Modeling Ion Channel Kinetics with HPC

*Allison Gehrke, #Katherine Rennie, +Timothy Benke, ++Daniel A. Connors, and *Ilkyeun Ra

*Dept. of Computer Science and Engineering, #Otolaryngology, +Pharmacology, +Pediatrics, +Neurology, and ++Electrical Engineering

University of Colorado, Denver

Denver, USA

{Allison.gehrke, Katie.rennie, Tim.benke, dan.connors, Ilkyeun.ra}@ucdenver.edu

Abstract— **Performance improvements for computational sciences such as biology, physics, and chemistry are critically dependent on advances in multicore and manycore hardware. However, these emerging systems require substantial investment in software development time to migrate, optimize, and validate existing science models. The focus of our study is to examine the step-by-step process of adapting new and existing computational biology models to multicore and distributed memory architectures. We analyze different strategies that may be more efficient in multicore vs. manycore environments. Our target application, Kingen, was developed to simulate AMPAR ion channel activity and to optimize kinetic model rate constants to biological data. Kingen uses a genetic algorithm to stochastically search parameter space to find global optima. As each individual in the population describes a rate constant parameter set in the kinetic model and the model is evaluated for each individual, there is significant computational complexity and parallelism in even a simple model run.**

Keywords- multicore; cluster; workload characterization; application profiling; kinetic modeling; scientific application; high performance computation; ion channel kinetics

I. Introduction

Ion channels are trans-membrane proteins that open and close to regulate the flow of ions (currents) across membranes in all cells. These ionic currents are critical to intra- and inter-cellular signaling. Ion channels are especially suitable biological entities for computational studies through kinetic modeling. These kinetic features critically influence the temporal coding of cell-signaling information. AMPA receptors (AMPARs), ligand-gated ion channels activated by the timed release of the neurotransmitter glutamate, are responsible for nearly all fast excitatory neuronal signaling in the central nervous system. Understanding the detailed kinetic properties of these receptors underlies our understanding of neurodevelopment, sensory processing, learning/memory and pathological states such as epilepsy and intellectual disability.

Simulations of ion channel kinetics allow the investigation of how different inputs (e.g. for AMPARs, the alteration of the relative amount and temporal characteristics of glutamate stimulation) influence reaction rates and transition states of the kinetic scheme. A kinetic scheme (model) describes states, or conformations of the protein (open, closed, or desensitized), and the transition rates between them (timing of the switching from one state to another). Our goal is to rapidly optimize (fit) rate constants of the receptors to experimental kinetic data. Numerous studies have demonstrated that these receptors activate, deactivate and desensitize in a complex fashion. Previously published kinetic schemes are unable to simultaneously describe all key biological characteristics of AMPAR that underlie complex neuronal signaling.

Implementation and optimization of kinetic schemes were initially coded sequentially and were found to be prohibitively time consuming (> 30 days). The execution time was so prohibitive to prevent its usefulness to explore the implications and validate existing and alternative models, to test a broader range of parameter sets and to conduct sensitivity analyses. An efficient method for modeling and simulation is a tremendous advantage to researchers.

We developed a process to adapt scientific applications to parallel architectures. We integrated a strategy recommended by Intel engineers and conducted our analysis level-by-level starting with system level analysis and drilling down to progressively finer levels of analysis (Fig. 1). There is, of course, some interplay and overlap between the levels but the approach defines a meaningful framework within our overall process.

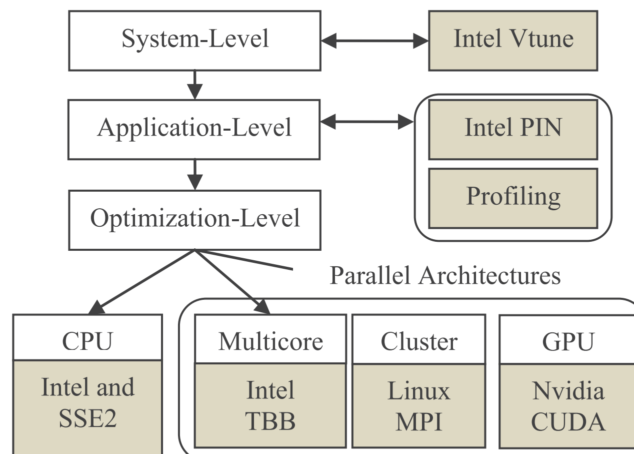


Figure 1. Model illustrating the process we followed to characterize, profile, optimize, and port our application to parallel architectures.

