Thread Group Multithreading: Accelerating the computation of an Agent-based Power System Modeling and Simulation Tool – GridLAB-D

Shuangshuang Jin and David P. Chassin

Pacific Northwest National Laboratory, Richland WA 99354, USA
E-mail: {shuangshuang.jin, david.chassin}@pnl.gov

Abstract

GridLAB-D™ is an open source next generation agent-based smart-grid simulator that provides unprecedented capability to model the performance of smart grid technologies. Over the past few years, GridLAB-D has been used to conduct important analyses of smart grid concepts, but it is still quite limited by its computational performance. In order to break through the performance bottleneck to meet the need for large scale power grid simulations, we develop a thread group mechanism to implement highly granular multithreaded computation in GridLAB-D. We achieve close to linear speedup on the multithread version running on general purpose multi-core commodity for a benchmark simple house model. The performance of the multithreading code also shows very favorable scalability properties, resource utilization, and much shorter execution time for large-scale complex power grid simulations.

1. Introduction

GridLAB-D™ is an open source, next generation agent-based smart grid simulator that provides unprecedented capability to model the performance of smart grid technologies and concepts in the power distribution system [1]. It incorporates advanced power system, thermal, control, and economic modeling techniques with high-performance solution algorithms to deliver fully coupled smart grid system models. These models include power flow solvers, load models, market models, distribution automation models, etc., to examine, in detail, the interplay of smart grid components.

Over the past few years, GridLAB-D has been used to conduct important analyses of smart grid concepts. But the scope of these analyses remains quite limited by the computational performance of the core solver, which must continuously synchronize the states of many thousands of heterogeneous agents. At this time, GridLAB-D is being used to model roughly 50 MW distribution systems composed of tens of thousands of complex agents. The topology of agents is so variable that it challenges conventional parallelization methods used in most ordinary/partial differential equation solvers. Although simulations of systems as large as 100 MW are possible, they can take hours or days to complete. Growing demand is expected for results from large GridLAB-D models for utility customer systems above 100 MW. It is urgent that the performance of GridLAB-D be significantly improved through effective parallelization. This improvement aims to optimize computation resource allocation for better scalability and reduce the time needed to complete large scale power grid simulations.

In this paper, we discuss a new thread group mechanism to implement multithreaded computation in GridLAB-D. It obviates the need for a conventional threadpool: significantly reduces the thread management overhead needed for job assignment, and keeps the advantage of not creating and destroying too many threads. The thread group implementation of GridLAB-D performs very well. It shows promise of excellent scalability, limited overhead, and close to linear speedup compared against the conventional single-threaded version running on general purpose multi-core commodity computing hardware.

This high-performance version of GridLAB-D has broad market and life expectancy. The potential user community is substantial. The long-term total user base includes at least 2000 small and large electrical power generation, transmission, distribution, and retail companies in North America. Multiple teams within those companies have an immediate business and regulatory need. Improved simulation and modeling capabilities can help them to evaluate fully the impact of smart-grid technology and business strategies on electricity delivery systems.

This paper is organized as follows: Section 2 introduces the agent-based simulator and data flow of GridLAB-D. Section 3 illustrates the techniques and
problems of the two previous multithreading approaches implemented on GridLAB-D prior to this current approach. Section 4 presents the detailed design and implementations of this new thread group mechanism. Section 5 presents the experimental results on multi-core PCs. Section 6 concludes the paper with a brief discussion of the future work.

2. GridLAB-D

GridLAB-D is one of the first open-source agent-based power distribution system simulation and analysis tools that provides information to users who design and operate electric power systems and who wish to examine the interplay of the smart grid technologies [2].

Agent-based modeling is used to represent and study dynamic systems of interacting and potential evolving agents, and considers the systems from a disaggregated (i.e., bottom-up) perspective. This level of detail involves the precise specification of many agent attributes and behaviors, and their interactions within a modern smart power distribution system. These interactions are determined dynamically at runtime by the internal structures, informational states, etc [3]. The advantage of an agent-based simulator in the context of smart grid analysis is that it makes no assumptions about the aggregate performance of either the individual technologies, or their combined effects in a complex system. Emergent behavior that is desired can be tested and designed, or when undesired, detected and eliminated. This attribute is unattainable with conventional power system simulation and modeling tools and is what makes GridLAB-D a truly unique tool in the panoply of power system modeling tools.

