MERLIN Status Report

S. Rowan\textsuperscript{1}, S. Tygier\textsuperscript{2}, and R. Barlow\textsuperscript{1}

\textsuperscript{1}University of Huddersfield
\textsuperscript{2}University of Manchester

1 Recap

The previous report “Merlin Release with Hollow Electron Lens”, dated 28/2/2017, briefly discussed the recent implementation of hollow electron lens (HEL) functionality in MERLIN, developed by H. Rafique for his PhD thesis [1], as well as the requested integration of MERLIN with the Eclipse IDE. That version is considered MERLIN 5.01. In this report, the developers discuss recent improvements in performance, accessibility and maintainability aimed for MERLIN 5.02. Currently available versions of MERLIN are 5.01 and a beta release of 5.02.

2 MERLIN 5.02 - The Goal

It is noted that, in considering long-term scientific software development, accessibility, compatibility and maintainability are just as important as functionality and performance [2]. As a result, the goal for MERLIN 5.02 is not only to focus on performance optimization, but also to improve the new user/developer experience and long-term code maintainability. Improvements implemented to address these issues include allowing MERLIN to support multiple install and build options and ensuring MERLIN runs smoothly on most common systems’ architectures and operating systems. Moreover, it was decided that time be invested in code refactoring such that the software adheres with long-term software sustainability practices. This, however, is a non-negligible task and refactoring will therefore be restricted to aspects of the software that are either the most computationally taxing or hold excessive dependencies. As such, performance optimization is carried out in parallel.

3 Development Team

The current active developer/user team comprises of Dr. Scott Rowan (Full-time, Huddersfield), Dr. Sam Tygier (Full-time, Manchester) and Prof. Roger Barlow (Part-time, Huddersfield). An internal collaboration with University of Huddersfield research software engineers for work specifically related to refactoring and sustainability practices is being discussed.
4 Accessibility Improvements

Following a series of discussions and user trials, it was concluded that the easiest and most intuitive means of accessing MERLIN was to have the main repository publicly accessible online via a conventional internet browser. As a result, MERLIN is now publicly available in its entirety on GitHub at the following URL, see Figure 1:

https://github.com/MERLIN-Collaboration/

Using GitHub has a number of advantages. Firstly, GitHub allows for any potential user to gain access to MERLIN without an explicit request to the current developers. This greatly increases the reach potential of the project. GitHub is also fully integrable with the Eclipse IDE, providing a lot of multiple-author development utilities, such as code reviewing, code hooks and dynamic version control. This allows for fast and frequency repository updates. GitHub also has a user-friendly browser interface, which provides intuitive access to all common git functionalities, such as version HEAD referencing, repository cloning, branching and forking etc. The GitHub repository replaces the dedicated local server. Although use of such a dedicated server is said to be faster, no appreciable difference has been noted and the gain in user convenience is considerable.

Other changes include the simultaneous availability of a cmake build option as well as an eclipse build option. Both build options also have branches including the last known stable version of MERLIN the current beta. Branches are denoted MERLIN 5.01 and MERLIN 5.02 (beta), respectively. It is the intention to keep a branch of each stable version available to preserve user script compatibility. A placeholder README repository has also been created which
will provide an introduction to the software and its capabilities as well as a full user guide (currently being written).

5 Build/Install Improvements

To improve compatibility, it was decided that more than one build option be maintained to allow users to work in their preferred environment. As a result, on GitHub you will find repositories for both MERLIN-eclipse and MERLIN-cmake. This allows users to download and run MERLIN with or without the requirement for an IDE. Both build formats have a README file in their respective repositories explaining the specific install and build test procedure. This replaces the previously mentioned online guide written for use with the University’s local server repository.

5.1 Build Tests & Simulation Examples

A series of small tests and common simulation examples now also come provided with each build format. Information on running each test is provided in the build formats accompanying README file. Running these tests as instructed will confirm that the download, installation and build procedures have been carried out successfully. Provided tests either perform simple checks such as confirming that you have installed all of the required plugins, that the library is properly linked or that input files are being properly read, etc.