Case studies continue to play an important role in addressing programming challenges in HPC and in maturing computational science [1]. We re-iterate the importance of

978-0-7695-4214-0/10 $26.00 © 2010 IEEE
DOI 10.1109/HPCC.2010.46

IEEE computer society

integrating analysis and performance optimization in transition plans to parallel architectures which are too often ignored or under-prioritized in practice. We achieved important gains by following the left-hand side of Fig. 1 from system-level to optimization-level analysis through sequential execution on CPU that was magnified in parallel. Careful analysis identified where the application spends more than 95% of its time (after optimization) which is never obvious through manual source code inspection alone for anything beyond a simple example. (It is easy for developers to think they fully understand their application's behavior but they are usually wrong and best practices dictate getting the data [2].)

We are researching how to leverage more advanced and difficult to exploit parallelism in our hotspot region which guarantees high returns on our efforts. We are examining our straight-forward implementation of a self-scheduling algorithm (or master-slave algorithm) using MPI since we discovered limitations with some workloads under genetic algorithm optimization that need to be explored further and we are porting the application to GPU.

This paper is organized as follows. Section 2 includes background and a brief discussion of related work. Section 3 demonstrates characteristics of Kingen through workload characterization and the methodology we used for application profiling. Section 4 describes our computing framework for kinetic simulations on multicore and cluster architectures including a coarse-grained parallel implementation of the model, developed using Intel's Threaded Building Blocks (TBB) on multicore systems and MPI on a 204-core cluster. In section 5 we explore our ability to efficiently port applications that scale with massively parallel architectures.

II. Background and Related Work

In this section we briefly summarize research efforts related to high performance kinetic modeling and application profiling and workload characterization.

A. High Performance Ion Channel Kinetic Modeling

Researchers in [3] developed software called kinetic preprocessor (KPP) that was designed as a general analysis tool to aid simulation of chemical kinetic systems. They maintain computational efficiency in generated code but don't explicitly support emerging architectures. However, [4] extended KPP to generate optimized code for multicore platforms and compared their implementation for the Weather Research and Forecast with Chemistry model on three multicore platforms: GPUs, Cell Broadband Engine, and quad-core CPUs.

Neural simulations that exploit multicore architectures are explored in-depth in [5]. They focus on solving linear systems of equations in parallel and automatic load-balancing. Ion channel modeling is one component in systems modeling of many types (the neural cell as a system, a network of cells involved in the same function as a system, etc.) and progress made at this level facilitates modeling at other levels.

B. Application Profiling and Workload Characterization

Analyzing program behavior is critically important to optimization goals and in guiding porting efforts to parallel architectures. Research in parallel debugging, performance prediction, auto-tuning, scheduling, as well as practical application in administration of HPC installations (supercomputing centers), all rely on application profiling and workload characterization. In general, the complexities of emerging systems are under intense scrutiny to address that errors are so easy to make, yet so difficult to understand and isolate.

C. Analysis Tools

There are many different analysis tools available that vary by compatibility, type of information, level of detail, runtime impact on code, scalability, and ease of use [6]. Yet, few agree that the tools they use in HPC satisfy all their needs. There is a strong need for tools that are non-intrusive, easy to use, and correct.

For this analysis we used Pin [7]. PIN is a JIT-based instrumentation engine that supports binary introspection on the IA32, EM64T, IPF and XScale platforms via the use of Pin Tools that export a rich user interface. Without applying instrumentation, the system can be viewed as a native-to-native binary translator. Pin performs various optimizations such as code caching, trace linking, inlining, register allocation and liveness analysis on the generated code to minimize the overhead incurred at run-time.

III. Application Characterization and Profile

Successful porting of scientific applications to multicore and manycore architectures heavily depends on primary characteristics of the application. We profiled Kingen and characterized our typical workload to optimize the application for performance and to identify how best to map it to parallel architectures.

