

ACCELERATING A METROPOLIS RANDOM WALK AND IMMERSION-METHOD SADDLE-POINT ALGORITHMS IN MULTIDIMENSIONAL NUCLEAR POTENTIAL-ENERGY SPACES

JUSTIN WILLMERT[†]

School of Physics and Astronomy
University of Minnesota
Minneapolis, MN
willmert@physics.umn.edu

KEMPER TALLEY[†]

Bredesen Center for Interdisciplinary
Research and Graduate Education
University of Tennessee–Knoxville
Knoxville, TN
ktalley5@utk.edu

UNDER THE MENTORSHIP OF

PETER MÖLLER

Nuclear & Particle Physics, Astrophysics & Cosmology, Group T-2
Los Alamos National Laboratory
Los Alamos, NM
moller@lanl.gov

AS PARTICIPANTS OF

LOS ALAMOS NATIONAL LABORATORY'S^{*} COMPUTATIONAL PHYSICS STUDENT SUMMER WORKSHOP

Sponsored by LANL's Advanced Scientific Computing Program

ABSTRACT

We present an analysis of and improvements on the Möller et al. [8] utility used to simulate the fission-fragment charge-yield distributions by the Metropolis method. Due to particular energy-surface properties, particularly as mass decreases, the random-walk procedure could wander for very long tracks and required prohibitively long run times to collect sufficient statistics. Our work has decreased the execution time by taking advantage of parallelization available to symmetric multiprocessing (SMP) systems exposed through the standard OpenMP language extensions and libraries [13]. Because of the intrinsic parallelizability of the random walk, our efforts have demonstrated that nearly linear speedups with the number of processors are possible within certain critical sections of the algorithm. The same code modernization and application of OpenMP constructs has consequently also resulted in a greatly accelerated minima search and associated saddle-point determination. Important for future work, the parallelization effort has resulted in modular code which can more easily be reused than the original monolithic program.

^{*}Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396.

[†]Also Nuclear & Particle Physics, Astrophysics & Cosmology, Group T-2

1 Introduction

High-performance computing has permitted many advances in all branches of physics including the theoretical work being done in nuclear physics. In the past, relatively simple models existed for calculating the energy of a ground-state nucleus by using the liquid-drop model. As computing capabilities have increased, though, so has the complexity—and maybe more importantly the associated computational costs—of the models. One such model is the macroscopic-microscopic model used to calculate nuclear parameters for a wide variety of shapes. Comparing the charge/mass distribution and associated asymmetries, such as in the particular case of ^{188}Hg , proves the efficacy of this model and therefore the utility of such computations. An important goal beyond the theoretical work to generate these models is then to implement them in computationally efficient algorithms.

2 Mass models

A naïve attempt at calculating the masses of nuclei would simply sum the total masses of all protons neutrons, and while this will provide a reasonable first estimate, detailed predictions are sensitive to the inaccuracies inherent in this method. The issue stems from Einstein’s energy-mass relation wherein the (negative) binding energy, which can be both theoretically and computationally difficult to obtain, has important consequences on the mass and on other important properties of the nucleus. An improved technique might attempt a full quantum-mechanical calculation of the mass, but because the nature of the strong force makes the requisite quantum chromodynamics calculations computationally impractical if not impossible, mass models are of critical importance to the study of nuclear interactions. Since masses can experimentally be measured with high precision, this allows mass to be a valuable instrument for testing various models. In the following sections, we review several important milestones in the mass models that ultimately lead to the specific model used in our calculations, the macroscopic-microscopic model (§2.5).

2.1 Liquid drop model

One of the first successful mass models utilized was the liquid-drop model (LDM). It was developed before the discovery and description of the strong force, and therefore it is a purely phenomenological model constructed from the patterns seen in experimental data [2]. Experiments show that the nucleon density is nearly constant, proportional to the nucleon number A . This naturally led to a description of nucleons as an incompressible fluid with associated volume (A) and surface ($A^{2/3}$) interaction terms. The volume term is the attractive term due to the mutual binding between nucleons while the surface term is a repulsive correction term because the surface nucleons have fewer neighbors than the interior nucleons. Finally, protons repel one another due to the Coulomb interaction, which is repulsive and gives rise to a term proportional to $Z^2 \cdot A^{-1/3}$.

Further corrections to the classical model can be included from quantum mechanics. An asymmetry term accounts for the energy balance involving the neutron excess, and the particular form for the term shown below (coefficient a_A) can be derived from considering the nucleus to be a two-fluid Fermi gas and keeping the first-order expansion term of the kinetic energy [10]. (In fact, the zeroth-order expansion term reproduces the volume term, confirming it as an appropriate choice.) The final term is the pairing term ($\delta \cdot A^{-1/2}$) which accounts for the observation that nuclei have increased stability when they have even numbers of protons and/or neutrons, interpreted as having complete spin pairs. These two additional quantum terms together with the classical terms produce the Bethe-Weizäcker or semi-empirical mass

formula:

$$M(A, Z) = Zm_p + (A - Z)m_n - \frac{E_B}{c^2} \quad (1)$$

where

$$E_B = a_V A + a_S A^{2/3} + a_C \frac{Z^2}{A^{1/3}} + a_A \frac{(A - 2Z)^2}{4A} + \frac{\delta}{A^{1/2}} \quad (2)$$

The coefficients (a_V , a_S , a_C , and a_A) are fitted constants over a range of nuclei while δ is fit separately for nuclei with even-even, even-odd, or odd-odd proton/neutron numbers.

Both the LDM and semi-empirical mass models as presented assume spherical (typically ground state) nuclei. For excited nuclei—and of particular interest, nuclei undergoing fission—the nucleus can be deformed. Appropriate modification of the classical terms in Eq. 2 allow calculations for any shape given by any chosen shape parameterization; here we will focus on the three-quadratic-surface shape parameterization used in the macroscopic-microscopic model in §2.3, but further discussion of the Weizäcker formula can be found in Povh et al. [10, §3.3] or Das and Ferbel [2, §5.2.1].

2.2 Shell model

In contrast to the LDM macroscopic interpretation, shell models use a microscopic view as that of independent particles moving within a nuclear potential, and the associated single-particle orbitals are solved quantum mechanically. Because the nucleus is not a dense system—the ratio of the closest packed volume of the nucleus to the actual volume is $\approx 1/100$ —nucleon-nucleon forces deviate very little from a mean effective potential [11, §2.1]. Thus the nucleus can be considered as made up of these independent particle states and solved in a manner analogous to the idea of the Hartree-Fock potential of the atom. Also like the electron in the atom, the single-particle solutions for spherical shapes have degenerate states that lead to distinct shells. These shells (the magic numbers) serve as the basis for further corrections to the binding energy, particularly in the Strutinsky method, in a manner parallel to the use of the noble gases for calculating ionization energies. The shell model theory is extensive, and as such, we refer the reader to Ring and Schuck [11] or Wong [16] for a more complete discussion. We instead move on to introduce the deformation which is instrumental in constructing the introductory macroscopic-microscopic model.

2.3 Deformation

Light nuclei are almost always spherical due to the nuclear force, but massive nuclei have a more detailed energy balance. The Coulomb force grows more powerful as Z increases, and since Coulomb forces are inversely proportional to the square of the distance, the most bound states result by putting protons as far away from one another as possible. Because nuclear volume is invariant, though, the shape must change. Adding the saturation property which leads to further decreases in binding energy per nucleon for higher A , the interplay between these forces may favor a non-spherical shape.

In particular, nuclei are spherical at closed shells (magic numbers), but as nuclei progress from one major shell to another they transition through prolate shapes (elongated along the z -axis) and then oblate shapes (flattened at the poles) as they approach the next major shell. The reason for this preference in shape is discussed in detail in Wong [16, Ch. 6]. The shape of the nuclei can be reasonably expected to change the distribution of the individual nucleons and as such will change the overall binding energy. Each shape corresponds to a distribution of nucleons that influence the nuclear binding energy considered in the calculations of the macroscopic-microscopic model (§2.5).

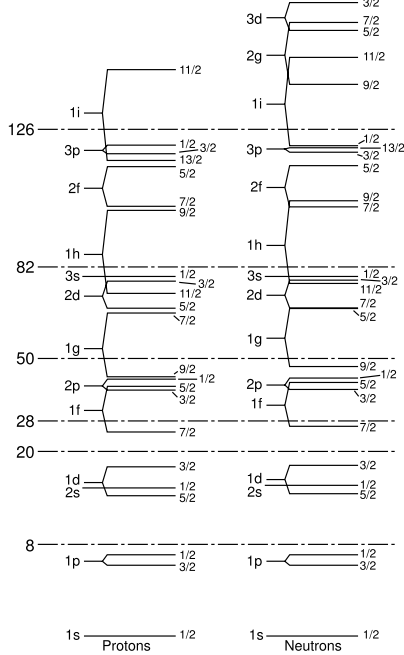


Figure 1: A schematic showing the nuclear levels in spherical nuclei for protons and neutrons. The right parts of the proton and neutron level schemes show how the extreme degeneracy is broken when spin-orbit coupling is included. Note the occurrence of magic numbers which correspond to particularly stable configurations. From Povh et al. [10].

