Folding Proteins at 500 ns/hour with Work Queue

Badi' Abdul-Wahid[*,‡,§], **Li Yu**[*,‡], **Dinesh Rajan**[*,‡], **Haoyun Feng**[*,‡,§], **Eric Darve**[†,¶,‖], **Douglas Thain**[*,‡], and **Jesús A. Izaguirre**[*,‡,§]

[*]University of Notre Dame, Notre Dame, IN 46656

[†]Stanford University, 450 Serra Mall, Stanford, CA 94305

[‡]Department of Computer Science & Engineering

[§]Interdisciplinary Center for Network Science and Applications

[¶]Department of Mechanical Engineering

[‖]Institute for Computational and Mathematical Engineering

Abstract

Molecular modeling is a field that traditionally has large computational costs. Until recently, most simulation techniques relied on long trajectories, which inherently have poor scalability. A new class of methods is proposed that requires only a large number of short calculations, and for which minimal communication between computer nodes is required. We considered one of the more accurate variants called Accelerated Weighted Ensemble Dynamics (AWE) and for which distributed computing can be made efficient. We implemented AWE using the Work Queue framework for task management and applied it to an all atom protein model (Fip35 WW domain). We can run with excellent scalability by simultaneously utilizing heterogeneous resources from multiple computing platforms such as clouds (Amazon EC2, Microsoft Azure), dedicated clusters, grids, on multiple architectures (CPU/GPU, 32/64bit), and in a dynamic environment in which processes are regularly added or removed from the pool. This has allowed us to achieve an aggregate sampling rate of over 500 ns/hour. As a comparison, a single process typically achieves 0.1 ns/hour.

I. Introduction

A significant challenge in the field of molecular dynamics is that of time scales. Many biomolecular processes, such as folding, occur in the millisecond to second timescales. However, traditional molecular dynamics is limited to time steps of 2 to 3 femtoseconds. While direct sampling using trajectories (possibly computed in parallel) has proven successful in understanding small biological systems, the cost in terms of computational time and data size is high. This is due in part to the unbiased nature of these simulations: the low-energy regions of the free-energy landscape are oversampled while the (arguably more interesting) high-energy transition regions are rarely observed.

Recently, guided sampling techniques have been developed to enhance sampling of low-probability (but critical) regions of the energy landscape, while providing unbiased or "exact" statistics. One such technique is the Accelerated Weighted Ensemble (AWE), which

can be used to predict both thermodynamic and kinetic properties [1]–[3]. This method partitions the conformational space of the protein into macro-cells, that can be mapped along the main transition pathways. An algorithm, used to enhanced the sampling efficiency, is used to enforce a fixed number of parallel "walkers" in each cell. By assigning a probabilistic weight to each walker, one can extract unbiased statistics from this technique.

Traditional molecular dynamics are usually run on dedicated parallel machines with a high-performance network and extensive software optimizations (see for example Anton [4], Desmond [5], NAMD, Gromacs, and LAMMPS). While powerful, such machines are expensive, limited in availability, and typically scheduled in a way that discourages long-running programs. In contrast AWE is designed to be able to run on a large collection of processes connected only through a high-latency low-band with network.

In this paper, we demonstrate how we can harness heterogeneous computing resources such as computer clusters, clouds, and grids consisting of thousands of CPUs and GPUs. The requirements of our parallel software framework is that it can:

1. Concurrently use heterogeneous resources

2. Use resources as they become available and while the simulation is on-going

3. Release resources that are not used so that they are available to other projects

4. Scale to a large number of processes

5. Minimize communication time and idle processes

We therefore consider that the computing resource is not static, but rather evolves dynamically with compute nodes being added and dropped as required. Such a parallel framework was implemented for this paper using Work Queue [6].

We applied AWE to the Fip35 WW domain, which is composed of 545 atoms. To our knowledge, this is the first application of AWE to simulating protein folding for an all-atom description of the protein. We achieve excellent scalability, averaging an aggregate sampling rate of 500 ns/hour, with peak throughput at 1 μs/hour. Additionally, we are able to use many different resources representing multiple computing environments to achieve this. They include GPU and CPU clusters, cycle-scavenge nodes using Condor, dedicated machines, and cloud-computing platforms, namely Amazon EC2 and Windows Azure. Note that all the resources are used concurrently and are able to accumulate the statistics required by AWE. Finally, we show that resources are dynamically allocated as the applications enter different stages, and that the application framework is robust to failure.

The remainder of the paper is organized as follows. Sections II and III describe the AWE algorithm and Work Queue framework. Section IV discusses the scaling issues, challenges, and results. Section V provides results of the Alanine Dipeptide validation of AWE and application to Fip35 WW domain.

II. Accelerated Weighted Ensemble

We briefly present the AWE algorithm. Direct molecular dynamics calculations of rates (the focus of this work) and free energy for large proteins (or other types of biomolecules) is made challenging by the presence of large meta-stable basins (minima of the free energy) that can trap the system for extended periods of time. Although many methods exist to calculate free energy, fewer are available to calculate rates. Most approaches rely on the existence of a single saddle point or transition point (the point of highest energy along a minimum free energy pathway) that determines the rate of transition between the reactant states and product states. This may correspond, for example, to protein folding or a change in the conformation of the protein (i.e., from inactive to active state). Those methods are typically not very accurate when multiple pathways exist, when the saddle point does not sharply "peak," and, in general, calculating such transition points is practically difficult. To circumvent these issues, methods have been developed to calculate rates in a more direct manner while avoiding the fatal slow-down resulting from the existence of meta-stable basins. These methods rely on a partitioning of the conformational space of the protein into a large number of macro-states or cells and a direct sampling of transition statistics between cells. This is the case for example with Markov State Models [7]–[9]. However, a drawback of these methods is that their accuracy depends on the Markovity of the system, which is in practice seldom observed unless "optimal" cells are used [10].

In contrast, the AWE method is also based on sampling macro-states but can be proven to always lead to an exact rate (assuming an absence of statistical errors from finite sampling) with no systematic bias. The method is based on running a large number of walkers or simulations in each cell. Walkers naturally tend to move out of cells that have low probabilies and move toward high-probability cells. As a result some cells get depleted while others get overcrowded. To overcome this and maintain the efficiency of the simulation, we employ the following procedure. At regular intervals, we stop the time integrator for all trajectories and consider how many walkers each cell contains. Each walker i is assigned a weight w_i. When the number of walkers is too low, walkers are duplicated and assigned reduced statistical weights. Similarly when a cell contains too many walkers, we randomly select a walker among a set of walkers S using their probabilistic weights and assign to it the sum of all weights in S. We proved that this can be done in a statistically unbiased manner. Readers interested in further details, including a python code implementation, are referred to [3].

If this procedure is applied as described, upon reaching steady-state the weights of walkers in each cell converge to the probability of the cell so that this procedure can be used to calculate the free energy. Because of the resampling procedure (splitting and merging of walkers), even low-probability cells can be sampled with great accuracy.