A. System Level Analysis

Performance increase is the driving factor for parallelizing an application, so a first check when porting a program to parallel architectures should be on thread utilization. Kingen's thread profile (across all cores, the threads are fully utilized 93% of the time, underutilized 4.8%, and serial 1.65%) demonstrates that all available cores are kept busy most of the time indicating that Kingen is processor-bound. Stalls due to memory or disk requests can't be happening very often and the way to increase performance for this application is through more cores.

B. Application Level Analysis

There are no system level concerns, so we continue with application level analysis. To port our application, it was important to understand where it spends most of its time during execution to focus on those regions of code that are actually impacting performance.

Our runtime is dominated by the simulation loop where each chromosome is evaluated. We removed code redundancies in this loop and the overhead inherent in function calls. These manual optimizations had a significant

impact on performance as shown in Fig. 2. We are currently evaluating parallel options to address the bottleneck in the simulation loop.

During analysis of the runtime hotspots, we also looked at cycles per instruction retired (CPI) and at floating point (FP) related metrics since it is clear Kingen is very compute-intensive due to its dependence on FP operations. Through experimentation on how to reduce the CPI and FP performance impacting issues it became clear we needed to upgrade our compiler and establish a new baseline. The upgrade changed, among other things, the instruction set default to require Intel SSE2 instead of X87 instructions. Significant FP operations affecting performance were addressed by upgrading an optimizing compiler.

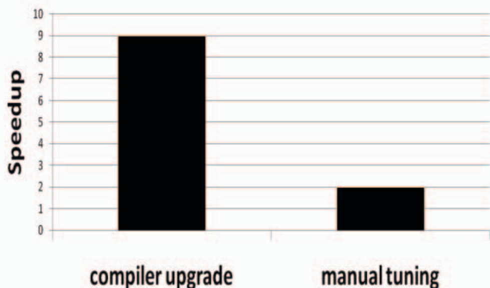


Figure 2. Speed-ups achieved by upgrading the compiler and through manual tuning.

IV. COMPUTING FRAMEWORK

We implemented Kingen on several different multicore architectures: four cores on 32-bit windows and 64-bit linux (Intel Quad Q6600), eight cores on 32-bit windows (Intel X5355) and a 204-core linux based cluster. We evaluate our parallel implementation, speedup, and computational complexity in the following subsections.

A. *Kingen Description*

Kingen simulates AMPAR ion channel activity and optimizes kinetic model rate constants to biological data. Kingen uses a genetic algorithm to optimize (fit) the rate constants of AMPA receptors. GAs are based on Darwinian evolution and are increasingly used to solve hard problems of practical interest in diverse fields [8]. GAs are known for their ability to find optimal solutions within a defined parameter space from initial random populations [9]. Kingen approximates the solution of a system of linear differential equations that kinetically describe AMPAR-mediated ionic currents using the Runge-Kutta 4th order method.

Kingen was carefully restructured to be a highly parallelized program (shown in Fig. 3). This program is designed to study AMPARs but the approach is also applicable to other types of ion channels [10, 11].

```
initialize chromosomes, model parameters, and random numbers
for (int i = 0; i < number of generations;  i++){
    // PARALLEL EXECUTION
    For (int a = 0; a < number of chromos; a++) {
        ic50 error
        ec50 for peak error
        ec50 for steady error
        entry to desensitization error
        tau error
        recovery from desensitization error
    }
    // Genetic algorithm runs sequentially
    genetic algorithm
        pick N random chromosomes and select best among them
        apply mutation operator
        apply crossover operator
```

Figure 3. Pseudo-code of coarse-grained multicore TBB parallelism. Each chromosome in the population describes a rate constant parameter set in the kinetic model and the program evaluates each individual in parallel.

B. *Coarse-grained Multicore TBB Implementation*

TBB is developed by Intel as a template library that extends C++ [12]. TBB abstracts CPU resources and allows parallelism to be expressed with constructs that were designed to be familiar to C++ developers.

We implemented coarse-grained parallelism with parallel_for, a TBB template function that parallelizes loops that have independent iterations. We parallelized the loop that iterates over each chromosome to evaluate each chromosome's "fitness". The iteration space is broken up into chunks of work and TBB runs each chunk on a separate thread.

