the PDF evaluation is the following:

- Create a N-sized sample array for the input *double* events;
- Create a N-sized array of *double* results;
- For i from 0 to N
    - Evaluate the PDF in x_i: $y_i = PDF_{\theta}(x_i)$
    - Evaluate the negative logarithm of y_i;
    - Compute the total NLL summing up all the logarithms.

When moving to parallel computation this algorithm must be slightly modified in order to split the work efficiently between the parallel workers. The OpenMP implementation, fully described in (2), follows the same steps sketched above. The advantage of working with arrays of data is exploited in the evaluation of the PDF that now is performed by a group of threads, each working on an independent part of the array. The *reduce* operation is thus performed parallel, with each thread summing its own results in a different accumulator; finally, the root thread sums up the few remaining accumulators. The sketch of the algorithm for such parallel implementation is the following (with the usual convention of bold letters for arrays)

- Create a N-sized sample array of input events;
- Create a N-sized array of *double* results;
- Split the input between *threads*, defining for each thread two vectors $\mathbf{x}_i$ and $\mathbf{y}_i$, respectively the input and the output to be processed by that threads;
- Each thread *t* within *threads*
    - Evaluate the PDF: $\mathbf{y}_i = PDF_\theta\left(\mathbf{x}_i\right)$
    - Evaluate the negative logarithm of $\mathbf{y}_i$
    - Sum all the logarithms within $\mathbf{y}_i$ in an accumulator s_i
- Sum all the accumulators to get the final NLL value: $NLL = \sum_i^{threads} s_i$

Since the goal of the present project is to exploit both MPI and OpenMP in the computation, this implies to modify the existing OpenMP parallel algorithm in order to take into account also MPI. In particular MPI can be consider *on top* of the OpenMP, since we operate MPI parallelization and then OpenMP parallelization for each MPI process. There are two main advantages of adding support to MPI: running the application on networked systems of multiple computing nodes and increasing performance on the single node reducing the number of synchronization needed by OpenMP.

Let's say we have an input dataset *I* composed of *N* events to process, and a set of *P* MPI parallel workers with *T* OpenMP threads each. Each MPI worker *w* holds s a copy of the whole input dataset. Since OpenMP uses shared memory, the input data chunk processed by OpenMP threads should be the most compact possible in memory to avoid cache misses. A clever way to fulfil this constrain is first split the input dataset between MPI workers into $N \div P$ subsets, map subsets one to one on MPI workers and then split again each subset between threads into chunks of events, each to be processed by a different thread t_w.

In the next section I explain how to split the work between the two technologies. Each *w* operates on a subset of *I* defined by two indices *iMPIStart, iMPIEnd*; each t_w in *w* has the chunks $\mathbf{x}_i$ and $\mathbf{y}_i$ defined by two other (sub)indices *iStart, iEnd.*

It should be clear that, within one *w*, the routine runs *exactly* the same as in the OpenMP implementation except for the different start and end indices. 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 the final result to be computed, a second parallel reduction must be performed thus summing up all the MPI partial results owned one by each worker. This reduction is performed via an MPI::COMM_WORLD.Allreduce() call; workers perform collective communication summing up all the accumulators (double-precision numbers) to produce the final sum. It must be underlined that this is the only communication the MPI application must do in the whole application, and each worker communicates only a double number. We expect a very small (and thus negligible) overhead due to MPI communication.

1.2 Work splitting between OpenMP and MPI

Ideally each worker will take $N \div P$ events and process them independently of the others, before summing up the results all together (reduction). If *N* is not multiple of *P*, $N\%P$ workers will compute one more event splitting also the remainder.

In the original OpenMP implementation, of course, *P* corresponds to *T*, so the work is split only between the parallel threads producing integer indices *iStart* and *iEnd* corresponding to the first and the last event to be processed by the current thread (see Code block 1).

```
      Int_t iStart, iEnd;

      // That's the total number of events that this process should take
care of
      Int_t nEvents = MY_EVENTS;

      // Now I split for OpenMP
      int numEventsIn = nEvents / omp_get_num_threads();
      int numEventsOut = nEvents % omp_get_num_threads();

      // OpenMP load balancing
      iStart = omp_get_thread_num()*numEventsIn;
      if (omp_get_thread_num() < numEventsOut) {
        iStart += omp_get_thread_num();
        iEnd = iStart + numEventsIn + 1;
      }
      else {
        iStart += numEventsOut;
        iEnd = iStart + numEventsIn;
      }

      // Update nEvents
      nEvents = iEnd - iStart;
```

Code block 1: Algorithm to split work between multiple OpenMP threads

What does it mean adding MPI on the top of OpenMP? First of all, the work must be split twice: between the MPI workers and then between OpenMP threads. Second, we must take care to set the right indices also if MPI or OpenMP are disabled. The new algorithm for work splitting looks similar to the previous, despite of the few MPI calls before the OpenMP splitting (see Code block 2).

```
      Int_t iMPIStart, iMPIEnd;
      Int_t iStart, iEnd;

      // That's the total number of events that this process should take
care of
      int Ev = nEvents;

      // Now I split for MPI
      int NumEventsIn = TotalEvents / MPI::COMM_WORLD.Get_size();
      int NumEventsOut = TotalEvents % MPI::COMM_WORLD.Get_size();

      // MPI load balancing
      iMPIStart = MPI::COMM_WORLD.Get_rank()*NumEventsIn;
      if (MPI::COMM_WORLD.Get_rank()<NumEventsOut) {
          iMPIStart += MPI::COMM_WORLD.Get_rank();
          iMPIEnd = iMPIStart + NumEventsIn + 1;
      } else {
          iMPIStart += NumEventsOut;
          iMPIEnd = iMPIStart + NumEventsIn;
      }
      Int_t MPIEvents=iMPIEnd-iMPIStart;

      // Now I split again for OpenMP
      int numEventsIn = MPIEvents / omp_get_num_threads();
      int numEventsOut = MPIEvents % omp_get_num_threads();

      // OpenMP load balancing
      iStart = omp_get_thread_num()*numEventsIn;
      if (omp_get_thread_num()<numEventsOut) {
        iStart += omp_get_thread_num();
        iEnd = iStart + numEventsIn + 1;
      }
      else {
        iStart += numEventsOut;
        iEnd = iStart + numEventsIn;
      }

      // Update nEvents
      nEvents=iEnd-iStart;
```

Code block 2: Algorithm to split work between multiple MPI workers and OpenMP threads

2 Adding MPI support

In this section I describe in detail the MPI implementation in the MLFit prototype, taking special care to the class design and the programming solutions in order to include MPI on the top of OpenMP without changing the external programming interface of MLFit. I describe also how to output to standard output in a parallel context without changing application code and how to perform a parallel reduction in MPI in a safe and deterministic way.

The application should be designed in such a way it can be compiled with or without MPI, without losing functionality or the original OpenMP parallelization. In addition, the less the user of RooFit has to modify its code, the better is; in particular, explicit calls to MPI, included MPI_Init() and MPI_Finalize() (or equivalent calls at the beginning and at the end of the execution) must be avoided by a smart design of the code.