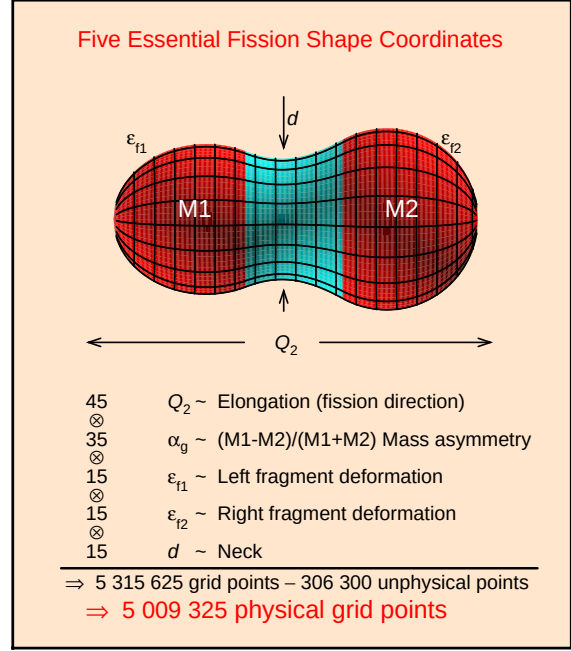


Figure 2: In the Möller et al. [8] presentation, five physical quantities parameterize the shape and energy of the nuclei. Each quantity can be related to the macroscopic-microscopic parameters typically found in the literature. Note that there are 306,400 unphysical points in the model that are included strictly for computational purposes. From Möller et al. [8].

2.4 Strutinsky method

Neither the liquid-drop model nor the shell model are independently capable of accurately describing the binding and mass energy of all nuclei. The liquid-drop model captures the bulk trend but cannot accurately describe individual nuclei. In contrast, the shell model can calculate the properties of magic nuclei well, but fails to predict the bulk properties. In order to span the gap between both models, Strutinsky created a shell correction procedure which combines the best qualities of both models [12]. It reproduces the experimental ground-state energies as well as their dependence on deformation parameters.

The Strutinsky method describes the oscillatory deviations of the LDM binding energies from the experimentally observed binding energies, which are due to shell closures. The oscillatory nature of the binding energy between shells has its maximum at the shell closures (i.e. magic numbers: 2, 8, 20, 50 ...). In the shell model one would calculate the energy by including this oscillatory part plus an “average” level density, however, this average was wrong. Strutinsky instead replaced it with the LDM energy while still keeping the fluctuating part from the shell model. More details of how the oscillatory energy can be extracted from the shell model are found in Strutinsky’s paper [12].

2.5 Macroscopic-microscopic model

The macroscopic-microscopic model uses smooth trends from a macroscopic model and local variations from a microscopic model to describe global properties. The total potential energy—dependent upon Z , N , and various shape parameters—is the sum of these macroscopic and microscopic terms.

$$E_{\text{pot}}(Z, N, \text{shape}) = E_{\text{macro}}(Z, N, \text{shape}) + E_{\text{micro}}(Z, N, \text{shape}) \quad (3)$$

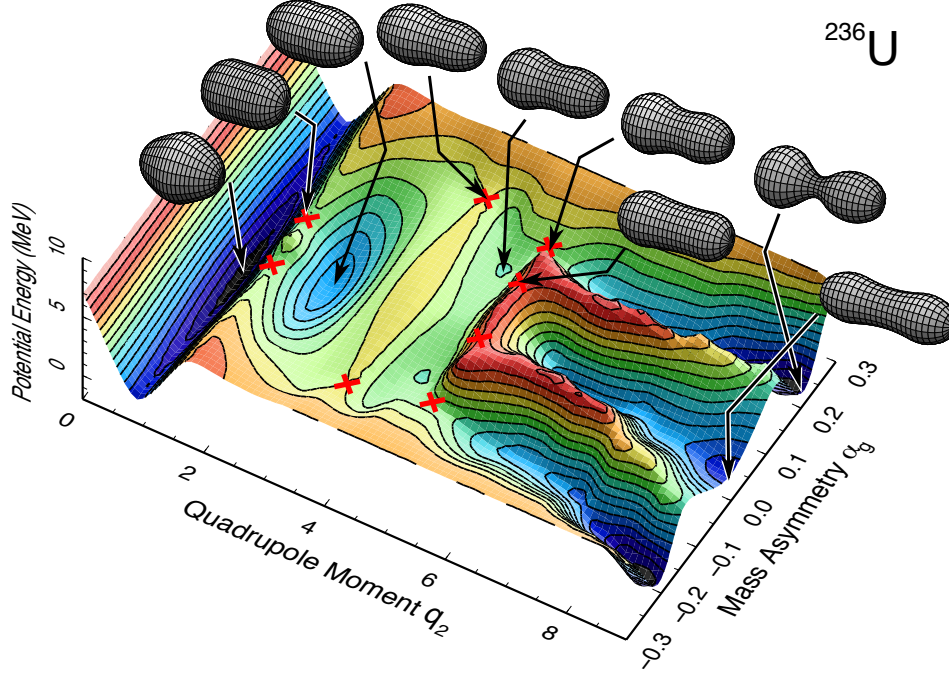


Figure 3: Two-dimensional surface approximation of the five-dimensional potential-energy surface used. Key features such as minima, saddle-points (red crosses), ridges, and valleys have been captured in this approximation. From Ichikawa et al. [6].

For a specific nucleus, the macroscopic term corresponds to the LDM energy, and the microscopic term is determined by the following prescription. (1) A shape is prescribed using all five parameters given in Fig 2. (2) A single-particle potential with this shape is generated, including the spin-orbit term. (3) The Schrödinger equation is solved for this deformed potential, generating the corresponding single-potential levels and wave functions. (4) The shell correction is calculated by use of the Strutinsky method. (5) Finally, the pairing correction is calculated in the BCS or Lipkin-Nogami method. By this method, the energy hypersurface can be computed for each of the 5 million shapes. A full description of the calculation can be found in Möller et al. [8, 9].

This energy surface—such as is shown in Fig. 3 in a two-dimensional approximation of the full five-dimensional surface—is then used as the base for the Metropolis method (discussed in §4) that calculates various properties about the fission of a selected nuclei. This energy surface can be used to search for minima and saddle points as well, both important features that are used to compute further properties of nuclear fission.

3 Saddles in multiple dimensions

Barrier energies provide powerful criteria on transitions in many physical systems. It is important, then, to extract the barrier energy from the data contained within the energy hypersurface. Computationally this corresponds to finding the saddle-point between a given pair of minima or between a minimum and the lowest energy for the most elongated set of shapes, which in our calculation corresponds to almost separated fission fragments. These saddle-points correspond to the critical deformation of the nucleus at which the nucleus will be irreversibly committed to fission. It is important to note that there are many saddle-points on such a hypersurface. However, the points of interest are those that correspond to barriers. Möller et al. [8] discuss at length how the typical path-minimization methods for determining saddles between minima in multiple dimensions is insufficient and can produce incorrect or

misleading results. We have assumed the use of the immersion method which Möller et al. argue is the correct strategy, but an interested reader can refer to their paper for a discussion on the pitfalls of the path-minimization methods. We first begin by introducing the immersion method in a concrete, physical example and describe the Möller et al. implementation specifically in §5.2.1.

3.1 Immersion method

The immersion method for determining saddle-points is described in an easily accessible way by Hayes [5]. He presents the method in the context of solving the continental divide problem, namely, how does one determine the location of the continental divide? In a single-dimensional landscape, the problem is trivial with the divide defined by the maximum value along the axis. Already in two dimensions, though, the maximum on the height gives a clearly incorrect answer; in an example using North America as the prototype for a two-dimensional surface, Hayes gives the example that “when you search out the highest point in the lower 48 states, you find yourself atop Mount Whitney, in California, elevation 4,418 meters. But Mount Whitney is nowhere near the continental divide.” A second method Hayes considered was to classify each point by the features of its 4 cardinal neighbors—whether a neighbor is higher or lower—and follow a continuous ridge across the continent. He concluded, though, that the 81 combinations in 16 classes of configurations was an untenable idea. In our case, the situation is drastically worse. Having 242 nearest neighbors in five dimensions leads to an incredible $2.9 \cdot 10^{115}$ configurations! Considering the problem further, he concluded that purely local methods are insufficient to determine an intrinsically non-local feature.

The solution as Hayes and others—particularly in geography and topography—have made use of (see Vincent and Soille [15]) is to instead perform a global operation and raise the sea level. The continental divide is then simply the line where the Pacific and Atlantic Oceans’ waters meet without mixing. Digitally this can be achieved by keeping track of which locations are dry and which are wet, incrementally increasing the water level on both sides at equal rates.

4 Metropolis random walk

The Metropolis random walk is a Markov chain Monte Carlo method for producing a random walk through the potential-energy hypersurface. A pictorial analogy is that of a drunk man walking around with his path more favorably choosing some directions over others due to topographical features such as hills or valleys. The drunk man is more likely to move down the hill than up it, but there is still a chance that he will climb it. In a similar manner, we simulate the shape evolution of a nucleus by having it “walk” through the potential-energy hypersurface. We initialize the algorithm by starting the walk at a minimum with a given excitation energy above the surface, and evolve the shape according to a simple algorithm until scission occurs: (1) Choose a neighboring cell at random, corresponding to an incremental shape change. (2) If the new shape has a lower energy than the current shape, immediately walk to the cell. (3) Otherwise, step to the new cell only with a probability equal to a Boltzmann-like factor from the energy difference and an effective temperature. (4) Check whether the neck radius is compatible with a scissioned state. If so, the algorithm is finished and the final shape parameters are recorded, otherwise the process repeats for another step in the Metropolis random-walk path.