However, to calculate rates, a further modification must be made. The reactant states are defined using a set A while set B is associated with product states. Each walker is assigned a color, either blue for A or red for B. At each step in the simulation whenever a blue walker enters B, its color changes to red, and vice versa. The rate from A to B is then directly obtained by computing the flux of blue particles that change color, and similarly for the B to

A rate. This scheme can be extended to multiple colors and sets, leading to not just a forward and backward rates, but multiple kinetic rates for the molecular system. From a simulation standpoint, the method therefore requires the following sequence of steps:

1. Run in parallel a large number of short trajectories. This represents the bulk of the computational work.

2. Parallel barrier. Collect cells statistics and determine how walkers should be split and merged. This step requires minimal exchange of data.

3. Go to step 1 for additional sampling if statistics are insufficient.

III. The Work Queue Parallel Framework

To implement a distributed version of AWE, we used Work Queue [6], a framework for implementing massively parallel data intensive applications. We briefly introduce it, and then focus on the aspects necessary to run AWE at massive scale.

The Work Queue (WQ) framework consists of a master program that coordinates the overall computation and multiple workers that carry out the individual tasks. The master program is written using the WQ API (which supports C, Python, and Perl) and is responsible for defining new tasks and processing their output. Each individual task consists of a standalone sub-program to run, along with a definition of the necessary input and output files for the task. The WQ library is responsible for dispatching tasks to workers, moving input and output files, and handling failures transparently.

To implement the AWE algorithm, the high-level AWE logic is contained in a WQ master program written in Python, while the individual tasks consist of standalone molecular dynamics (MD) simulations. These MD codes are off-the-shelf codes that do not need to be modified. For this paper, we used a code that has the capability of running on multicore CPUs and GPUs.

To run a WQ program, the end user simply starts the master program in a suitable location, then starts up worker processes on whatever resources he or she has available. Workers can be started by hand on individual machines, or they can be started on distributed resources such as clusters, clouds, and grids, including Condor [11], SGE [12], Amazon EC2, and Windows Azure. As the workers become available, the master will start to transfer data and execute tasks.

For a small-scale application (100 workers or less), the mechanism described so far is sufficient. However, as we scale up to thousands of workers running millions of tasks for multiple masters over long time scale, more careful management is required. The following features have been implemented:

Discovery

In the simplest case, one can direct workers to contact a master at a given network address. However, this requires one to reconfigure workers every time a new master program is started, or if an existing master must be moved. To address this, we provide a *Catalog*

Server to facilitiate discovery. WQ masters periodically report their name, location, and resources needs to the catalog server. Workers query the catalog server for appropriate masters by name, and use that information to connect. In this way, masters may be started, stopped, and moved without reconfiguring all of the workers.

Worker Management

Sustaining a large number of workers on a given cluster, cloud, or grid requires constant maintenance. For example, the submission itself may take considerable time; the submissions might be temporarily denied; the workers might be rescheduled or terminated. On the other hand, if a master has run out of tasks to execute, then it would be wasteful to reserve workers that would sit idle.

To serve this need, we provide the *Worker Pool* tool, which interfaces with multiple resource management systems to maintain an appropriate number of workers over an extended time period. The Worker Pool queries the catalog for currently running masters, sums the current need for workers and then interfaces with the desired system to submit or remove workers as needed. This is done with a degree of hysteresis to prevent resonance, and within some hard limits set by the user to avoid overloading the system.

Load Management

At very large scales, there is always the danger of having *too many* workers for a given master to handle. This could be due to a simple logical mismatch: a master with 1,000 tasks simply does not have enough tasks to keep 2,000 workers busy. Or it could be due to a performance mismatch: a master with 10,000 tasks might not be able to keep 1,000 workers busy if the IO-CPU ratio of the tasks is too high for the underlying network to support.

To address this problem, the master explicitly tracks its capacity to support workers. The *logical capacity* is simply the number of tasks currently in the master's queue. The *performance capacity* is the number of workers that can be kept busy when the network is completely utilized. (This is computed by tracking the CPU-IO ratio of recently executed tasks.) Both capacity numbers are reported to the catalog, in order to inform the Worker Pool how many workers are actually needed.

In addition, if the number of workers actually connected to a master exceeds the logical or performance capacity, the master releases the workers, so that they may look for another master to serve.

IV. Experience with AWE and Work Queue

We ran multiple AWE applications on the scale of several thousand cores for several days at a time. In this section, we will share some of experience in running applications at large scale, demonstrating the fault tolerance, dynamic reconfiguration, and attainable throughput of the system.

As an example, we share a timeline of three AWE applications started at different points over a three day period. All three masters report to the same catalog server and share the

available workers as their needs change. Over the course of the three days, we ran workers on a variety of computing resources available to us, each with varying availability. Each of the following resources was managed by a single *Worker Pool*:

Notre Dame HPC: A 6000-core HPC cluster shared among many campus users, managed by the Sun Grid Engine [12] batch system. **Notre Dame Condor:** A 1200 core campus-scale Condor pool [13] that harnesses idle cycles from Notre Dame and also "flocks" with with Condor pools at Purdue University and the University of Wisconsin-Madison. **Stanford ICME:** A dedicated cluster at Stanford University, consisting of about 200 CPUs and 100 NVIDIA C2070 GPUs. **Amazon EC2:** Standard virtual machines from the Amazon commercial cloud. **Microsoft Azure:** Virtual machines obtained through the Azure "worker role" running the Work Queue software built via the Cygwin compatibility layer.

Fig. 2 shows a timeline of the run. Each of the three graphs shows the number of workers connected to each master. We began by starting Master 1, then starting the Worker Pool for ND-HPC and ND-Condor. Master 2 was started at hour 12, and it began using workers that master 1 no longer required. About hour 15, the ICME Worker Pool was launched, and assigned workers to Master 2, which had a greater need than Master 1. Master 3 was started at hour 17 and gained unused workers from the other masters. About hour 45, we started the Worker Pool for Amazon, which contributed to all three masters.

After running three masters for several days, we removed Masters 2 and 3 and allowed Master 1 to continue running for a week. Fig. 3 displays two of the states that workers can be in when connected to a master. A worker is in the "ready" state when it has successfully connected to a master but has not yet been assigned a task. The "busy" state indicates that a worker has been given a task to complete. Over the course of the week we can observe that the available workers fluctuates between 1700 and 2500. This is expected, since these experiments overlapped with the peak usage times for SGE, ICME, and Condor. We typically observed the highest number of workers during the nights and weekends.

Fig. 4 displays the distribution of task execution times we observed over the run, indicating the non-uniform properties of the combined resources. While there were very few GPU machines available, due to their ability to execute the tasks rapidly they were able to run a large portion of all the tasks. The ND-HPC, Stanford-ICME CPU and Amazon EC2 workers have similar performance. ND-Condor workers displayed a wide distribution of execution times, while the Azure workers were only able to return a very small percentage of the results due to network constraints.