In order to match these requirements, the software is designed with three levels of decoupling that abstract RooFit user from low level MPI calls. First, all the calls to MPI APIs are masked by special precompiler macros: the compiler is allowed to completely switch off MPI (not even linking is required) only by defining a macro. Second, within RooFit there are no direct calls to MPI APIs; all the calls are mediated by a special class, MPISystem, that takes care of calling MPI if it is enabled or just pass-by otherwise. Third, moving library function calls from MPISystem to its derived classes via protected virtual methods, it is possible to support other message-passing libraries different from MPI.

Finally, designing MPISystem with the singleton design pattern it is possible to call automatically MPI_Init() and MPI_Finalize() without any code modification in the application on the top of RooFit.

In the next sections I explain in more details how these levels of decoupling are implemented.

2.1 Class design

As sketched above, all the calls MPI 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 values valid for OpenMP-only execution. This is actually performed automatically if the code is not compiled with the MPI compiler (mpic++).

- **ENABLE_MPI:** the main macro to configure the application for MPI usage. If not defined, all the calls to MPI are removed from the code even at compile time, avoiding to link the unnecessary libraries. This macro is not defined if the compiler doesn't support MPI;
- **MPISafeCall**(func): call to *func* is added to the code only if a ENABLE_MPI is defined;
- **MPIElse**(funct): call to *func* is added to the code only if ENABLE_MPI is not defined.

The trivial C++ source code to implement those macros is contained in Code block 3.

```
// the purpose of this macro is shutting down MPI
#ifdef ENABLE_MPI
    #define MPISafeCall(p)       p
    #define MPIElse(p)
#else
    #define MPISafeCall(p)
    #define MPIElse(p)           p
#endif
```

Code block 3: C++ implementation of the macros for automatic removal of MPI calls from the code

From now on, all these macros are omitted in the next code blocks for readability.

All the other levels of abstraction are achieved by the smart design of MPISystem class: it takes care of MPI initialization and shutdown, inserts a layer of abstraction between RooFit and real MPI APIs calls and add support to other message passing libraries such as RCCE (4).

The singleton design pattern is particularly effective since MPI_Init() call, if placed in the default constructor of the singleton class, will be called only once at the first time the user requests MPI support. A subtle point, particularly important to match the requirement, is the choice of the singleton design. There exist many slightly different singleton patterns in C++, while the most common are Gamma singleton and Meyers singleton, see (5) and (6 p. 32). The Meyers singleton design is particularly interesting for our purposes since it offers automatic deletion of the instance (see Code block 4); in this way the MPI_Finalize() call can be easily placed in the default destructor of MPISystem class.

```
// modified Meyers singleton design pattern
class MySingleton {
protected:
    // default constructor
    MySingleton() {
        // add your initialization code here
    }

    // copy constructor locked
    MySingleton(const MySingleton &Ref) {

    }

    // default destructor
    ~MySingleton() {
        // add your shutdown code here
    }

public:
    // getinstance method
    static MySingleton *Instance() {
        static MySingleton Instance();
        return &Instance;
    }
};
```

Code block 4: modified Meyers singleton design pattern used for MPISystem

All MPISystem APIs are abstracted by public methods of MPISystem, e.g if the user wants to call MPI_Wtime(), he should call instead MPISystem::Instance()→WTime(). Each public method internally calls a protected virtual method of MPISystem, e.g. MPISystem::DoWTime(), to be overloaded by children classes of MPISystem with the implementation-specific code. This way of exploiting inheritance is often referred as *implementation inheritance*, since children of MPISystem decouple function call logic from all the possible implementations.

This fact is particularly important; it provides the last important functionality of MPISystem, an easy way to extend support to another existing message passing libraries with same logic but different interface than MPI.

Up to now, only two libraries have been added to MLFit: OpenMPI and RCCE. OpenMPI is an open source MPI standard library that operates via network both for accessing remote nodes and local workers. RCCE (‘rookie’) is a tiny MPI-like library written from Intel Corporation to run on the Intel Single-chip Cloud Computer; it has reduced functionality than standard MPI and a slightly different interface. The main purpose of adding support to RCCE is to benchmark the code also on this new research processor.

For reference, the complete source code of MPISystem class is available in Appendix A: MPISystem class details. For the implementation-specific code, see Appendix B: Adding support to OpenMPI and Appendix C: Adding support to RCCE.

2.2 Parallel reduction in MPI

The parallel reduction in MPI, executed just after the OpenMP reduction, have to produce the final result summing up all the partial OpenMP sums in at least the root process. It can be realized in many ways, the most simple of which is calling MPI::COMM_WORLD.Allreduce with MPI_SUM as reduction operation.

The call to the API has to be performed to keep OpenMP working with MPI disabled (see Code block 5).

```
        double Sum = 0.0;

        // OpenMP reduction
        // ... the value of sum is set to the partial sum of this worker ...

        // MPI reduction
        MPISafeCall(
            int WorkerId = MPISystem::Instance()->GetWorkerId();
            int RootId = MPISystem::Instance()->GetRootId();
            double ReceiveBuf = Sum;

            MPISystem::Instance()->WorldReduce(
                &Sum,
                &ReceiveBuf,
                1,
                MPI_DATATYPE_DOUBLE,
                MPI_OP_SUM
            );
            if (WorkerId == RootId) {
                Sum = ReceiveBuf;
            } else {
                Sum = 0.0;
            }
        )
```

Code block 5: code for performing MPI reduction

3 Printing to standard output within a parallel context

The first problem to take care of when moving to MPI is printing to standard output: a group of parallel processes accessing standard output at the same time will produce inconsistent string of characters. OpenMP doesn't suffer the same problem; it is quite trivial to perform thread-safe output with critical zones.

There are many conceptual ways of handling output in a multi-process context. In most of the cases, standard output is managed as a spinlock in order to avoid inconsistencies. Here I present only the two methods used in MLFit, root print and ordered print, before discussing how to practically implement them.

The first way to manage standard output is enabling printing only by one process (let's refer to this process as *root*). All the other processes in the system are not allowed to write to standard output unless *root* status is moved to one other process. If one process asks for printing but doesn't have the *root* privileges, its request is just discarded. This is the way used in MLFit to print the final results of the computation. The MPI implementation of this output is trivial.

The second way, ordered print, is particularly useful in debugging: each process prints its output sorted by ascending process id. A bit of communication is needed to perform this print, since the workers have to synchronize themselves to produce the sorted output. The algorithm is quite simple:

- at the beginning process 0 is *root* and is allowed to print, while the others are waiting to receive the permission in a *blocking* way;
- after finishing the print, *root* tells the other processes that process 1 is the new *root* and begins waiting the end of print, discarding any further communication;
- then process 1 prints, while the others are waiting;
- process 1 increments the value of *root* and broadcast to the others, before idling;
- *… repeat for all processes in the set …*
- the algorithm finishes when the processes receive a communication stating that the new *root* worker is equal to the number of processes.