Kingen's coarse-grained parallelism encloses a great deal of computation complexity. Fig. 4 is a graph of time as a function of the number of chromosomes being evaluated in each generation. The computation complexity between generations under several computing frameworks is apparent as the problem size starts to grow exponentially quickly.

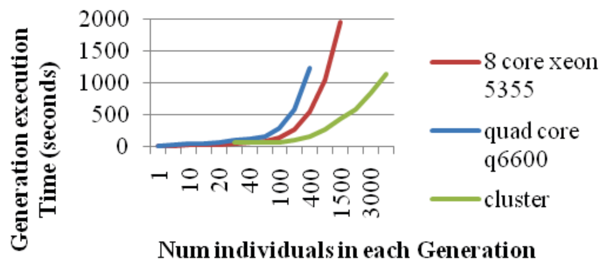


Figure 4. Computational complexity. Genetic algorithms rely on genetic diversity to improve convergence. As you increase the number of individuals per generation the problem starts to grow exponentially.

C. *204-Core Cluster with MPI Implementation*

We also ported Kingen to a Linux-based computing cluster that has 17-nodes (1 master, 16 compute nodes) with 2 x 2.2 GHz AMD Opteron six core processors per node.

The cluster computing power includes 204 cores (12 core on master node and 192 cores on compute nodes). The cluster enables us to examine much larger populations in the genetic algorithm. This is quantified in Fig. 4 where you can see how many individuals we can use in each generation on the cluster before our execution time grows exponentially (~600).

We parallelized the application at the same coarse-grained level as we did on the multicore architectures using MPI and a master-slave algorithm to automatically handle load distribution to the available cores. As soon as a given compute core is done processing one chromosome it sends its results back to the master, and the master, knowing which core just became available for more work, sends that core another chromosome to be processed. This algorithm is elegant in that no one compute node is ever held up waiting for some other process to finish. Each compute core gets more work assigned as soon as it has completed its task for as long as there are more chromosomes to process.

D. Fine-grained Parallel Opportunities?

Speedup analysis on the cluster indicates that our coarse-grained parallelism can be too coarse when working on multicore and smaller scaled systems. On more moderately parallel systems (as opposed to massively parallel systems) common in many research departments and test frameworks it makes sense to explore dynamic scheduling algorithms and/or finer-grained parallel approaches within the application. For example, we calculate the error for a given rate constant (chromosome) many times under different conditions for many different kinetic processes and at different points in the curves that describe these processes (e.g. inhibition of currents by glutamate and the peak current, time course of entry into desensitization, time course of current in response to 5mM glutamate and recovery from desensitization under different concentrations of glutamate). Each of these errors is summed together to get the "fitness score" for each chromosome and each error can be run in parallel. Our coarse-grained parallelism at the chromosome level runs each of the simulations sequentially.

The workload can be distributed differently for more efficient use of the idle cores that impact the speedup at different combinations of the number of execution cores and number of chromosomes under smaller workloads as seen in Fig. 7 (speedup was also evaluated for 100 chromosomes with similar results, not shown) by evaluating each error simultaneously. Of course, you have to be careful of the relationship between the number and size of messages and communication overhead in MPI that is known to impact performance. We are evaluating different approaches to determine the effects they have on efficient mappings to multicore and manycore architectures.

The simulation loop computes 4th order Runge-Kutta formulas to numerically integrate differential equations that describe our kinetic scheme. ODE solvers are inherently serial since they time-step the solution over an interval and each time step is dependent on the calculations from the previous time step. This is true in our simulation loop. However, any gain made here will have a large impact on overall performance since this is where Kingen spends the vast majority of runtime.

The simulation loop performs many complex calculations for each state in the kinetic model. Fig. 5 graphs this complexity by measuring the dependence height, number of operations, and number of memory operations involved in each state computation. Fig. 6 shows the dependence graphs for each state using several as examples. These two figures demonstrate there are parallel opportunities within the sequential constraints of the simulation loop. The differential equation for each state represents 22-way parallelism, but they are hard to exploit.

