High-Performance Spatial Simulations and Optimisations on 64-Bit Architectures

K. A. Hawick and H. A. James and C. J. Scogings

2006

Many numerical simulations applications continue to require large computing budgets to allow their use in state-of-the-art parameter regimes. We report on a number of optimisation techniques that are especially applicable to spatial simulations. We employ trade-off techniques based on the use of high-memory and especially 64-bit addressable memory to boost the performance of simulations that need extensive geometric maths function evaluations. We also describe data structures and indirect addressing techniques to improve spatially-oriented simulation algorithms. We present performance data and a discussion and analysis on these techniques which we believe generalise to many spatially-based scientific simulations codes. We illustrate these methods in the context of simulations for: diffusion; cluster-cluster aggregation; Monte-Carlo annealing; and artificial-life models.

Keywords: spatial simulation; maths functions; performance optimisation; data structures; high-memory usage

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-026,
        author = {K. A. Hawick and H. A. James and C. J. Scogings},
        title = {High-Performance Spatial Simulations and Optimisations on 64-Bit
                Architectures},
        booktitle = {The 2006 International Conference on Modeling, Simulation and Visualization
                Methods},
        year = {2006},
        editor = {H.R.Arabnia},
        series = {CSTN-026},
        pages = {129-135},
        address = {Las Vegas, Nevada, USA},
        month = {26-29 June},
        publisher = {CSREA Press},
        note = {ISBN 1-932415-99-8},
        keywords = {spatial simulation; maths functions; performance optimisation; data
                structures; high-memory usage}
}
```

High-Performance Spatial Simulations and Optimisations on 64-Bit Architectures

K.A. Hawick, H.A. James and C.J. Scogings

Institute of Information and Mathematical Sciences

Massey University – Albany

North Shore 102-904, Auckland, New Zealand

Email: {k.a.hawick,h.a.james,c.scogings}@massey.ac.nz

Tel: +64 9 414 0800   Fax: +64 9 441 8181

10 October 2005

Abstract

Many numerical simulations applications continue to require large computing budgets to allow their use in state-of-the-art parameter regimes. We report on a number of optimisation techniques that are especially applicable to spatial simulations. We employ trade-off techniques based on the use of high-memory and especially 64-bit addressable memory to boost the performance of simulations that need extensive geometric maths function evaluations. We also describe data structures and indirect addressing techniques to improve spatially-oriented simulation algorithms. We present performance data and a discussion and analysis on these techniques which we believe generalise to many spatially-based scientific simulations codes. We illustrate these methods in the context of simulations for: diffusion; cluster-cluster aggregation; Monte-Carlo annealing; and artificial-life models.

Keywords: spatial simulation; maths functions; performance optimisation; data structures; high-memory usage.

1 Introduction

Numerical simulations of physical and other real life systems continue to present a demand for computational cycles [12]. Many such applications require simulated systems sizes that are big enough to capture emergent collective effects of the modelled system and therefore these applications have absolute minimum compute requirements to be worthwhile. Generally these applications require carefully crafted codes in languages like C, C++, Fortran or Java and the model regimes of interest cannot yet be explored with problem-solving environments and higher-level languages.

The general trend we have observed is that high performance numerical computing has come full circle. When numerical simulations were becoming popular processors were slow; we were compute-bound. The amount of memory that was available to users and processes was not as critical as the actual amount of cycles that could be quickly devoted to a process. As processors became faster and broke the 1GHz barrier we soon realised that we had become memory-bound: the 32-bit operating systems that ran our machines were no longer able to hold the size of data structures that we desired to simulate. It is interesting that now we have 64-bit operating systems, and the ability to address far in excess of 4GB of memory, we are once again compute-bound: we now have to wait considerable time for our algorithms to update the large number of entities we are simulating. There is little doubt that as processors increase in speed and the price-performance breakpoint of dual- and quad-core processors becomes more attractive, our processes shall again be memory-bound.

In this paper we describe several simulations and our attempts to encode them with various optimisation techniques to allow us to explore worthwhile model regimes. We discuss the sometimes counter-intuitive effect of various data structures to enhance computational performance and report on various measurements of maths function primitives and some simulations code algorithm kernels. Although all the codes we discuss are our own, we believe these ideas generalise and will be of use for other simulations code developers.

As a preamble to our discussion of various simulations codes we first consider (section 2) some of the expensive maths functions that occur in simulations. In section 3 we describe a diffusion-limited cluster aggregation simulation and two optimising approaches we have employed involving `sqrt` functions and random number generation. In several simulations the way the data structures are organised can affect the performance considerably and we illustrate this effect for a cluster-cluster interactions code in section 4 and for a Monte-Carlo [16] phase transition simulations code in section 5. A sophisticated code for modelling phenomena like complexity in artificial life has several interacting performance constraints and we discuss these in section 6. Many of the optimisation we discuss in simulations are dependent upon trading off memory for performance and in section 7 we discuss some of the 64-bit high-memory access issues for simulations codes. Finally we draw some conclusions about future porting and optimisation issues in simulations codes in section 8.

2 Timing Common Maths Function Primitives

Table 1 presents some timing data for various combinations of modern processor architectures [2, 3], operating systems and their support libraries. The data was obtained using a loop of 10^9 operations involving the sum of the respective mathematical function shown in the rows of the table. The system random number generator `random()` [5] was used to generate an argument for each function – since this nicely exercises the function across a wide range of input values and avoids any timing pathologies due to internal lookup table implementations un-beknownst to the user. The computer's own internal clock was used and the times presented for the maths functions have had the base loop times with the plain sum of `random()` subtracted out so that for instance column 2 shows that the `sqrt` function takes around 130 machine cycles (10^9 operations on a 1GHz clock machine) and

even on our fastest 64-bit architecture `sqrt` still takes around 29 machine cycles (12×10^9 operations $\times 2.4$Ghz). All timing data presented can be taken as accurate and reliable to within 1 second. We chose to make measurements over 1 billion (10^9) operations since it gives us meaningful comparisons between machines in a regime where this clock accuracy is adequate and reliable enough for our purposes in this paper rather than employ architecture-specific techniques to obtain super-accurate timing measurements [7] for a more rigorous benchmark. While the `sqrt` function is the most heavily used, all the primitives in table 1 are used quite extensively in the codes reported in sections 3 and 6.

