Using a source-to-source transformation to introduce multi-threading into the AliRoot framework for a parallel event reconstruction

Stefan B. Lohn
CERN, CH-1211 Geneva 23, Switzerland
E-mail: stefan.lohn@cern.ch

Xin Dong
Northeastern University, Boston, Massachusetts 02115, USA
E-mail: xindong@neu.edu

Federico Carminati
CERN, CH-1211 Geneva 23, Switzerland
E-mail: federico.carminati@cern.ch

Abstract. Chip-Multiprocessors are going to support massive parallelism by many additional physical and logical cores. Improving performance can no longer be obtained by increasing clock-frequency because the technical limits are almost reached. Instead, parallel execution must be used to gain performance. Resources like main memory, the cache hierarchy, bandwidth of the memory bus or links between cores and sockets are not going to be improved as fast. Hence, parallelism can only result into performance gains if the memory usage is optimized and the communication between threads is minimized. Besides concurrent programming has become a domain for experts. Implementing multi-threading is error prone and labor-intensive. A full reimplementation of the whole AliRoot source-code is unaffordable.

This paper describes the effort to evaluate the adaption of AliRoot to the needs of multi-threading and to provide the capability of parallel processing by using a semi-automatic source-to-source transformation to address the problems as described before and to provide a straightforward way of parallelization with almost no interference between threads. This makes the approach simple and reduces the required manual changes in the code.

In a first step, unconditional thread-safety will be introduced to bring the original sequential and thread unaware source-code into the position of utilizing multi-threading. Afterwards further investigations have to be performed to point out candidates of classes that are useful to share amongst threads. Then in a second step, the transformation has to change the code to share these classes and finally to verify if there are anymore invalid interferences between threads.

1. Introduction
Introducing multi-threading to parallelize sequential source-code is a labor intensive challenge. Many techniques have been investigated to automatize the process of parallelization, in order to reduce the development effort and avoid error prone custom solutions. The GNU C/C++
Compiler [1] provides automatic parallelization of inner nested loops following the polyhedral model [2]. Compiler based parallelization uses the internal low-level source-code representation of a given program as available during the compilation process to facilitate optimization just before generating the final executable machine code. However, it prevents the user from intervening in the parallelization procedure and to customize inefficient automatic parallelization of certain loops. Besides, the parallelization becomes dependent on specific compilers. To keep compiler independence and to establish the possibility to intervene, a source-to-source transformation is the better choice. Frameworks like the rose compiler [3] are optimized for static analysis and source-to-source transformation. They aim to add thread managing primitives [4] like pragmas from OpenMP to introduce parallelism even for more abstract source-code with object-oriented features like the ones available with the C++ STL.

But full automatized approaches to parallelize source-code leaves much to be desired. Automatic parallelization only affect source-code that is proven to fulfill constraints of the respective transformation scheme. For instance the detection of data dependency has to follow a conservative strategy to avoid breaking probable dependencies and can refuse the parallelization of loops that are possibly be parallelizable. Another problem is that loops might be parallelized even if results are leading to performance degradation. Hence, most of these techniques are focused on computationally intensive tasks with processing intensive computations in inner nested loops.

But many large object-oriented frameworks have an extended execution path and intense numeric operations aren’t either efficiently located in one routine or they are displaced to become a sideline of the whole processing procedure. The objective of this approach is to analyze the performance of a specific semi-automatic general purpose parallelization that has been developed in the Geant4-MT project [5] and to point out issues and prospective solutions.

This paper has the following structure, first the former approach of the Geant4-MT project will be summarized in section 2. In section 3 necessary changes will be discussed and an overview of the application for ROOT and AliRoot will be presented. Section 4 will briefly go into concerns of correctness and section 5 summarizes first results of performance testing.

1.1. ALICE experiment
The ALICE experiment has to face computationally intensive tasks as they are common in HEP computing and especially heavy ion physics. During collision of either Proton-Proton (p-p) or Lead-Lead (Pb-Pb) particles big chunks of data, 100 MB/s and 1200MB/s respectively, are recorded for the subsequent track reconstruction and physical analysis.