A single Metropolis random walk is insufficient to provide insights into the dynamics, so an ensemble of paths are collected. From the ensemble of results, various properties such as the mass/charge distributions of the fissioning system can be obtained. Confidence in the conclusions drawn requires sufficiently large statistics, and because the procedure must be run for thousands of nuclei, producing a computationally efficient implementation of the Metropolis random walk is important to active research goals.

5 Implementation

We have made an attempt to follow Donald Knuth’s advice as stated in his popular quotation, “We should forget about small efficiencies, say about 97 % of the time: premature optimization is the root of all evil” [7]. Our improvement efforts were directed by the results that were collected from various performance analyzers. We did this for two primary reasons. First, the time we had to complete our work was limited, so we could not afford to spend time on “optimizations” which later proved to be ineffective. Second, highly optimized code often looks very different than its unoptimized form, usually at the expense of readability. Just for the sake of maintaining easily understandable and maintainable code it is important to avoid unnecessarily obfuscating the code.

For example, an early preliminary idea of what caused the original algorithm implementations to perform poorly was poor cache use. The common solution to such a problem is to change the data storage model to be more cache-friendly. These data structures, though, are often very complex and using them directly in algorithm implementations is nontrivial and error prone. In a language like C++, powerful template metaprogramming techniques can be used to abstract the data structure so that a simple interface is presented while simultaneously mitigating the run-time cost through compile-time optimizations. Fortran does not have the capacity for the same type of compile-time optimized metaprogramming, so we would have had to choose between using an error-prone direct implementation (thereby increasing the debugging time) or sacrificing some performance to build and use an abstract interface. Furthermore, Fortran’s lack of an extensive standard libraries (like C++’s STL or Boost) on top of which to build more complex data structures and algorithms would have required writing the basic building blocks from scratch, greatly increasing the implementation time.

Our first goal was to transform the original serial implementation of the random walk into one that could run individual walks in parallel, but before parallelization constructs could be used, the existing code needed to be updated and cleaned. The reasons will be discussed in §5.1. In the process of updating the random-walk code, several reusable sections of code made the reimplementing of the saddle-point algorithm convenient, and several algorithmic changes we introduced will be discussed in §5.2. Throughout this process we made use of two performance analysis tools: the GNU Project’s `gprof` [4] and Valgrind’s `cachegrind` tool [14]. Because the outputs from both programs tend to be long, we do not present the results and analysis inline and have instead placed the discussions in Appendices A.1 and A.2. We also explored the use of the Tuning and Analysis Utilities (TAU) [3] which makes use of hardware counters available on many modern architectures. Configuration problems and associated complications made this method of analysis ineffectual within our time table and did not factor into our analysis. As a consequence, TAU will not be further discussed here, but it presents a promising task for a future project to expand upon the analysis we have already performed.

5.1 Updating the code base

The first stage in implementing parallelization features into the random walk and saddle determination algorithms was to update the code base from FORTRAN77 to Fortran 95. The original FORTRAN77 code was not structured in a way that allowed for the use of OpenMP [13], our choice of parallelization technology. Another motivation for the modernization was to create a self-consistent software package. We were provided the complete FORTRAN77 source for random walk routines from Peter Möller and code snippets for reusable routines such as routines to load the data from disk, but we were independently implementing the saddle-point algorithms from scratch. To make best use of our time, we preferred a consistent coding style and set language features.

The first stage of changes focused solely on updating the syntax from FORTRAN77 style to Fortran 95. These changes included moving away from the fixed column layout to free form so that logical

code blocks could follow a consistent indentation style, replacing use of labeled CONTINUE statements with equivalent end do or end if block constructs, and converting extensive use of GOTO statements with if/else pairs. An ongoing effort began at this stage as well to explicitly declare all variables rather than using implicit typing. Requiring explicit variable declarations aided development by enabling compile-time warning and error messages that were capable of catching more type mismatches or unused variables, greatly helping the cleanup effort. Because the code had to be disentangled, though, often the implicit none feature could not be enabled for a procedure or module until several more stages of our code updates had also been completed.

The next stage of development was to extract the monolithic code base into independent modules. There are several procedures which were reused in every application, but the original implementation used a copy-paste methodology for providing the necessary routines. Modules also have the advantage that an application programming interface (API) can be generated which declares public interfaces and hides routines intended for internal use only. The utility of this strategy is demonstrated by example of the routines used to load the matrix data from disk. The data is stored as ASCII (plain text) values, and the original loading routine provided an interface to read the data and arrange it into the five dimensional matrix that is used in subsequent calculations. Parsing the ASCII, though, is a slow process; binary files can be loaded into memory in a fraction of the time, but they also pose a portability problem (data alignment, endianness, etc). We introduced two new specialized methods to load either the original ASCII data which is guaranteed to be portable or load a binary dump of the in-memory matrix that may be compiler, architecture, or Fortran version dependent. The original interface was then used to automatically load the binary cache of the data if it existed or read the ASCII file and generate a cache file for the next use. In this way, all programs compiled against the module automatically gained a new feature without requiring any additional work.

The final major stage of code cleanup was to replace extensive use of COMMON blocks to share data between procedures with explicit passing in procedure calls with user-defined derived types. In addition to improving code readability by making possible side-effects more obvious, the step was also critical to adding OpenMP support. COMMON block require that the OpenMP scope (such as threadprivate or shared) be declared at every usage. This is error-prone since any missing update could lead to both compile-time and run-time errors. With derived types, though, a variable must only declare its scope once where the parallelization instructions are used. Subsequent use of the data in calling child procedures automatically make use of the correct data, whether that be a local thread copy or a shared global resource.

5.1.1 Parallelization with OpenMP

The large effort necessary to update or rewrite code was rewarded through the simplicity with which parallelization was added using the OpenMP framework. In an ideal situation, parallelizing a loop without OpenMP directives would effect no extra changes to the code. In more realistic situations, though, some shared state typically needs careful consideration and possibly specialized code. Keeping special cases to a minimum, though, helps ensure that the program output and behavior is identical whether run serially or in parallel on an SMP system. In Fortran, OpenMP declarations are added as comment lines with a special syntax that instructs an OpenMP-aware compiler while simultaneously maintaining compatibility with OpenMP-ignorant compilers.

We added !\$OMP PARALLEL DO statements surrounding two so-called hot loops—critical loops that consume the majority of computational resources: one in the random walk algorithm and one in the virtual flooding used to identify saddle points. In the random-walk algorithm, we parallelized over multiple paths. The act of a random walk is completely dependent on its history so an individual path cannot split into multiple work units. We were interested in generating an ensemble of paths, though,

and since every path is inherently independent from any other, it is almost a prototype for parallelizable algorithms; only a small amount of end-state data must be collated across all paths. In the flooding algorithm, a complete and coherent flood map is required prior to expanding the flood, so parallelizing must be restricted to a single step, where a single step is defined as growing the flood to only its nearest neighbors (subject to energy constraints). Utilizing the popular idea of double-buffering from computer graphics, though, each individual step can be parallelized across the examination of the greater than five million elements in the flood map and energy matrices. Given an input flood map, a parallelized flooding step can iterate across all cells in the matrix, examining each cell's nearest neighbors in the input map, and outputting any changes to a different output map. This guarantees that all threads see the same input state no matter in which order the map is updated.

A few modifications required to have the OpenMP parallelization function properly are of note: (1) large stack-allocated variables must be avoided, and (2) we swapped out the original random number generator for that used in the Monte Carlo N-Particle Transport Code version 5 (MCNP5) software package [1].

Stack variables pose problems for OpenMP parallelized code because the program stack can quickly be exhausted, leading to segmentation-fault errors. This is because every instance of the procedure running in parallel requires its own copy of stack variables, so the memory requirements scale with the number of processors. The simple fix for this is to make use of dynamically allocated memory that is carved out of the heap. The heap is typically much larger than the stack, and can grow dynamically even larger if necessary. The downside is that extra effort is required to release the memory appropriately to avoid leaks that consume extra resources.

We switched to the MCNP5 random number generator because the original generator was incompatible with running in parallel. First, time would have been required to properly protect common blocks and global variables for use in parallel code. Doing so would have been possible, but without a skip-ahead procedure, the effort would have been wasted. A critical goal of parallel algorithms is to produce identical results no matter how many threads are used during execution. In an algorithm that makes use of random number sequences, this means that a particular identifiable part of the problem should start from a definite location in the pseudo-random number sequence. The original generator could only accomplish a skip ahead by generating every value in the intermediate sequence, a very costly process. In contrast, the MCNP5 generator, because it was designed to be used in a parallel code, was created with an efficient skip ahead procedure; only a few algebraic operations are required to skip to any arbitrary location in the sequence. We made use of this feature to produce identical results whether our code was run on a single processor or sixteen.

In addition to adding parallelization to the flooding process, several algorithmic changes were implemented. These changes will be discussed in the following section before we return in §6 to present the run-time performance gains we measured.