In addition to these basic tests, a few example simulations are provided. This includes among others, lhc_collimation_test, and now a basic and diffusive_hollow_electron_lens_test showcasing the various functionality and analysis outcomes implemented in the previous release. There are currently 13 automated tests, see Figure 3. More will be developed and provided more tests with each new release.
5.2 Eclipse Build

The Eclipse build is now designed for users and new developers. It comes with a deliberately clean workstation, providing only the library and a UserProject by default - tests and examples are optional imports. Acquiring MERLIN in this format can be done through Eclipse’s internal egit plugin, importing from the URL provided above with extension /merlin-eclipse.

5.3 Python Integration

Due to the growing global adoption and open nature of the language [3], Python has been agreed upon as the main language for data and analysis visualiza-
tion for MERLIN. In fact, as can be seen in Figure 3, several tests including have_python.py requires that Python and accompanying plugins, scipy and numpy, to be installed. Although the developers will continue to ensure users can run MERLIN without Python integration, it is the intention to actively support front-end data analysis in the language.

6 Profiling & Performance Improvements

One of the goals for 5.02 is to ensure MERLIN runs smoothly on multiple operating systems and systems architectures and that simulation performance is optimized where possible. The developers have used various methods of profiling analysis to identify areas of the code where functions either have high execution times and/or are called frequently.

6.0.1 Cdash Test Suite

To confirm MERLIN operated smoothly on multiple systems architectures/operating systems, an cdash test suite has been constructed. Multiple servers of various architectures (x86/ARM) operating systems (Fedora/Ubuntu/CentOS) have cdash scripts running, such that on a nightly basis the most recent commit to the main MERLIN GitHub repository is downloaded and the aforementioned basic test series is run. Test outcomes are then upload to the following URL:


Figure 5 shows the information provided daily by the test suite. Note that it is the intention to add more systems in the future.

Figure 5: Cdash test suite running the automated MERLIN test series on various different servers systems, qualifying multiple architectures and operating systems as well as the compatibility of any recent commits.
6.1 Time Profiling & Performance Optimization

Although the cdash suite provides basic profiling information such as test code coverage and potential test script memory issues, it does not provide in-depth information on function calls/execution times. For such an analysis the developers have used a common debug package Valgrind [4]. In particular, the Callgrind tool [5] is used in conjunction with the Kcache|grind GUI [6] for visualization and ease of interpreting outcome results.

6.2 Callgrind Outcome

Figure 6 shows the 5.01 Callgrind outcome of the provided lhc_collimation_test. The number of times each function is called as well as the total cumulative percent of execution time spent within the function is provided. This allows for easy localization of so called software hotspots, i.e. functions with significant call times/numbers. The top 5 hotspots, excluding standard library functions, are listed in Table 1. Note that lhc_collimation_test is a deliberately short test of only 20 particle turns and that the call numbers would scale, accordingly.

![Figure 6: Callgrind outcome for lhc_collimation_test.](image)

Table 1: Top lhc_collimation_test hotspots in MERLIN 5.01.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Function Calls</th>
<th>Call Time [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTMap::Apply</td>
<td>1,395,213,914</td>
<td>33.11</td>
</tr>
<tr>
<td>RMap::Apply</td>
<td>1,395,213,914</td>
<td>15.73</td>
</tr>
<tr>
<td>InterpolatedRectEllipseAperture::PointInside</td>
<td>169,087,309</td>
<td>5.20</td>
</tr>
<tr>
<td>ppDiffractiveScatter::PomeronScatter</td>
<td>50,000,457</td>
<td>3.01</td>
</tr>
<tr>
<td>RectEllipseAperture::PointInside</td>
<td>1,120,384,967</td>
<td>2.56</td>
</tr>
</tbody>
</table>
It is clear that, with the exception of the Apply functions which simply contains large vector iterators and are called every time a particle moves, the most significant hotspots are the PointInside aperture functions. In particular, the Interpolated function, which is called $\sim1/10$th of the time but accounts for double the cumulative execution time. The PointInside functions check if a particle is within the local element aperture boundaries. This function is also called every time each particle moves. The following details the optimization process carried out. Note that optimizing how the Apply functions operate would involve a non-negligible re-design of a significant part of the tracker and, although being investigated, is not being consider for 5.02.