The delivered processing capacities in certain computing centers are already below the requirements of the computing model [8] to reach the desired time for extracting physics results. Distributed computing is already exploited with the ALICE Grid and CERN Analysis Facility. But exploitation of processing power of the computer nodes is low due to new and unsupported hardware features like vectorization (SIMD), hardware threads (SMT) and Chip-Multiprocessing (CMP). This approach attempts to utilize the latter feature with multiple threads for a parallel execution to improve processing time, share data and hence reduce memory consumption as both are decisive factors to determine the computing model.

1.2. AliRoot framework
AliRoot [9] is a million lines of code framework, written in an object-oriented fashion in C++ and is based on the physics analysis framework ROOT [10]. ROOT is a huge framework that gives support for HEP experiments to analyze huge chunks of data. ROOT contains CInt, a C/C++ interpreter for fast prototyping, which delivers type information to ROOT objects to provide reflection information (reflex data).

Important tasks of AliRoot are simulation, track reconstruction and physical analysis. First
Parallelization efforts are described by Tadel [6] and for the Monte-Carlo simulation Geant4-MT by Dong [5] [7]. Since simulation is already addressed, this paper focuses on the parallel track reconstruction.

The input for the reconstruction is given in digits, as they usually come from simulation, or raw data, as they are used to be produced from real detector modules. The local reconstruction is specific for each detector module in the ALICE detector and contains tasks like cluster finding algorithms. Results are stored into one file per module, which contains the reconstruction points. These information are subsequently used to precisely discriminate the primary vertex, to perform the global track reconstruction and to carry out the particle identification. Final results are stored as event summary data (ESD), which can then be used for further physical analysis.

1.3. Former parallelization approaches

The parallelization attempt of Tadel shows the negative effect of a straightforward custom parallelization approach on scalability, caused by memory and I/O intensive operations. Additional, it demonstrates the labor intensity and the necessity to evaluate techniques for source-code re-engineering to reduce the required manpower of professionals in parallel programming. In this sense, the Geant4-MT approach of Dong seems to be an interesting solution. The idea of the Geant4-MT project is to investigate a general semi-automatic parallelization approach that can be used for a variety of sequential frameworks which rely on processing of independent event data to obtain event parallelism. This paper is presenting the parallelization techniques of the Geant4-MT project applied for the track reconstruction, which is here referred to as event reconstruction.

2. Semi-Automatic parallelization

2.1. Overview

Semi-automatic parallelization seems to be a reasonable strategy [11] to deal with the diverging architectural design approaches of large scale software frameworks. Such kind of parallelization can be interactive [15] or uses specific manual intervention as described here. The Geant4-MT project developed a source-to-source transformation tool [7] to transform the 700k source lines of code framework Geant4 to obtain parallel and scalable source-code. The adaption is divided into several steps:

- **Recognition** of global and static declaration
- **Privatization** of detected (recognized) declarations
- **Sharing Memory** by sharing instances of classes with selected thread-specific member fields
- **Debugging** to detect data races and unintended memory access to shared instances

During recognition information about global and static declarations are collected. Global state in a program is a peril for multi-threading because it shares memory and is hence prone for race conditions. Privatization reduces the global state and makes those declaration thread private by using thread-local storage. Both, recognition and privatization, are parts of the Transformation of Thread-Safety (TTS) that intends to make source-code unconditionally thread-safe. The basic idea is to remove all relevant global and static declaration.

For this purposes the thread-local specifier [14] with its keyword _thread of the C99 language specification of C is used to provide fast and scalable access to thread specific content of plain old data types (PODs). Listing 1 shows how thread-local storage is used to privatize complex types such as classes. Classes and arrays of classes are transformed into a pointer, as done in line 6. Due to the missing capability of dynamic initialization of the C99 thread-local specifier, corresponding declarations will later be initialized in the head of each function, as shown in line
10. This kind of initialization is also known as lazy initialization. To avoid that every subsequent usage has to be rewritten, the pointer is assigned to a reference with the original name as shown in line 15.