5.2 Algorithmic updates

As part of an introductory exercise to the code base and methods, our mentor tasked us with implementing our own version of the minima identification and saddle-point determination algorithms. Fruitfully, this effort provided more than just an educational exercise: it also resulted in several enhancements to the algorithm over the original implementation. We were provided high-level, abstract descriptions of the algorithm, but in the absence of a reference code, our approach and implementation choices were different. In the next few sections, we outline these differences and describe how they have contributed to our more efficient solution.

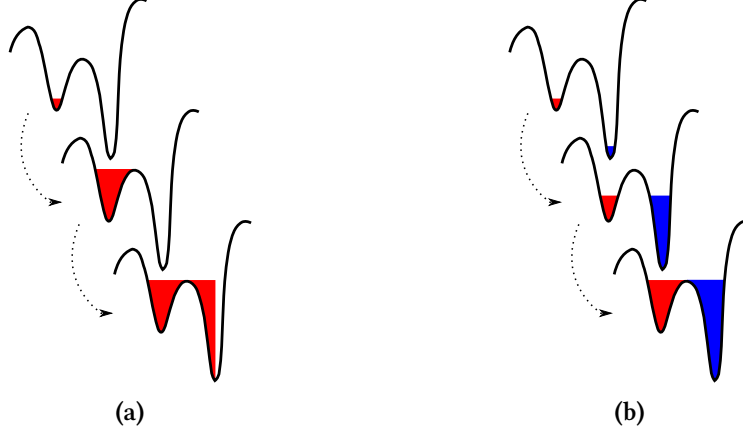


Figure 4: A simplified comparison of the original and new flooding algorithms implemented for saddle determination. (a) In the single-flooding algorithm, only one of the minima are made wet. The flood is stepped successively higher until it floods over the top of the saddle point, and then it must flow down until the second minimum is detected as having been made wet. (b) For simultaneous flooding, both minima are initialized as wet, but with two different, distinguishable floods. The lower minimum's flood is raised individually until at the same level with the higher, and then the process progresses such that both floods are always at the same elevation. Both are raised until the floods collide at the saddle point.

5.2.1 Original Möller et al. implementation

Möller et al. search for the saddle point between two minima labeled the entry and exit points. An auxiliary matrix is created which associates a property of “wetness” with each cell in the original energy matrix. This auxiliary matrix—which we refer to as the flood map—tracks the virtual flood used in the immersion method. They begin by setting only the entry point to be wet in an otherwise completely dry flood map. The flood is expanded by repeatedly processing every cell in the flood map. A particular cell is made wet if it simultaneously has a corresponding energy less than a given threshold and has a neighbor which is already wet. They choose the initial threshold to be 1 MeV above the entry point's energy. The flood is expanded until either an iteration where no changes are made in the flood map occurs—signaling that the flood has expanded as far as is permitted by the energy surface and the threshold limit—or the exit point becomes wet. In the former case, the exit point is examined to determine whether it is wet. If it is not then the threshold was lower than the saddle energy between the entry and exit points, and the threshold can define a lower bound on the saddle energy. The threshold is then incremented by another 1 MeV, and the process is repeated. If instead the exit point is wet, then the threshold is an upper bound on the saddle energy. Together the lower and upper bounds define an interval within which the saddle energy must exist. The saddle energy can be defined to a desired precision by continually decreasing the increment size from the initial 1 MeV by a factor of 10 when necessary.

5.2.2 Simultaneous flooding

One of the first optimizations we made was to decrease the number of times that the entire energy matrix had to be traversed by flooding both minima simultaneously. Möller et al. mention the possibility of doing this by making use of two flood maps—one for each flood—but dismiss the implementation as being needlessly complex. Rather than making use of two flood maps, though, we were able to achieve simultaneous flooding with only a single flood map by simply identifying each flood with a different unique integer and non-flooded cells with the zero value. In theory, this method can be used to simultaneously flood as many minima as unique values can be stored in the map (255 for the one byte cells currently used). Simultaneous flooding reduces the number of iterations required by avoiding unnecessary flooding. With over 5 million cells in the matrix, shaving off iterations can rapidly save

computational time. Möller et al.’s method iterates until the flood map is left unchanged after a pass over the entire matrix. They are then able to check when the the bound was too high or too low based on the exit point’s “wetness”. Our implementation, though, is capable of detecting collisions between the simultaneously expanding floods, possibly long before all eligible points have been flooded. If more than one collision occurs, then we already know the threshold is too high and can restart the search with a tighter bound. Additionally, we gain the ability to automatically know when the process has reached sufficient convergence by noting when only a single collision is generated during a single pass.

5.2.3 Saddle convergence

Another small optimization came from a different choice in the convergence technique used to identify the saddle energy. As described in §5.2.1, the Möller et al. implementation converged on the energy by using a linearly increasing threshold with step sizes that decreased by an order of magnitude when the algorithm detected that the energy had been overestimated. In our implementation, we instead use a binary-search style convergence algorithm. The lower bound is set by the minima energies, but there is no method a priori to know what the upper bound should be. We determine this upper bound by stepping the threshold by 1 MeV steps much like the Möller algorithm until the first overestimate is detected. After both the lower and upper bounds have been determined, though, the process proceeds by a binary search. The threshold is set to the midpoint between the upper and lower bounds. If the threshold is still an overestimate, the upper bound is decreased to the threshold, and the process is repeated. If instead the flood reaches the threshold and an entire pass results in no changes to the flood map, then the lower bound is increased to the threshold, and the process is again repeated. This process of bisection and flooding is continued until a single collision point is detected which indicates that the threshold is a good estimate for the true saddle energy.

The gains measured, though, were modest at best. Raw timings of the time to complete a single saddle determination were lower for the binary search case than when reprogrammed to make use of a linear search like Möller et al.’s original implementation. In addition, we identified using `gprof` (see §A.1 for more information) that a particularly expensive operation—creation of a backup flood map for use when the upper bound is too high and the process needs to restart from a state before any collisions have occurred—requires a smaller fraction of the total run time in the binary search case. On a particular test case, the relative weights decreased from 0.20% for the linear search to 0.03% for the binary search. The significance to overall run time is very low, though, so any future performance gains are unlikely from exploring this particular implementation detail further.

5.2.4 Bounded-box flooding

The reason the flooding operation can be so computationally expensive is because the algorithm potentially examines the more than five million cells over many complete iterations. The motivation for the bounded box addition was to decrease the number of elements which must be examined on each element by excluding regions which are known to be completely dry and cannot become wet in the next iteration. Figure 5 demonstrates the method in a simplified two-dimensional case, and the method is directly expanded to the five dimensions contained in our problem. When the two flood points are initialized at each respective minimum point, the maximum and minimum extents are remembered in each dimension. The bounding box is then defined by subtracting 1 from the minimum values and adding 1 to the maximum values. Since the flood can only grow by a single cell into a neighbor during a single iteration, this is sufficient to define the next maximum possible extents of the flood. The flooding algorithm must then only loop over cells contained within the bounds of the hypercube during a given iteration. If during the loop the flood is extended to reach a face of the bounding hypercube, the extent in that direction is

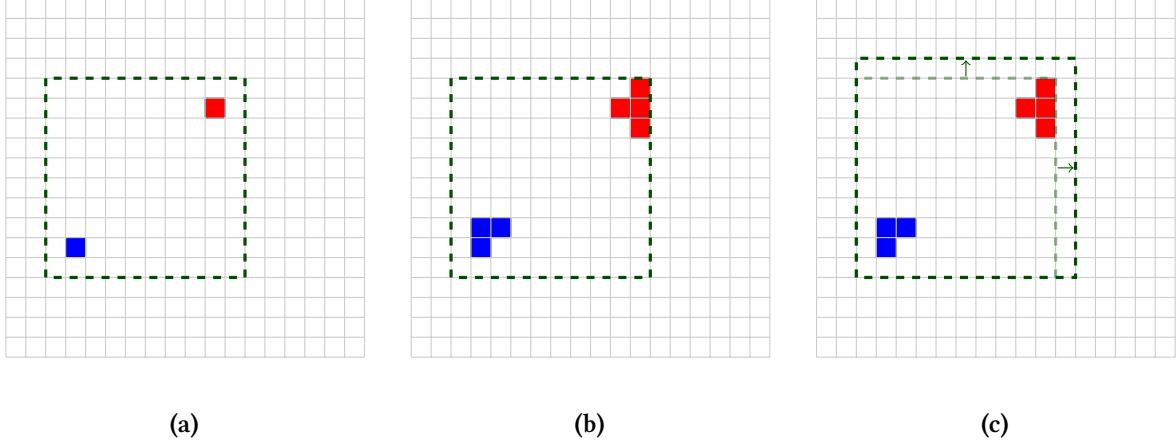


Figure 5: The three major stages in the bounding-box optimization to the immersion method for a simplified 2D example. (a) The flood map is initialized with an appropriate bounding box. The matrix traversal starts at the top-left corner of the bounding box and proceeds in column-major order within established bounds. (b) During traversal, cells with a wet neighbor are also made wet within the constraints of the energy surface values (not depicted). Some expansions may cause a flood to expand into the boundary such as at the top right. (c) The bounding box is expanded in any dimension's limits to re-establish a single-cell border around the outermost flooded cells. The process restarts as in (a) from the new box's top-left cell.

pushed outward (toward zero for a minimum and toward infinity for a maximum extent) to maintain a hypercube sufficient to contain the *next iteration's* maximum possible size.