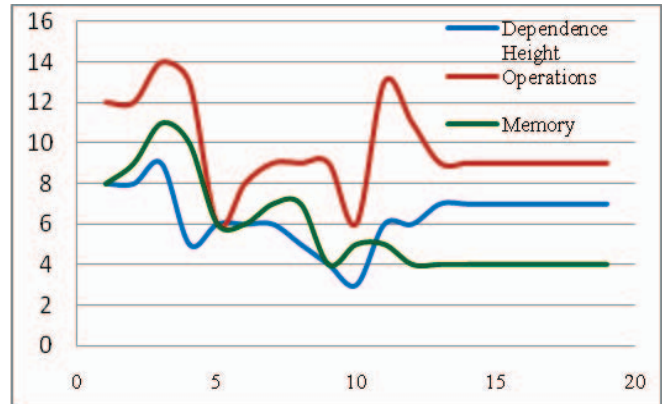


Figure 5. Calculation complexity for the differential equations describing each state in the kinetic model over dependence height, number of operations, and number of memory operations. The x-axis represents the ODE for each state in the kinetic model. The y-axis is the state calculation complexity.

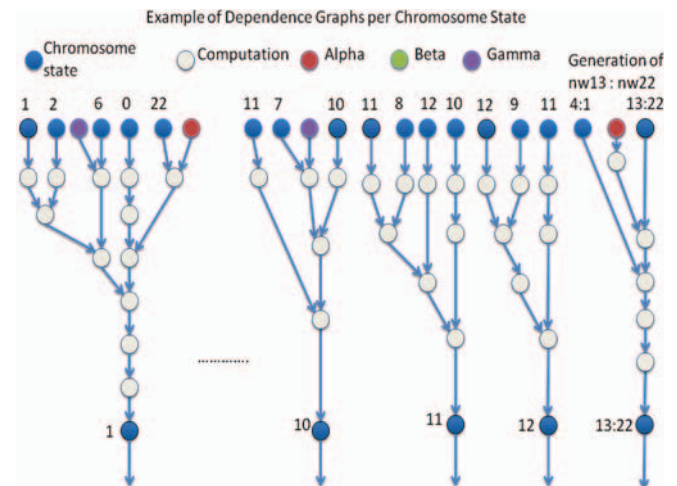
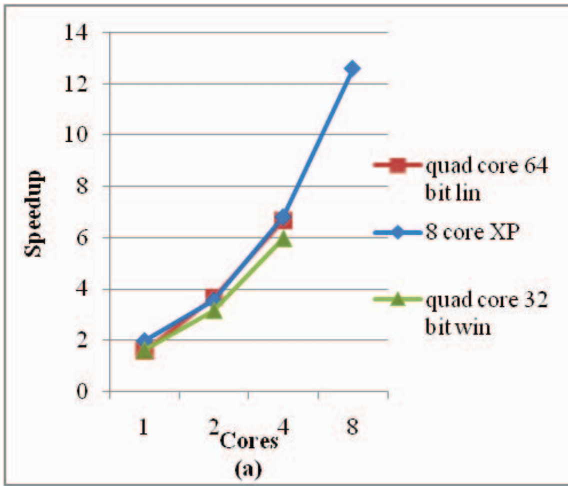


Figure 6. Dependence graphs per state. Each are independent and point to further optimizations opportunities in code.

V. Experimental Results and Analysis

A. Simulation and Experimental Set-up

Fig. 7 shows speedups we have obtained as a result of our step-by-step process and coarse-grained parallelism. Note that Fig. 7 is based on 50 chromosomes per generation. This means that 51 cores is the most we can efficiently use with this workload under this implementation (one core for the master and one core to evaluate each chromosome) which explains the leveling off in Fig. 7b between 60 and 70 cores (see speedup until 51 cores but didn't measure until 60 cores; flat-line between 60 and 70 is because we are not using the additional cores for this workload). As the number of chromosomes per generation increases, the performance improvement is magnified as shown in Fig. 8.