In a next step, the Transformation for the Memory Footprint Reduction (TMR) is used to share memory and to reduce the memory consumption by explicitly returning memory to global state. The idea is to return memory of specific declaration back to global state if the content leaves unchanged during parallel processing. Because it is unlikely to find whole classes with constant content, single member fields can be redirected to thread-specific storage (thread-local storage) to allow a certain amount of thread specific class members. Fields of shared classes can either be accessed read-only during concurrent processing, which we refer to as relative read-only, or transitory if thread specific. The transformation according to the TMR is demonstrated in listing 2 for a corresponding header file. Basically four changes are needed if one field has to become transitory. The original declaration has to be uncommented as in line 6. An additional thread-local array of the same type for all instances that are going to be created has to be added as in line 1 to replace the missing member field. Now, each instance will get a contiguous thread id as index to the corresponding array field like in line 8. If the variable name in the corresponding implementation is used, it can be replaced by a #define directive, like: #define var var_array[class_id].

Since selection of member fields for candidates of transitory declaration is done by hand, a validation method to identify undesirable write access to the resting fields, which ought to be relative read-only, was introduced. An efficient way for analyzing shared data is to use the Memory Write Protection Tool (MWPT) to investigate if write access to shared memory occurs.

2.2. Debugging
The concept of MWPT is to start your adapted code with a single thread as inferior process, where every write access to shared memory must be detected. Remember that all shared memory has to be read-only during the parallel execution. Once the process is started all shared instances are allocated into a memory region that will be protected using the POSIX system call mprotect. Write access to these memory region would result into a segmentation violation. Just before the threaded execution is started, a second process, the superior, is attached as child process to return a backtrace with ptrace if a segmentation violation in the inferior has been detected. Additional, a signal handler in the inferior is configured to remove the protection of the memory
Figure 1. Basic shape of a source-to-source transformation. The existing code is parsed and the Abstract-Syntax-Tree is created. Subsequently a user defined static analysis is performed to collect information and initiate the final source-code rewriting to the privatized source-code.

region and to re-execute the same function again. This makes it possible to detect race conditions of contained data during a single threaded execution. Afterwards, the protection is enabled again and processing is continued to collect further unintended write accesses.

2.3. Source-code Rewriting with GCC and ELSA

The rewriting tool of the Geant4-MT project is divided into a GCC 4.2 compilation step with an augmented parser for the recognition stage and an ELSA based source-code rewriting for the privatization procedure. ELSA is an open source parser for C/C++, which is not maintained anymore. The token-stream delivered by ELSA is analyzed case-based to extract facts from source-code for rewriting purposes. The tool suite provides the following functionality:

(i) Return position of all global, static or extern declaration
(ii) Privatize selected declaration with the thread-local specifier
(iii) Add lazy initialization of global or static declarations in related files or classes

A negative side effects of the fragmentation between recognition and privatization is that the full source-code representation, the abstract-syntax-tree (AST), is not available during the rewriting of source-code. Hence, the following issues are observed:

- Misplaced/Missing lazy initialization in functions with access to global declarations
- Type names of POD declarations are hard coded
- Adapting nested-name-specifier in qualified identifiers failed, due to missing context (e.g. my_array[my_class::inner_class::size])

To fix these issues to bring more flexibility and to add more prospective features, the Clang front-end of the LLVM project is a good candidate with already existing examples in static source-code analysis and source-code rewriting.

3. Application to AliRoot source-code

3.1. Source-code Rewriting with Clang

Clang is an upcoming front-end of the LLVM compiler project [18]. To traverse the AST of Clang the RecursiveASTVisitor template is used. To install custom code for static analysis and
to initiate the rewriting process the DeclVisitor, StmtVisitor and TypeLocVisitor templates of Clang are used to visit specific types of nodes. This way information about declaration can be collected and source-code rewriting can be conducted during the AST keeps its information. For ROOT and AliRoot the source-code basis has turned out to be too heterogeneous than a case-based tool, as the one described in the previous section, could accomplish. Many mistakes have been corrected by hand. Using Clang for the rewriting process, the most important problems of the former tool suite are abolished. The additional abilities are:

- Extraction of positions of function definitions that contain access to global declaration
- In-build classification to distinguish types: POD, reference, pointer or array types
- Rewriting declaration names, replacing #define directive for the TMR transformation

For the rewriting process, as shown in figure 1, positioning information can be obtained directly from a node of the AST. To process further global or static declaration for privatization or to add corresponding lazy initialization into the head of a function definition, each change in code need to be registered in a position map. The position of subsequent demanded declaration can then be obtained by the original position of the AST plus the offset demanded from the position map.