Function	Machine 1 1.3GHz G4 (OSX 10.4)	Machine 2 1GHz G4 (OSX 10.4)	Machine 3 2.4GHz G5 (OSX 10.3)	Machine 4 2.4GHz G5 (OSX 10.4)	Machine 5 2.4 GHz dual Opt250 (Linux 2.4.21-20)
`random`	35	36	16	15	16
`sqrt`	125	130	24	12	12
`sin`	122	126	59	51	110
`cos`	118	124	60	52	116
`tan`	194	202	115	106	290
`atan`	129	134	56	53	64

Table 1: Times (in seconds) for executing various maths function primitives 10^9 times on various processor/operating system architecture combinations. All are using the GNU gcc compiler (version 4.0) with optimisation flags `-O6`. Machine data columns 4 and 5 have 64 bit architecture features enabled. Column 3 and 4 data represent identical hardware but different operating system versions – column 4 data is 64-bit enabled. Times are accurate to ± 1 second.

In all cases the GNU compiler version 4.0 with optimisation flag `-O6` was used and in the case of native architecture flags being available to activate 64-bit capabilities, these were also applied.

3 Square Roots and DLA

Simulations based on diffusion limited aggregation have been popular as a basis of investigating growth and structural change in physical systems since the 1980's. The core diffusion limited aggregation (DLA) algorithm [23] is based on diffusing particles that stick to a growing aggregate when they encounter it. Particles are often released in a circular (2 dimensions) or spherical pattern (3 dimensions) and the initial aggregate is simply a single stationary particle. The model can be simulated on a lattice quite efficiently [17], but it can also be run using continuous coordinate particles. In either case, although some savings can be made using squared coordinates from the present centre of the aggregate's frame of reference, at some point in the algorithm the square root function is needed in the inner code loop. As seen in table 1 square-root evaluations continue to be computationally expensive and most processors do not have special purpose hardware or instructions for their evaluation [20].

We developed a hyper-dimensional DLA code capable of simulating the DLA process in 2, 3, 4, 5, 6 and in principle higher dimensions. In practice we rapidly run out of memory and interestingly sized aggregates are only really feasible in dimensions up to 4. Memory limitations not withstanding, we are running this code to generate a large statistical sample of aggregates as a set of independent jobs on a cluster computing system. A detailed profile analysis of the code reveals that the simulation is dominated by the geometry calculations – and specifically the square root radius evaluations, and the random number generator computational cost. The latter can be reduced slightly by using a faster (but lower quality) generator algorithm. It is however not feasible to reduce the quality of the square root evaluations. The correlation properties of the aggregates are sensitive to symmetries and the enclosing hyper-sphere for releasing particles that diffuse, must be computed properly.
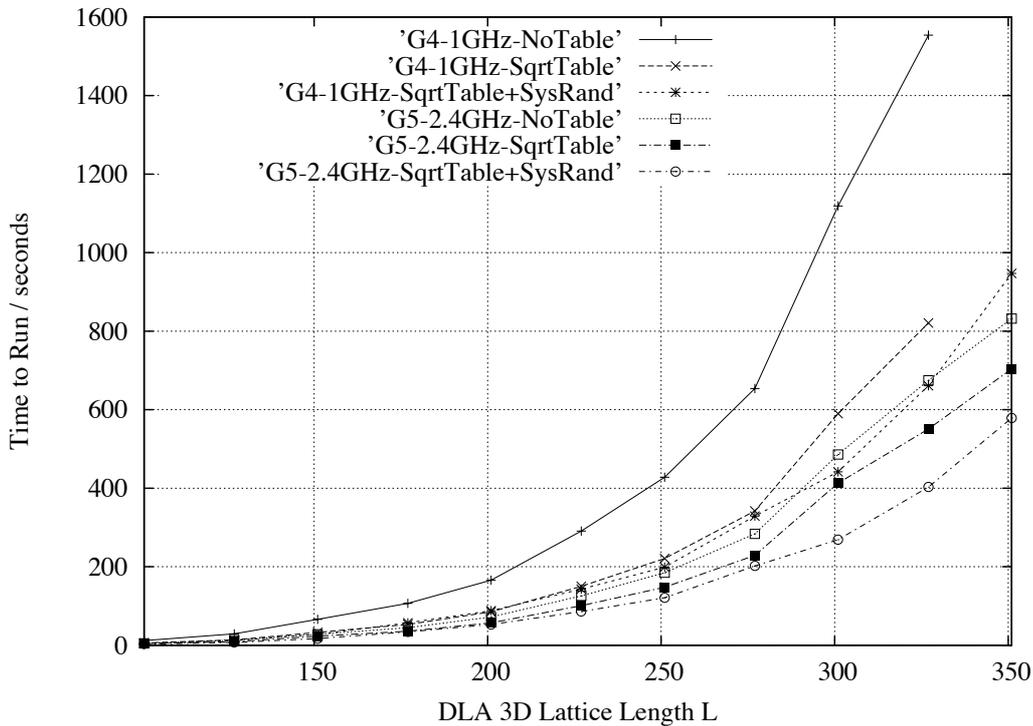

Figure 1: Time to run 3-D DLA simulation (seconds) vs enclosing Lattice Length. Introducing a fairly simple optimisation such as a square-root lookup table leads to a significant increase in performance. A further change from the default C language random number generator to a Unix-specific generator leads to an additional but not entirely expected performance increase.

Figure 1 shows the effect of applying two code optimisations to the DLA simulation program on a 32-bit G4 processor and on a 64-bit G5 processor. The first optimisation involves the table of pre-calculated `sqrt` double values outside of the inner simulator loop. This makes almost a factor of two improvement, for the code overall on the 32-bit G4 where `double-sqrt` calculations are particularly expensive. Even on the 64-bit G5 however it still

gives a 25% speed improvement.

