ADVANCES WITH MERLIN - A BEAM TRACKING CODE

J. Molson, H. Owen, A. M. Toader, R. J. Barlow
Manchester University, UK and the Cockcroft Institute, UK

Abstract

MERLIN is a highly abstracted particle tracking code written in C++ that provides many unique features, and is simple to extend and modify. We have investigated the addition of high order wakefields to this tracking code and their effects on bunches, particularly with regard to collimation systems for both hadron and lepton accelerators. Updates have also been made to increase the code base compatibility with current compilers, and speed enhancements have been made to the code via the addition of multi-threading to allow cluster operation on the grid. The developments allow for simulations with large numbers of particles to take place. Instructions for downloading the new code base are given.

INTRODUCTION

Merlin is a beam tracking code developed in C++ by N. Walker et al. [1, 2]. It is easy to extend due to its modular process nature, and the code is clean structured C++. Merlin exists as a set of library functions, where one writes one's own simulation program and makes use of this provided simulation system. This provides great flexibility in what the code can do, as demonstrated in the example files available in the Merlin distribution. We have taken responsibility for developing and maintaining the code, and have added several new features and enhancements.

CODE IMPROVEMENTS

Many improvements have been made to the Merlin code base. We have implemented proton scattering in collimation systems, enhanced the resistive wakefield processes, and have given the code base a major speed increase. In addition many pure code alterations have been made. We have brought the code base up to date to be compatible with current compilers, and now use a modern version control system (SourceForge) to host the code and track changes.

Scattering Physics

Scattering physics for protons has been added to the code. These scattering processes include: multiple coulomb, elastic proton-nucleus, inelastic proton-nucleus, elastic proton-nucleon, quasi-elastic single diffractive proton-nucleon, and Rutherford scattering. Enhancements to the proton scattering physics in collimators are discussed in detail in a separate paper [3].

Resistive Wakefield Enhancements

Previous versions of Merlin assumed a fixed charge for each macroparticle in the bunch [4]. We have added a new ParticleBunchQ class, which allows each macroparticle to have its own charge. This allows us to simulate core beam particles with a large charge with a halo of lower-charge particles. This will give a more accurate simulation of the effect of wakefields on halo particles, with the core beam charge producing the field that acts on the halo. In addition, the WakefieldProcess has been enhanced to work with MPI (see below), allowing transfers from other compute nodes. The collective wakefield from multiple systems is calculated, and the macrparticles on each node are updated.

Minor Changes

Collimator apertures had previously to be added to the Merlin input files (MAD XSIF format) by hand. These are now read instead from a user-defined collimator database file, allowing easy changes to collimator settings. There is also now a unified material class and a material database class: here materials objects are created, are filled with the relevant material properties and are then pushed back onto a C++ vector for easy access and searching. Since the material data does not change between runs this information is held within the source code itself, instead of in an external configuration file; given the correct material properties and cross sections, it is now trivial to add new materials to Merlin.

We have implemented bunch load functions, allowing checkpoint features for long simulation runs, where the bunch must be saved and reloaded at a later time in the same state. We have tested the code base with gcc 4.5 builds to ensure compatibility with current compilers, and have eliminated many outstanding code warnings. Many minor code design and layout enhancements have also been made, and copious code comments added.

Symplectic Tracking

Since the code was initially developed for single pass linacs, symplecticity was not previously a design concern. However, for large multi-turn hadron machines such as the LHC it can become an issue; tests on an LHC lattice (V6.503 design optics) show that there is a small drift in emittance size over several million turns. A set of symplectic integrators have been developed for Merlin by A. Wolski [5], and will be implemented in the next release of Merlin.

INCREASING TRACKING SPEED

Improving tracking statistics is CPU-intensive. We have explored several methods to parallelise the computation, as described below.

