Conditioning Simulated Particle Pulses

Lawrence Campbell, Andrew Colin, and Brian McNeil

Version 3 : July 31, 2015

1.1 Abstract

This paper describes technical aspects of a program designed to couple the two main components of a Free Electron Simulator, namely, an accelerator simulator and an undulator simulator. The main task of the program is to replace the crude approximation of a high-speed electron stream produced by the accelerator with a much finer approximation suitable for the undulator.

The main topics covered are

- Splitting a *macro-particle* that represents some thousands of electrons into a large number of *micro-particles* each carrying a much smaller charge.
- Managing internal storage (of which a great deal is needed for the process).
- Implementing the algorithm on a multi-core computer

1.2 Introduction

Free Electron Lasers ("FELs") produce X-ray pulses of very high intensity and short duration. They are important scientific tools, and allow for experiments which would not be feasible with any other type of optical pulse.

FELs are massive machines. They are time-consuming to build and expensive to operate. It is therefore useful to have FEL *simulators*, so that design parameters and operating modes can be checked before major costs are incurred.

Simulators for Free Electron Lasers have two main active components: a simulated electron gun, to model an accelerator that produces short electron pulses of very high energy, and a simulated 'wiggler' which converts some of the energy in each electron pulse into a very short, intense X-ray pulse.

A practical problem arises because the output of a typical electron gun simulator is not directly compatible with the input needed by the wiggler simulator. In particular, the gun simulators produce streams of 'macro-particles' each of which has a known position in six dimensions (three physical, three momentum) and a charge which is typically about 10000 e, where e is the charge on an electron, 1.6022×10^{-19} coulombs. On the other hand, the wiggler simulator gives the most accurate results if it is fed much smaller 'micro-particles' with an average charge of about $10e$. It should be noted that both "macro-particles" and "micro-particles" are hypothetical constructs consisting of groups of electrons with identical properties. They are approximations, of increasing fidelity, to the real world where each electron is unique.

The Simulated Electron Conditioner is a program that serves to link the electron gun and the wiggler simulators. The conditioner separates each macro-particle emitted by the gun into about 1000 micro-particles, using the statistical methods defined in [1].

The conversion process is not trivial. The main problems are the volume of data to be handled, and the need to run the program on a multi-core computer, so as to ensure an acceptable turn-round time. Apart from a very brief introduction to FELs (which the reader may skip) the paper concentrates on the computational techniques used in the program.

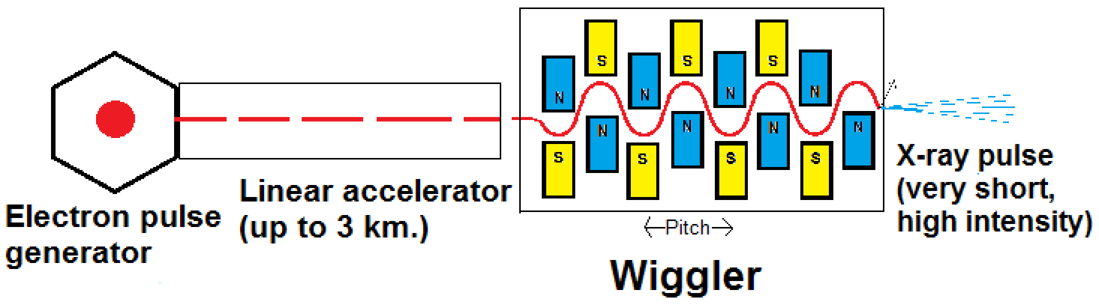
Electron pulse generator
Linear accelerator (up to 3 km.)
S
N
←Pitch→
Wiggler
X-ray pulse (very short, high intensity)

Figure 1.1: Schematic of a Free Electron Laser

1.3 Free Electron Lasers

Figure 1 shows the two key components of a Free Electron Laser [3, 4]. The accelerator on the left is fed a short burst of electrons typically released by a laser pulse. As the electrons travel along the tube they are accelerated by magnets until they reach a speed close to that of light. They may have γ values of 2000 or more where

$$\gamma = 1/\sqrt{1 - \nu^2/c^2}, \tag{1.1}$$

c is the speed of light, and ν is the speed of the electrons.