A second optimisation concerns the quality of the random number generator used. We have employed Marsaglia's lagged-Fibonacci generator safely and without problems for some years in various simulation work. This is based on floating point calculations and is relatively expensive for the DLA code. We experimented with two other generators – the Unix system supplied `rand` which is very fast but very poor in terms of quality of deviates. A careful analysis shows it does not adequately span the whole surfaces of the particle release hyperspheres around the growing aggregate and thus introduced unacceptable statistical biases. An intermediate choice is the newer Unix system supplied `random` generator. It is a better generator than `rand` and is faster than the lagged-Fibonacci. The figure shows timings for the DLA code when we use this system random number generator in combination with the lookup table and different architectures.

Plotting these timing curves on a log-log scale (not shown) simply reveals the result that they all (as expected) are dominated by a $O(L^x)$ complexity with $x \approx 4$. A least squares fit to $\log - \log$ plots reveals $x \approx 4.018$. This is expected as $4 = d + 1$ for an embedded dimension $d = 3$ simulation. The scale factors for the different timing curves do vary considerably however. The use of the `sqrt` lookup-table yields a factor or 2 or 1.25 on the G4/G5 architectures and the use of the random generator yielding a further factor of 1.15 or 1.25 respectively. Taken over simulations taking CPU months this is significantly useful.

As the figure 1 shows, computing a square-root is expensive, even with all compilation optimisations activated. The importance of this is due to the `sqrt` calculation being in the inner loop of the simulation. Our code is already using considerable memory in a non trivial cache hitting pattern to accommodate the growing aggregate embedded in the lattice. It tuned out therefore that the cache missing caused by introducing a large pre-calculated table of square roots for all possible integer values that can be encountered in a lattice simulation is not too high.

Figure 1 shows the performance improvements from use of a square-root table in the DLA code. This is surprisingly high, and the table values are not an approximation, just precomputed outside the main loop. The lattice code in 3-dimensions for example cannot encounter square-root arguments of greater that $3 \times L^2/4$, for an embedding lattice length size of L, with the growing aggregate centred. This still represents quite a large table size for typical L values of around 1024 or higher, although it is slower scaling than the embedding lattice size itself which is $O(L^d)$. It is worthwhile to precompute these `sqrt` values, providing memory is available.

We have encountered similar issues for computationally expensive functions such as trigonometric functions, which are used to compute spherical (in 3-D) and higher-dimensional release coordinates in our model. It is generally not feasible however to pre-identify a finitely countable set of argument values that will be encountered in an arbitrary simulation code and therefore it is not feasible to use a precomputed lookup table.

The animat simulation code, discussed in section 6 involves a similar use of square roots to ensure correct behavioural symmetries. In this case, much of the calculation can be formulated in terms of comparisons amongst squared values, and the expensive `sqrt` function

need not be evaluated in the inner loops.

4 Cluster-Cluster Aggregation

In a similar thread of research to that reported in section 3, we have also been experimenting with a Diffusion Limited Cluster Aggregation (DLCA) [11,18] code. Whereas the DLA code features a single stationary particle and at each step a single particle is released and allowed to diffuse until it aggregates with the existing particle aggregate, the DLCA code maintains a 'soup' of particles which are allowed to individually diffuse within the boundaries of our simulation. Akin to the DLA model, when the particles collide they aggregate. Thus, the system may initially contain many clusters of very small size. Over time the clusters aggregate to form larger and larger clusters. Clusters diffuse in certain directions according to their relative mass. A snapshot of our system is shown in figure 2.
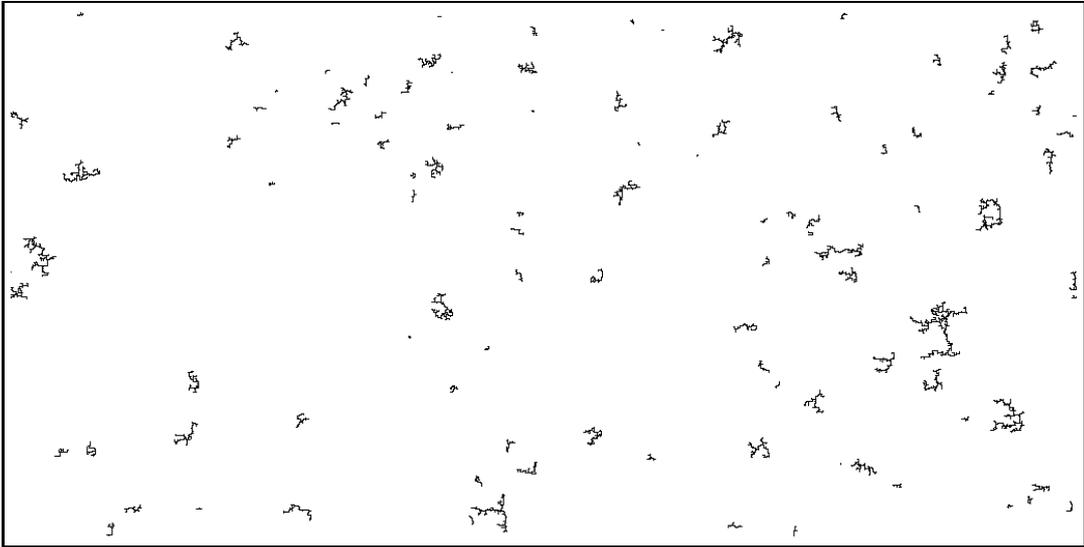

Figure 2: Snapshot of our system at simulation time 508 when run with Peclet number of 1.0 A drift force from right to left is applied, which biases the random choice of particle movement.

Internally, we represent the system as a lattice array, each of which specified an individual grid point within an N-dimensional space. Each particle has a unique index and this index is stored in the lattice array. Figure 3 shows an indirect addressing scheme that is useful for hyper-cubic symmetry lattice models in arbitrary dimension d. The lattice lengths $L_i, i = 1, ..., d$ in each dimension are fixed and hence particle positions $x_i, i = 1, ..., d$ on the lattice can be encoded as a single integer $k = x_1 + x_2 \times L_1 + x_3 \times L_2 \times L_1$ and so forth. These "k-pointers" can be used to identify the location of the m'th member particle in the j'th cluster.