GridLAB-D’s simulation model is event-driven. The simulation core has an advanced agent synchronization algorithm that simultaneously coordinates the state of independent devices, each of which is described by multiple differential equations solved only locally for both state as a function of time and time as a function of state. Aside from the fundamental difference in how emergent behavior is addressed, the advantages of this algorithm over the traditional finite difference-based simulators and traditional differential-based solvers are:

- It allows integration of new modules and third-party simulation systems; and
- It is not necessary to integrate all the agent’s behaviors into a single set of differential equations that must be solved concurrently [1].

GridLAB-D concerns itself with objects, the class of each of those objects, and the modules that implement those classes. Each object has a synchronization method that performs two essential functions. First, it updates the state of an object to a designated point in time, and second it lets the core solver know when the object is next expected to change state. This is vital for the core to determine the time to which the clock will be advanced for the next expected state change and to which the objects will next be synchronized.

2.1. Data flow

The core in GridLAB-D sets up the main simulation, initializes and ranks the objects, and runs the simulation until it either a) the simulation end-time is reached, b) it settles to equilibrium (i.e., there are no further state changes expected), or c) it runs into a problem that prevents further processing.

The whole simulation is composed of a number of arbitrarily long time steps. The length of each time step is calculated based on which object will next change state. At each time step, the core’s executive scheduler iterates through until all the objects converge on a globally consistent state. The scheduler passes control to each objects synchronization routine, giving it the timestamp of the next expected state change. The synchronization routines attempt to advance each object’s internal clock to the time indicated. If successful they return the time of the next expected state change. Then the executive advances the global clock and processes the new states for all the objects in the model for that time step.

If any object does not converge, it returns an indication that it is expecting other objects to perform an iterative process until convergence is reached. If the number of iterations without converging is too large, the simulation will abort. This prevents the simulation from locking the clock indefinitely, a situation that is typically caused by a modeling error or time resolution inconsistency, and generally must be remedied by the user.

The core continues synchronizing objects and advancing the global clock until all objects indicate that no further state changes are expected or desired. This is the stopping condition and the simulation consequently ends. Figure 1 shows the data flow of this simulation process.
2.2. Object synchronization

Object synchronization is an intricate sequence of calls to various object synchronization functions. These are executed by rank in three passes through the objects, followed by a commit pass. A rank is defined as a group of object that shares a common dependence on objects of higher rank and no dependency on each other. The first pass (called the pre-top-down pass) through the ranks is a top-down pass during which most objects will initialize variables that lower ranked objects depend on. The second pass is a bottom-up pass in which lower rank objects are given the opportunity to update accumulators in objects they depend on. The third pass is a post-top-down pass during which objects may complete updates based on changes to accumulators in higher ranked objects, if any. In each iteration of the main iteration loop, the executive scheduler calls the synchronization function for every object in the order of these three passes, as illustrated in Figure 2.

Pre-top-down and post-top-down sync calls start from the objects with the most dependents working down the rank tree. The bottom-up sync calls work from the objects with no dependents and proceed up the rank tree. All of the objects at a given rank are processed before any objects in the next rank are processed. Presync is called during the pre-top-down pass. Sync is called from the bottom-up pass. Postsync is called with the post-top-down pass. The synchronization function requires the time to which the object is supposed to synchronize. The function returns the time at which the object’s internal state will next change.

Figure 1. Data flow of GridLAB-D’s simulation process.
Figure 2. Object synchronization – presync (a), sync (b), and postsync (c) pass sequences.

3. Previous approaches

A thread is an independent stream of instructions that can be scheduled to run as such by the operating system [4] but shares access to a common memory space. The pthread library used in GridLAB-D is the POSIX “C” API thread library that has standardized functions for using threads across different platforms [5]. To realize GridLAB-D’s performance enhancements on multi-core commodity PCs, we selected pthread as the development tool to implement the multithreaded optimization in GridLAB-D.

The object synchronization requirements in GridLAB-D preclude updating objects at a given rank while any objects at another rank are processed. As a result, only the objects within a same rank may be synchronized simultaneously. Figure 3 illustrates the data flow inside the main object synchronization loop, and indicates where are the places that multithreading implementation is applicable.

Figure 3. Multithreading can only be implemented at object rank list level to contain the synchronization integrity.

Two approaches are examined as initial tests of multithreaded implementation of object synchronization in GridLAB-D. The first one is a simple pthreads implementation.