3.2. Transformation of Thread-Safety
The TTS is fully automatic and is adapting only parts of AliRoot and ROOT. The interpreter CInt as such is not needed during reconstruction and will be shared as described in section 3.3. CInt is not able to become transformed according the TTS because of the usage of a source-code generator to create dictionaries with intensive use of global declarations. In total 1120 static (620 in ROOT and 500 in AliRoot) and 192 global declaration (150 in ROOT/11 in AliRoot) plus 77 extern declaration (46 in ROOT/31 in AliRoot) has been privatized and more than 2000 lines are added for lazy initialization. In comparison, not more than 200 lines are added manually to have thread-specific files for collecting intermediate results. After the TTS the selected source-code, involved during execution, is thread-safe.

The source-code contains ROOT containers, histograms, core files and others. For AliRoot all files in the STEER, ITS, TPC, some in the HLT and RAW sub-folders have been transformed. Files involved in reconstruction can be pointed out by generating a calligraph, e.g. with callgrind [12].

3.3. Memory Reduction of Footprint
Due to problems with CInt the testing candidate can still not run in parallel after the TTS. Fortunately the TMR can also be used to conduct some design changes to solve this problem. Remember, that AliRoot has to run multiple reconstruction threads, but since CInt is not transformed, the interpreter can only be initialized once. Hence, on the one side, each thread must own his own set of singletons for supervising a run (AliRun), configuring a run and it’s folder structure with containers for tasks like the reconstruction process (AliConfig), loading and accessing Condition Data Base (CDB) entries during reconstruction (AliCDBManager), accessing geometry (AliGeoManager) and steering the event reconstruction (AliReconstruction). On the other side ROOT contains the CInt interpreter interface (TInterpreter, TCInt), an I/O system based on shared reflex data (TStreamerInfo), base classes to add reflex data and ROOT I/O support (TClass, TObject, TClassTable) and a steering class for initialization and bookkeeping (TROOT), where some member fields of these classes must be shared, some can be shared and others must be excluded from parallel execution.

Since TROOT initializes CInt, it is necessary to keep TROOT a singleton. Normally the approach needs to make all instances to become thread-local. But the pattern of the TMR described in section 2.1 can be used to redirect a calling thread to thread-specific member fields
and therefore TROOT can become a singleton with diverging content for different threads. As result Clnt is initialized once along with the singleton TROOT, but AliRoot is able to start multiple reconstruction threads.

The following steps are needed to share the interpreter Clnt:

- Remove the thread specifiers of gClnt and gInterpreter, the pointers to the Clnt interface
- Exclude the execution of interpreting macros by pre-compiling them with ACliC
- Prepare the parallel execution by loading type information in advance

Even Clnt is shared, it is not possible to use the interpreter safely during multi-threaded execution. Since there is no need to execute the interpreter during reconstruction, this is not a limitation. Instead Clnt functionality has to be executed before- or after parallel processing. Invalid access to Clnt functionality can be found by data race detection methods, but have not been observed.

AliConfig utilizes the ROOT folder structure fRootFolder in TROOT. Since AliConfig will be instantiated per thread, the ROOT folder structure must be transformed into a transitory field. The necessary recreation of thread specific objects is done in related getters in TROOT. Further containers can be shared, if elements can be added on a thread-safe way, if their state leaves unchanged during parallel processing and are not removed before the last thread is terminated. Otherwise they have to become transitory. MWPT can be used to investigate access pattern to protected objects that are moved to the memory protected regions by changing the new operator to one with a specific address, which is returned by a static function in TROOT.