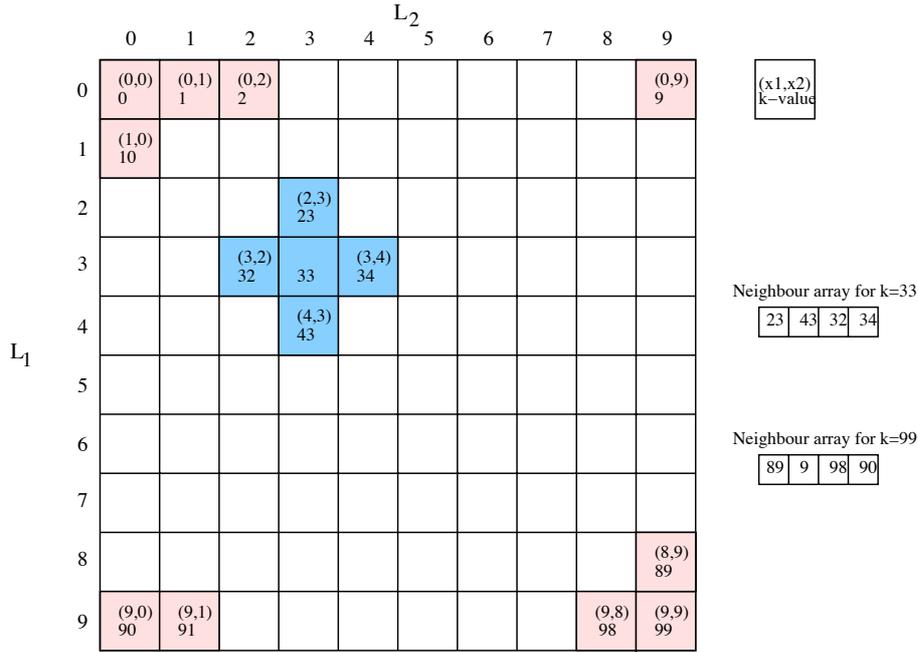Thus, while finding an individual particle involves searching through the whole lattice array,

Figure 3: In a 2-dimensional system with L[1]=10 and L[2]=10, we can assign a 'k' value (bottom number in each grid box) to each unique (x_1, x_2) lattice point (top number in each grid box). Functions can be written to convert between the 'k' value and the 'x-vector'. Using the 'k' value as an indirect referencing mechanism allows us to write our code for an arbitrary number of dimensions. Using periodic boundary conditions this system has two clusters; using non-periodic boundary conditions the system has five. Our simulations typically utilise periodic boundary conditions.

this is not a common operation in our current code. The main way in which we address our particles and clusters of particles is through the clusters array. When the system is first (randomly) initialised each of the particles will be a mono-particulate cluster. Each cluster is numbered and has an entry in the clusters array.

Our code uses a cluster data structure as shown in figure 5. The physical lattice uses conventional Cartesian coordinates (x, y, z) or $(x_1, x_2, x_3, x_i, .., .x_d)$ for a d dimensional system. The lattice is of size L_i in each dimension and we can encode the x_i into a single integer storage coordinate which we term a k index. The k index values are effectively pointers into our $\mathcal{R}^d$ lattice space and make it easy to group clusters in terms of arrays of k values.

The actual representation of cluster is also indirect. As the actual clusters change rapidly due to aggregation, it is beneficial to retain an indirect reference to these cluster numbers that are always packed in a dense array from 0 up to $n_{clusters}$ for ease of iteration across the list of clusters. All vectors in our code have elements numbered $0 \leq i \leq n$: element 0 is used to store the actual number of elements, i.e. $x[0] = n$ and the elements 1 through n are used for the data. This prevents us from having to maintain a separate data structure representing the number of elements in each array. This can be seen in the cluster lists in
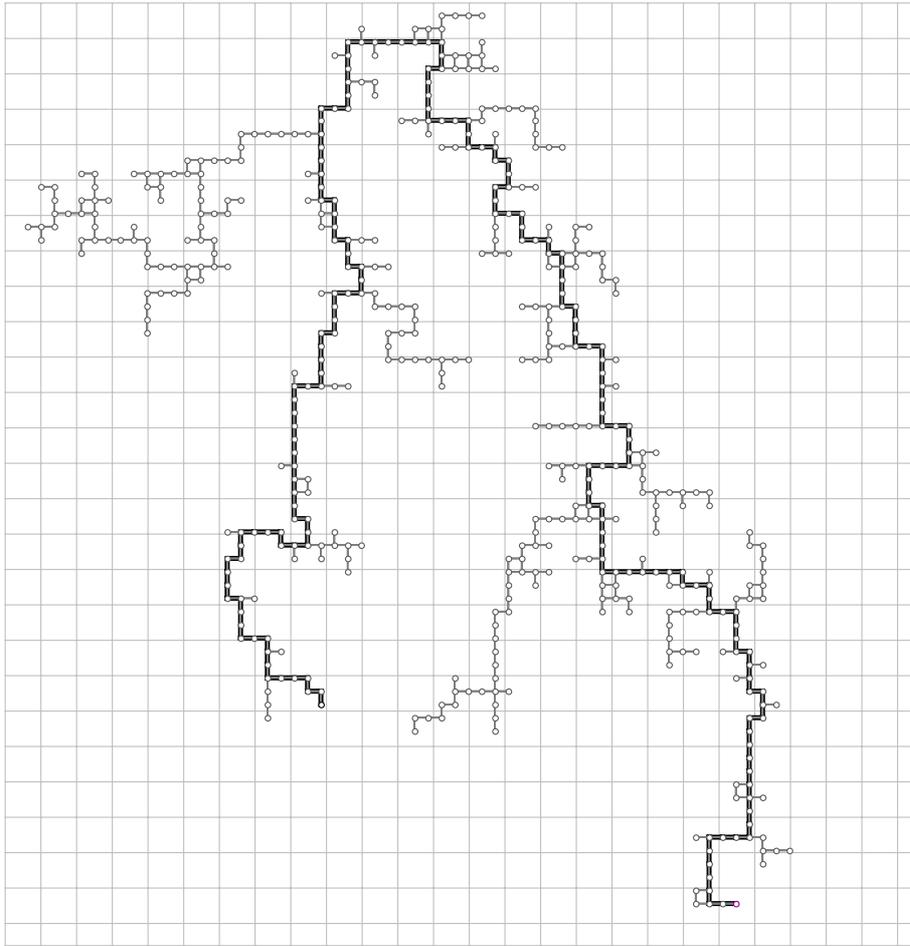
Figure 4: Structure of the largest cluster in a DLCA system simulation.