The pulse of high-speed electrons is fed to the wiggler, which consists of an array of magnets arranged as shown in the diagram. The electrons follow a roughly sinusoidal path, keeping to a plane perpendicular to the page. The X-ray pulse is formed by two effects:

- As the electrons follow their sinusoidal path they accelerate (specially when near the magnets). This makes them emit synchrotron radiation, of all frequencies and in all directions. The radiation emitted in a forward direction, and at a wavelength equal to the pitch between the magnets, will be in phase with the radiation emitted at the next magnet, so that the energy is additive. The same is true of radiation at sub-multiples of the pitch, but all the other radiation is just noise and tends to cancel. Neglecting relativistic effects, we would expect the wiggler to produce a pulse of radiation of frequency c/pitch and its harmonics.

  But relativity modifies this result considerably. It can be shown that the wavelength of the emitted radiation falls by a factor of γ^2. For example, if the pitch is 5 cm, and γ is 2000, the radiation will be at a wavelength of $0.05/2000^2$ or 0.0125 nM., in region of hard X-rays.

- A further effect that increases the shortness and strength of the X-ray pulse is 'bunching'. As the electrons travel through the wiggler they tend to gather in bunches, so that the amount of radiation emitted at a later stage is much higher than for an earlier stage.

FELs have the advantage that the output frequency can be tuned, simply by altering the speed of the incoming electrons.

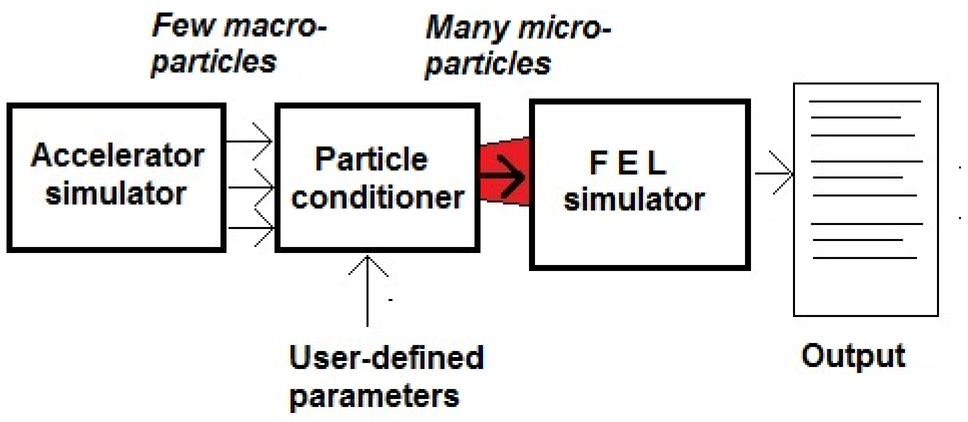

Figure 1.2: An FEL Simulator

1.4 The role of the Particle Conditioner

The role of the Particle Conditioner,(“**PACO**”) as a component of an FEL simulator, is shown in figure 1.2. Its task is to replace each macro-particle from the gun simulator by a large number of smaller micro-particles, whilst keeping the total charge constant.

The design of **PACO** seeks to fulful three key aims:

- The program should be usable with streams of particles of any type. The immediate application that led to its development used electrons, but it should also work with (for example) a stream of molecules of mixed chiralities, fed to a device that separates the left-hand and the right-hand molecules [2].
- The program must be adaptable to the user’s needs. For example, it should be able to accept data in many different formats, and offer a choice of different mathematical models to control the splitting of the macro-particles.
- It must be efficient, so as to offer a fast turn-round time. The basic algorithm is computationally intensive and uses a large amount of memory; it needs to run on a machine with multiple processors. Such machines generally queue the jobs submitted to them, and allocate priorities to jobs in inverse proportion to the resources (cores and time) they need. The user must have the information to find the optimum balance of cores and time estimate to get the job run quickly, while avoiding the risk of premature termination.