The performance gains were again modest, on the order of a few percent or less which is not to be entirely unexpected in our limited tests upon only a couple of energy surfaces. We expect that the most significant gains will occur at for early steps where the flooded regions are small, especially if the minima are located near each other in the energy surface. At further steps, the flood can spread to span across the extent of some or all dimensions, thereby requiring a traversal of nearly every cell in the matrix and eliminating any performance gains. In particular, the common case of locating the saddle between the ground and scissioned states almost completely spans the elongation parameter, and therefore a large fraction of the entire hypercube is included within the bounds already at initialization. Furthermore, confident conclusions are only reasonable after examining the performance change for a large number of energy surfaces, which is goal of future studies.

6 Performance analysis

The final analysis results shown in the following subsections makes use of timers wrapped around simple stub routines which are capable of running an isolated algorithm. This required special care to ensure that extraneous and uninteresting processes such as loading the energy matrix or auxiliary data is excluded from the timings by performing them before the timing begins. In all presented results, the timing gives the average time required to complete the operation over five invocations, and the standard deviation in times is used to the estimated statistical error. We made use of two types of timings: wall (or real) time and CPU time. The wall time is the physical, world time that elapses between two events, as if you were to watch a clock on the wall and note the time interval. This is in contrast to CPU time which is the sum total of time that the processor(s) were busy in computations. On a single processor machine, the wall and CPU times should correspond almost exactly. For SMP systems, though, CPU time will be greater than the wall time since multiple cores were computing in parallel during the same physical time interval. Under perfect circumstances we'd expect the wall time to be inversely related to the number of processors used to compute the problem while maintaining constant CPU time. Because of various forms of overhead, any

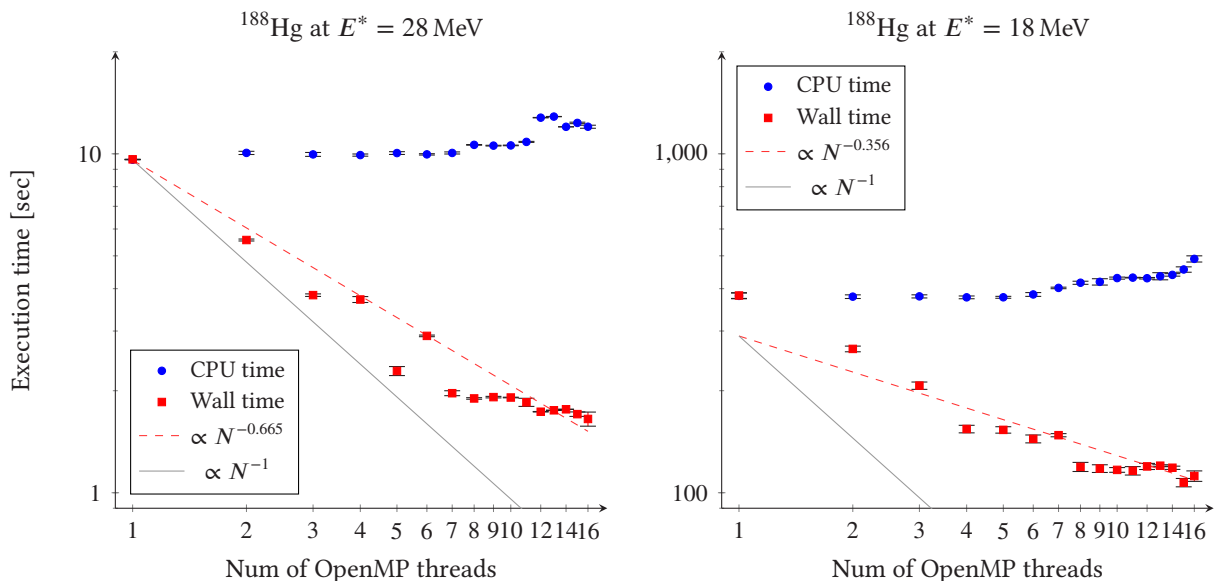


Figure 6: Shown are the run-time performance analyses for 16 random walk paths over the ^{188}Hg energy hypersurface for excitation energies of 18 and 28 MeV. A perfectly parallelized algorithm would be expected to have constant CPU time while decreasing the wall time proportional to the number of OpenMP threads. Taking the lesser performance of the 18 MeV case together with the systematic zig-zag deviations in wall time, long running single paths which hold completion of the process is a significant factor in total completion time.

real-world result will fall short of these expectations. We collected timings for both the random-walk and saddle-search algorithms in 16 timing trials which sequentially incremented the number of processors in use from 1 to 16.

6.1 Random walk performance

The random-walk algorithm was tested using the ^{188}Hg energy hypersurface with excitation energies E^* of 18 MeV and 28 MeV. Each trial was asked to calculate the yield distribution for 16 paths, chosen to correspond to the number of processors available so that at maximum utilization, each processor is assigned a single path. The results are presented in Fig. 6. In both cases, a log-log fit was generated for the wall time and compared against a perfect N^{-1} response.

We see that the wall-time performance as a function of number of threads depends on the problem context strongly; the $E^* = 28 \text{ MeV}$ case, which already required over an order of magnitude less time than the 18 MeV case when executed serially, conforms more closely to the ideal and gains a larger speedup factor. Despite the low convergence rate of $N^{-0.356}$, though, the results are promising since ^{188}Hg represents one of the most difficult energy hypersurfaces for the Metropolis method. Furthermore, the effects of exceptionally long paths are visible in the results. The zig-zag pattern in the timings are not a statistical artifact—the error bars show the standard deviation in timings but are almost invisible on the plots. It is caused by one or two long running paths that continue to run on a single processor after all other paths have finished. Given these considerations, we expect that more well-behaved energy hypersurfaces will achieve more consistent speedups.

Additionally, the CPU time is qualitatively well constrained to a constant; we observe small increases in total CPU time as a function of the number of OpenMP threads, but the rise is not more than would be reasonably expected from overhead inherent to coordinating and synchronizing additional threads.

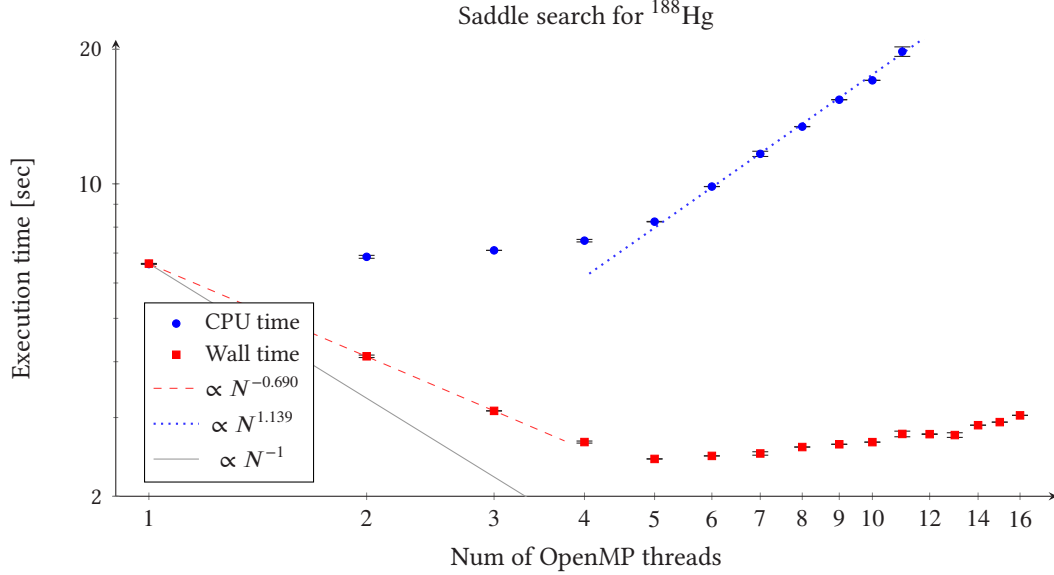


Figure 7: The timing results to find the saddle point between a single pair of minima. The algorithm does not scale beyond approximately 4 processors—simultaneously the wall time hits a plateau while the CPU time increases proportional to the number of processors. A critical bottleneck, likely to be waiting on main memory, currently limits the calculation rate for the saddle-search algorithm.

6.2 Flooding performance

Determining the saddle point for a single pair of minima in ^{188}Hg was also performed to examine the performance characteristics of the flooding algorithms as a function of the number of parallel threads. The results are shown in Fig. 7, and they differ from those of the random-walk tests starkly. First, the convergence proportional to $N^{-0.690}$ only extends to about 4 or 5 processors (the first 4 points were used in the fit) before the performance gains plateau and begin to decrease. In this case, we cannot continue to reduce the real-world computational time by simply throwing a larger number of processing cores at the problem.

In fact as the CPU time shows, we actually waste resources by asking that all 16 processors on our test machine work on the task. Up through roughly the same limit where the wall time decreases, we also see a relatively constant total CPU time. Beyond that limit where the wall-time hits a plateau, though, we see that the CPU time increases dramatically—at a rate approximately proportional to the number of processors in fact! The flooding algorithm makes much more demands on memory, and we suspect that cache misses become significant and each thread begins to wait for data to be retrieved from main memory. Therefore, the most efficient use of computing resources for the saddle-finding procedure is to utilize approximately 4 processors.