The MPI implementation of this way of printing is included in code block 6.

```
        // prints buffer starting from zero
        int RootId = 0;
        int WorkerId = MPI::COMM_WORLD.Get_rank();
        int WorkerNum = MPI::COMM_WORLD.Get_size();

        while (RootId != WorkerId) {
            // this worker should wait
            int NewRootWorker=-1;
            // get new root worker id
            MPI::COMM_WORLD.Bcast(&NewRootWorker,1,MPI_DATATYPE_INT,RootId);
            if (NewRootWorker >= 0) {
                RootId=NewRootWorker;
            } else {
                std::cerr << "Invalid new root worker id, aborting"
                          << std::endl;
                exit(1);
            }
        }

        // this worker got root permissions: do the print
        std::cout << " worker " << WorkerId << " says hello world!"
                  << std::endl;

        // increment root id and broadcast
        RootId++;
        MPI::COMM_WORLD.Bcast(&RootId, 1, MPI_DATATYPE_INT, WorkerId);

        // wait until finish
        while (RootId != WorkerNum) {
            // this worker should wait
            int NewRootWorker=-1;
            MPI::COMM_WORLD.Bcast(&NewRootWorker,1,MPI_DATATYPE_INT,m_RootId);
            if (NewRootWorker >= 0) {
                RootId=NewRootWorker;
            } else {
                std::cerr << "Invalid new root worker id, aborting"
                          << std::endl;
                exit(1);
            }
        }
```

Code block 6: MPI code for sequential printing in parallel context

There is still a major problem left: change way of printing to standard output without modifying original application source code (i.e. explicitly calling a custom function to perform ordered print). In the next sections I explain a way of solving this problem when dealing with C++ STL default print.

3.1 Intercepting user calls to std::cout

Many user applications using ROOT and RooFit are written in C++, using STL iostream to output to screen. There is an elegant trick to hook the calls to std::cout in the C++ way and substitute the default non-parallel print with a custom parallel-safe one.

First, it must be created a child class of *streambuf*, the object of STL managing stream outputting. Let’s call this new class *mpibuf*. Within *mpibuf*, two virtual methods of *streambuf* have to be overridden: *overflow* and *sync*. The first is called when a new character is inserted into the stream; the latter is called when a flush signal is received and the print have to be performed. *overflow* should take care of inserting the new character into a buffer that will contain the string to print at the moment of *sync*.

Then, an instance of *mpibuf* must be created and the default std buffer substituted by the new one with a call to std::cout.rdbuf() function. In code block 7 an example of std::cout output hooking is shown.

```
    // std::cout hooking interface
    namespace std {
        class mpibuf: public streambuf {
        protected:
            std::string m_Buffer;

            mpibuf() {
                m_Buffer.reserve(4096);
            }

            virtual int_type overflow(int_type c) {
                if (c != EOF) {
                    m_Buffer.push_back(c);
                }
                return c;
            }

            virtual int sync() {
    #if 1
                // root print
                MPISystem::Instance()->RootPrint(m_Buffer.c_str());
    #else
                // ordered print
                MPISystem::Instance()->OrderedPrint(m_Buffer.c_str());
                MPISystem::Instance()->Sync();
    #endif
                m_Buffer.clear();
                return 0;
            }
        };
    };

    std::mpibuf *m_MpiBuf;
    std::streambuf *m_OldBuf;

    void InitHooking() {
        m_MpiBuf=new std::mpibuf;
        m_OldBuf=std::cout.rdbuf(m_MpiBuf);
    }

    void StopHooking() {
        std::cout.rdbuf(m_OldBuf);
        delete m_MpiBuf;
    }
```

Code block 7: sample C++ code for hooking std::cout calls and redirect output to a custom print function

4 Measuring time within a parallel context

Measuring time in a parallel context is a delicate matter. First of all, when moving from single threaded to parallel, the definition of 'total execution time' must be updated: the *total execution time* is the time taken by the slowest worker in the set to complete the execution.

Moreover, in many occasions it is not even simple to have a common time reference between multiple processes: if they are running on the same CPU with known synchronizations between the cores it's quite trivial to measure execution time, but the things get much more complicating when involving network communication. In this case, it should be kept in consideration that the time measurement by itself take some time to complete and may need process communication and, thus, synchronization.

To solve these complications it's common practice to use profiling tools, such as TAU (7): TAU is an easy to use profiling tool able to profile software using both MPI and OpenMP by instrumenting code before compilation.

In the next sections I describe how to measure time at runtime using MPI and OpenMP native calls in a transparent way for the user, and how to profile the application with TAU to measure MPI and OpenMP overhead.

4.1 Runtime application timing

Both MPI and OpenMP provide an API to measure time efficiently and thread-safe. In particular, the total execution time measurement is quite trivial in OpenMP, since the application is executed only by one process spawning threads when needed.

Similarly to OpenMP, also with MPI the measurement of total execution time via the provided API is taken on each process. To match the definition of total execution time in parallel context given before, a reduction must be performed keeping only the maximum time measurement between all the processes.

We still need to fulfill the constraints of no code modification by the user of MLFit and leave him choose whether to use MPI or not. In other words, we have to think a way of develop an interface to time measurement independent of the chosen parallel library (if any).

Again, class inheritance helps creating interface independent of implementation. In Code block 8 the timer class design exploiting the bridge design pattern for decoupling abstraction refinement inheritance and implementation inheritance is shown.

```
// timer types
enum {
    TIMER_LOCAL=0,
    TIMER_GLOBAL
};

// pure virtual prototype of back end class
// that implements the timer
class BaseTimer {
protected:
    int m_Type;

public:
    BaseTimer(int Type=TIMER_LOCAL);

    virtual void Reset()=0;
    virtual double Mark()=0;
    virtual unsigned long long GetTickCount()=0;

};

// class that should be used by the user (interface)
class Timer {
protected:
    BaseTimer *m_Impl;

public:
    Timer(int Type=TIMER_LOCAL);
    virtual ~Timer();

    void Reset();
    double Mark();
    unsigned long long GetTickCount();
};
```

Code block 8: Declaration of main Timer classes exposing bridge design pattern

The user of MLFit simply uses a Timer object in its code, specifying if the timer has to perform only the local measure or have to perform the reduction to get the right execution time. When a Timer is instantiated, in its constructor a BaseTimer child is automatically created choosing the one that best fits user library selection.

MPI timer, for example, is instantiated and used only if MPI is enabled in compilation; if both OpenMP and MPI are active at the same moment, MPI timer is preferred since OpenMP timer is not able to measure correctly the total execution time of the process set. Complete source code of the MPI timer is available in Appendix D: MPITimer class details.

4.2 External profiling with TAU

TAU is a powerful external profiling tool supporting both OpenMP and MPI. It is divided in two different applications: a script backend for instrumented code generation and a GUI frontend for graphical results plotting.

Once installed, it's enough to replace the compiler command from the usual gcc/icpc/mpicc to tau_compile.sh script to automatically build a profiled version of the program. After execution, a profile file is generated in the same directory with precise function timings, memory occupancy and communication latency measurements.

The GUI processes those profile files generating easily readable charts. Figure 2 demonstrates TAU ability to profile applications with both OpenMP and MPI running.