GPUs: OpenCL and CUDA

The advent of high performance GPUs with hundreds of processors has reduced the cost of many highly parallel tasks (such as particle beam tracking), but to date only single-precision arithmetic is efficient. This problem is to be somewhat rectified with the new generation of Fermi processors from NVidia, where the double precision performance is up to 50% that of single precision. Other beam tracking codes have been implemented using single-precision GPUs, and have been shown give significant deviations compared to double-precision codes [6]. For collimation studies this difference is important, although our group has previously implemented a GPU-based tracking code based on NVidia's CUDA [7, 8, 9].

There are now several different APIs for GPU programming, most of which are vendor-dependent, such as CUDA. A general-purpose GPU API has been created: OpenCL (Open Compute Language) [10]: this will allow a compute "kernel" to be written once, and executed on any available compute element on a system, be that a CPU or GPU. OpenCL is not yet sufficiently mature, but with the advent of such a language and high speed double precision GPUs, there is the possibility of creating beam tracking integrators that can run on any GPU. For now, more standard compute methods have been used.

OpenMP

When using one physical machine, a common multi-threading technique is to use OpenMP. This is achieved via the addition of a #pragma definition in the source code. For the transport code, a map is applied to each particle in turn. This loop can be parallelized:

```
#pragma omp parallel for
for(size_t i = 0; i<bunch.size(); i++)
{
amap->Apply(bunch.GetParticles()[i]);
}
```

The compiler itself (gcc 4.3 for this work) will handle all parallelisation work, making this a trivial method to gain a considerable speed increase. The same method can be applied to other processes, such as collimation.

Using Computing Grids - MPI

For a further increase in speed, one can use multiple physical machines. The industry-standard communication method is Message Passing Interface (MPI).

We have implemented particle distribution routines into Merlin in order to split tracking over multiple computers.

Table 1: Scaling of OpenMP-enabled MERLIN with 10 laps of an LHC lattice, and 100k particles. Wakefields are enabled.

No. of Processor Cores	Time (seconds)
1	1067
2	738
3	632
4	569

Our first design decision was to only exchange particles where required. All tracking, collimation, and other independent processes will take place on individual CPU nodes, with particle exchange taking place for collective effects only. These include processes such as initial bunch creation, wakefield effects, emittance calculations, and space charge effects (not yet implemented in Merlin).

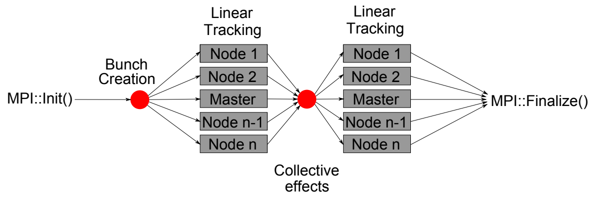

Figure 1: Illustrating of code branching in the MPI version of Merlin.

This does mean the limiting issue for any speed increases are any single-threaded collective effect algorithms.

MPI code is placed where required within existing Merlin functions, and a compile time define ENABLE_MPI allows selection of the different tracking methods. A new MPI datatype was created for transfer of particles; this is defined as MPI_PARTICLE. This is implemented as a structure, allowing the addition of extra types: currently it consists of 6 doubles for the 6d phase space coordinates (7 including the per-particle macrocharge). Addition of non-double types such as the particle type (electron, proton, etc) is possible due to this formalism. MPI send and receives require a continual block of memory to act as a buffer for transfers. The particle bunches in Merlin are C++ vector types, hence to be transferred, they must be first copied particle by particle into a suitable buffer array. They are then sent, and the receive buffer array is converted into a PSVector type (a single particle), which is then pushed back onto a ParticleBunch vector one by one.