The following observations can be made:

Concurrency changes require worker balancing

As can be seen in both Fig. 2 and 3, each of the applications expresses a very large concurrency initially, which then drops down as the current iteration waits for stragglers to complete. While there are techniques that can be used to reduce stragglers, some level of variability is inevitable. Thus, to maximize overall throughput, it is essential for masters to release unused workers so that they can be applied elsewhere.

Varying resource availability demands fault-tolerance

Our framework demonstrated fault-tolerance with workers becoming unavailable; this did not affect the correctness of the overall calculation or significantly impact performance.

Using whole programs as tasks has acceptable overhead

Each instance of a "task" in WQ requires starting a new instance of the molecular dynamics simulation. This is potentially costly, because the application requires 34 MB of program files (task initialization) and 100 KB of input files (task allocation) for each task. Because WQ caches the needed files at each worker, the larger cost is only paid when a new worker must be initialized. For AWE, we observed that less than 2% of all the tasks sent required task initialization, the remaining 98% where task allocations. Due to the effects of caching, the average communication time observed was 54 millisec (of 617385 completed tasks). Thus, even though the cost for a task initialization is high, the overhead remained small: the total communication time (9.3 hours) amounts to 2.1% of the master wall clock time (433 hours).

Speedup and efficiency are acceptable

After allowing one master to run for several weeks we compute the following. By computing the total CPU time from all the tasks we compute speedup as follows:

$$S = \frac{\sum\limits_{\text{Task}i} T_{\text{CPU},i}}{T_{\text{wall,master}}}$$

This yields a parallel speedup of 831 times over running AWE with a single worker for Fip35. Currently, the main bottleneck is the straggling workers, as the iterations complete the majority of their tasks in 6 to 7 hours, but spend the remaining time waiting for a few hundred tasks to return. This results in an efficiency of 0.5. This is reflected in Fig. 3 as the periodicity exhibited by the connected workers.

Another measure of efficiency is the ratio of wasted to useful work, or the effective efficiency. We define useful work (W_{useful}) as done by the "busy" workers and wasted work (W_{wasted}) as that done by "ready" workers (see Fig. 3; a ready worker is sitting idle waiting for work from the master).

$$E_{\text{effective}} = 1 - \frac{W_{\text{wasted}}}{W_{\text{useful}}}$$

By taking the ratio of the areas under the respective curves our effective efficiency is 0.89.

V. AWE Results

An advantage of the AWE method is that it allows decoupling of the conformation sampling procedure from the kinetics analysis. As part of the preprocessing steps, sampling along transition paths may be obtained using techniques such as replica exchange, high temperature simulations, low viscosity implicit solvent simulations, etc. Once the

conformational space has been partitioned into cells, AWE will compute the reaction rates of the given system. This section presents a benchmark of AWE running with Work Queue on a large biomolecular system as well as presenting two pathways recovered from the resultant network that had been previously posited in the literature.

We applied AWE to the Fip35 WW domain, a fast folding mutant of the WW domain with 33 residues whose native state is formed by three beta sheets separated by two short loops. The experimental folding time of Fip35 is known to be in the low microsecond range and is estimated at 13 μs [14].

Our procedure was as follows: we ran a long simulation (over 200 μs) in which multiple folding and unfolding events are observed, using the Amber 96 force field with Generalized Born implicit solvent, and a time step of 2 fs. This simulation was run on an Nvidia GeForce GTX 480 GPU using Open MM [15]. The viscosity parameter (γ) was set to 1 ps^{-1} to accelerate conformational sampling. Conformations are stored every 1 ns. We then prepared several initial Markov State Models (subsampling the long simulations at 500 ns) using MSM Builder [16], constraining the total number of cells ("micro states" in MSM Builder terminology) to range between 100 and 5000. Comparisons were done using the RMSD between the α and β carbons of the beta sheets, with a cutoff of 3 Å. Furthermore, we forced the native state to be considered as one of these microstates before assigning the remaining conformations.

One property of the MSM is that reaction rates can be predicted by choosing varying lag times between snapshots, and computing the transition matrix. As this matrix is a stochastic matrix, the eigenvalues provide the implied timescales. We selected an MSM with 1000 states as it provided an acceptable tradeoff between separation of timescales (the Markovian property) and presence of noise. The slowest implied timescale or mean passage time computed by the MSM converged to approximately 10μs.

The next step was to run 2 iterations of AWE using these cell definitions as input. One of the important parameters for AWE is the length of the trajectory computed by each walker. If the trajectories are too long, there is a risk of cells getting empty and loss of efficiency. If they are too short, frequent global synchronizations are needed. This trial run was used to establish a trajectory length of 500 ps. Another important parameter is the partitioning of the conformational space, and the definition of the cells. Since the unfolded ensemble of conformations can be extremely large we need to reduce the time AWE spends sampling this region, and enrich the cells in the transition and folded basins. We define conformations with an RMSD to the native structure in the range of [0, 3], (3, 6], (6,+∞] Å to correspond respectively to folded, transition, and unfolded states. Therefore, using this initial MSM, we constructed a new MSM where the number of cells in the unfolded regions is reduced, which is then used for the final runs of AWE. The timescales of this final MSM are comparable to those of the MSM, around 10μs. Note that MSM and AWE are different models and therefore their predictions are not expected to match. Upon convergence AWE is always more accurate.

The final setup for AWE was the following: 601 cells, 20 walkers per cell, walker length of 500 ps using $\gamma = 50$ ps^{-1} with Amber96 and the Generalized Born model of Onufriev, Bashford and Case (OBC). We define the unfolded and folded cells as the "blue" and "red" cores. Walkers not belonging to either of these cores are assigned a random color, otherwise they take the core color. We then ran AWE for 50 iterations and examined the flux between the unfolded and folded regions, as well as the network of transitions between the cells. Fig. 5 displays the folding and unfolding fluxes from AWE.

Additionally, we constructed the interaction network (Fig. 6) based on the transitions between cells as observed during the AWE procedure. We computed the betweenness centrality of all nodes. For node i, it is defined as the number of pairwise shortest paths from all vertices to all others that pass through node i. This gives a measure of how important node i is for connecting different portions of the network. In order to favor more probable paths we reweight the graph using the negative log of the probability of the edge, before applying a weighted shortest paths algorithm. As a result we are able to recover two folding pathways from the network, as shown in Fig. 7. These pathways are distinguished by cells 591 and 404, which respectively show hairpin 2 and 1 forming before the other. Furthermore, these pathways are corroborated by previous work (there is an extensive literature on this topic, see e.g. [17]–[20]), which posits that Loop 1 formation is initiated at near the loop region and "zippers up" via hydrogen bonding, and Loop 2 is formed by a hydrophobic collapse (a transition which may be inferred from the $404 \rightarrow 600$ edge).