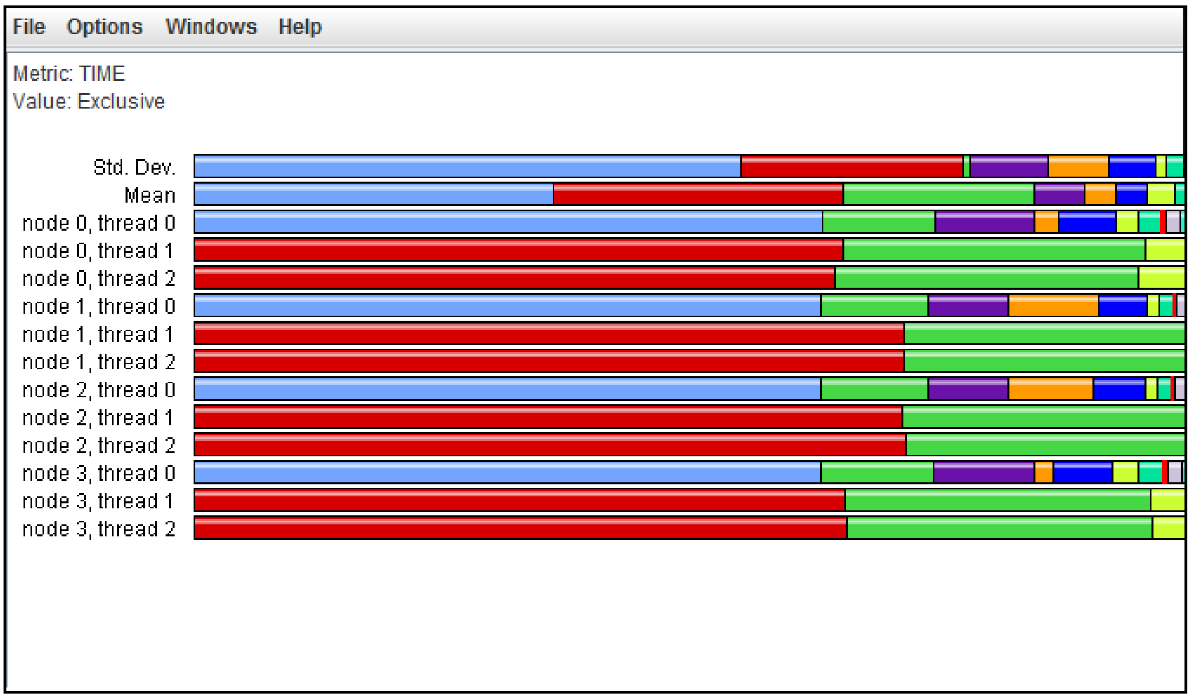
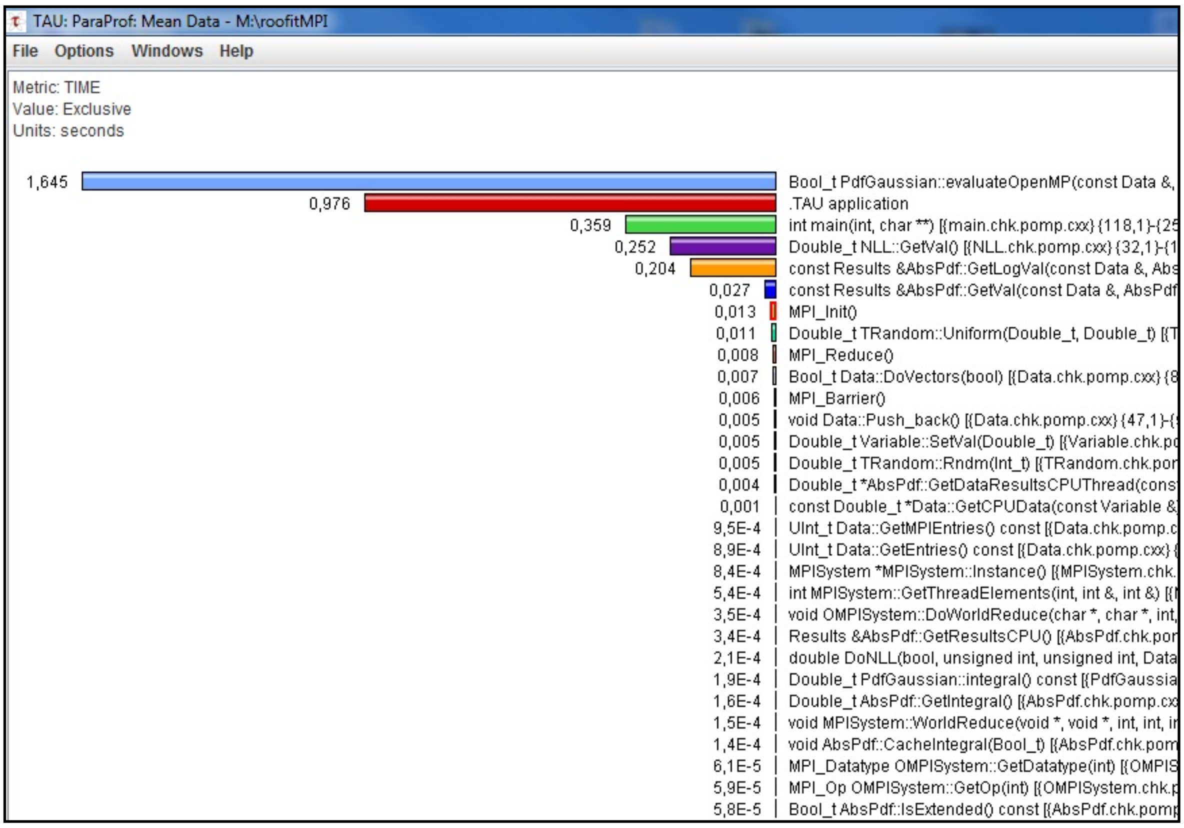


Figure 1: TAU graphical user interface screenshot demonstrating the profiling of an application using both MPI and OpenMP (respectively, 4 MPI processes with 3 OpenMP threads each).