3.1. Pthreads implementation

For each of the object rank list, we create n threads, where n is the number of objects in that rank, and assign each thread a list of objects to synchronize. These threads are destroyed after the synchronizations of all the objects in this object rank list are completed. Figure 4 shows the implementation of this approach.

Figure 4. Simple Pthreads implementation in GridLAB-D.

This implementation is simple and the speedup is obvious for small power grid simulation cases. However, it has two problems. First, each thread has too little work relative to the overhead of starting and terminating. The high ratio of thread setup time to thread computation time would presumably lead to poor performance. Second, software threads require memory for their stacks and private data structures. In extreme cases, such as running large simulation cases that involve many objects in frequent iteration loops, there can be so many threads that the program runs out of memory [6]. The solution is therefore to limit the number of threads created by reusing them employing a threadpool.

3.2. Threadpool implementation

The creation and destruction of large numbers of threads comes with high system overhead costs. Employing a threadpool avoids the overhead of frequently creating and destroying thread and consists of a fixed number of threads to which incoming jobs are to be assigned in an efficient and easy way [7].

In GridLAB-D, a global threadpool (of m threads) is created before the first call of the object synchronization, as shown in Figure 5. The threadpool is composed of thread lists, mutexes, and condition variables to signal available and accepted jobs from the
working queue of the jobs. For each object rank synchronize event, the threadpool dispatches an available thread to complete the synchronization task for the next object. At most, \( m \) synchronization jobs are allowed to operate at any given time. When all threads in the pool are busy, dispatch will block until a thread becomes free.

![Figure 5. Global threadpool implementation in GridLAB-D.](image)

This implementation overcomes the deficiencies of the simple pthreads creation/destruction scheme and the thread creation/destruction overhead is easily mitigated. Better performance and system stability for large simulation are achieved. However, it is still not an optimal solution for GridLAB-D’s multithreading implementation. GridLAB-D’s rigid requirements to control the order of synchronization events create a high demand for thread synchronization objects, which reduces scalability.

Different simulation cases have different number of object rank lists in each synchronization pass. The number of objects in each object rank list can vary from one to many thousands. Using the same predefined number of global threads to parallelize each different object rank list most likely leads to highly unbalanced thread usage. For object rank lists with few objects, most of the threads resources are underutilized. For object rank lists with large numbers of objects, the performance gain of multithreading is limited by the number of global threads. Furthermore, the dispatch of threads and management of the queue of jobs are so frequent that the overhead may undermine the benefit of multithreading. Given the high model structure variability known to exist for many GridLAB-D models, the generality of the threadpool method seems limited and the performance gains are mixed, depending on the structure and size of the model.

Another disadvantage of the global threadpool implementation for GridLAB-D is that we have to introduce additional blocking to hold the threads from running beyond one object rank lists to another. The sequencing requirements for object rank lists means they must be executed completely before the next rank starts. If any “fast” thread takes the job from the next object rank list after finishing all its jobs in one object rank list, while other threads are still doing work in the previous object rank, a loss of synchronization can be caused in the simulation. Implementing of additional blocking that restricts the working range of threads in one object rank list before all its objects are processed avoids this problem. But obviously it is inefficient for the multithreading performance and fails to employ the natural, albeit complex, sequencing behavior of the single-threaded code.

4. Multithreading design and implementation

Based on the evaluation of the two initial multithreading approaches, and the special synchronization characteristics of the GridLAB-D simulator, a specific multithreading mechanism was developed using a thread group mechanism. The implementation fits exactly into GridLAB-D’s event sequencing structure, retains the benefits of a threadpool, resolves the problems related to memory management, and optimizes GridLAB-D’s performance in multithreading environments.

4.1. Thread group mechanism

GridLAB-D’s simulation core is composed of potentially thousands of concurrent heterogeneous main iteration loop, depending on the size of each simulation case. However, because all the objects are initialized and their ranks are set up before the main iteration loop starts, the number of object rank lists, the number of objects in each object rank list, and the execution order of each object rank list in each different synchronization pass can be determined prior to entering the main loop.

The basic idea is to create exclusive thread groups for each type of synchronization events and reuse the threads of these groups for each pass of the main iteration loop. A thread group is not a real threadpool as described in section 3.2, which manages threads
dispatch and maintains a job queue. It is a simpler threads implementation using mutexes and condition variables to control many dedicated threads pool and performs the work of a threadpool separately for each synchronization event.