figure 5, where, cluster j has 7 elements.

The current state of the art in DLCA simulations has produced systems with hundreds of clusters with many million particles in each. For this reason it is necessary to be able to produce simulations of very large 2-, 3- and higher-dimensional systems. As can be gleaned from the discussion above, it is not merely sufficient that we represent our very large lattice: there are quite a few subsidiary data structures that can grow quite large, too. This requires the ability to access those regions of memory higher than the 4GB limit imposed on current 32-bit operating systems, or the 2GB addressable by signed 32-bit pointers. We have been successfully simulating very large systems using our Macintosh G5 processing nodes running the 64-bit Tiger operating system that we were not able using previous versions. Also required is the ability to instantiate a single variable with sufficient range to be able to effectively use memory above the 2GB limit. This is discussed further in section 7, but in our 64-bit version of this code we treat `size_t` structures as if they are `int`s for the purposes of array indexing, etc.
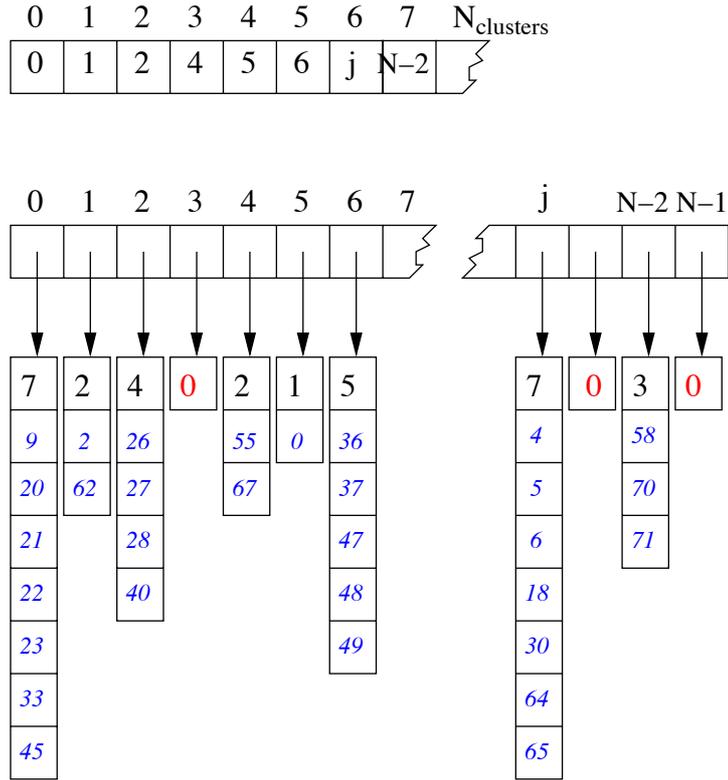
8

Figure 5: Clusters are accessed via an indirect array to allow for the frequent operation of iterating through clusters. The left-hand array is always densely packed, while the centre array, indexed by actual cluster number, may contain clusters with no elements. This centre array points to the actual cluster data array. For those clusters that contain particles, the first element is the size of the cluster, followed by the 'k' values of the particles.

One of the initial features that we built into this code was the use of a neighbour array. This array held, for each point in the lattice, the neighbouring 'k' values. Elements in the array were ordered: north, south, west and east, and in three dimensions, closer to and further away from the current array slice. Thus, we theorised, when moving a cluster or checking to see what would happen when we moved a cluster in a particular direction, we could always use the same positional index in each particle's neighbour array. We soon realised that this optimisation was not really one at all: we were expending a significant amount of valuable memory in representing a neighbour list that would not change: the lattice structure of our system was regular, as opposed to a Small World model [21]. On the off chance of making an improvement we changed the simulation to use a neighbour function, which computed the relevant neighbour's 'k' value on the fly. This turned out not to just save us the memory due to the neighbour array, but also sped up the code significantly. So what we originally thought was an optimisation was not really after all.

Another optimisation of this model was the introduction of Grid Boxing. This idea involved the division of the complete lattice into smaller grid boxes, into which clusters were over-

layed. Only searching the required grid boxes meant that we did not have to perform an $O(N^2)$ search. This technique is further discussed in [10].

5   Wolff and Metropolis Ising Simulations

Another high performance code we have been investigating is the behaviour of the Ising model of ferromagnetic substances when experiencing quenching [4]. We are particularly interested in the behaviour of this model when certain modifications have been made to the underlying lattice structure.

This model again simulates the behaviour of particles arranged on a regular lattice structure. Each particle is allowed a simple spin of 'up' or 'down'. When the model is 'hot' particles are able to change their spin quite easily; when 'cold' this transition is more difficult. In practice we observe that as the temperature of the model is lowered the particles tend to arrange themselves into domains of a similar spin.

The protocol for deciding whether a particle should change its spin is based on the type of update model that is used. In the Metropolis update model a particle is chosen at random in each time-step and a simple test is made to determine whether changing the spin will result in a lower energy configuration for the system (which is preferred). On average, every particle is sampled once during each update time-step. Thus, time-steps bear some close resemblance to physical simulation time.

In contrast, in the Wolff update model tries to construct a cluster of particles which will all change their spin. A site is chosen at random. Neighbouring sites, with the same spin, are added to the Wolff cluster according to some probability that is based on the temperature of the system. When no more particles are added to the cluster, all the particles' spins are changed. In this update model not all sites are sampled once every time-step, so there is little resemblance between the time-step and simulation time.

Figure 6 shows the time taken per update for the different update models. It can be seen that the Metropolis algorithm takes far longer per time-step. This is primarily because the algorithm hits, on average, each particle once per time-step, whereas the Wolff algorithm does not. However, when the system is quite hot, and there are very few neighbouring particles that can be coalesced into clusters of like spin for modification, the Wolff algorithm does take a long time to make a significant difference to the simulation state.

The slope of the lines in figure 6 is also quite instructive: Wolff updates (bottom line) show a slope of 3.118 ± 0.09 and Metropolis updates (top line) show a slope of 3.40 ± 0.07. That these lines are approximately straight in the $\log - \log$ plot suggests there is a power-based scaling law in evidence: the average time taken by the Wolff update algorithm is approximately $O(L^{3.118})$ and for Metropolis $O(L^{3.40})$.