4. Correctness and Debugging

4.1. Correctness

The original source-code is taken as the underlying specification for the source-code transformation, which is presumed to be correct. Two important aspects of the transformation ensure that the adapted source-code behave according the specification. First, the control-flow during parallel processing has not changed in comparison to the original. Statics and globals became thread-local, but they still have the same scope, duration and linkage. Second, all shared memory is relative read-only to avoid unintended crosstalk amongst threads. For ideal source-code, the transformed program would follow the same data-flow as the original program if processed on the same set of data. Unfortunately real source-code violates this approach if untransformed source-code is involved, e.g. in external libraries or generated source-code. For ROOT and AliRoot it is not feasible to transform all external libraries. For instance some functions of the POSIX.1 core services are non-reentrant and returning references to same memory locations. Additional, Clnt is generating source-code to setup reflex data and summarizes type information in dictionaries.

4.2. Data races and shared memory access

A violation of correctness can be a race condition in a simultaneous call of a non re-entrant routine, a false classified read-only data member that is shared amongst threads or a missing privatized global declaration. To validate these issues, dynamic debugging and data race detection tools can be used. Unfortunately they have to be executed frequently and slow down the development process. To point out if variable content is accidentally shared as read-only declarations, related instances can be allocated using the MWPT tool to investigate access pattern and to find data races on allocated read-only memory more efficient.

Objects that are going to be shared can be allocated into the protected part of the heap. It makes sense to keep this protection during parallel runs to detect all further inappropriate write accesses. Since validation is done for specific test cases dynamically it is possible that some issue
might remain unseen or could be added by additional user code. That’s why it is important to keep this kind of protection to assure correctness during usual production runs.

5. Performance analysis

5.1. Test Setup

For testing a machine with two Intel Westmere 2.6 GHz processors with six cores each, 12 MB of 3rd level cache and 24 GB of main memory has been chosen. The current processing is limited to ITS as detector module. Other detectors and quality assurance have been switched off. The test cases are derived from AliRoot ppbench and PbPbbench examples and is reconstructing tracks from raw data generated by Geant3.

The event reconstruction is controlled by a pre-compiled macro that is executing the following steps:

(i) Sequential initialization:
   (a) Loading type information in advance (reflex data).
   (b) Extracting event information from file for each thread.
   (c) Pushing event data onto a queue. (dyn. load balancing)

(ii) Parallel execution:
   (a) Spawn multiple threads.
   (b) Reinitialize global declarations.
   (c) Start initialization of reconstruction.
   (d) Get event information from the event queue. (dyn. load balancing)
   (e) Execute event reconstruction.
   (f) Exit thread.

(iii) Merging results.

(iv) Cleanup and Termination.

During sequential initialization, reflex data have to be loaded first (i.a). Then, the event data tree has to be loaded from file and cloned to guarantee distinct objects for each thread (i.b). Event numbers are stored onto a queue (i.c) for a simple dynamic load balancing. Afterwards, threads are spawned (ii.a) and parallel initialization starts. The global objects are reinitialized (ii.b) in the same way as done in the TROOT constructor and then the configuration and initialization of the reconstruction process (ii.c) is carried out. Afterwards, each thread requests an event number from queue (ii.d) and event reconstruction starts processing (ii.e). After the threads are terminated, results must be merged (iii) and the program quit (iv).

| threads | static (Vir/Res) | full (Vir/Res) |
|---------|------------------|----------------|
| 0       | 591/138          | 591/138        |
| 1       | +164/+132        | +164/+132      |
| 4       | +22/+18          | +137/+112      |
| 8       | +18/+14          | +133/+108      |

200 pp events

| threads | static (Vir/Res) | full (Vir/Res) |
|---------|------------------|----------------|
| 0       | 591/138          | 591/138        |
| 1       | +452/+429        | +452/+429      |
| 4       | +199/+185        | +239/+215      |
| 8       | +146/+133        | +203/+183      |