As shown in Figure 6, exclusive groups are created during the first main iteration loop for each object rank list. Threads in each of these groups only do synchronization work for the objects in its own object rank list. After the completion of its run, threads in this group are locked and reserved for the next run. Only when the exact object rank list is executed again in the next iteration loop, the threads of its corresponding threads pool are unlocked to pick up the object synchronization work for this object rank list.

**Figure 6.** Thread group mechanism in GridLAB-D.

There are three advantages of such a thread reservation lock and unlock mechanism over the previous approaches. First, the number of threads pool is fixed according to the number of object rank lists. There’s no unpredictable overhead of starting and terminating threads, and hence no memory usage issues which may affect the system performance. Every object rank list has its own thread group to do its work. They exist until the end of the whole simulation and are reused thousands of times during the entire run.

Second, each thread group in its object rank list works as a separate threadpool. Mutex and condition variables are used to indicate the availability of a thread. It is more efficient and lower in cost compared to a global threadpool, particularly with respect to thread management, job dispatch, and queue maintenance.

Third, there is more flexibility to adjust the number of threads used for each group to parallelize the object synchronization. This helps to avoid the situation of running too few or too many jobs for a group and right-sizes the multithreading resource utilization during the simulation.

### 4.2. Thread group implementation

The thread group implementation falls into four parts: the declaration of mutex and condition variables, the creation of thread group, the implementation of threads lock and unlock, and the realization of memory locking.

#### 4.2.1 Mutex and condition variable

Four arrays of mutexes and condition variables are used in the implementation: `startlock` mutexes, `start` condition variables, `donelock` mutexes, and `done` condition variables. Each object list has its own `startlock/donelock` mutexes and `start/done` condition variables. The `startlock` mutex with `start` condition variable signals the unlock of waiting threads to start synchronizing objects in the object rank list. The `donelock` mutex with `done` condition variable signals the lock of working threads upon the completion of all the object synchronizations in this object ranks till the next run. These are allocated and initialized for object rank lists before the start of the main iteration loop.

There is another variable called `donecount`, which is initialized to the total number of objects expected to be synchronized in an object rank list. When `donecount` reduces to zero during synchronization events, it indicates the end of the processing of one object rank list.

#### 4.2.2 Thread group creation

Thread groups for each of the object rank lists are created in the first iteration of the main iteration loop during the simulation. Taking one object rank list as an example, the following steps describe how a thread group is created:

1. Determine the number of threads needed based on the number of objects and the number of available processors.
2. Allocate and initialize the thread group.
3. Assign starting object for each thread and construct the object list associated with it.
4. Create and initialize the threads in a stopped state.

#### 4.2.3 Threads lock and unlock

Threads lock and unlock are implemented inside two parts: the iteration loop and the object synchronization function.

Inside the iteration loop, the `donelock` mutex and `done` condition variable are used to assure that the threads only work inside their owning object rank list. The `startlock` mutex and `start` condition variable
triggers the threads to pick up object synchronization work for this object rank list.

The \textit{lock/unlock} mechanism inside the iteration loop:

1) Lock access to \textit{donelock}
2) Initialize \textit{donecount}
3) Lock access to \textit{starlock}
4) Update \textit{start} condition
5) Signal all the threads for the updated \textit{start} condition
6) Unlock access to \textit{starlock}
7) Begin waiting for update on \textit{donecount}
8) Unlock \textit{donelock}

Inside the object synchronization function, the \textit{starlock} mutex and \textit{start} condition variable indicate when the object synchronization starts, and the \textit{donelock} mutex and \textit{done} condition variable signal back when the whole object synchronizations in this object rank list are completed.

The \textit{lock/unlock} mechanism inside the iteration loop:

1) Lock access to \textit{starlock}
2) Wait for thread \textit{start} condition
3) Unlock access to \textit{starlock}
4) Process the object lists for this thread
5) Signal completed condition
6) Lock access to \textit{donelock}
7) Signal this thread is \textit{done} for now: \textit{donecount}--
8) Signal update in \textit{done} condition
9) Unlock access to \textit{donelock}

4.2.4 Memory locking. Memory locking is implemented using compare and exchange methods. Each time, more than one object can concurrently write to the same region of memory. It is implemented to prevent one object from overwriting the changes made by another. For example, more than one link can simultaneously update the admittance and current injection accumulators in nodes. Thus it is required to prevent two objects from simultaneously reading the same accumulator and posting their modifications without considering the other's contribution.