To steer **PACO**, the user must supply a *parameter file* which defines the format of the input data, splitting model and other details.

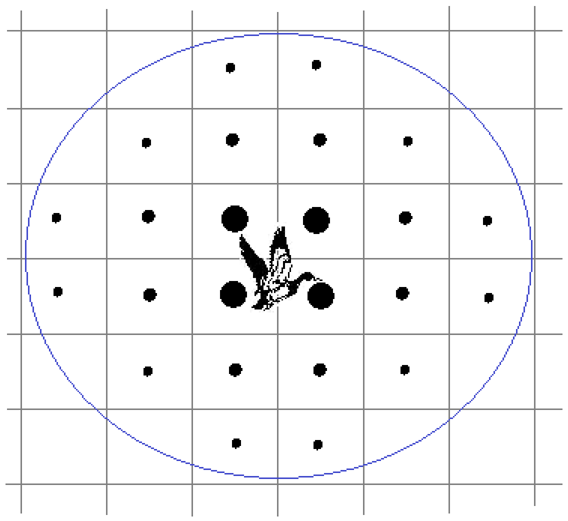

Figure 1.3: Duck shooting

1.5 Splitting macro-particles

In this section we describe the method used to split each macro-particle into a large number of micro-particles.

Every particle in **PACO** is handled as a point in a six-dimensional space: three spatial axes X, Y and Z, and three momentum axes Px,Py and Pz. The actual extent of each dimension depends on the range of that dimension in the input data, but it is convenient to divide the extent into sectors of equal size. The number of sectors in a dimension is called its *resolution*, and can be set by the user. A representative resolution for the Z and Pz axes is 1000, because this is the direction of motion, and a high resolution leads to accurate results. Resolutions for the other four axes can be much smaller; 25 is a typical figure. We can imagine a hyothetical six-dimensional matrix, where each macro-particle can be placed somewhere in one of the cells.

A useful two-dimensional analogy is the behaviour of a cartridge fired from a shotgun. Initially all the pellets move towards the target, but as they approach they spread out, in a roughly Gaussian pattern, both in direction and speed.

Figure 1.3 shows a duck flying in front of a rectangular grid. When the hunter takes a well-aimed shot at the duck, the majority of the pellets hit cells close to the target (some of them may even hit the duck), but a substantial fraction land in cells some distance away. Each blob in figure 1.3 shows roughly

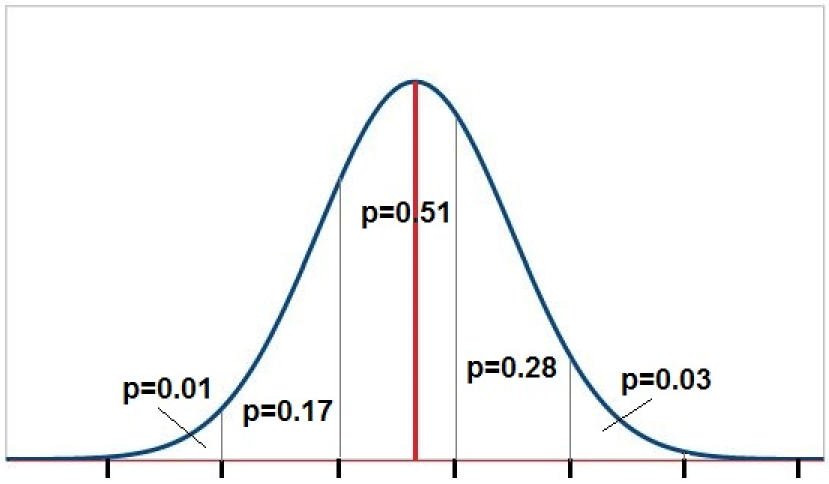

Figure 1.4: Gaussian distribution

how many pellets hit each cell [1].