VI. Conclusions

We have presented the application to protein folding of Work Queue, a simple yet powerful framework for dynamic distributed applications. An important aspect is that Work Queue can be used with any molecular dynamics (MD) code and only interacts through input and output files. The master script is written in Python such that other algorithms can be easily implemented. Our application bears some resemblance to Copernicus [21]. Compared to the latter, our adaptive sampling method is unbiased (does not require Markovity). In addition, whereas Copernicus relies on custom-made solutions for every aspect of the infrastructure and computation, we rely on Work Queue and its Python bindings and can accommodate any MD code.

VII. Acknowledgements

References

1. Huber GA, Kim S. Weighted-ensemble Brownian dynamics simulations for protein association reactions. Biophys. J. 1996 Jan.70(1):97–110. [PubMed: 8770190]

2. Zhang BW, Jasnow D, Zuckerman DM. Efficient and verified simulation of a path ensemble for conformational change in a united-residue model of calmodulin. Proc. Natl. Acad. Sci. USA. 2007 Nov.104(46):18 043–18 048.

3. Darve, E.; Ryu, E. Innovations in Biomolecular Modeling and Simulations. Royal Society of Chemistry; 2012 May. Computing reaction rates in bio-molecular systems using discrete macro-states.

4. Shaw DE, Deneroff MM, Dror RO, Kuskin JS, Larson RH, Salmon JK, Young C, Batson B, Bowers KJ, Chao JC, Eastwood MP, Gagliardo J, Grossman JP, Ho CR, Ierardi DJ, Kolossvary I, Klepeis JL, Layman T, Mcleavey C, Moraes MA, Mueller R, Priest EC, Shan Y, Spengler J, Theobald M, Towles B, Wang SC. Anton, a special-purpose machine for molecular dynamics simulation. Communications of the ACM. 2008; 51(7):91–97.

5. Bowers, K.; Chow, E.; Xu, H.; Dror, R.; Eastwood, M.; Gregersen, B.; Klepeis, J.; Kolossvary, I.; Moraes, M.; Sacerdoti, F.; Salmon, J.; Shan, Y.; Shaw, D. ACM/IEEE SC 2006 Conference (SC'06). ACM/IEEE SC 2006 Conference (SC'06); IEEE; 2006. p. 43-43.

6. Bui P, Rajan D, Abdul-Wahid B, Izaguirre J, Thain D. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. Workshop on Python for High Performance and Scientific Computing at SC11. 2011

7. Swope WC, Pitera JW, Suits F. Describing Protein Folding Kinetics by Molecular Dynamics Simulations. 1. Theory. J. Phys. Chem. B. 2004 May; 108(21):6571–6581.

8. Chodera JD, Singhal N, Pande VS, Dill KA, Swope WC. Automatic discovery of metastable states for the construction of Markov models of macromolecular conformational dynamics. J. Chem. Phys. 2007; 126(15):155101. [PubMed: 17461665]

9. Pande VS, Beauchamp K, Bowman GR. Everything you wanted to know about Markov State Models but were afraid to ask. Methods. 2010 Sep.52(1):99–105. [PubMed: 20570730]

10. Vanden-Eijnden E, Venturoli M, Ciccotti G, Elber R. On the assumptions underlying milestoning. J. Chem. Phys. 2008; 129(17)

11. Litzkow, M.; Livny, M.; Mutka, M. Condor - a hunter of idle workstations; Eighth International Conference of Distributed Computing Systems; 1988 Jun.

12. Gentzsch, W. Sun Grid Engine: Towards Creating a Compute Power Grid; Proceedings of the 1st International Symposium on Cluster Computing and the Grid; 2001. ser. CCGRID '01

13. Thain, D.; Tannenbaum, T.; Livny, M. Condor and the Grid. In: Berman, F.; Hey, A.; Fox, G., editors. Grid Computing: Making the Global Infrastructure a Reality. John Wiley; 2003.

14. Liu F, Du D, Fuller Aa, Davoren JE, Wipf P, Kelly JW, Gruebele M. An experimental survey of the transition between two-state and downhill protein folding scenarios. Proc. Natl. Acad. Sci. USA. 2008; 105(7):2369–2374. [PubMed: 18268349]

15. Friedrichs MS, Eastman P, Vaidyanathan V, Houston M, Legrand S, Beberg AL, Ensign DL, Bruns CM, Pande VS. Accelerating molecular dynamic simulation on graphics processing units. J. Comp. Chem. 2009; 30(6):864–872. [PubMed: 19191337]

16. Bowman GR, Huang X, Pande VS. Using generalized ensemble simulations and Markov state models to identify conformational states. Methods. 2009 Oct.49(2):197–201. [PubMed: 19410002]

17. Zheng W, Qi B, Rohrdanz MA, Caflisch A, Dinner AR, Clementi C. Delineation of folding pathways of a β-sheet miniprotein. J. Phys. Chem. B. 2011 Nov.115(44):13 065–13 074.

18. Shaw DE, Maragakis P, Lindorff-Larsen K, Piana S, Dror RO, Eastwood MP, Bank JA, Jumper JM, Salmon JK, Shan Y, Wriggers W. Atomic-Level Characterization of the Structural Dynamics of Proteins. Science. 2010; 330(6002):341–346. [PubMed: 20947758]

19. Noe F, Schutte C, Vanden-Eijnden E, Reich L, Weikl TR. Constructing the equilibrium ensemble of folding pathways from short off-equilibrium simulations. Proc. Natl. Acad. Sci. USA. 2009; 106(45):19 011–19 016.

20. Pande V. Meeting halfway on the bridge between protein folding theory and experiment. Proc. Natl. Acad. Sci. USA. 2003 Mar.100(7):3555–3556. [PubMed: 12657736]