7 Future work

As occurs in any project, there are places where there is additional work to be done: (1) Previously mentioned in §6, more reliable and accurate performance analysis can be performed by making use of the TAU instrumentation suite to provide access to the hardware counters built into many computer architectures. The sampling and simulation techniques used by `gprof` and `cachegrind`, respectively, proved to be useful tools to do a broad optimization of the algorithms, but hardware counters that provide production, run-time data may give insights into remaining performance bottlenecks. In particular, more

advanced methods may provide further insight into the complex interactions parallelized algorithms can have with the underlying hardware (as suggested by the saddle finding performance in Fig. 7). (2) Extensive performance tests across a wide range of nuclei could provide further insights into bottlenecks or limitations in these updated implementations. (3) Modifications to the saddle-finding algorithm may be possible that would allow the simultaneous identification of saddle points between multiple pairs of minima. The current implementation finds saddle points between pairs serially. (4) Changes to the structure of the energy matrices may lead to further enhancements. For example, eliminating the need for the unphysical points simply for computational purposes could reduce resource requirements. Another option would be to use a different choice of parameterizations which produce computationally favorable energy surfaces; in particular, the flooding algorithms are sensitive to the extents of the flood, so choosing a parameterization which preferentially constrains to a subspace could amplify the effectiveness of the bounding boxes. (5) Alternative storage of the matrices in memory—possibly multiplexing or an as-yet unconsidered format—could provide more efficient cache usage if point 1 can prove where this method would be effective.

8 Acknowledgements

We'd first like to thank in general Los Alamos National Laboratory's Advanced Scientific Computing Program; it is responsible for generously supporting the Computational Physics Student Summer Workshop in which we have participated. A special thank you must go to our mentor Peter Möller for his guidance and in particular his patience as one of us (Justin) was introduced to an unfamiliar field and the other (Kemper) who has much less coding experience. This work would also not have been possible without the use of his code and data as a starting point. Next, a thank you to Scott Runnels for all the organizational work done to make this workshop a success, from the application process and in-processing paperwork to all the lectures provided for our benefit. Lastly, we want to thank all of our peers and other workshop mentors who have provided great sounding boards against which to help refine ideas and strategies.

A Performance profilers

As first mentioned in §5, we made use of the `gprof` [4] and `cachegrind` [14] tools to help guide our implementation strategies. In the following two sections we will introduce each tool, discuss the advantages and pitfalls of each, and provide a short discussion of the conclusions we drew from an example output.

A.1 The GNU profiler

The GNU profiler `gprof` is a sampling profiler. This means it interrupts the execution of the program of interest at periodic intervals in order to record the location of execution within the program. This has several important consequences for how a program is profiled and what conclusions can be drawn from the resultant information.

First, because the profiler samples a running program, the collected profiling data will only reflect the performance for a particular code path. In a complex piece of software such as a web browser, it can be very challenging to ensure that all code paths of interest are executed since the browser may execute very different sections of its code based on its interaction with the user. In our case, though, both the random walk and flooding procedures are simple, single-purpose utilities which do not have alternate modes of usage. This means that we can be confident that our programs are being sufficiently covered to provide representative samples. Further work is required to reach strong conclusions, though—it is known that not all nuclei are equally computationally expensive and make unique demands on resource usage, so a comprehensive review of many nuclei is still needed.

Second, the sampling occurs at periodic intervals which may systematically never overlap with a particular line of code. If we were concerned with corner case performance or micro-optimizations, we would have more reason to worry, but because we were interested in the performance of only the internal hot loops which are executed very many times, we are not concerned with the fact that several iterations may not be counted. The programs still spend an overwhelming fraction of their time working within the hot loops and therefore a fair representative sample should still result.

Additionally, matching executing instructions with specific lines of original code can be inaccurate or misleading. Without any optimizations, the machine instructions can be traced back directly to the code that generated them, but the use of optimizing compilers can make the correspondence less accurate or completely misleading. This is because the optimizers make use of very complex passes which can rewrite the machine instructions in order to make best use of the available resources, but this may also mangle the code lines so that the code flow is no longer sequential. A decision must then be made. One choice is to sacrifice program performance (and deviate from the production executable that would actually be executed) to retain accurate code-to-instruction correspondence. The other choice uses the optimized program at the cost of generating profile information which cannot be entirely trusted.

Finally, the sampling method only provides performance data by counting the number of times a particular line of code is executing when the sample is taken. This provides information about where the code is spending its time, but not why. In order to properly optimize a particular line of code, it is necessary to know the reason for the long execution time. For example, the method for reducing the number of cache misses in a line of code will vary greatly from the strategy used to reduce the number of cycles required on the CPU. The output from `gprof` must therefore either be interpreted using heuristics or another tool which can provide extra information, and this is the reason for also making use of `cachegrind`.

Listing 1: A sample of the output generated by the `gprof` analysis tool during execution of the saddle finding utility. The particularly useful data is given in the first and last columns. The first column shows the cumulative fraction of the time spent in executing a particular line of code. The least column then gives the file and line number of the corresponding code.

```

1 Flat profile:
2
3 Each sample counts as 0.01 seconds.
4  %   cumulative   self           self       total
5  time    seconds seconds    calls   Ts/call   Ts/call   name
6  75.76    22.54    22.54           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:91 @ 401de4)
7   3.90    23.70     1.16           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:68 @ 401b39)
8   2.64    24.48     0.79           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:101 @ 401fe4)
9   2.61    25.26     0.78           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:71 @ 401cda)
10  2.07    25.87     0.62           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:65 @ 401b24)
11  1.78    26.41     0.53           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401d7b)
12  1.65    26.90     0.49           0             0             0 __moller_flooding_MOD_saddle_point (moller_flooding.f95:363 @ 408172)
13  1.56    27.36     0.47           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:91 @ 401da9)
14  1.13    27.70     0.34           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:91 @ 401dbc)
15  1.06    28.01     0.32           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:91 @ 401dd1)
16  0.99    28.31     0.30           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401dce)
17  0.84    28.56     0.25           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401db5)
18  0.44    28.69     0.13           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:91 @ 401dcb)
19  0.32    28.78     0.10           0             0             0 __moller_flooding_MOD_saddle_point (moller_flooding.f95:363 @ 40815d)
20  0.29    28.87     0.09           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401de0)
21  0.27    28.95     0.08           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:63 @ 401b6a)
22  0.20    29.01     0.06           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:88 @ 401d24)
23  0.18    29.06     0.06           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401d01)
24  0.15    29.11     0.05           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:88 @ 401b1d)
25  0.13    29.15     0.04           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401fe0)
26  0.13    29.19     0.04           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401fc3)
27  0.12    29.22     0.04           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:101 @ 401fd9)
28  0.12    29.26     0.04           0             0             0 __moller_flooding_MOD_saddle_point (moller_flooding.f95:287 @ 408121)
29  0.10    29.29     0.03           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:88 @ 401d0f)
30  0.10    29.32     0.03           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401d17)
31  0.10    29.35     0.03           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:91 @ 401d9a)
32  0.10    29.38     0.03           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:101 @ 401fb6)
33  0.10    29.41     0.03           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401fbc)
34  0.08    29.43     0.03           0             0             0 __moller_flooding_MOD_saddle_point (moller_flooding.f95:363 @ 408138)
35  0.07    29.45     0.02           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401c70)
36  0.07    29.47     0.02           0             0             0 __moller_flooding_MOD_saddle_point (moller_flooding.f95:342 @ 40802e)
37  0.07    29.49     0.02           0             0             0 __moller_flooding_MOD_saddle_point (moller_flooding.f95:287 @ 408141)
38  0.07    29.51     0.02           0             0             0 __moller_flooding_MOD_saddle_point (moller_flooding.f95:363 @ 408150)
39  0.07    29.53     0.02           0             0             0 __moller_flooding_MOD_saddle_point (moller_flooding.f95:287 @ 408162)
40  0.05    29.55     0.02           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401da2)
41  0.03    29.56     0.01           0             0             0 __moller_flooding_MOD_flood_min_pair (moller_flooding.f95:230 @ 404000)
42  0.03    29.57     0.01           0             0             0 __moller_flooding_MOD_flood_min_pair (moller_flooding.f95:267 @ 4043ff)
43  0.03    29.58     0.01           0             0             0 __moller_flooding_MOD_flood_min_pair (moller_flooding.f95:255 @ 404780)
44  0.03    29.59     0.01           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 4019ba)
45  0.03    29.60     0.01           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401abe)
46  0.03    29.61     0.01           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:91 @ 401d5c)
47  0.03    29.62     0.01           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401d6e)
48  0.03    29.63     0.01           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401daf)
49  0.03    29.64     0.01           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401dc8)
50  0.03    29.65     0.01           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:94 @ 401f7f)
51  0.03    29.66     0.01           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:95 @ 401f87)
52  0.03    29.67     0.01           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401f9b)
53  0.03    29.68     0.01           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:101 @ 401fd3)
54  0.03    29.69     0.01           0             0             0 __moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401fd6)