In **PACO** the splitting process is first done separately for each of the six dimensions. In Figure 1.4, the marks on the horizontal axis represent the cell boundaries in a given dimension. A macro-particle lands on one of the cells (not necessarily in the middle). Its position is shown by the red line. Assuming a Gaussian distribution of scattering, we superimpose a gaussian curve centred on the position of the macro-particle and with a standard deviation that can be determined by the user (usually 2, 3 or 4 cell sizes). We then use the error function to calculate the area under the curve for each division, that is,the probability that an electron from the macro-particle will land in that slice of the cell matrix, as shown in the figure.

Once this process has been repeated for all six dimensions, the probability that an electron from the original macro-particle lands in a given cell of the matrix is the product of the six probabilities from the individual dimensions. The expectations, (probabilities $\times$ charge on the macro-particle) which are generally fractional at this stage, are then distributed according to these probabilities. Each micro-particle is initially placed in the centre of its cell. The micro-particle with the largest charge will end up in the original target cell, but many others will be sent to cells nearby.

[1]No duck was killed in the writing of this paper.

We have described the process for a Gaussian distribution, but the user is free to specify different scattering patterns. A pattern is best described as a cross-section of the distribution along one dimension. For example, a "top hat" distribution ensures that the scattering is uniform over a certain range, with no 'tails' in either direction. A 'circular' distribution gives a maximum in the middle, but also ensures no scattering outwith the radius of the circle.

Using the parameter file, the user can specify the scattering model and its spread independently for each of the six dimensions.

Once all the data has been read and processed, the results can be output to a file. Each non-empty cell undergoes two transformations:

- The expectation in the cell is replaced by a Poisson variate, which gives an actual (integral) number n of electrons.
- The position of the micro-particle adjusted by noise. Let p be a rectangularly distributed random number in the range $-0.5 <= p <= 0.5$. The position in each dimension is moved by a fraction f of the cell size, where

$$f = p/\sqrt{n} \tag{1.2}$$

The output record for each cell consists of six real coordinates and an integral charge [2].

This process raises an important practcal point. If two or more macro-particles are close to one another, it is probable that some cells will receive charges from more than one source, These charges must be summed before the final Poisson and 'noise' adjustments are made to the resulting micro-particle. This implies that the cell contents cannot be output as they are generated, but must be stored until all the macroparticles have been processed.

1.6 The internal data model

We have suggested using a hypothetical six-dimensional matrix to store our intermediate results. A conventional matrix is impractical. The resolutions divide the space into cells of equal size. Given our figures, there will be $1000^2 \times 25^4 = 3.9 \times 10^{11}$ cells. This implies a memory size that is not available on even the largest current computers.

Now consider the input to **PACO**. Suppose (for the sake of argument) that the pulse consists of 40000 macro-particles with an average charge of $10^4 e$. The total number of electrons is 4×10^8. If **PACO** were to distribute these electrons

[2]This rule implements the algorithm given in [1]

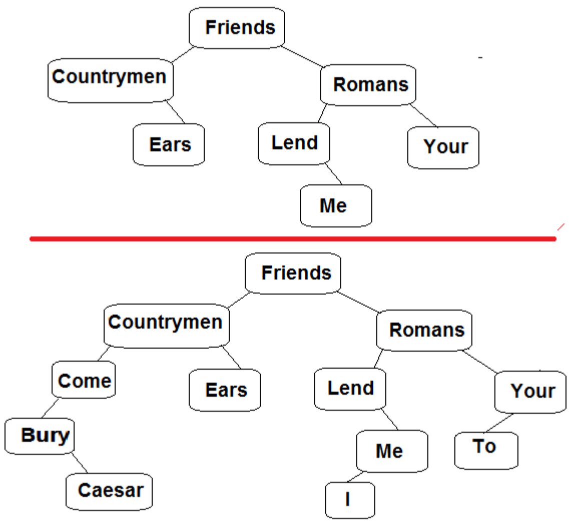

Figure 1.5: Two stages in growing a binary tree

so that each one occuped a single cell on its own, only 0.1% of the cells in our matrix would be occupied. In practice, many of the electrons will be directed to the same cell, so the proportion of filled cells will be even less.

Clearly the cell matrix is exceedingly *sparse*, and in a highly irregular way (unlike - say - a band matrix). An effective way to store the matrix without wasting space on the mostly empty cells is to use a binary tree [5, 6].