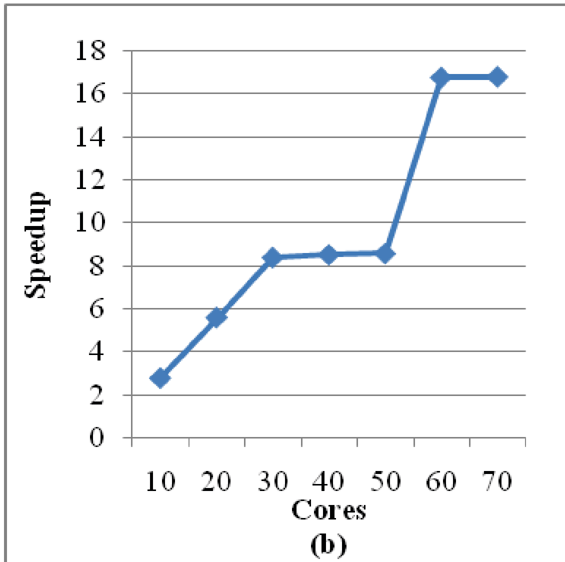


Figure 7. Speedup on several architectures with a baseline of 50 chromosomes. Timings were averaged over 10 executions: (a) Quad core and 8 core windows and linux multicore architectures. We set the baseline to our current model in serial before redundancies in the hotspot region or other optimization opportunities were addressed. You see ~2 speedup even sequentially (1 core) because the sequential baseline is post compiler upgrade and pre-manual code optimizations; (b) 204 core linux based cluster with MPI. This baseline includes all code optimizations.

Figure 7b shows a flattening out of the linear speed-up we were achieving between 30 and 50 cores. This is explained when you look at how the chromosomes (individuals in the genetic algorithm) are sent to each core using MPI messaging and the master-slave algorithm (see table I). The workload modeled in Fig. 7 has 50 chromosomes to process. At first all available compute cores receive one chromosome to process. The only exception is if there are less chromosomes than available cores in which case most receive 1 (depending on the number of chromosomes) and the rest are idle; this happens with this workload when the number of cores is 52 or greater. With 51 execution cores, there is one master and 50 compute cores, one for each of the 50 chromosomes, and execution is bound by the time it takes to evaluate one chromosome per generation on average.

After the first distribution of work, some compute nodes receive more chromosomes to process. For example, the first row of table I is read as: with 10 execution cores, 1 is master and 9 are compute cores; 9 compute cores process 5 chromosomes each on average (9*5 = 45 chromosomes accounted for), and ultimately 5 of those 9 must compute an additional chromsome or 6 total (for a total of 50 chromosomes among all cores). It doesn't matter if all 9 cores compute 6 chromosomes or if just 1 core computes 6 chromosomes; the runtime is bound by the core(s) with the most work to do.

TABLE I. Chromosome Distribution

Number Execution Cores	Available Compute Cores	Number chromosomes processed by each compute core on average	Number chromosomes processed by most tasked compute core(s)
10	9	5	6
20	19	2	3
30	29	1	2
40	39	1	2
50	49	1	2
60	59	1	1

Between 30 and 50 cores, where the speedup levels off, you can see from table I that the number of chromosomes processed by the most tasked core(s) is the same, two. This is why there is no speedup in this range. You see linear speedup between 10 and 20 cores and between 20 and 30 cores because the cores with the most work went from 6 chromosomes to 3 chromosomes and 3 chromosomes to 2 chromosomes, respectively. The execution is bound by the number of chromosomes processed by the heaviest tasked core (s) through the distribution of work. We don't see the speedup affected on the quad core and 8 core multicore systems because with any reasonable workload (50 is about

the fewest chromosomes we can consider) we don't have enough cores to see this effect.

Figure 8 demonstrates that speedup with a more substantial workload doesn't have the same dependence on the combination of the number of cores and the number of chromosomes as the application does with smaller populations. We tested 1,000, 3,000 (shown in Fig. 8) and 5,000 chromosomes and they all had similar speedup lines on the cluster. We believe there is a tipping point between 100 and 600 chromosomes where the effect seen in Fig 7b is active and it makes sense to devote more effort to keeping idle cores busy. This point is likely related to where the execution size starts to grow exponentially (see Fig. 4) and we are currently quantifying it more precisely. There is a compelling need to improve performance with smaller workloads on moderately parallel systems for more simple models, test, and proof of concepts common within these ranges.