In the context of this particular application, TAU is used for precise measurements of MPI communication overhead, verifying that MPI contribution to total execution time is totally negligible when compared to the time spent by PDF evaluation floating point operations, as clearly shown in Figure 3. The most expensive MPI call is MPI_init, taking only 0.013 s to be executed.



```
TAU: ParaProf: Mean Data - M:\roofitMPI
File  Options  Windows  Help

Metric: TIME
Value: Exclusive
Units: seconds

1,645  Bool_t PdfGaussian::evaluateOpenMP(const Data &,
0,976  .TAU application
0,359  int main(int, char **) [{main.chk.pomp.cxx} {118,1}-{25
0,252  Double_t NLL::GetVal() [{NLL.chk.pomp.cxx} {32,1}-{1
0,204  const Results &AbsPdf::GetLogVal(const Data &, Abs
0,027  const Results &AbsPdf::GetVal(const Data &, AbsPdf
0,013  MPI_Init()
0,011  Double_t TRandom::Uniform(Double_t, Double_t) [{T
0,008  MPI_Reduce()
0,007  Bool_t Data::DoVectors(bool) [{Data.chk.pomp.cxx} {8
0,006  MPI_Barrier()
0,005  void Data::Push_back() [{Data.chk.pomp.cxx} {47,1}-{
0,005  Double_t Variable::SetVal(Double_t) [{Variable.chk.p
0,005  Double_t TRandom::Rndm(Int_t) [{TRandom.chk.por
0,004  Double_t *AbsPdf::GetDataResultsCPUThread(cons
0,001  const Double_t *Data::GetCPUData(const Variable &
9,5E-4  UInt_t Data::GetMPIEntries() const [{Data.chk.pomp.c
8,9E-4  UInt_t Data::GetEntries() const [{Data.chk.pomp.cxx} {
8,4E-4  MPISystem *MPISystem::Instance() [{MPISystem.chk.
5,4E-4  int MPISystem::GetThreadElements(int, int &, int &) [{
3,5E-4  void OMPISystem::DoWorldReduce(char *, char *, int,
3,4E-4  Results &AbsPdf::GetResultsCPU() [{AbsPdf.chk.por
2,1E-4  double DoNLL(bool, unsigned int, unsigned int, Data
1,9E-4  Double_t PdfGaussian::integral() const [{PdfGaussia
1,6E-4  Double_t AbsPdf::GetIntegral() [{AbsPdf.chk.pomp.cx
1,5E-4  void MPISystem::WorldReduce(void *, void *, int, int, i
1,4E-4  void AbsPdf::CacheIntegral(Bool_t) [{AbsPdf.chk.pom
6,1E-5  MPI_Datatype OMPISystem::GetDatatype(int) [{OMPIS
5,9E-5  MPI_Op OMPISystem::GetOp(int) [{OMPISystem.chk.p
5,8E-5  Bool_t AbsPdf::IsExtended() const [{AbsPdf.chk.pomp
```

Figure 2: TAU graphical user interface screenshot showing the precise time fractions of any function in the program, included TAU overhead, in a common runtime scenario with 100k events and 12 MPI workers.

5 Testing the prototype

MLFit prototype is tested in different scenarios: first with only OpenMP or MPI working demonstrating good scalability with both technologies, then in the hybrid scenario with both OpenMP and MPI at the same time. Very interesting results come out of this last series of tests, since MPI proves to be more effective than OpenMP when scaling to multi-socket systems.

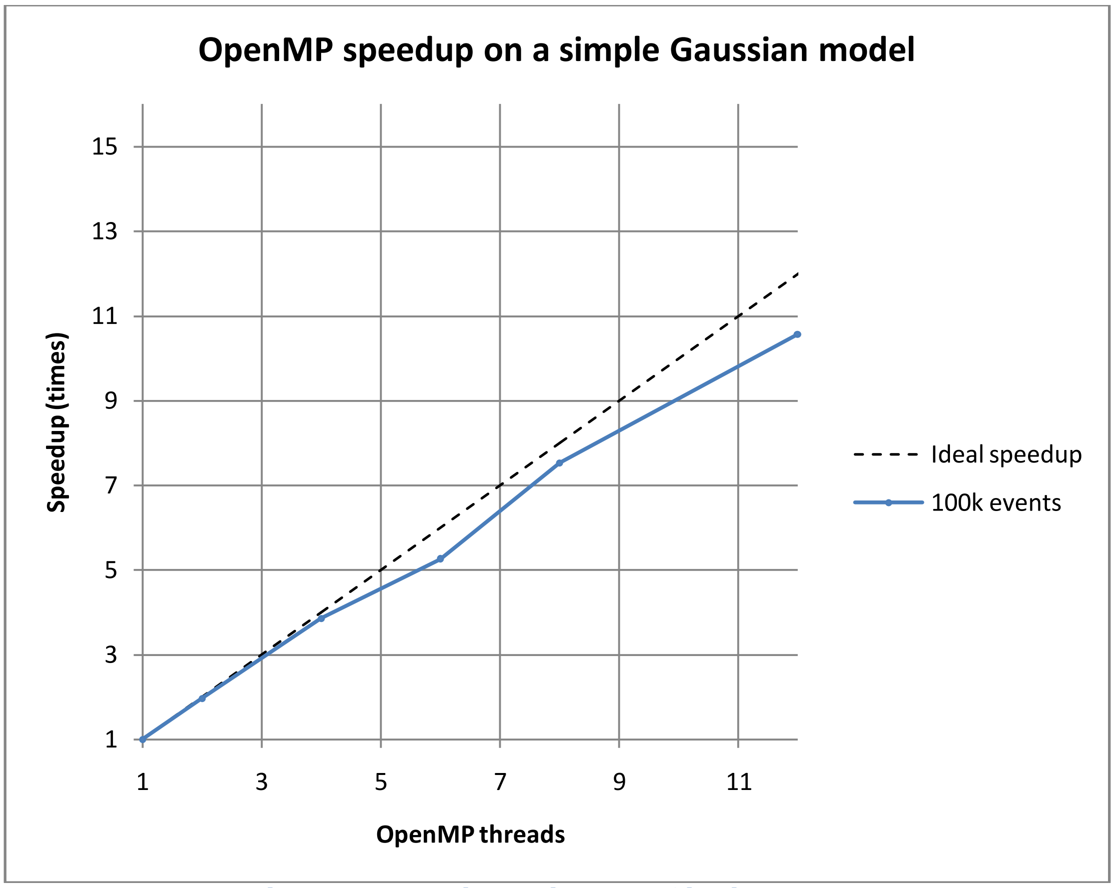
The hardware where the application has been tested is a dual socket Intel system with Westmere-EP architecture; each CPU is a 6 cores SMT Xeon running at 2.67 GHz. All the tests run with 100,000 input events with a simple PDF: a Gaussian distribution. In order to have a good amount of workload each program run repeats 10.000 times the whole evaluation.

It should be pointed that the system has only 12 physical cores, but I expect best performance with 24 parallel workers because of SMT capabilities of the CPU.

5.1 Running with OpenMP

Here I present OpenMP scalability results. MLFit prototype is compiled with both OpenMP and MPI, but no MPI parallelization is performed (process number is fixed to 1), while OpenMP thread number is varying to measure only OpenMP scalability.

Graph 1 shows the results obtained. OpenMP achieves good scalability, up to 10.6x with 12 cores involved.