In each of these models there is a very strong reliance on a fast, good, random number generator function. After initialisation, the code spends a significant amount of its run-time generating random numbers. We currently use the well-known Marsaglia random number generator [15] which has been shown to be free of correlations and has a large period.
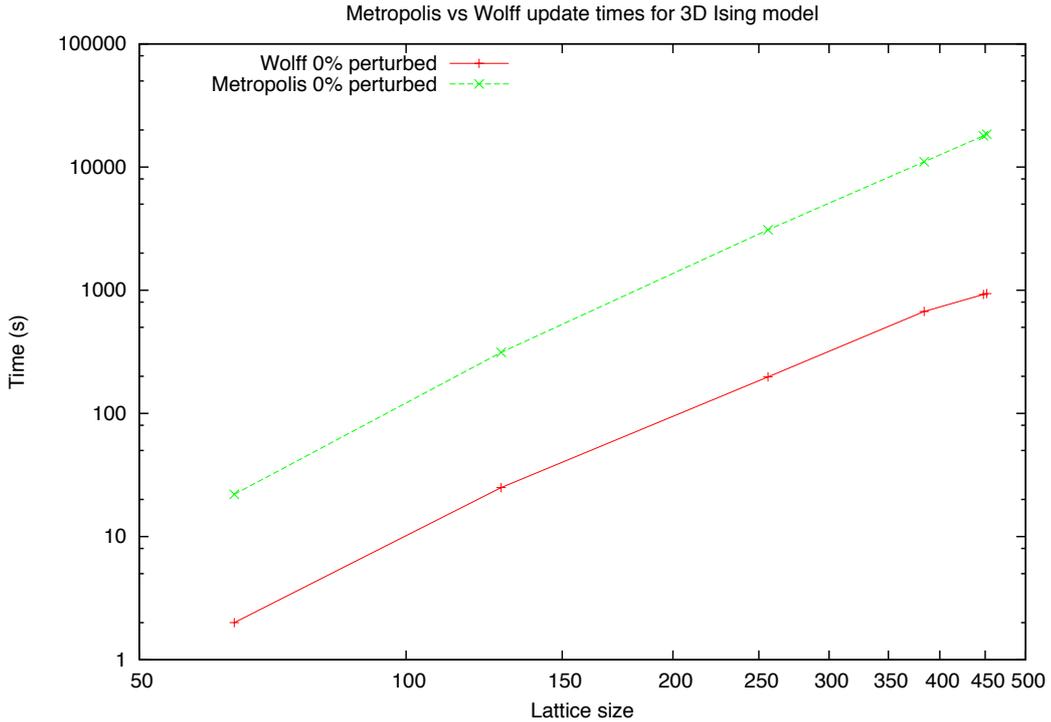
Figure 6: Timing information for different Ising update strategies over different lattice sizes in 3-dimensions. This graph shows the data on a $\log - \log$ plot. That the lines in are very close to linear in $\log - \log$ space leads us to hypothesise the relationship between average update time is related to the lattice size in 3 dimensions via a power-law.

Internally, we re-use the 'k-value' ideas discussed in section 4 and shown in figure 3: 'k-values' are a very convenient tool for representing a lattice site and utility functions can be written to convert to and from 'x-vectors'.

The most significant differences between the Ising code and the DLCA code is that firstly the particles do not move, and secondly, the lattice features a particle at every site. We must maintain an array of the current spins of each particle. We also need a temporary data structure in which we can create clusters of particles for the Wolff update model. We find representing the temporary data structures using `int`s or even `size_t`s that we do not require all the bits for this purpose. We are able to cram more than one use into this array by sharing it between routines with different purposes.

The memory requirements of the major structures in our model are approximately as shown in the following table. Note that there is an extra overhead in each case due to temporary variables and the processes' run-time stack, etc.

Dimensions	Structure	Memory Requirements
3	Neighbours array	sizeof(`unsigned int`) $\times L^3 \times (6+1)$
	Temporary array	sizeof(`unsigned int`) $\times L^3$
	TOTAL	sizeof(`unsigned int`) $\times L^3 \times 8$
4	Neighbours array	sizeof(`unsigned int`) $\times L^4 \times (8+1)$
	Temporary array	sizeof(`unsigned int`) $\times L^4$
	TOTAL	sizeof(`unsigned int`) $\times L^4 \times 10$

Before we had access to a 64-bit operating system we were bound to a maximum lattice size of 370 in 3 dimensions and 77 in 4 dimensions. Now we are once again compute-bound instead of memory-bound. The largest practical system we can simulate using a system with 4.5Gb of RAM (not in single-user mode) is 475 in 3 dimensions, an extra 1.1 million lattice points.

6   Animat ALife Simulation

Our animat ALife code is a specialist simulation, but exhibits many of the properties and algorithmic features that are common to other simulation areas. It is based on a set of N microscopic detailed components (animat [1, 22] agents) that interact locally and at non-trivial spatial separations. It exhibits many emergent behaviours [19] and properties that can only be effectively studied with large N and for a large number of system-evolutionary timesteps. Ideally we also want to repeat runs with the same parameters but different stochastically chosen starting conditions to statistically sample and appropriately average over some parts of the model's phase space.

The animat model is used to study emergent behaviour arising from the interaction of two distinct animat species - the predators and the prey. Each animat reacts to other animats in its vicinity, for example, a prey animat will attempt to flee from an adjacent predator, a hungry predator will move towards the nearest prey and so on. The model and the arising emergent behaviour have been described in [8, 9].

Thus it is necessary to calculate the nearest neighbours for each animat in every iteration. These calculations rapidly become lengthy as the number of animats rises because the check for nearest neighbours is an $O(N^2)$ problem. We have employed two techniques in the animat model to reduce the time required to locate neighbours. One approach has been to recognise that it is not necessary to check every other animat as the nearest neighbours must always be located reasonably nearby. Thus the area of the model has been divided into a grid and each animat has only to check the animats within its own grid box, except that animats near the edge of a grid box have to also check the adjacent box(es). This work has been described in [10].