6.2.1 PointInside Optimization

In addressing the interpolated aperture hotspot, aperture interpolation was made optional as even knowing where particles are lost at an element level is sufficient for most analyses. With interpolated apertures turned off, for the length between defined aperture entries, the aperture is constant. This also means that the PointInside function need only be called at every aperture entry, however, this has not been implemented yet. Changes to PointInside function algorithms have, however, been made.

The current method of checking whether or not a particle is within the aperture boundaries is shown in Figure 7 (Note that a RectElliptical aperture is modelled in MERLIN as the superposition of a rectangle and an ellipse).

```cpp
bool RectEllipseAperture::PointInside (double x, double y, double z) const
{
  if( (x*x + y*y + z*z) > Workspace)
    return false;
  else if(std::fabs(x) > RectHalfWidth || std::fabs(y) > RectHalfHeight)
    return false;
  else
    return true;
}
```

Figure 7: MERLIN 5.01 RectEllipseAperture::PointInside code.

The function first checks if the particle lies within the ellipse, then subsequently if it lies within the rectangle. A faster method is to use a more in-depth algorithm, utilizing other simple geometries, such as 45-degree-rotated squares (diamonds). Such an algorithm should also be ordered taking into account the most common particle scenario. The devised algorithm is graphically depicted and shown in Figure 8.

A series on additional steps are added, each checking the particle lies within ever growing diamonds. This allows for most checks to be passed by calculations involving additions rather than more computationally taxing multiplications.
Figure 8: MERLIN 5.02 RectEllipseAperture::PointInside code (bottom) and corresponding graphical depiction (top).

Several check algorithms were benchmarked against the current one. The average of 10 loops of 1 billion function calls is taken for each. Results are shown in Table 2.

<table>
<thead>
<tr>
<th>Check Type</th>
<th>Time Avg [s (stdev)]</th>
<th>Difference [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current</td>
<td>36.65 (0.49)</td>
<td>n/a</td>
</tr>
<tr>
<td>Current (swapped)</td>
<td>37.2 (0.95)</td>
<td>+1.49</td>
</tr>
<tr>
<td>Swapped + 1 diamond</td>
<td>34.52 (0.66)</td>
<td>-5.80</td>
</tr>
<tr>
<td>Swapped + 2 diamonds</td>
<td>34.58 (0.74)</td>
<td>-5.65</td>
</tr>
</tbody>
</table>

It is clear that the addition of a diamond check results in a significant 4.09% performance improvement. A further diamond step, however, in fact decreases performance, though only by 0.15% and is within error margins. It is therefore concluded that a single diamond step is optimal and that any further change due to additional steps would either be negligible or detrimental. It is also noted that, although multiplications are more computationally taxing, simply swapping the check order to exclude particles outside the rectangle without additional diamond steps decreases performance by 1.49%. This is a prime example of the importance of taking into account probability, i.e. where the particles are most likely to be.
Callgrind results following implementation of the above changes are shown in Table 3. There is a clear reduction in the cumulative percentage of execution for all PointInside functions. Moreover, on running a non-random version of the test before and after, a significant reduction in overall execution time from 264 s to 251 s is noted. Similar algorithmic optimization has been done for all aperture geometries in MERLIN 5.02.