21. Pronk S, Bowman G, Beauchamp K, Hess B, Kasson P, Lindahl E. Copernicus : A new paradigm for parallel adaptive molecular dynamics. Supercomputing. 2011
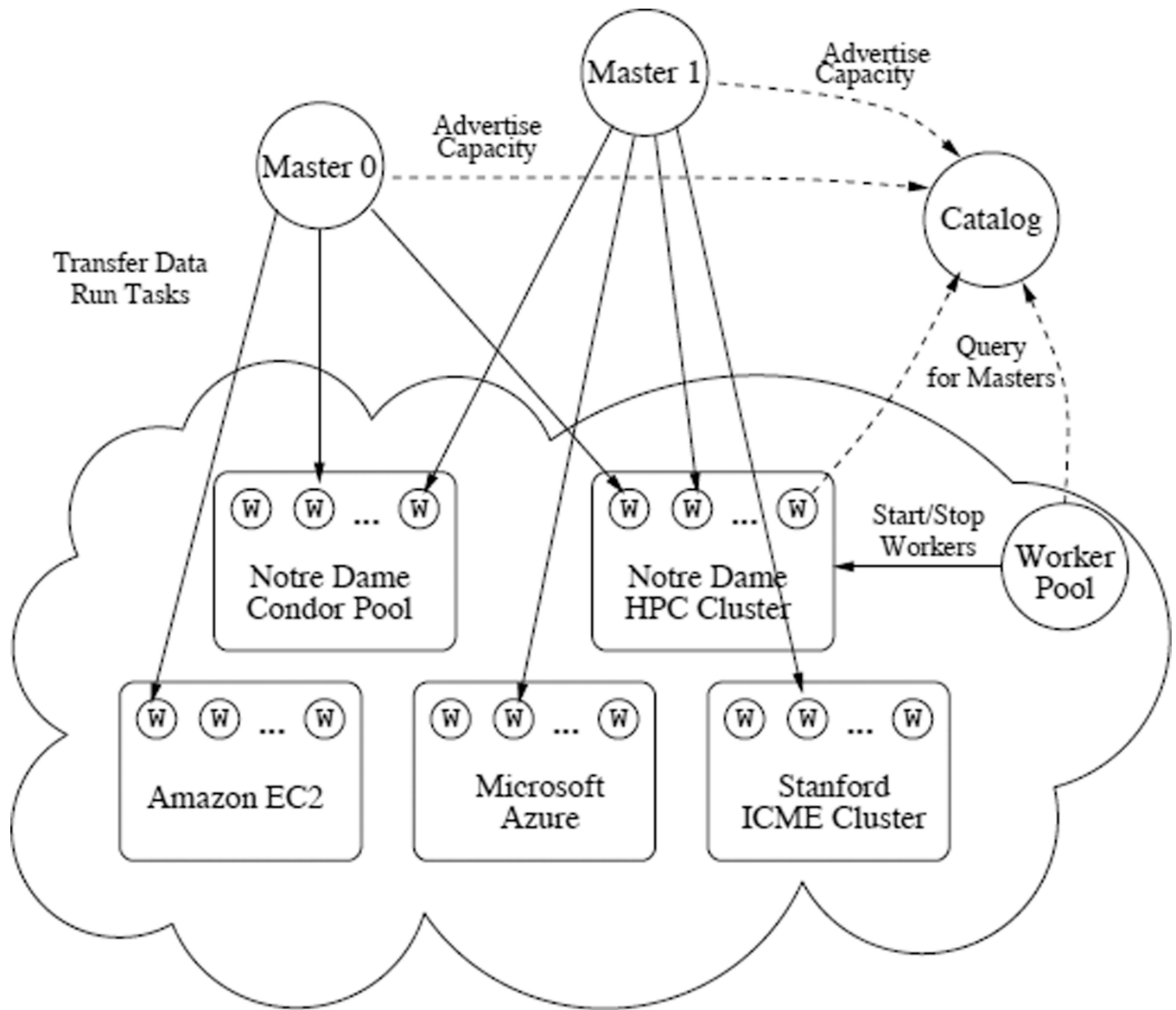
Fig. 1.
An overview of the Work Queue framework. Master programs coordinate tasks and data on Workers running on multiple resources. The Catalog tracks running masters and makes this information available to Worker Pools, which maintain an appropriate number of workers on each resource.
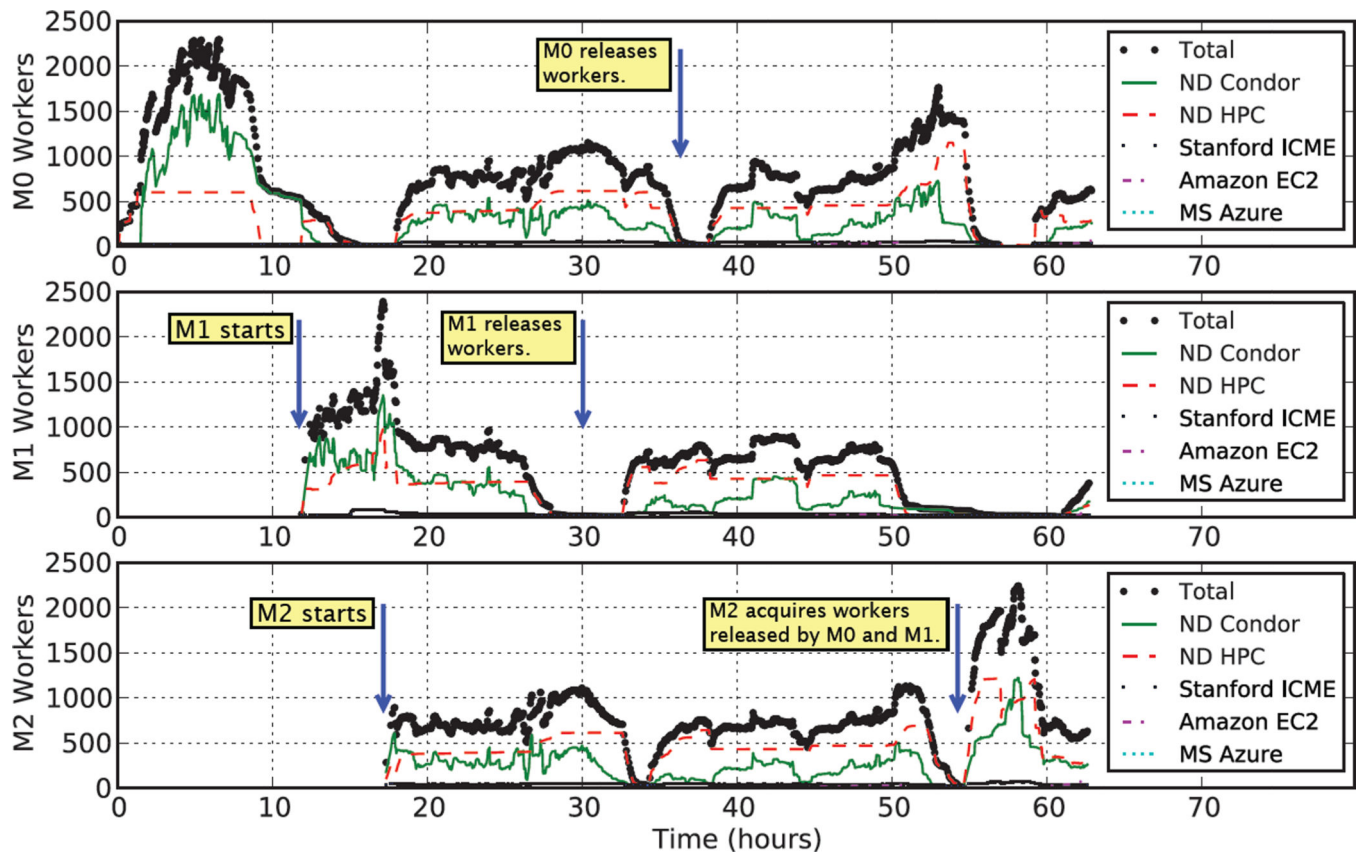
Fig. 2.