A second approach to speeding up the search for nearest neighbours has been to avoid the use of square roots when calculating distances. The distance between an animat with coordinates (x_1, y_1) and another with coordinates (x_2, y_2) is calculated as $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. This involves a call to the `sqrt` function which requires significant time to return a result.
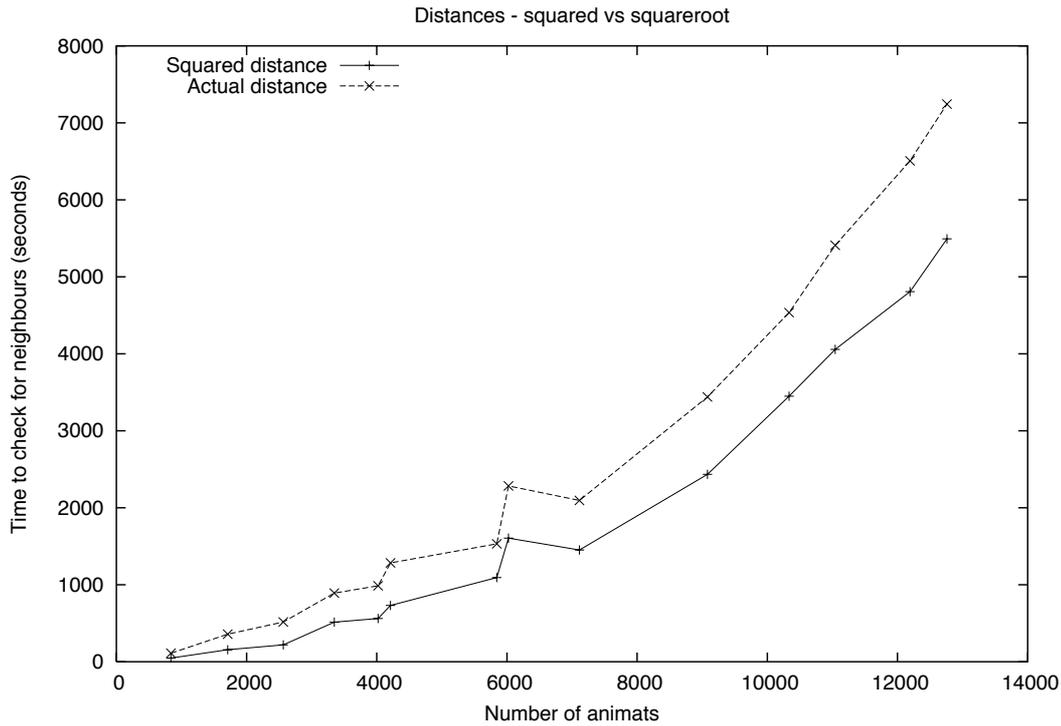
Figure 7: This graph shows the increase in time taken to find nearest neighbours as the number of animats increases. The decrease between 6000 and 7100 is caused by animats spreading out into more grid boxes, thus temporarily reducing the number of possible neighbours that need to be checked. It is always more efficient to use "squared distances" between animats rather than calculating actual distances which requires the use of the `sqrt` function.

During the search for nearest neighbours the actual distance between animats is not required. What is required is a comparison between distances to find which is smaller and hence which animat is closer. This comparison can be performed by using the squared distance $(x_1 - x_2)^2 + (y_1 - y_2)^2$ instead of the actual distance and thus the expensive `sqrt` function can be dispensed with.

A graph showing the effectiveness of this approach appears in Figure 7. The time taken to find the nearest neighbours generally increases as the number of animats increases although the slow down varies due to the effect of the grid system. As animats spread out into different grid boxes, the overall time taken may even decrease – see for example the decrease from 6000 to 7100 animats. But the graph clearly shows that the actual distance calculations, involving the `sqrt` function, are always slower than the squared distances which avoid calling the `sqrt` function.

The animat model also lends itself to the use of a square root lookup table as described in section 3. The vast majority of distances calculated are less than a known maximum (the size of a grid box) and thus actual distances could be calculated by looking them up in a table in order to avoid the `sqrt` function. The lookup table has not yet been implemented

as actual distances between animats are not required. However, as the model grows and becomes more complex it is possible that a lookup table may be incorporated.

7 High Memory Utilisation

Some of the performance achieving techniques we have described make considerable use of high memory and are effectively algorithms for trading memory off against computation. Recent introduction of 64-bit processor architectures with 64-bit memory bus structures means that in principle standard desktop and commodity computers are able to address more than the 32-bit addressable 4 Gigabytes of memory. Falling memory costs are also making the affordable, and Operating Systems are becoming available that are able to exploit such high memory amounts.

We recall the problematic issues for code portability that arose in the 1980's when 32-bit PC's were introduced to ultimately replace the then common 16-bit architectures. It was quite common then to have programs that needed to address single data structures that were bigger than 2^{16} addressing could accommodate and several *ad hoc* capabilities such as long and multi-segment addresses had to be introduced into operating systems [14].

We are now encountering similar issues where we now find it quite likely that one of our simulations will address more memory than a 2^{32} address system can accommodate. In practice, many programs are based on the assumption that 2's complement (signed) integers are used for array indexing and this means that many programs will implicitly be limited to 2^{31} elements.

The `size_t` type is a standard part of both the C and C++ programming languages and is meant to provide a maximal (limited by the processor architecture) unsigned integer suitable for memory allocation and addressing. It is not however a simple matter to upgrade all our software to use `size_t` instead of `int` since quite often index calculations, as part of the simulation algorithm or model go temporarily negative as parts of intermediate calculations. Unfortunately the weak typing support of C/C++ means that the compiler does not help us much and unsigned integer calculations that accidentally involve signed intermediates will silently overflow or wrap around in the 2's complement bit representation. Some considerable care is needed in porting codes to safely use data-structure addressing above 2^{31}.

The GNU C/C++ compiler suite [6] is still one of the most useful in terms of portability and performance. It offers the following useful data types:

- `int` - typically 32 bits, independent of processor architecture.

- `long` - typically also 32 bits, independent of processor architecture.

- `long long` - typically 64 bits, independent of processor architecture.

- `size_t` - typically the size of the native processor's register or bus structure.

We find the use of `size_t` is necessary as we often write our simulations to do array-index calculations as a single k-index, and thus to simulate multidimensional array-indexing. As it turns out this has proved fortunate as not all compilers will successfully support multidimensional array indexing if the resultant (internal) memory address can transcend 2^{31} addressable segments.