```

55	0.03	29.70	0.01				__moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:102 @ 402281)
56	0.03	29.71	0.01				__moller_flooding_MOD_saddle_point (moller_flooding.f95:324 @ 407b63)
57	0.03	29.72	0.01				__moller_flooding_MOD_saddle_point (moller_flooding.f95:287 @ 4080fa)
58	0.03	29.73	0.01				__moller_flooding_MOD_saddle_point (moller_flooding.f95:339 @ 408441)
59	0.03	29.74	0.01				rang (mcnp_random.f95:180 @ 407b2d)
60	0.02	29.74	0.01				__moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:62 @ 401ab2)
61	0.02	29.75	0.01				__moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:88 @ 401ab9)
62	0.02	29.75	0.01				__moller_flooding_MOD_flood_step._omp_fn.0 (moller_flooding.f95:91 @ 401db2)
63	0.02	29.76	0.01				__moller_flooding_MOD_saddle_point (moller_flooding.f95:287 @ 408159)
64	0.00	29.76	0.00	112	0.00	0.00	__moller_flooding_MOD_flood_step (moller_flooding.f95:38 @ 402740)
65	0.00	29.76	0.00	89	0.00	0.00	frame_dummy
66	0.00	29.76	0.00	2	0.00	0.00	__moller_flooding_MOD_saddle_point (moller_flooding.f95:287 @ 4076c0)
67	0.00	29.76	0.00	1	0.00	0.00	__mcnp_random_MOD_rn_init_problem (mcnp_random.f95:232 @ 406ea0)
68	0.00	29.76	0.00	1	0.00	0.00	__moller_datafiles_MOD_load_tensor (moller_datafiles.f95:186 @ 406490)
69	0.00	29.76	0.00	1	0.00	0.00	__moller_datafiles_MOD_load_tensor_cache (moller_datafiles.f95:123 @ 405ce0)
70	0.00	29.76	0.00	1	0.00	0.00	__moller_flooding_MOD_flood_min_pair (moller_flooding.f95:135 @ 402980)
71							
72	%	the percentage of the total running time of the					
73	time	program used by this function.					
74							
75	cumulative	a running sum of the number of seconds accounted					
76	seconds	for by this function and those listed above it.					
77							
78	self	the number of seconds accounted for by this					
79	seconds	function alone. This is the major sort for this					
80		listing.					
81							
82	calls	the number of times this function was invoked, if					
83		this function is profiled, else blank.					
84							
85	self	the average number of milliseconds spent in this					
86	ms/call	function per call, if this function is profiled,					
87		else blank.					
88							
89	total	the average number of milliseconds spent in this					
90	ms/call	function and its descendents per call, if this					
91		function is profiled, else blank.					
92							
93	name	the name of the function. This is the minor sort					
94		for this listing. The index shows the location of					
95		the function in the gprof listing. If the index is					
96		in parenthesis it shows where it would appear in					
97		the gprof listing if it were to be printed.					

A.1.1 Program preparation & tool execution

Using `gprof` requires additional compile-time tooling to include extra machine instructions necessary for collecting the samples, and therefore `gprof` is only suitable for analyzing programs that can be compiled from source—embedded debug symbols in the program are insufficient. The advantage of the compile-time tooling is that no extra effort is required to generate profiling information; only a single compiler option must be added, and the rest of the work is performed automatically by the compiler. For the GNU Compiler Collection (GCC), the flag `-pg` must be added to compilation. The resultant executable is then profiling-enabled.

Generation of the profiling information is automatic upon execution of the program with the data being written to the file `gmon.out` within current working directory. `gmon.out` is not human readable, though, and must be post-processed to generate the output shown in Listing 1 with a command sequence similar to

```
$ gprof --line flood_test gmon.out > flood_test.gprof-annotate
```

The sequence takes as arguments the path to the binary and profile data (here, both in the current working directory) and emits the human readable form to the standard output (here being redirected to a file). The extra option tells `gprof` that we want line-by-line annotations rather than just function totals.

A.1.2 Profile interpretation

Listing 1 shows the flat profile generated by `gprof`. We have enabled line-by-line counting, so most of the output is for specific lines of code, and function totals only show up near the end of the listing. We were most interested in the lines that account for the greatest proportion of the run time, and `gprof` makes this clear for us, sorting the entries by individual percentage cost by default (the left-most column). In Listing 1, the first line shows that the call at line 91 of `moller_flooding.f95` accounts for over 75% of the run time. Looking up the line in source, this corresponds to a search over neighbors cells in the flooding routine for a flooded neighbor.

With a line identified, we can finally expend the time and effort to explore optimization opportunities. The line is a critical part of the flooding algorithm, so there is not a way a trivial way to reduce the number of times the line is executed. Further, the line is a call to a Fortran intrinsic, so we can safely assume that a hand-written method would be less efficient than the code produced by the compiler and support library maintainers. For this specific case, we identified the largest cost in the program and concluded that we could not optimize this specific line within the time constraints and goals of our research, and therefore moved on to other optimization efforts.

The output from `gprof` was used in a similar manner to help with smaller optimizations already discussed in §5. For example, the linear vs. binary saddle convergence described in §5.2.3 was in part analyzed by noting how the costs changed between `gprof` profile outputs; we identified that the number of calls to a particularly expensive memory call were reduced in the binary search implementation over the linear search.

A.2 Valgrind

In contrast to `gprof`, Valgrind is a simulation profiler which runs an interpreter to track every line as it executes. As a consequence Valgrind will never miss an execution of a line of code, but execution time is significantly increased. The major benefit that makes the cost worth paying, though, is that the simulator is capable of simulating how the code interacts with any software or hardware feature that Valgrind and its plugins are capable of simulating. For `cachegrind`, this means that we gain the ability to simulate

Listing 2: At runtime, cachegrind outputs a summary of the collection information to the terminal in addition to generating an auxiliary data file which can be analyzed later (see Listing 3). The most interesting data occurs in lines 41 and 42 which give the cache miss rate for data-related reads and writes. We see that the rates are less than 1%, initially suggesting cache misses are not a significant effect, but see §6.1 and §6.2 for a more complete discussion.

```

1 ==18642== Cachegrind, a cache and branch-prediction profiler
2 ==18642== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote et al.
3 ==18642== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
4 ==18642== Command: ./randwalk
5 ==18642==
6 --18642-- warning: Unknown Intel cache config value (0x76), ignoring
7 --18642-- warning: Unknown Intel cache config value (0xff), ignoring
8 --18642-- warning: L2 cache not installed, ignore LL results.
9 Please specify the nuclear parameters
10 Z:
11 A:
12 Enter a new value, or leave blank to accept the default.
13
14 Number of iterations to perform [ 50000 ]:
15 Set the excitation energy [ 6.540 ]:
16 Set the shell correction damping factor [ 60.000 ]:
17
18 Excitation energy (wrt ground) : 6.540 MeV.
19 Absolute ground state energy : -2.110 MeV.
20 Absolute excitation energy : 4.430 MeV.
21 EDAMP : 60.000 MeV.
22
23 All manual input has been entered. Please be patient as the random
24 walks are performed. Note that if no cache exists, reading in the
25 data files may take some time.
26
27
28 Total rnd count : 8062781
29 Mass normalization check: 199.999985
30 Charge normalization check: 200.000000
31 ==18642==
32 ==18642== I refs: 1,791,441,715
33 ==18642== I1 misses: 2,301
34 ==18642== LLi misses: 2,298
35 ==18642== I1 miss rate: 0.00%
36 ==18642== LLi miss rate: 0.00%
37 ==18642==
38 ==18642== D refs: 853,671,979 (555,202,464 rd + 298,469,515 wr)
39 ==18642== D1 misses: 4,036,351 ( 1,055,017 rd + 2,981,334 wr)
40 ==18642== LLd misses: 2,910,170 ( 3,988 rd + 2,906,182 wr)
41 ==18642== D1 miss rate: 0.4% ( 0.1% + 0.9% )
42 ==18642== LLd miss rate: 0.3% ( 0.0% + 0.9% )
43 ==18642==
44 ==18642== LL refs: 4,038,652 ( 1,057,318 rd + 2,981,334 wr)
45 ==18642== LL misses: 2,912,468 ( 6,286 rd + 2,906,182 wr)
46 ==18642== LL miss rate: 0.1% ( 0.0% + 0.9% )