When running on shared computing systems such as the grid (where there is no guarantee that the code has exclusive usage over the CPUs available) we found that execution was frequently held up waiting for calculations on a busy node to complete. To work around this problem, we implemented a load balancing system, where the time per particle on each node is measured between collective effects, and when particles are redistributed, this is done

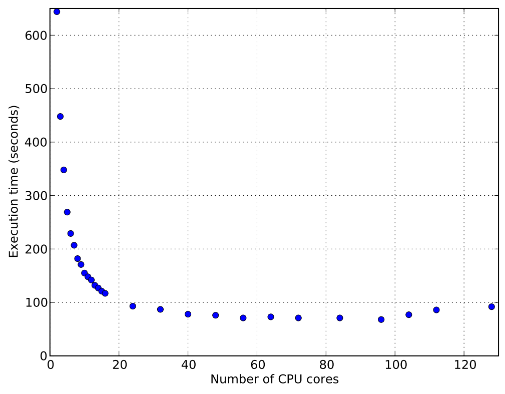

Figure 2: Variation of execution time for a test LHC tracking run versus number of CPU cores in the MPI version of Merlin.

according to the load on each CPU, with more heavily loaded CPUs receiving fewer particles. This also allowed increased performance on heterogeneous clusters, where the particles would be distributed according to individual CPU speed.

We have tested the scalability of the MPI code, and have found a limit corresponding to 60 processors as shown in Fig. 2. This is due to the remaining single threaded wakefield code, and the initial accelerator lattice loading.

When benchmarking the new MPI code base against the old, we found a small divergence in tracking results. We believe that the difference is that in the single-threaded version variables are implicitly held at 80bit due to the x87 FPU enhanced precision. In the MPI code, particles are transported between physical machines as doubles, hence an implicit precision truncation occurs. Differences can be avoided by forcing the x87 FPU to operate in 64bit precision mode, via the use of the fldcw assembly instruction (FPU load control word). We point out this difference to others considering using MPI as a solution. One can also use long doubles throughout, but these will result in increased computational time.

ACCESS AND DISTRIBUTION

The current release of the source code is available from SourceForge [11], and we actively encourage new developers to join the Merlin project. As part of our development efforts, we have switched to the git distributed version control system; this allows individual developers to make their own branches and track their own changes without modifying the main source tree.

FUTURE WORK

We intend to implement the symplectic integrators into Merlin and verify their precision. Lepton scattering will be fully implemented for future linac and light source studies. Scattering will be moved to be a function of a particle bunch, such that we can create arbitrary bunches of protons, electrons, muons, etc. We also plan to implement the effects of long range inter-bunch wakefields, on top of the short range intra-bunch resistive wakefields. The wakefield code is still the limiting factor in the speed of tracking, and enhancements can be made via attempting to parallelise the bunch slicing and kick calculations. General speed increases can also be found in the MPI code via increased use of non-blocking send and receive calls.

CONCLUSIONS

In conclusion, we have added new physics effects such as proton scattering in collimators, enhanced wakefield effects, and parallel tracking. Merlin is currently under active development and we welcome new developers.

ACKNOWLEDGEMENTS

We would like to thank Nick Walker and Andy Wolski for their huge efforts in the earlier versions of Merlin. Also, we thank CSED staff at STFC Daresbury Laboratory for providing computational resources.

REFERENCES

[1] http://www.desy.de/∼merlin/

[2] D. Kruecker, F. Poirier, N. Walker. Merlin-based start-to-end simulations of luminosity stability for the ILC. PAC2007.

[3] A.Toader et al. Simulations of the LHC collimation system (These proceedings).

[4] A. Toader and R. Barlow, Computation of Resistive Wakefields. PAC2009.

[5] A. Wolski. Private communication.

[6] G. Sterbini. An early separation Scheme for the LHC Luminosity Upgrade. PhD Thesis.

[7] 'High performance stream computing for particle beam transport simulations', R.B. Appleby et al, 2008 J. Phys.: Conf. Ser. 119 042001 (10pp)

[8] 'Beam dynamics using the stream processing code GPMAD', Appleby, Bailey and Salt, EuroTeV report 2008-022.

[9] http://developer.nvidia.com/object/gpucomputing.html

[10] http://www.khronos.org/opencl/

[11] http://sourceforge.net/projects/merlin-pt/