Table 3: Top lhc_collimation_test hotspots in MERLIN 5.02.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Function Calls</th>
<th>Call Time [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTMap::Apply</td>
<td>1,378,425,116</td>
<td>33.40</td>
</tr>
<tr>
<td>RMap::Apply</td>
<td>1,378,425,116</td>
<td>15.86</td>
</tr>
<tr>
<td>InterpolatedRectEllipseAperture::PointInside</td>
<td>167,051,974</td>
<td>3.64</td>
</tr>
<tr>
<td>ppDiffractiveScatter::PomeronScatter</td>
<td>50,000,469</td>
<td>3.08</td>
</tr>
<tr>
<td>RectEllipseAperture::PointInside</td>
<td>1,106,895,569</td>
<td>2.89</td>
</tr>
</tbody>
</table>

6.3 Unnecessary Bunch Copies

Merlin stores the particle bunch in a C++ vector. This is a very efficient data structure as it is continuous in memory and the location of any particle can be calculated in O(1) time. However, during aperture checks, any particle can be lost from the bunch and a vector does not support efficient erasure of the elements. Each particle that is erased from the middle of the bunch would cause all the particles following it to be moved along by 1 position to preserve the continuous layout. In order to avoid having to make multiple moves at each aperture, Merlin avoids using the erase method. Instead, it creates a new bunch and copies the surviving particles to the new bunch. This means that a particle is moved at most once at any given aperture. However, this has a side effect of increasing cache use, as two copies of bunch will exist simultaneously. It is also speculate that such movement is detrimental to the CPUs caching algorithm as the location of the bunch changes.

To minimise these effects, a pre-check was added to make sure that the bunch is only copied when needed. If no particles are lost at a given aperture, there is no need to copy the bunch. Figure 9 shows the performance comparison for simulations with and without unnecessary bunch copying. There is a clear and significant performance increase in implementing the pre-check.
Figure 9: Performance comparison across two CPU types, both with (old) and without (new) preventing unnecessary bunch copies.

6.4 Compiler Optimization

There are numerous methods of compiler optimization [7]. GCC has several options built-in, -O1,-O2,-O3, each optimizing more processes than the previous. To determine the most suitable build option, a non-randomised version of lhccollimation_test was ran which all performance build options. The result are shown in Table 4.

Table 4: Execution time variation between compiler optimization options.

<table>
<thead>
<tr>
<th>Compiler Optimization</th>
<th>Execution Time [s (stdev)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>272.18 (3.49)</td>
</tr>
<tr>
<td>O2</td>
<td>252.88 (2.16)</td>
</tr>
<tr>
<td>O3</td>
<td>254.01 (3.16)</td>
</tr>
</tbody>
</table>

Interestingly, even though -O3 has optimizes more process, it would that -O2 optimization is slightly faster. This may be due to the fact that some -O3 optimizations increase binary size. Note that the difference, taking into account the error margins, is, however, near-negligible.

7 High-Throughput Computing

Another means of reducing the cumulative simulation time is to take advantage of scalable computing techniques. For example, in loss map simulations it is typical to model of order $10^8$ particles to reach sufficient statistics. Due
to particle tracking being so computationally taxing, on a single CPU, such a simulation may take several days. However, due to the independence nature of the each particle, i.e. no simulated particle-particle coupling, it is trivially parallelizable. The simplest method is to split the simulations into $N_j$ jobs of $N_p$ particles and then to use a batch system such as HTCondor [8]. The final result is found by summing over the loss maps from each job.

As cluster resources are limited, it is important to optimise both individual CPU and overall execution times. Balancing $N_j$ and $N_p$ is an important part of this optimisation as the runtime of any job does not scale linearly with $N_p$. There are some fixed start up costs, such as loading input files, constructing the lattice and setting up cross section tables. There are also other overheads that are independent of $N_p$, such as iterating through the lattice and checking which physics processes to apply. As a result, when $N_p$ is small, the start up time and overheads dominate, which would suggest that larger $N_p$ simulations will be more efficient. However, there are downsides to simulating with a large $N_p$. One example found is the limitation of available CPU cache. Modern computers have multiple levels of processor cache, L1, L2, L3, etc, each increase in size, but having lower bandwidth and higher latency. Simulating with a smaller $N_p$ allows for the whole working data set to fit within the CPUs L3 cache, which is typically around 1 MB per core. Note that high-throughput computing is only one method of addressing this issue. Potential benefits of High-Performance Computing (HPC) methods of parallelization, such as MPI and OpenMP, will be investigated in the future.