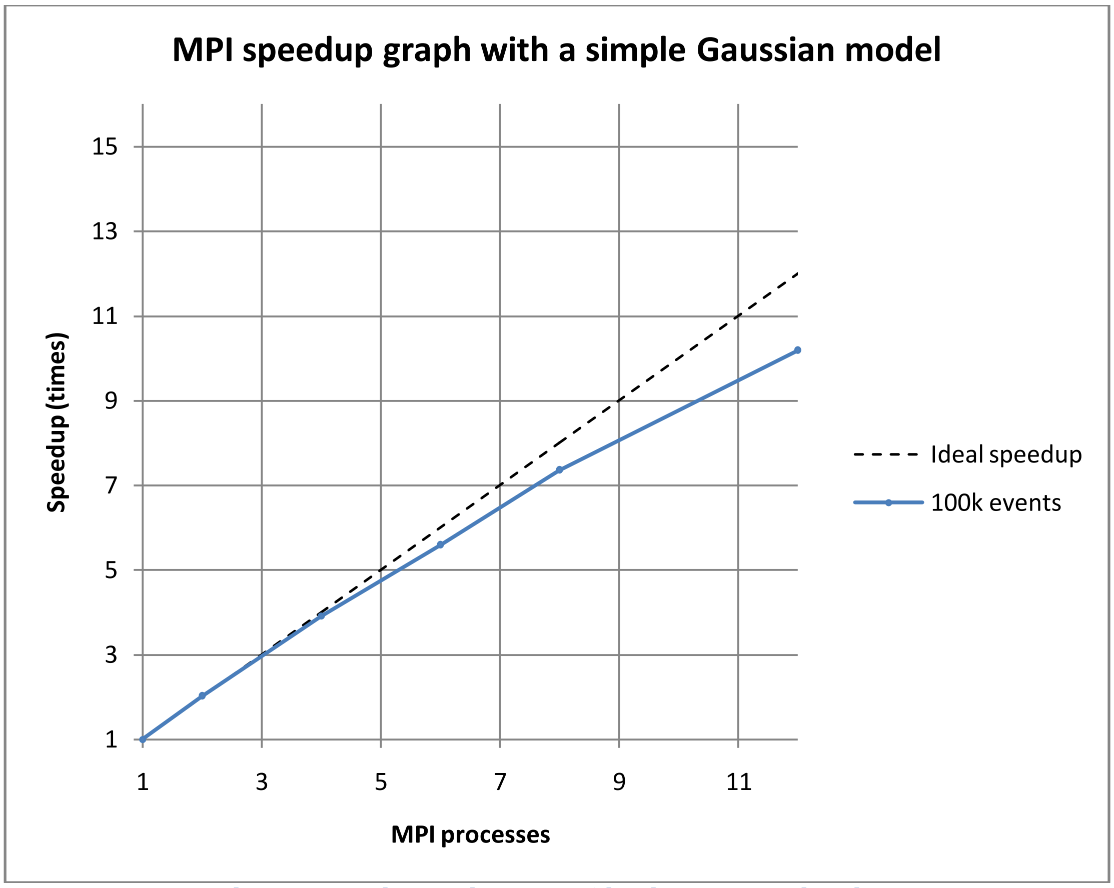


Graph 1: OpenMP speedup graph test run with only 1 MPI process

5.2 Running with MPI

In this section I present MPI scalability results. MLFit prototype is compiled with both OpenMP and MPI in the same way used to measure OpenMP scalability, but now OpenMP thread number is fixed to 1 while MPI process number is varying.

Graph 2 shows the results obtained. MPI achieves good scalability, up to 10.2x with 12 cores involved. Compared to the OpenMP, MPI is slightly slower when comparing mean times. Although, if we compare minimum execution time in each scenario, MPI is performing slightly better than OpenMP. The bigger variance in MPI execution time is probably due to the loopback network communication, which requires more operating system support.



Graph 2: MPI speedup graph test run with only 1 OpenMP thread

5.3 Running with OpenMP and MPI

In this section I present results obtained in the hybrid scenario, with both OpenMP and MPI running at the same time. In order to find the best balance between the two technologies, I fix the number of total parallel workers to K varying the number of MPI workers, P, and the number of OpenMP threads, T, under the constraint that $P \cdot T = K$.

Since the system has 12 physical cores with SMT, two series of tests are shown with a total of $K = 12$ and $K = 24$ parallel workers.

The execution time results of $K = 12$ tests are shown in Table 1. Here the best compromise is obtained with 2 MPI processes spawning 6 OpenMP threads each. This result is perfectly understandable considering of the topology of the system: two CPU, 6 cores each, with shared L3 cache able to contain all the input data. Since OpenMP is especially minded to work in shared memory, it's exploiting the L3 cache better than MPI that needs network communication.

MPI	OpenMP	Execution time (s)	Best execution time (s)
1	**12**	3.00 ± 0.031	2.90
2	**6**	2.97 ± 0.030	2.91
3	**4**	3.04 ± 0.176	2.94
4	**3**	3.60 ± 0.464	2.93
6	**2**	3.18 ± 0.398	2.93
12	**1**	3.09 ± 0.223	2.93

Table 1: Execution times with both OpenMP and MPI. A total of 12 parallel workers are active in each test, running with 100k events and 10k program repetitions. The best results

are achieved in a hybrid scenario where OpenMP is dominating and have the same number of threads as the number of cores per CPU.

The execution time results of $K = 24$ tests are shown in Table 2. In this scenario the best performance possible within this system is achieved, since all the SMT units are fully exploited with a 21,4% speedup from the $K = 12$ runs.

The best compromise is achieved by 12 MPI processes spawning 2 OpenMP threads each. When enabling SMT, MPI is performing much better than OpenMP.

MPI	OpenMP	Execution time (s)	Best execution time (s)
1	**24**	4.85 ± 1.40	2.29
2	**12**	4.40 ± 0.97	2.55
3	**8**	2.91 ± 0.67	2.23
4	**6**	2.77 ± 0.58	2.14
6	**4**	2.63 ± 0.32	2.15
8	**3**	2.61 ± 0.93	2.12
12	**2**	2.34 ± 0.15	2.10
24	**1**	4.19 ± 0.12	4.05

Table 2: Execution times with both OpenMP and MPI. A total of 24 parallel workers are active in each test exploiting Xeon SMT capabilities, running with 100k events and 10k program repetitions. The best results are again achieved in a hybrid scenario where MPI is dominating.

6 Conclusions and future work

Tests show that MPI on the top of OpenMP, other than allowing to run the application on a cluster system, is giving a boost to overall performance. In the future different hardware solution can be tested. Currently, other than the Xeon platform, MLFit has been also tested only on the Intel Single-chip Cloud Computer research processor on the top of RCCE library, achieving very good performance and scalability.

Much bigger speedups can be imagined in Microserver solutions, where an high number of low-power local CPUs are interconnected by an higher latency PCIExpress bus, or in distributed memory systems.

Appendix A: MPISystem class details

```
class MPISystem {
protected:
    MPISystem(bool DoInit=false);
    MPISystem(const MPISystem &Ref);
    virtual ~MPISystem();

    int m_WorkerId;
    int m_WorkerNum;
    int m_RootId;
    std::mpibuf *m_MpiBuf;
    std::streambuf *m_OldBuf;
    bool m_EnableOutputHanging;
    int m_OutputMode;

    virtual void DoInit() {}
    virtual void DoFinalize() {}
    virtual bool DoIsInitialized() const {return false;}
    virtual int DoGetWorldRank() const {return 0;}
    virtual int DoGetWorldSize() const {return 1;}
    virtual void DoWorldBarrier() const {}
    virtual void DoWorldBcast(
        char *Buf,
        int Size,
        int Datatype,
        int Root
    ) {}
    virtual void DoWorldReduce(
        char *Send,
        char *Receive,
        int Size,
        int Datatype,
        int Operation,
        int Root
    ) {}
    virtual void DoWorldAllreduce(
        char *Send,
        char *Receive,
        int Size,
        int Datatype,
        int Operation
    ) {}
    virtual double DoWtime() const {return 0.0;}
    virtual double DoWtick() const {return 0.0;}

public:
    static MPISystem *Instance();

    // main
    void Init(
        bool EnableOutputHanging=true,
        int OutputMode=MPI_PRINTMODE_ONLYROOT
    );
    void Shutdown();
    void Sync() const;
    bool Initialized() const;
    int GetWorkerId() const;
    int GetWorkerNum() const;
    int GetRootId() const;

    // io functions
    void OrderedPrint(const char *Buffer);
    void RootPrint(const char *Buffer);

    // cooperative work functions
    void WaitRootEvent(int NewRootId);
    void ChangeRootWorker(int DestWorkerId);
    void NextRootWorker();

    // mpi-calls
    void WorldBarrier() const;
    void WorldBcast(
        void *Buf,
        int Size,
        int Datatype,
        int Root
    );
    void WorldReduce(
        void *Send,
        void *Receive,
        int Size,
        int Datatype,
        int Operation,
        int Root=-1
    );
    void WorldAllreduce(
        void *Send,
        void *Receive,
        int Size,
        int Datatype,
        int Operation
    );
    double Wtime() const;
    double Wtick() const;

    // specific functions
    int GetThreadElements(
        int TotalEvents,
        int &Start,
        int &End
    );
};
```