20 Lead-Lead events

Table 1. Shows memory allocation for one more additional opened file of raw event data per thread respectively for static and dynamic load balancing. With 0 threads, no file is opened and no reconstruction but the initialization was running.
5.2. Speedup of the PP and PbPb event reconstruction

As shown in figure 2 the total speedup of a p-p event reconstruction grew only to 3.5 with 8 threads. The parallel processing takes around 38 seconds instead of 132 seconds in sequential mode. Meanwhile the speedup of the concurrent event processing rose to 5. The limitation of speedup during the parallel run has two major reasons. First, the running algorithms have not been optimized by this approach and might be inefficient as figure 3 indicates. Running local reconstruction, the task with the most intensive I/O operations, in parallel leads to performance degradation. Global tracking seems not to have performance issues till 8 threads and vertexing is scalable till the 7th thread. Second, the parallel initialization is not yet optimal and needs to be locked for certain cases. For instance if information are loaded from a file by multiple threads has to be serialized.

Results of the same test for simulated Lead-Lead collisions are shown in figure 2. Because of an extended runtime, 107 minutes for reconstructing tracks of 20 events, the serialized preparation for parallel processing does not fall into account. Total speedup and speedup of the concurrent event processing are close together. With 8 threads 19.1 minutes are needed resulting in a speedup of 5.6.

Runs with a short event processing time have shown a parallel overhead of more than 10%. The overhead is nearly compensated for Lead-Lead runs with a long processing time per event. Since the reconstruction in sequential mode contains at least a main and a worker thread this penalty is payed for resting mutual exclusions in the ROOT framework, e.g. in TStorage.

5.3. Memory footprint of the event reconstruction

The memory footprint of the ppbench test case with a single thread has around 600 MB resident and 770 MB virtual memory. Each additional thread is allocating 260 MB resident and 400 MB virtual memory. A single threaded process of the PbPbbench example utilizes 1040 MB resident and 1460 MB virtual memory. One additional thread is allocating 660 MB resident and 800 MB virtual memory more. For each thread, additional memory is allocated to open a further set of files and the retrieved objects. Table 1 shows additional memory allocated for another opened file of raw events and its corresponding TTree. Since 203 MB of virtual memory is used for another opened full (all 20 events) event file of Lead-Lead data, only 600 MB are allocated.
by the reconstruction algorithms. 
Per reconstruction thread around 100 MB are saved by sharing CInt. The rest of the saved memory comes from the fact, that additional opened files do allocate less additional memory, as it is described in table 1 for opening multiple files of raw events. Static and dynamic load balancing can be used, where in the static case the raw event data file is divided into $n$ parts for $n$ threads and hence needs less memory (see Figure 5) than the whole file of raw events for dynamic load balancing, but has an extended runtime of up to 8% for a Lead-Lead reconstruction job of 20 events. Each thread needs his own file of raw events in the dynamic case, since otherwise resources between threads would be shared, what violates the approach. 
Figure 4 demonstrates how speedup is developing in comparison to the ratio of memory consumption in the $i$-th and the first thread. More and more memory is saved until the 5th thread, when the speedup does not scale anymore.

6. Conclusion
The transformation is following simple rules. It can transform a framework within days. However, transforming large-scale frameworks like AliRoot and ROOT turned out to be difficult. Many issues have been fixed by using the Clang based rewriting tool. For source-code generating software, like the CInt interpreter, an appropriate automatic adaption is not feasible. Similar, access to thread-unsafe external libraries needs further manual intervention and reduces, in most cases, performance and scalability because of necessary synchronization. Unfortunately, both negative effects reduces the applicability to such a transformation for the production process in ALICE with regular stable releases.

On the other side, the approach has still shown its value. Privatized data do not need to be protected by mutual exclusion and hence synchronization time is low during parallel event processing. Other issues that come from the utilized algorithms can be observed by this prototype and their efficiency can still be improved.

The TMR is currently used for a decent architectural redesign to exclude CInt functionality during parallel processing. But further memory can be shared, as for instance detector geometry or data from the CDB.
Acknowledgement

This work was supported by the Wolfgang-Gentner-Programme of the ”Bundesministerium für Bildung und Forschung“ (BMBF).