The binary tree was first developed to construct alphabetised dictionaries of words. It relies on the fact that any two words are either the same, or can be placed in alphabetical order.

Figure 1.3 shows how a binary tree can be grown from Mark Anthony's speech[3] in *Julius Caesar* [7]. Once the the root word ("Friends") has been been established, every other word in the text can be assigned to the left branch of the tree if it is less (in dictionary order) or the right branch if it is greater. The tree is constructed by using this rule repeatedly.

The tree has several advantages. Memory space need only be allocated when it is known that a particular entry actually exists. Under ideal conditions, when the tree contains n different words, the number of comparisons needed to locate a given word or enter a new one is approximately $\log_2(n)$, as opposed to $n/2$ if the words were just stored in the order they occurred in the text. The cost

[3]It begins, *Friends, romans, countrymen, lend me your ears. I come to bury Caesar ...*

Key type	**Depth of tree**	**Performance figure**	**Comparison with ideal**
Unmodified key	356	156	13.5%
Reflected key	53	27	77.8%
Ideal	23	21	100%

Table 1.1: Performance measures for binary trees

is that each word must be accompanied by a couple of pointers to indicate its left and right descendants. The contents of the tree can be converted to a list in alphabetical order, simply by recursively printing the left branch of a node, the node itself, and then its right branch.

In an ideal tree, each layer (except usually the last) is fully populated, so that layer p would contain 2^p words. The trees in figure 1.3 already show departures from this ideal. We can define a *performance figure* to assess the quality of a given tree : the average number of comparisons needed to reach any item, starting from the top. For example, the number of comparisons required to reach the word “Ears” is 3, and to reach “Caesar” , 5. The average over the whole tree is 3.17. On the other hand, in an ideal tree the mean (with this number of words) would be 3.00. With larger trees we can expect this discrepancy to increase.

As a the tree grows, its adherence to the ideal depends on the words being supplied in a random alphabetical order. If the word at the head of the tree happens to be “Aardvark”[4] then all the other words will be placed in the rght branch, and the tree will be *unbalanced*, with a performance figure greater than the optimum.

When the tree is constructed in **PACO**, all the coordinates of the microparticles are integers within their respective resolution ranges, because the splitting process places each microparticle into the *middle* of its cell. The 'key' to each entry, which we use instead of a word, is built by placing each coordinate (in binary) into a 64-bit binary word, arranged so that they do not overlap. This gives a unique key for each entry, but leads to a poorly balanced tree.

A simple way to ensure a better balance is to “reflect” the key from left to right, so that the least significant bit becomes the most significant. and vice versa. The outcome, for a tree with 6818254 entries, is shown in table 1.1. This slight modification leads to a considerable improvement in speed

Figure 4 shows the layer occupancy for both the ideal and the reflected key cases:

[4]Aardvark: A nocturnal African mammal with a long snout, that lives on termites.