Appendix B: Adding support to OpenMPI

```
    // the purpose of this macro is shutting down OMPI
    #ifdef ENABLE_OMPI
        #define OMPISafeCall(p)     p
        #define OMPIElse(p)
    #else
        #define OMPISafeCall(p)
        #define OMPIElse(p)         p
    #endif

    class OMPISystem: public MPISystem {
        friend class MPISystem;
    protected:
        OMPISystem();
        OMPISystem(const OMPISystem &Ref);
        ~OMPISystem();

        MPI_Op GetOp(int Operation);
        MPI_Datatype GetDatatype(int Datatype);

        void DoInit();
        void DoFinalize();
        bool DoIsInitialized() const;
        int DoGetWorldRank() const;
        int DoGetWorldSize() const;
        void DoWorldBarrier() const;
        void DoWorldAllreduce(
            char *Send,
            char *Receive,
            int Size,
            int Datatype,
            int Operation
        );
        void DoWorldReduce(
            char *Send,
            char *Receive,
            int Size,
            int Datatype,
            int Operation,
            int Root
        );
        void DoWorldBcast(
            char *Buf,
            int Size,
            int Datatype,
            int Root
        );
        double DoWtime() const;
        double DoWtick() const;
    };


    OMPISystem::OMPISystem() : MPISystem() {
        Init();
    }

    OMPISystem::OMPISystem(const OMPISystem &Ref) : MPISystem(Ref) {
        Init();
    }

    OMPISystem::~OMPISystem() {
        Shutdown();
    }

    inline MPI_Op OMPISystem::GetOp(int Operation) {
        MPI_Op MPIOp;
        switch (Operation) {
            case MPI_OP_MAX:
                MPIOp=MPI_MAX;
                break;
            case MPI_OP_SUM:
                MPIOp=MPI_SUM;
                break;
            default:
                LOG("Invalid operation");
                exit(1);
                break;
        }
        return MPIOp;
    }

    inline MPI_Datatype OMPISystem::GetDatatype(int Datatype) {
        MPI_Datatype MPIDt;
        switch (Datatype) {
            case MPI_DATATYPE_INT:
                MPIDt=MPI::INT;
                break;
            case MPI_DATATYPE_FLOAT:
                MPIDt=MPI_FLOAT;
                break;
            case MPI_DATATYPE_DOUBLE:
                MPIDt=MPI_DOUBLE;
                break;
            default:
                LOG("Invalid data type");
                exit(1);
                break;
        }
        return MPIDt;
    }

    void OMPISystem::DoInit() {
        OMPISafeCall(
            MPI::Init();
        )
    }

    void OMPISystem::DoFinalize() {
        OMPISafeCall(
            MPI::Finalize();
        )
    }

    bool OMPISystem::DoIsInitialized() const {
        OMPISafeCall(
            return MPI::Is_initialized();
        )
        return false;
    }

    int OMPISystem::DoGetWorldRank() const {
        OMPISafeCall(
            return MPI::COMM_WORLD.Get_rank();
        )
        return 0;
    }

    int OMPISystem::DoGetWorldSize() const {
        OMPISafeCall(
            return MPI::COMM_WORLD.Get_size();
        )
        return 1;
    }

    void OMPISystem::DoWorldBarrier() const {
        OMPISafeCall(
            MPI::COMM_WORLD.Barrier();
        )
    }

    void OMPISystem::DoWorldAllreduce(char *Send, char *Receive, int Size,
                                      int Datatype, int Operation) {
        MPI_Op Op = GetOp(Operation);
        MPI_Datatype Dt = GetDatatype(Datatype);
        OMPISafeCall(
            MPI::COMM_WORLD.Allreduce(Send, Receive, Size, Dt, Op);
        )
    }

    void OMPISystem::DoWorldReduce(char *Send, char *Receive, int Size,
                                   int Datatype, int Operation, int Root) {
        MPI_Op Op = GetOp(Operation);
        MPI_Datatype Dt = GetDatatype(Datatype);
        OMPISafeCall(
            MPI::COMM_WORLD.Reduce(Send, Receive, Size, Dt, Op, Root);
        )
    }

    void OMPISystem::DoWorldBcast(char *Buf, int Size,
                                    int Datatype, int Root) {
        MPI_Datatype Dt = GetDatatype(Datatype);
        OMPISafeCall(
            MPI::COMM_WORLD.Bcast(Buf, Size, Dt, Root);
        )
    }

    double OMPISystem::DoWtime() const {
        OMPISafeCall(
            return MPI_Wtime();
        ) OMPIElse (
            return 0.0;
        )
    }

    double OMPISystem::DoWtick() const {
        OMPISafeCall(
            return MPI_Wtick();
        ) OMPIElse (
            return 0.0;
        )
    }
```