A timeline of the number of workers working with three masters, showing the resources from which the workers came. Progression is as follows: Master 0 (M0) is started and run alone for 11 hours. M1 is then started. Since M0 has entered the straggling phase, M1 begins to pick up the workers that are released by M0. When M2 is started at hour 18, the pool allocates new workers and migrates workers from M0 and M1 to M2. Hours 30, 38, and 55 show similar behavior: as one master enters a slow phase, others are able to use the available resources. The masters report their resource needs and usage to the catalogue server. These data are then used by the pools to allocate and migrate workers. In practice, this worker migration scheme is not completely visible in the plot due to several factors: the total number of workers is not constant and the time to start workers can change due to environmental factors (such as job allocation policies, network issues, resource usage by other users). Nevertheless, the utility of the pool is evident: once started, the pools managed the workers without any user intervention.
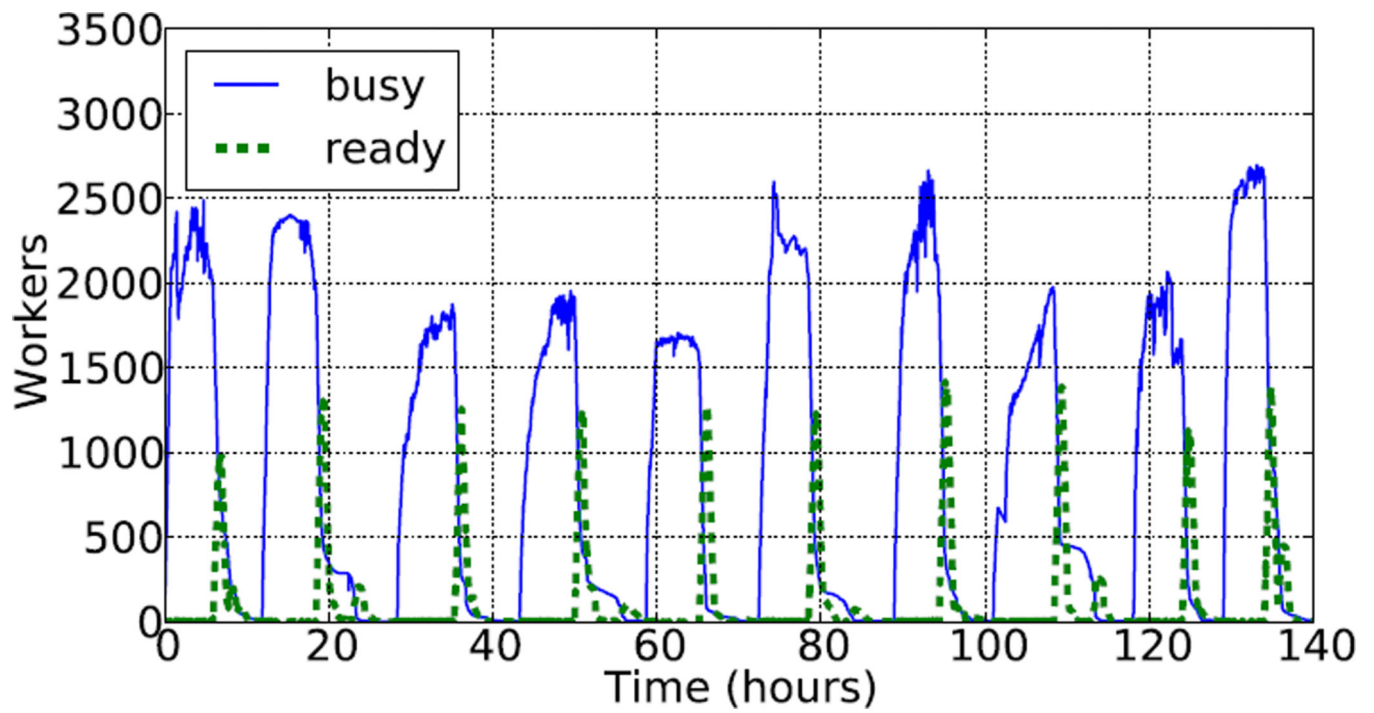
Fig. 3.
A snapshot of the number of workers busy with or ready to be assigned a task over the course of one week. The periodicity is due to stragglers that must be completed before the next iteration can begin. The areas under the "busy" and "ready" curves reflect the useful and wasted resource usage.
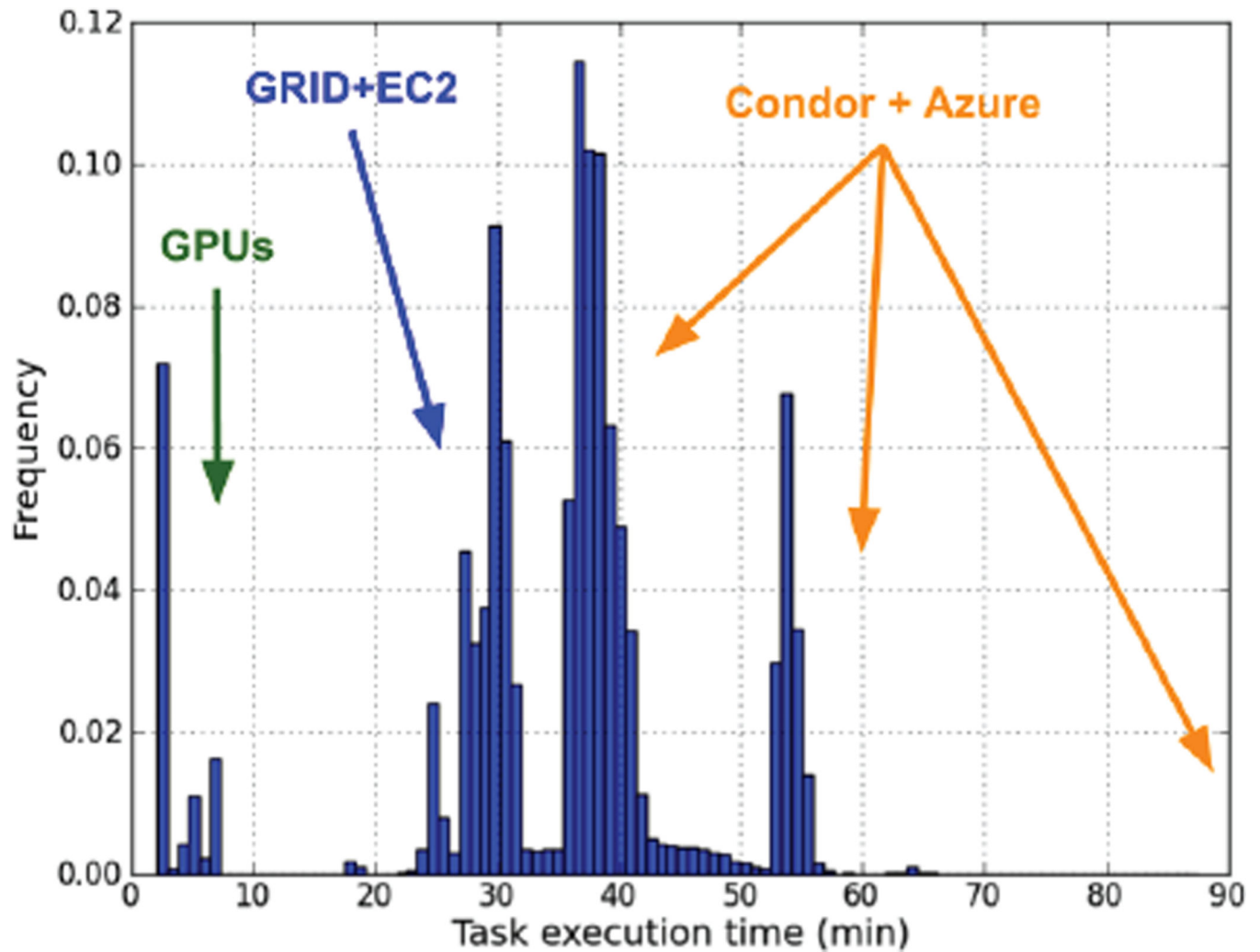
Fig. 4.
Distribution of task execution times. The fastest tasks were completed on the GPU while the middling times fell mainly to the grid and EC2 machines. The machines available through Condor displayed a wide range of performance, while the Windows Azure instances were the least performant for our application.
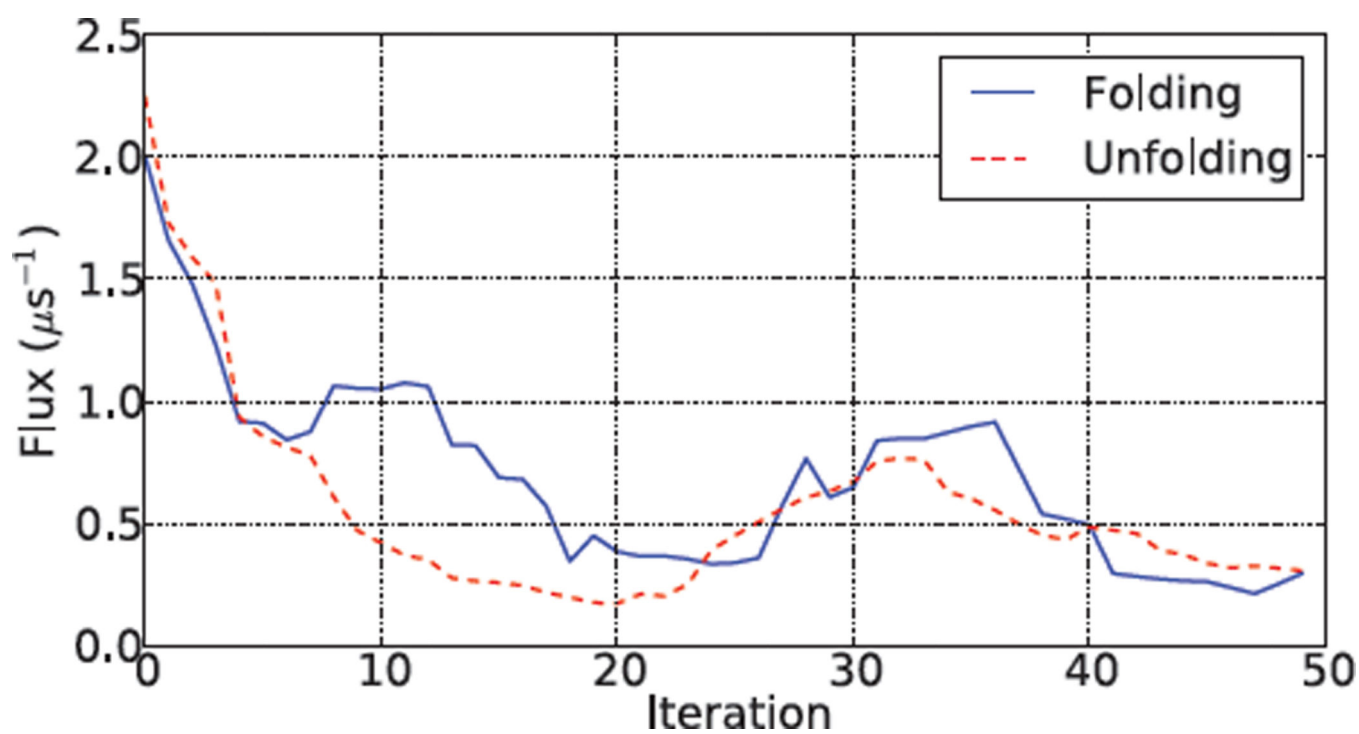
Fig. 5.
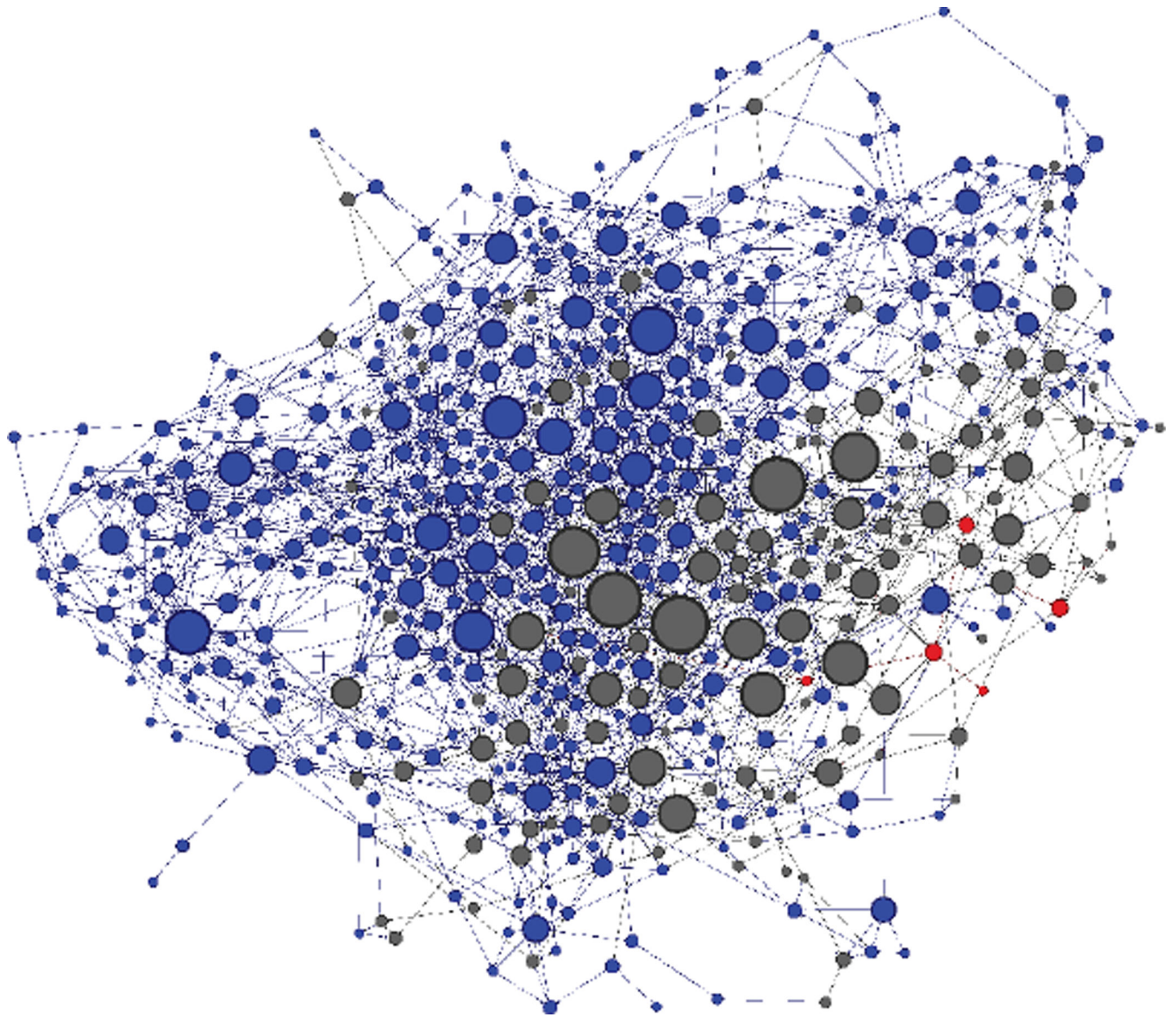Folding and unfolding flux for Fip35 for each AWE iteration.