```

Listing 3: The annotated cachegrind output. (For the run-time output summary, see Listing 2.) Performance is given in terms of instruction or data cache reads and writes per function. Note lines 26 and 29 are two of the three most expensive operations in terms of cache misses for both read and writes, but—because they are 1-time initialization procedures—do not factor into the most critical hot loops.

```

1 -----
2 I1 cache:      67108864 B, 64 B, 2-way associative
3 D1 cache:      67108864 B, 64 B, 2-way associative
4 LL cache:      268435456 B, 64 B, 8-way associative
5 Command:       ./randwalk
6 Data file:     randwalk_test.cachegrind
7 Events recorded: Ir I1mr I1Lmr Dr D1mr D1Lmr Dw D1mw DLmw
8 Events shown:  Ir I1mr I1Lmr Dr D1mr D1Lmr Dw D1mw DLmw
9 Event sort order: Ir I1mr I1Lmr Dr D1mr D1Lmr Dw D1mw DLmw
10 Thresholds:   0.1 100 100 100 100 100 100 100 100
11 Include dirs:  ~/code-hg/
12 User annotated:
13 Auto-annotation: off
14
15 -----
16           Ir  I1mr  I1Lmr           Dr      D1mr  D1Lmr           Dw      D1mw      DLmw
17 -----
18 1,791,441,715 2,301 2,298 555,202,464 1,055,017 3,988 298,469,515 2,981,334 2,906,182  PROGRAM TOTALS
19
20 -----
21           Ir  I1mr  I1Lmr           Dr      D1mr  D1Lmr           Dw      D1mw      DLmw  file:function
22 -----
23 501,645,738      8      8 137,148,861      223      61 56,922,291           29           0  ???:__ieee754_expf
24 240,338,562      2      2 88,545,786           0           0 63,246,990           0           0  ???:fesetenv
25 188,439,137      0      0 47,137,234 79,641           0 19,408,216           141          0  moller_randwalk.f95:__moller_randwalk_MOD_randwalk_path
26 159,558,964      0      0 60,770,911 580,005           0 22,767,659 635,989 635,976  moller_randwalk.f95:__moller_randwalk_MOD_randwalk_preload
27 145,468,077      2      2 50,597,592           4           1 25,298,796           0           0  ???:expf
28 126,493,980      3      3 63,246,990           0           0 44,272,893           13          0  ???:feholdexcept
29 113,844,582      1      1 31,623,495           0           0 25,298,796           0           0  ???:fesetround
30 74,913,240      0      0 20,351,262 390,146           0 20,351,260 1,346,412 1,271,952  moller_randwalk.f95:__moller_randwalk_MOD_reflect_mass_asymm
31 65,682,948      0      0 7,892,244           0           0 1,315,374           0           0  mcnp_random.f95:__moller_randwalk_MOD_randwalk_path
32 39,539,501     11     11 9,410,084      121     117 9,045,079 996,819 996,818  ???:memcpy
33 31,623,500      0      0 6,324,700           0           0           0           0           0  ???:finitef
34 22,960,500      1      1 4,592,151           0           0           0           0           0  ../../gcc-4.7.0/libgomp/config/linux/wait.h:gomp_barrier_wait_end
35 21,364,660      8      8 6,581,480           0           0 2,936,456           1           1  ../../gcc-4.7.0/libgfortran/io/transfer.c:_gfortran_transfer_array
36 14,175,000      5      5 4,252,500           0           0 2,480,625           0           0  ../../gcc-4.7.0/libgfortran/io/transfer.c:read_block_direct
37 14,047,182      6      6 3,937,256           0           0 2,165,311           0           0  ../../gcc-4.7.0/libgfortran/io/unix.c:buf_read
38 8,505,000       3      3 2,480,625           0           0 1,771,875           0           0  ../../gcc-4.7.0/libgfortran/io/transfer.c:unformatted_read
39 6,710,481     134    131 6,709,517           74          4          423          2          2  ???:???
40 4,592,173       2      2           17           2           0           0           0  ../../gcc-4.7.0/libgomp/config/linux/x86/futex.h:gomp_barrier_wait_e
41 2,361,330       1      1          472,296           1           0           0           0  ../../gcc-4.7.0/libgomp/config/linux/wait.h:gomp_team_barrier_wait_e
42 2,126,250       2      2 1,417,500           0           0           0           0           0  ../../gcc-4.7.0/libgfortran/io/transfer.c:iolength_transfer

```

cache memory operations, and we can estimate which code operations are causing the greatest number of cache misses.

As another run-time tool, a couple of concerns outlined for gprof also apply to the profiles generated by cachegrind. First, the same code path concerns apply—we cannot draw any conclusions about code that was not executed during a particular run. Again, because our programs are single-purpose utilities, this caveat is not a critical concern for us. Similarly, the optimization passes performed by the compiler can also cause the correlation between machine instructions and source code to become unclear, so the same trade-off between production-equivalent executable and accurate results must be considered.

We gain the ability for Valgrind and cachegrind to provide information about the cache performance of our algorithms with a caveat—the simulator does not support OpenMP parallelization. The consequence of this deficiency is demonstrated in Figure 7; the performance likely fails to increase with the number of cores because memory and cache misses become a bottleneck in the algorithm. Despite the serial constraint, though, any poorly performing code identified in the serial case is only going to degrade as the memory bus is stressed during simultaneous access from multiple cores.

A.2.1 Program preparation & tool execution

No special preparation is required during compile time since the program is executed in a simulator which has access to all machine instructions, CPU, and memory internals at all times. Profiling is performed by executing the valgrind tool, instructing it to use the cachegrind tool, and specifying the program to profile. An example invocation may be,

```
$ valgrind --tool=cachegrind --cachegrind-out-file=randwalk.cachegrind \  
> ./randwalk > randwalk.cachegrind-stdout
```

The file redirection produces the output shown in Listing 2, and use of the cg_annotate utility post-processes the file given by --cachegrind-out-file to produce a human-readable summary:

```
$ cg_annotate -I~/code randwalk.cachegrind > randwalk.cachegrind-annotate
```

The resultant annotated, human-readable output is shown in Listing 3.

A.2.2 Profile interpretation

The easiest results to interpret are shown in lines 41 and 42 of Listing 2. These two lines show that the simulated cache miss rate for reads and writes is less than 1%; from this we infer that the cache miss rate is not a significant contributor to the run time in the serial case. As has already been discussed in §6.2, though, this interpretation is likely not correct in the parallel case, particularly on SMP systems with a large number of cores.

More detailed information is contained within Listing 3. The sort order is by the number of instruction reads (analogous to the call count in gprof), and we see that the exponentiation function is called the most; note that the statistics are only for whole function calls unlike the output shown in Listing 1, and therefore we would expect the exponential function to be called often since it is used within the random walk inner loop. We are most interested in examining the columns marked with *mr* or *mw* in the names; these indicate the cache miss counts for the cache levels for both read and write operations, respectively. Lines 26, 30, and 32 immediately stand out. Both lines 26 and 30 do not pose a significant problem since these are both routines which must manipulate the entire energy matrix during the program initialization; a large number of cache misses are expected as caches are populated. The memory copy in line 32 is also expected to cause cache misses, but without further tooling and more specific profiling, we cannot conclude what causes the copies. Memory copies which occur due to loading data from disk is of no concern, but managing copies that occur during the random walk loop is an important task.

References

- [1] Forrest Brown. *The MCNP5 Random Number Generator*. LA-UR-07-7961. Los Alamos National Laboratory. URL: https://laws.lanl.gov/vhosts/mcnp.lanl.gov/pdf_files/la-ur-07-7961_mcnp_rn.pdf (visited on 08/13/2013).
- [2] Ashok Das and Thomas Ferbel. *Introduction to Nuclear and Particle Physics*. 2nd ed. World Scientific, 2009. ISBN: 978-981-238-744-8.
- [3] Department of Computer and Information Science, University of Oregon. *TAU Performance System*. University of Oregon, Los Alamos National Laboratory, and Research Center Jülich, July 25, 2013. URL: <http://www.cs.uoregon.edu/research/tau/home.php>.
- [4] Jay Fenlason. *The GNU Profiler*. The GNU Project, July 29, 2013. URL: <http://www.gnu.org/software/binutils/>.
- [5] Brian Hayes. “Dividing the Continent”. In: *American Scientist* 88.6 (2000), p. 481. DOI: 10.1511/2000.6.481. URL: <http://www.americanscientist.org/issues/pub/2000/6/dividing-the-continent>.
- [6] Takatoshi Ichikawa et al. “Contrasting fission potential-energy structure of actinides and mercury isotopes”. In: *Phys. Rev. C* 86 (2 Aug. 2012), p. 024610. DOI: 10.1103/PhysRevC.86.024610. URL: <http://link.aps.org/doi/10.1103/PhysRevC.86.024610>.
- [7] Donald E Knuth. “Structured Programming with go to Statements”. In: *ACM Computing Surveys* 6.4 (1974), pp. 261–301. URL: <http://portal.acm.org/citation.cfm?doid=356635.356640>.
- [8] Peter Möller et al. “Heavy-element fission barriers”. In: *Physical Review C* 79.6 (June 2009), p. 064304. DOI: 10.1103/PhysRevC.79.064304. URL: <http://link.aps.org/doi/10.1103/PhysRevC.79.064304>.
- [9] Peter Möller et al. “Nuclear Ground-State Masses and Deformations”. In: *Atomic Data and Nuclear Data Tables* 59.2 (Mar. 1995), pp. 185–381. DOI: 10.1006/adnd.1995.1002. URL: <http://dx.doi.org/10.1006/adnd.1995.1002>.
- [10] Bogdan Povh et al. *Particles and Nuclei. An Introduction to the Physical Concepts*. Trans. from the German by Martin Lavelle. Springer, 2008. ISBN: 978-3-540-79367-0.
- [11] Peter Ring and Peter Schuck. *The Nuclear Many-Body Problem*. Springer, 2004. ISBN: 3-540-21206-X.
- [12] Vilen Mitrofanovich Strutinsky. ““Shells” in deformed nuclei”. In: *Nuclear Physics A122* (1968), pp. 1–33.
- [13] *The OpenMP API specification for parallel programming*. The OpenMP Architecture Review Board, July 30, 2013. URL: <http://www.openmp.org/>.
- [14] Valgrind Developers. *Valgrind*. July 25, 2013. URL: <http://valgrind.org/>.
- [15] Luc Vincent and Pierre Soille. “Watersheds in digital spaces: an efficient algorithm based on immersion simulations”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13.6 (1991), pp. 583–598. DOI: 10.1109/34.87344. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=87344>.
- [16] Samuels S.M. Wong. *Introductory Nuclear Physics*. second. John Wiley & Sons, Inc., 1998. ISBN: 0-471-23973-9.