The performance profiler tracks how many times locks are requested and how many times each request is delayed to estimate the overall locking efficiency of the system.

5. Performance and results

The thread group method is implemented as described in Section 4 on a general purpose 8-core commodity PC with the following configuration:

- Intel(R) Xeon(R) CPU E5504 @ 2.00 GHz
- Installed memory (RAM): 6.00 GB
- System type: 64-bit Operating System

To assess the computational performance of the multithreading implementation for GridLAB-D, we select two typical power grid models, one that is intrinsically parallel and one that is not intrinsically parallel.

5.1. Test case 1: simple house model

This model only contains the residential module (models single-family homes and various home appliances) and the climate module (loads weather files and provides the current weather data for a model) [8]. All but one of the objects in this model are houses. 10000 houses are defined and simulated for one day from January 1, 2000 to January 2, 2000. These houses can be synchronized simultaneously in multithreading environment since they all belong to a same rank list during each object synchronization pass. Therefore, we expect to see close to linear speedup when the number of threads in use is increased.

To measure the resource utilization and scalability of the multithreading implantation on the 8-core commodity PC, we record the GridLAB-D execution performances for the simple house model case against different number of processors. A sample GridLAB-D execution profile, the resource utilizations of the multi-core system, and the scalability plots will be given in the following subsections.

5.1.1 Execution profile. As shown in Figure 7, the total GridLAB-D simulation time is profiled in two parts: core time and model time. Core time includes the compiler time and synchronization time for built-in objects such as schedules, loadshapes, enduses, and transforms. Model time is the overall ranked object synchronization time. Each type of these object updates is frequently called and highly parallel, but the types must be updated sequentially in each iteration loop. A thread group with thread reservation lock and unlock mechanism is also implemented for each of these built-in object types.

![Figure 7](image-url)
5.1.2 Resource utilization. When running GridLAB-D in multithreaded mode, the performance can degrade significantly because of the granularity of the core parallelization. The normal OS schedule algorithm in symmetric multiprocessing operating systems can be adversely impacted by GridLAB-D. CPU migration can be a very large drain on resources because of cache invalidation issues with objects that have significant memory accesses. The way to control this problem is to establish CPU affinity for each thread in advance. This requires that the core determine which threads get assigned to which CPUs very early on such that optimal CPU balance is maintained throughout the run.

5.1.3 Scalability. A 1-day simulation of the simple house model is executed for different thread counts. Using different number of threads, the total time taken to run the 24-hour simulation decreases from 736 seconds to 97 seconds, which are 117x to 891x real time, as shown in Table 1. The scalability is close to linear when the number of threads in use is increased, as shown in Figure 9. We can also observe that the core time and model time are also proportionally increased with the number of threads in use, which indicates that the multithread implementation for both built-in objects and ranked objects are scaling very well without excessive overhead.

Table 1. Time distribution of running the simple house model using different number of threads.

<table>
<thead>
<tr>
<th># of threads</th>
<th>Core time (s)</th>
<th>Model time (s)</th>
<th>Total time (s)</th>
<th>Simulation rate (s realtime)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>117.9</td>
<td>618.1</td>
<td>736</td>
<td>117</td>
</tr>
<tr>
<td>2</td>
<td>70.6</td>
<td>301.4</td>
<td>372</td>
<td>232</td>
</tr>
<tr>
<td>3</td>
<td>55.9</td>
<td>216.1</td>
<td>272</td>
<td>318</td>
</tr>
<tr>
<td>4</td>
<td>43.3</td>
<td>148.7</td>
<td>192</td>
<td>450</td>
</tr>
<tr>
<td>5</td>
<td>41.5</td>
<td>110.5</td>
<td>152</td>
<td>588</td>
</tr>
<tr>
<td>6</td>
<td>35.9</td>
<td>90.1</td>
<td>126</td>
<td>686</td>
</tr>
<tr>
<td>7</td>
<td>32</td>
<td>77</td>
<td>109</td>
<td>793</td>
</tr>
<tr>
<td>8</td>
<td>31.7</td>
<td>65.3</td>
<td>97</td>
<td>891</td>
</tr>
</tbody>
</table>

5.2. Test case 2: Comprehensive power flow model

The degree to which multithreading enhances performance is highly dependent on the structure of the model itself. Generally, models that have few densely populated ranks will typically parallelize much better than models that have many sparsely populated ranks. Furthermore, models in which the agents within a rank are more decoupled will typically parallelize much...
better than models that have a lot of inter-agent data exchange.