Appendix C: Adding support to RCCE

```
// the purpose of this macro is shutting down RCCE
#ifdef ENABLE_RCCE
    #define RCCESafeCall(p)     p
    #define RCCEElse(p)
#else
    #define RCCESafeCall(p)
    #define RCCEElse(p)         p
#endif

class RCCESystem: public MPISystem {
    friend class MPISystem;
protected:
    int m_State;

    RCCESystem();
    RCCESystem(const RCCESystem &Ref);
    ~RCCESystem();

    int GetOp(int Operation);
    int GetDatatype(int Datatype);
    int GetDatatypeSize(int Datatype);

    void DoInit();
    void DoFinalize();
    bool DoIsInitialized() const;
    int DoGetWorldRank() const;
    int DoGetWorldSize() const;
    void DoWorldBarrier() const;
    void DoWorldAllreduce(
        char *Send,
        char *Receive,
        int Size,
        int Datatype,
        int Operation
    );
    void DoWorldReduce(
        char *Send,
        char *Receive,
        int Size,
        int Datatype,
        int Operation,
        int Root
    );
    void DoWorldBcast(
        char *Buf,
        int Size,
        int Datatype,
        int Root
    );
    double DoWtime() const;
};

RCCESystem::RCCESystem() : MPISystem() {
    m_State = 0;
    Init();
}

RCCESystem::RCCESystem(const RCCESystem &Ref) : MPISystem(Ref) {
    m_State = 0;
    Init();
}

RCCESystem::~RCCESystem() {
    m_State = 0;
}

inline int RCCESystem::GetDatatypeSize(int Datatype) {
    int Bytes = 0;
    switch (Datatype) {
        case MPI_DATATYPE_INT:
            Bytes=sizeof(int);
            break;
        case MPI_DATATYPE_FLOAT:
            Bytes=sizeof(float);
            break;
        case MPI_DATATYPE_DOUBLE:
            Bytes=sizeof(double);
            break;
        case MPI_DATATYPE_LONGLONG:
            Bytes=sizeof(long long);
            break;
        default:
            exit(1);
            break;
    }
    return Bytes;
}

inline int RCCESystem::GetOp(int Operation) {
    int RCCEOp;
    switch (Operation) {
        case MPI_OP_MAX:
            RCCEOp=RCCE_MAX;
            break;
        case MPI_OP_SUM:
            RCCEOp=RCCE_SUM;
            break;
        default:
            exit(1);
            break;
    }
    return RCCEOp;
}

inline int RCCESystem::GetDatatype(int Datatype) {
    int RCCEDt;
    switch (Datatype) {
        case MPI_DATATYPE_INT:
            RCCEDt=RCCE_INT;
            break;
        case MPI_DATATYPE_FLOAT:
            RCCEDt=RCCE_FLOAT;
            break;
        case MPI_DATATYPE_DOUBLE:
            RCCEDt=RCCE_DOUBLE;
            break;
        case MPI_DATATYPE_LONGLONG:
            RCCEDt=RCCE_LONG;
            break;
        default:
            exit(1);
            break;
    }
    return RCCEDt;
}

extern int *_argc;
extern char ***_argv;

void RCCESystem::DoInit() {
    RCCESafeCall(
        RCCE_init(_argc, _argv);
        RCCE_barrier(&RCCE_COMM_WORLD);
    )
    m_State = 1;
}

void RCCESystem::DoFinalize() {
    RCCESafeCall(
        RCCE_finalize();
    )
    m_State = 0;
}

bool RCCESystem::DoIsInitialized() const {
    return (m_State == 1);
}

int RCCESystem::DoGetWorldRank() const {
    RCCESafeCall(
        return RCCE_ue();
    ) RCCEElse (
        return 0;
    )
}

int RCCESystem::DoGetWorldSize() const {
    RCCESafeCall(
        return RCCE_num_ues();
    ) RCCEElse (
        return 1;
    )
}

void RCCESystem::DoWorldBarrier() const {
    RCCESafeCall(
        RCCE_barrier(&RCCE_COMM_WORLD);
    )
}

void RCCESystem::DoWorldBcast(char *Buf, int Size,
                              int Datatype, int Root) {
    int Bytes = GetDatatypeSize(Datatype);
    RCCESafeCall(
        RCCE_bcast(Buf, Size*Bytes, Root, RCCE_COMM_WORLD);
    )
}

void RCCESystem::DoWorldAllreduce(char *Send, char *Receive, int Size,
                                  int Datatype, int Operation) {
    int Op = GetOp(Operation);
    int Dt = GetDatatype(Datatype);
    RCCESafeCall(
        RCCE_allreduce(Send, Receive, Size, Dt, Op, RCCE_COMM_WORLD);
    )
}

void RCCESystem::DoWorldReduce(char *Send, char *Receive, int Size,
                               int Datatype, int Operation, int Root) {
    int Op = GetOp(Operation);
    int Dt = GetDatatype(Datatype);
    RCCESafeCall(
        RCCE_reduce(Send, Receive, Size, Dt, Op, Root, RCCE_COMM_WORLD);
    )
}

double RCCESystem::DoWtime() const {
    RCCESafeCall(
        return RCCE_wtime();
    ) RCCEElse (
        return 0.0;
    )
}
```

Appendix D: MPITimer class details

```
// MPI timer performs local and global measurements
// when MPI is enabled
class MPITimer: public BaseTimer {
protected:
    double m_StartTime;
    double m_TickFrequency;

public:
    MPITimer(int Type=TIMER_LOCAL) : BaseTimer(Type) {
        m_StartTime=m_TickFrequency=0.0;
    }
    void Reset() {
        // we must assure that MPI is inited before proceeding
        if (!MPISystem::Instance()->Initialized()) {
            std::cerr << "MPI is not inited!" << std::endl;
            exit(1);
        }
        switch (m_Type) {
            case TIMER_GLOBAL:
                // need to resynchronize workers
                // for getting reference time
                MPISystem::Instance()->Sync();
                break;
            case TIMER_LOCAL:
            default:
                break;
        }
        m_StartTime = MPISystem::Instance()->Wtime();
    }

    double Mark() {
        double Local = MPISystem::Instance()->Wtime() - m_StartTime;
        switch (m_Type) {
            case TIMER_GLOBAL: {
                // Mark local and allreduce to get
                // the max time of execution
                double Received=0.0f;
                MPISystem::Instance()->WorldAllreduce(
                    &Local,
                    &Received,
                    1,
                    MPI_DATATYPE_DOUBLE,
                    MPI_OP_MAX
                );
                return Received;
            }
            case TIMER_LOCAL:
            default:
                break;
        }
        return Local;
    }
};
```

Bibliography

1. *Evaluation of likelihood functions for data analysis on Graphics Processing Units.* **S. Jarp et al.** Anchorage : to be published on the proceeding of "The 25th IEEE International Parallel & Distributed Processing Symposium", 2011. proceeding of "The 25th IEEE International Parallel & Distributed Processing Symposium".

2. *Parallelization of maximum likelihood fits with OpenMP and CUDA.* **S. Jarp et al.** Taipei : to be published on Journal of Physics: Conference Series, CERN-IT-2011-009, 2010. proceeding of "The International Conference on Computing in High Energy and Nuclear Physics".

3. **Cowan, Glen.** *Statistical data analysis.* Oxford : Clarendon Press, 1998.

4. **Intel Corporation.** RCCE Specification. *Many-cores Applications Research Community.* [Online] http://communities.intel.com/community/marc?view=documents#/?page=2.

5. **Gamma E., Helm R., Johnson R., Vlissides J.** *Design patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison-Wesley, 1994. 0-201-63361-2.

6. **Meyers, S.** *Effective C++.* s.l. : Addison-Wesley, 2005.

7. **University of Oregon.** *TAU - Tuning and Analysis Utilities.* [Online] http://www.cs.uoregon.edu/research/tau/home.php.

8. **Pacheco, Peter S.** *Parallel programming with MPI.* San Francisco : Morgan Kaufmann Publishers, Inc., 1997.

9. **A. Lazzaro, L. Moneta.** MINUIT package parallelization and applications using the RooFit package. *J. Phys.: Conf. Ser. 219.* 2010. 042044.

10. **H. Albrecht et al. (ARGUS collaboration).** 1990. 278.

11. **B. Aubert et al.** B meson decays to charmless meson pairs containing η or η' mesons. *Phys. Rev.* 2009. Vol. D80, 112002.

12. **Chapman B., Jost G., Van der Pas R.** *Using OpenMP*. Cambridge, Massachussets and London : The MIT press, 2008.