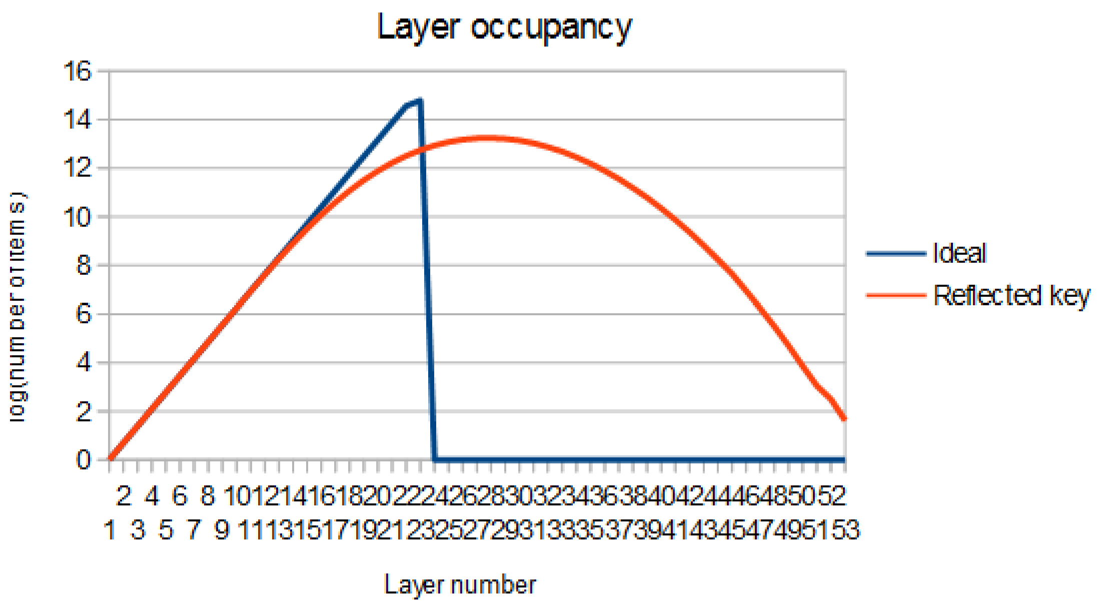

Figure 1.6: Layer occupancies

1.7 PACO in a multi-processor machine

The algorithm we have described in not suitable for a computer with only one central processor. When the input file is large, or if high resolutions are chosen for the axes, the code will be slow, and is likely to run out of memory altogether.

Here we describe how the algorithm can be adapted to run on a multi-processor machine.

A multiprocessor machine consists of many *cores*, each with its own central processor and an allocation of memory. The cores run asynchronously, and can communicate with one another by sending and receiving messages of arbitrary length. Once a core has announced that it expects a message from another core, it waits until that message arrives. The cores are numbered from 0 up. All the cores execute the same program but each can follow a different sequence of instructions by branching on its core number. The underlying operating system used here is the **Message Passing Interface** "MPI" [8].

To make good use of the multicore machine, the work should be distributed among the cores as evenly as possible. There is always one core, called the "root" which handles input and output, and distributes data to all the cores, including itself. Clearly the root must do more work than the other cores, but if the initial setting up and final output are both fast, the discrepancy need not be great.

The raw data for the program consists of a set of records generated by the accelerator simulator. Each one describes a macro-particle, giving its six coordinates and its charge. An example might be:

```
-2.5545e-07 1.3450e-22 -4.4189e-07 2.3766e-22 2.3386e-05 1.3091e-19 9.3760e+03
```

The program runs in five phases:

Phase 1: The root core reads the parameter file, which *inter alia* specifies the resolution for each axis, and picks the axis with the highest resolution. This is the"key axis" , which will normally be Z or Pz. Then the root reads the data, and applies a linear transformation to each axis so the values lie between zero and the resolution for that axis, and sorts the records into ascending order of the key axis. Next it distributes the records evenly across all the cores, assigning an equal number of records to each one. As the distribution in the key axis need not be uniform, the range sent to each core will not in general be the same. For example, suppose there are four cores, 5000 records and the key dimension has a resolution of 1000. If the records are bunched towards the centre of the key axis, the distribution might well be

- Core 0: 1250 records with key dimension 0 to 409
- Core 1: 1250 records with key dimension 410 to 498
- Core 2: 1250 records with key dimension 499 to 612
- Core 3: 1250 records with key dimension 613 to 1000

These dimensions are called the 'core boundaries' for each core.

Phase 2: Each core sets to work on its own set of records. Each macro-particle is split into a large number of micro-particles, using the rule described in section 1.5. If the macro-particle is close to either core boundary, some of the resultant micro-particles may lie beyond the core boundaries. The core stores the results in three separate trees:

- The *white* tree, which holds all the micro-particles which fall within the core boundaries. This will normally be the large majority.
- The *red* tree, which holds all the micro-particles with key axis coordinate less than the lower core boundary.
- The *green* tree, which holds all the micro-particles whose key axis coordinate is greater than the upper core boundary.

This process is illustrated in Figure 1.7.