Other than the intrinsically parallel simple house model, which benchmarks the performance of the thread group methodology, a highly coupled comprehensive model is also tested to evaluate the effects of intrinsic model properties on the multithreaded performance.

Unlike the simple house model, which only contains a single object rank list in each of the three passes with 10000 independent house objects, the comprehensive model contains multiple different modules and objects, such as the powerflow module, the residential module, schedules, transformers, implicit schedules, and water heaters, which disperse to 469 various objects.

These objects are not intrinsically parallel. Overall, there are 13 object rank lists in each of the synchronization passes. Eleven of them contain one object only, which is not worthy of any parallelization. On the contrary, it may even introduce unnecessary memory allocation and thread management overheads.

Only two object rank lists contain 150+ objects, which can be parallelized. However, these objects are with disparate object types other than the simple house model case; each of them may have different inter-agent data exchange and require different times to complete its synchronization. The time to finish the synchronization of one object rank list is then determined by the object which takes the longest time to complete its own synchronization. Therefore, the result is expected to be more variable than the simple house model, where all the house objects can almost complete their synchronizations in the same time frame. As a result, the performance of parallelizing the synchronization of diverse objects in this comprehensive model can still be expected to scale, but it won’t improve to the fully extent of close to linear speedup as the simple house model does.

To better illustrate the performance of parallelizing the object synchronization, we focus on the scalability of the corresponding Model time (s). Table 2 gives the time distribution of the object synchronization time during the simulation of this comprehensive power flow model for one month using 1 to 8 threads, respectively. Figure 10 shows the scalability of the model.

As shown in the results, when 8 processors are used, the Model time drops from 403.9 second to 89.1 second, which demonstrates that, despite the lack of linear improvement, the multithreaded technique still shows a significant benefit to utilizing multiple cores for parallelization. Very favorable performance gains can be achieved using the thread group method to parallelize the large-scale comprehensive models in GridLAB-D.

Table 2. Model time (s) of running the comprehensive power flow model using different number of threads.

<table>
<thead>
<tr>
<th># of threads</th>
<th>Model time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>403.9</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>278.4</td>
<td>1.451</td>
</tr>
<tr>
<td>3</td>
<td>200</td>
<td>2.0195</td>
</tr>
<tr>
<td>4</td>
<td>177.1</td>
<td>2.281</td>
</tr>
<tr>
<td>5</td>
<td>128.5</td>
<td>3.143</td>
</tr>
<tr>
<td>6</td>
<td>109.2</td>
<td>3.699</td>
</tr>
<tr>
<td>7</td>
<td>98.4</td>
<td>4.105</td>
</tr>
<tr>
<td>8</td>
<td>89.1</td>
<td>4.533</td>
</tr>
</tbody>
</table>

Figure 10. Scalability of parallelizing the object synchronization for the comprehensive power flow model with 1 to 8 threads. The blue line shows the ideal linear scalability. The red line shows the actual scalability achieved.

6. Conclusions

The thread group mechanism has many advantages over the pthreads and threadpool implementation in GridLAB-D’s parallelization. The raw pthreads approach scales well, but is not reliable because of its potential memory problem. The threadpool approach is reliable, but doesn’t scale well because of its thread scheduling overhead. The thread group approach is both reliable and scalable because it fits exactly into GridLAB-D’s agent-based event sequencing structure.

The implemented thread group management system optimizes GridLAB-D’s simulation performance in multithreaded environment based on its current data structure. The new method shows excellent scalability and efficiency for intrinsically parallel model and very favorable performance for diverse complex model on general purpose multi-core commodity PCs.
In the future, we intend to investigate the more general problem of how to further scale agent-based models, and look into the thread imbalance problem due to GridLAB-D’s non-uniform rank membership property.

Other potential sources of parallelism for GridLAB-D’s may also include: autonomous agents interacting, communicating, speculative scheduling of tasks, and exploiting weak coupling and independent synchronization by means of globally partitioned ranks.

7. Acknowledgments

This work is supported by the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL) under funding for Office of Electricity. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC06-76RL01830. The authors would like to acknowledge Jason C Fuller, Matthew Hauer, Francis K Tuffner, and Yousu Chen, all from the Pacific Northwest National Laboratory, for providing productive discussions and help for the project.

8. References