References

[1] K. Trifunovic et al., ”GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation“, ALCHEMY - INRIA Saclay - Ile de France , IBM Thomas J. Watson Research Center , Xi’an Jiaotong University [Chine] , Department of Informatics and Mathematics, Programming Group - FIM , Compiler Performance Engineering [Austin] , IBM Research [HAIFA] , Advanced Micro Devices [Austin] - AMD, GCC Research Opportunities Workshop (GROW’10), 2010
[2] M. Le Fur, J.-L. Pazat, F. Andr, An array partitioning analysis for parallel loop distribution, EURO-PAR ’95 Parallel Processing, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, P. 351-364, vol. 966, 1995
[3] Quinlan, D., Schordan, M. Philip, B. Kowarschik, M. The Specification of Source-To-Source Transformations for the Compile-Time Optimization of Parallel Object-Oriented Scientific Applications, Submitted to Parallel Processing Letters, also in Proceedings of 14th Workshop on Languages and Compilers for Parallel Computing (LCPC2001), Cumberland Falls, KY, August 1-3 2001
[4] L. Chunhua, D. Quinlan, J. Willecock, T. Panas, Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, International Journal of Parallel Programming, Springer Netherlands, P. 361-378, volume 38, issue 5, 2010
[5] X. Dong, G. Cooperman, J. Apostolakis, Multithreaded Geant4: Semi-automatic Transformation into Scalable Thread-Parallel Software, College of Computer Science, Northeastern University, Boston, Euro-Par 2010 - Parallel Processing, Lecture Notes in Computer Science, Springer Berlin Heidelberg, P. 287-303, volume 6272, 2010
[6] M. Tadel, F. Carminati, Parallelization of ALICE simulation - a jump through the looking-glass, Journal of Physics: Conference Series, volume 219, number 3, P. 032024, 2010
[7] X. Dong, Source-Level Transformation of Legacy Sequential Programs into Scalable Thread-Parallel Code, Northeastern University, PhD proposal, 2011
[8] F. Carminati, Y. Schutz et al. Technical Design Report of the Computing, CERN, Geneva, 2005
[9] R. Brun, P. Buncic, F. Carminati, A. Morsch, F. Rademakers, K. Safarik, Computing in ALICE, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, Volume 502, Issues 23, 21 April 2003, Pages 339-346, ISSN 0168-9002
[10] R. Brun, F. Rademakers, ROOT: An object oriented data analysis framework, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment”, vol. 389, nr. 12, P. 81-86, Elsevier, 1997
[11] Pankratius, V. Schaefer, C. Jannesari, A. Tichy, W. F. Tichy, Software engineering for multicore systems: an experience report, Proceedings of the 1st international workshop on Multicore software engineering, IWMSE ’08, 2008, ACM
[12] J. Weidendorfer, Sequential Performance Analysis with Callgrind and KCachegrind, in Tools for High Performance Computing, P.93-113, Springer Berlin Heidelberg, 2008
[13] T. Grosser, H. Zheng, R. Aloor et al., Polly - Polyhedral optimization in LLVM, First International Workshop on Polyhedral Compilation Techniques (IMPACT11), Chamonix, France, Apr. 2011.
[14] U. Drepper, ELF Handling for Thread-Local Storage, Red Hat, 2003
[15] Adam R. Smith, Prasas A. Kulkarni, Localizing globals and statics to make C programs thread-safe, Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, P. 205-214, CASES ’11, ACM, New York, 2011
[16] N. Nethercote and J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, SIGPLAN Not., vol. 42, nb. 6. P. 89-100, ACM, New York, USA, 2007
[17] M. Buss, S.A. Edwards, Bin Yao and D. Waddington, Pointer analysis for source-to-source transformations, Fifth IEEE International Workshop on Source Code Analysis and Manipulation, P.139-148, 2005
[18] Chris Lattner, Vikram Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, Proceedings of the International Symposium on Code Generation and Optimization (CGO’04), Palo Alto, California, 2004