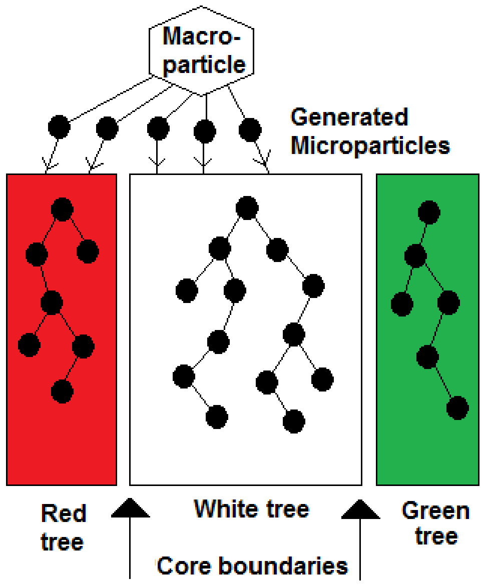

Figure 1.7: Distribution of micro-particles within a core

It is worth noting that all the micro-particles in the red tree of any core (except core 0) really belong in the white tree of the previous core. Similarly, the micro-particles in the green trees of all the cores (except the last) belong in the white tree of the following core.

Phase 3: In phase three, each core transfers the contents of its red tree to white tree of the previous core.The red tree cannot simply be appended to the white tree as a single entity, because some of its micro-particles may occupy the same cells as those already present in the white tree, and the contents must be merged correctly. The micro-particles of the red tree need to be added to the white tree, one by one.

To achieve the transfer, the contents of the red tree are first extracted, by a recursive function, and formed into a linear array. It is vital the this array should **not** be in ascending order of the keys, as otherwise this would lead to a grossly unbalanced white tree when the micro-partcles are added. To ensure this, we use a version of the tree exploration function which decides at random, at each stage, which to explore first - the left or the right dependent tree.

Next, the linear array is passed to the previous core as a message.

Finally, the previous core adds all the macro-particles in this array to its own white tree.

Phase 4: This is exactly the same as Phase 3, except that the contents of the green tree are passed forward to the next core.

Phase : This is the final output phase. Each core extracts the data from its white tree, and passes it back to the root core. Here

- The summed fractional expectation of each micro-particle is converted to an integer charge using the Poisson function
- Particles with a zero charge are dropped,
- The position of the macro-particle is adjusted away from the centre of the cell, in each dimension, by a random amount proprotional to $1/\sqrt{w}$, where w is the charge
- The record for each micro-particle is sent to the output file.

As the root core can only handle the data from the other cores one at a time, this is the slowest part of the process. As the data is (presumably) being written to a rotating disk with a single read/write head, no advantage would be gained if each core were to do its own writing.

1.8 Conclusion

The source code for **PACO**, together with detailed user instructions, may be found on (give a web site).

Bibliography

[1] B.J.W. McNeil, M.W. Poole and G.R.M.Robb
Unified model of electron beam shot noise and coherent spontaneous emission in the helical wiggler free electron laser
Physical Review Special Topics - Accelerators and Beams, Vol 6, 070701 (2003)

[2] R.P.Cameron, S.M. Barnett and A.M.Yao
Discriminatory optical force for chiral molecules
New Journal of Physics 16 (2014) 013020

[3] F.R. Elder, A.M. Gurewitsch, R.V. Langmuir and H.C. Pollock
Radiation from Electrons in a Synchrotron
Physical Review, vol. 71, Issue 11, pp. 829-830 (1947)

[4] J. Madey
Stimulated emission of bremsstrahlung in a periodic magnetic field
J. Appl. Phys. 42, 1906 (1971)

[5] A.D. Booth and A.J.T. Colin
On the Efficiency of a New Method of Dictionary Construction
Information and Control, Vol. 3, 327-334 (1960)

[6] Donald Knuth
The Art of Computer Programming : Searching and Sorting
Reading, MA : Addison-Wesley 1973

[7] W. Shakespeare
Julius Caesar
Ed. John Cox, Internet Shakespeare Edition

[8] Peter S. Pacheco
Parallel programming with MPI
Morgan Kaufmann (San Franciso) 1997