Figure 10 shows the tracking performance for 10 LHC turns for a range of CPUs with different caches sizes - aperture checking is disabled. In each case, the efficiency increases as $N_p$ increase, saturating where the bunch size approaches that of the CPU cache. Keeping the storage size of the bunch smaller that the CPU cache gives a 5-10% performance improvement. Note that for benchmarking, only a single CPU core was used. In practice, cache is shared between all CPU cores.

8 Code Maintainability

In view of MERLIN being maintained long-term, it is important that the code is user-friendly/accessible and that it adheres to software sustainability practices, i.e. readable, modular, well-structured, etc. The developers will therefore begin to re-factor any aspects of the code which do not meet these standards. This includes, “C-style” functions and classes which are tasked with too much or are inefficient in their execution. In doing so, the developers will utilize object orientated design patterns [2] as well as high levels of abstraction to greatly reduce the length of a typical user script, currently an excessive 500+ lines of code.
Figure 10: Absolute (top) and normalized (bottom) tracking speed for a range of CPUs with different L3 cache sizes. Cache size shown in brackets.
A working example of this is a redesign of the aperture implementation code. Currently, each aperture entry is read from an input file and is translated by if and switch statements, correlating a type string to an enumerated type. This type is then used to determine which derived class member functions are to be use. This code is then duplicated in each derived class, for example RectEllipseAperture as well as CircularAperture. It is also duplicated in each complimentary derived class, for example InterpolatedRectEllipseAperture or similarly an offset or titled aperture class. The working solution is to redesign this implementation utilizing factory and decorator design patterns. This maps any pointer to a base class type with any specific derived class member function based on a string input (factory pattern) and allows any modifiers (tilt, offset, interpolation) to be added accordingly (decorator pattern). This implementation, currently in development, will be integrated prior to the full release of 5.02.

Other recent improvements include code clean-up, particularly in relation to HEL process code, see Figure 2, as well as the use of not previous available C++1y techniques and function. Legacy code written in C++98 is being brought up-to-date alongside any related re-factoring, for example, the use of nullptr’s instead of 0’s.

8.1 Dependences

When considering the structure or architecture of a software package, it is important to consider class and function dependencies, i.e. how files/classes/functions rely on each other. An overly dependant software package can be very hard to modify as a small change in one class may require numerous, if not hundreds, or other changes elsewhere to build and run correctly. In contrast, a well-structure and minimally dependant architecture can allow multiple developers to work of different sections without risk of interference or compatibility issues.

TITAN [9] is a software architecture visualization and analysis tool which is designed to assist developers in locating dependency issues. The developers will carry out a full TITAN analysis and to re-design the base architecture, accordingly.

8.2 User Guide

One of the biggest issues new users/developers can have is not knowing what the code is capable of or how to write user scripts. The will therefore detail every aspect of MERLIN, including architecture, classes, functions and user-script writing, in a full, citable user guide.

8.2.1 RSE 2017

The developers will be giving an invited talk and contributing a paper at Research Software Engineering 2017 conference in Manchester [10]. The talk will
focus on the above outlined software engineering specific improvements to MERLIN.

9 MERLIN Roadmap

The following gives a brief look at what is currently planned for MERLIN in future.

5.03
- Composite materials
- High level abstraction
- User Guide

5.04
- Highly parallelizable and optimized tracker
- GPU-accelerated computation options

5.05
- FCC and beyond

10 Conclusion

Following the implementation of hollow electron lens simulation functionality, MERLIN has undergone significant performance optimizations, has had numerous test example programs written and provided, and is currently undergoing significant changes to improve accessibility and long-term code sustainability. There is also now a full time, albeit small, development team working on further improving the software and a plan for future developments has been laid out.

Overall, MERLIN is now a very capable and efficient particle tracking software package with both basic and advanced collimation simulation functionalities.

References