High-dimensional simulations with lattice edges of 4 or higher can easily give rise to such large array-indexes.

8   Summary and Conclusions

We have described several worthwhile tactics to optimisation of performance of various simulations codes. We estimate these have already saved us around a factor of three on our annual cluster computing budget. While we continue to agree with Hoare (and Knuth) that "premature optimisation is the root of (much) evil" [13], we believe that "lack of any optimisation at all is the root of much wasted resources". It has unfortunately become fashionable to dismiss performance optimisation in the mistaken belief that modern compilers can somehow do all the work for the programmer. This is clearly untrue and careful consideration of the memory tradeoffs and data structures we use in high performance simulations codes is effort well spent.

Since the C and C++ and indeed most modern languages will generally only have a native maths library that is optimised for double precision calculations (64-bit doubles) it is now very useful that 64-bit microprocessor architectures are available and affordable. It has been an troublesome issue for recent years that a numerical code usually has to be written carefully to compromise between using 32-bit floats or 64-bit doubles. It is a not widely remembered fact that most C/C++ compilers and their run-time systems will simply promote 32-bit floats to doubles for maths calculations, call the double precision maths library function in question, then coerce the result back into a 32-bit float data item. This is good for numerical precision in general but slow and does not necessarily represent a good value for cycles budget for an old code running on a modern architecture. Ideally in future we want a hardware architecture that supports 64-bit bus structures for high-memory access as well as 64-bit floating point to support a high quality and fast maths function library. We also need an operating system and run-time library that fully supports this and our applications codes need to be able to make use of the full 64-bit capability. Modern processor architectures and operating systems are now meeting these needs.

We believe that applying techniques like we describe allows the best use to be made of existing compute resources. We believe that many of the simulations codes we work with, especially for spatial-based simulations, benefit from these considerations and that performance optimisation is still a worthy art form.

References

[1] Adami, C. Avida (Digital Life Laboratory) Available at http://dllab.caltech.edu/avida

[2] Advanced Micro Devices, Inc. Opteron 64-bit processors. Available from http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8825,00.html

[3] Apple Computer, Inc. G4 and G5 Systems architecture. Available from http://www.apple.com/hardware Last visited October 2005.

[4] Baillie, C.F., Gupta, R., Hawick, K.A., and Pawley, G.S., Monte-Carlo Renormalisation Group Ising Calculations, Phys.Rev.B 45, 1992, PP10438-10453.

[5] UNIX Berkeley Software Distribution `random` manual page.

[6] Free Software Foundation, GNU Compiler Collection. Available from http://gcc.gnu.org, last visited October 2005.

[7] Grove, D.A., Performance Modelling of Message-Passing Parallel Programs. PhD Thesis, the University of Adelaide, 2003.

[8] Hawick, K.A., James, H.A. and Scogings, C.J. Web site, containing model Snapshots and Movie sequences: *http://www.massey.ac.nz/~kahawick/alife* Massey University 2004-5

[9] Hawick, K.A., Scogings, C.J. and James, H.A. Defensive Spiral Emergence in a Predator-Prey Model in Proc. Complexity 2004, Cairns, Australia, December 2004.

[10] Hawick, K.A., James, H.A., and Scogings, C.J., Grid Boxing for Spatial Simulation Performance Optimisation, submitted to 39th Annual Simulation Symp, Huntsville, AL, April 2006.

[11] Hellén, E.K.O., Salmi, P.E., and Alava, M.J., Cluster Persistence in One-Dimensional Diffusion-Limited Cluster-Cluster Aggregation, Physical Review E 66, 051108 (2002). http://de.arXiv.org/abs/cond-mat/0206139

[12] James, H.A. and Hawick, K.A., Scientific Data Management in a Grid Environment, Journal of Grid Computing, 20 September 2005, DOI; 10.1007/s10723-005-5464-y; ISSN: 1570-7873 (Paper) 1572-9814 (Online). (SpringerLink)

[13] Hoare, C.A.R. as restated by Knuth, D.

[14] Kuznetsov, A., Integer 64-Bit Optimizations, Dr. Dobb's Journal March, 2005.

[15] Marsaglia, G., A Current View of Random Number Generators, *Computer Science and Statistics, The Interface*, Elsevier Science Publishers B. V. (North Holland) L. Billard (ed.), 1985.

[16] Metropolis, N., Rosenbluth A.W., Rosenbluth M.N., Teller, A.H. and Teller E., Equation of state calculations by fast computing machines, in J. Chem. Phys., 21(6), pp 10871092, June 1953.

[17] Meakin, P. Diffusion-controlled cluster formation in 2–6-dimensional space, Phys. Rev. A 27, 1495–1507 (1983).

[18] Peltomäki, M., Hellén, E.K.O., and Alava, M.J., No self-similar aggregates with sedimentation, J. Stat Mech, JSTAT (2004) P09002.

[19] Ronald, E.M.A., Sipper, M. and Capcarrère, M.S. Testing for Emergence in Artificial Life, In Advances in Artificial Life: Proc. 5th European Conference on Artifical Life (ECAL'99), Switzerland, 1999 Ed. Dario Floreano and Jean-Daniel Nicoud and Francesco Mondada, Pages 13-20, Pub Springer-Verlag ISBN 3-540-66452-1.

[20] Soderquist, P. and Leeser, M., Division and Square Root: Choosing the Right Implementation, IEEE Micro, July/August 1997(Vol. 17, No. 4) pp. 56-66.

[21] Watts, D.J. and Strogatz, S.H., Collective dynamics of small-world networks, Nature (393), 4 June 1998.

[22] Wilson, S. The Animat Path to AI in *From Animals to Animats 1: Proceedings of The First International Conference on Simulation of Adaptive Behavior*, (pp. 15-21); Meyer, J-A. & Wilson, S. (eds), Cambridge, MA: The MIT Press/Bradford Books (1991).

[23] Witten, T.A., and Sander, L.M., Diffusion-Limited Aggregation, a Kinetic Critical Phenomenon, Phys. Rev. Lett. 47, 1400–1403 (1981)