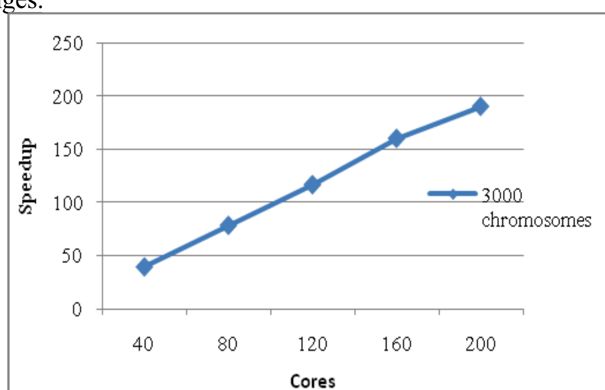


Figure 8. Speedup using many more chromosomes on the 204-core linux cluster. We had to experimentally derive the baseline sequential runtime (3000 chromosomes takes prohibitively long). We measured the average execution of 1 chromosome (over 20 examples) and found little variation in the mean. We multiplied this average by 3000 for the baseline sequential value in the graph. We compared this approach with smaller populations (10, 20 and 50 chromosomes) over multiple generations and found this approach yields accurate results.

VI. CONCLUSIONS

Impressive performance gains for Kingen were achieved with a process that systematically proceeds through different levels of analysis. Improving the sequential version is important as those gains were magnified in parallel. On a multicore architecture, the TBB implementation achieves a ~15x reduction in run time (2-4 days with more demanding workloads as compared with >30 days sequentially with a simpler model). Our analysis presented here suggests there is even more parallelism to exploit. The most efficient map depends on the architecture and is probably significantly different for multicore architectures and manycore architectures. In addition, this application may be ideally suited for GPU acceleration.

Scientific applications need to be thoroughly profiled and typical workloads characterized to reach performance goals and to map to new architectures. Overall, the improvement of the time-intensive optimization of kinetic models will accelerate discovery in neuroscience. Furthermore, the methodology to do so will be applicable to a broad range of scientific applications accelerating discovery in computer science. Researchers in HPC need reliable tools to facilitate porting to emerging systems that reduce errors and increase throughput.

ACKNOWLEDGMENT

The authors gratefully acknowledge the following support: NIDCD DC008297 (KJR and AG), NINDS NS056090 (TAB), Achievement Rewards for College Scientists (AG), and institutional support from the Department of Pediatrics and The Children's Hospital Research Institute (TAB and AG).

REFERENCES

[1] D. E. Post, R.P. Kendall, and R. F. Lucas, "The Opportunities, Challenges, and Risks of High Performance Computing in Computational Science and Engineering," in Advances in Computers, vol. 66, M. V. Zelkowitz Ed. New York: Academic Press, 2006, pp. 239–301.

[2] R. Sites, "What your mother never taught you about multicore programming". Presentation at Front Range Architecture Compilers Tools and Languages Workshop, Fall 2009.

[3] V. Damian, A. Sandu, M. Damian, F. Potra, and G.R. Carmichael. "The kinetic preprocessor KPP – a software environment for solving chemical kinetics". Computers and Chemical Engineering, 26, 1567-1579, 2002.

[4] J. C. Linford, J. Michalakes, M. Vachharajani, and A. Sandu. "Multi-core acceleration of chemical kinetics for simulation and prediction". Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC). November 2009.

[5] H. Eichner, T. Klug, and A. Borst. "Neural simulations on multi-core architectures". Frontiers in Neuroinformatics 3 (21), 1-15. 2009.

[6] D. Skinner, "Integrated performance monitoring: understanding applications and workloads". Presentation at Center for Scalable Appication Development Software Summer Workshop, 2008.

[7] C-K. Luk and et al., "Pin: building customized program analysis tools with dynamic instrumentation", Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, June 2005.

[8] D. Goldberg. The Design of Innovation: Lessons from and for Competent Genetic Algorithms. Boston, MA: Kluwer Academic Publishers, 2002.

[9] D. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Boston, MA: Addison-Wesley Professional, 1989.

[10] I.J. Youngs, "Exploring the universal nature of electrical percolation exponents by genetic algorithm fitting with general effective medium theory," J.Phys. D: Appl. Phys. 35, pp. 3127-3137. 2002.

[11] M. Gurkiewicz, A. Korngreen, "A numerical approach to ion channel modelling using whole-cell voltage-clamp recordings and a genetic algorithm," PLoS Comput Bio, 3(8), pp. 1633-1647. 2007.

[12] J. Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. Sebastopol, CA: O'Reilly Media, 2007.