Fig. 6.
Network of cell connections after running AWE for Fip35. Blue, grey, and red nodes
represent the unfolded, transition, and folded cells. Node size indicates betweenness
centrality. The unfolded regions is large, with many transitions between unfolded
conformations. In order to fold, blue states must pass through the transition states (grey)
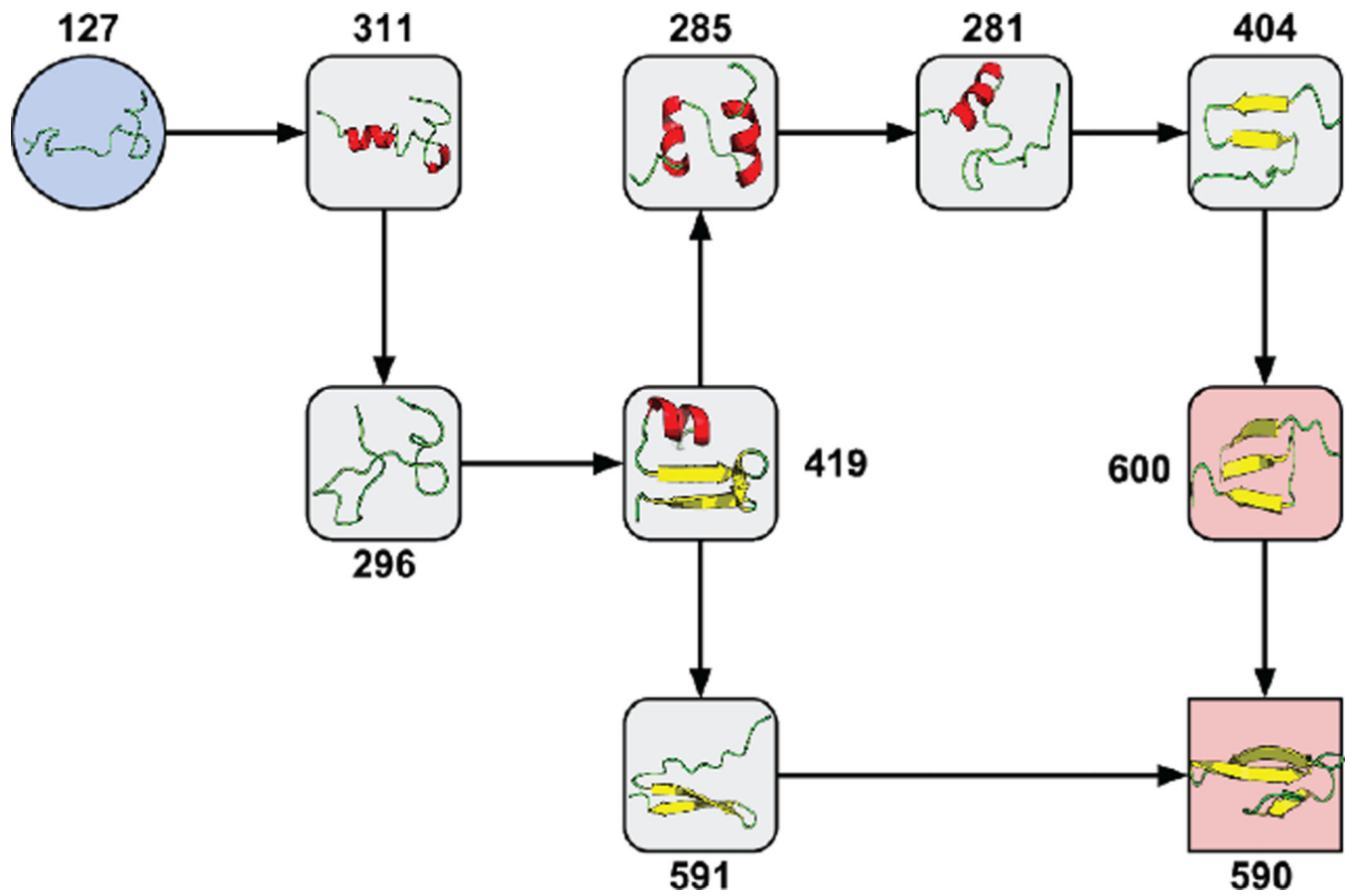before attaining the folded (red) conformations.

Fig. 7.
Two Fip35 folding pathways as found from the AWE network. Colors blue, grey, and red represent "unfolded", "intermediate", and "folded" conformations, respectively. These paths are corroborated by literature studies of the WW domain that indicates pathways exist in which Loop 1 forms first, and vice versa. The mechanism of these pathways indicate that Loop 1 formation may be initiated near the loop region and "zip" up via hydrogen bonding to form the beta sheet, while formation of Loop 2 can be seen as a "hydrophobic collapse." Cells 404 and 591 correspond to these Hydrophobic Collapse and Zipper intermediates in the folding pathways, respectively.
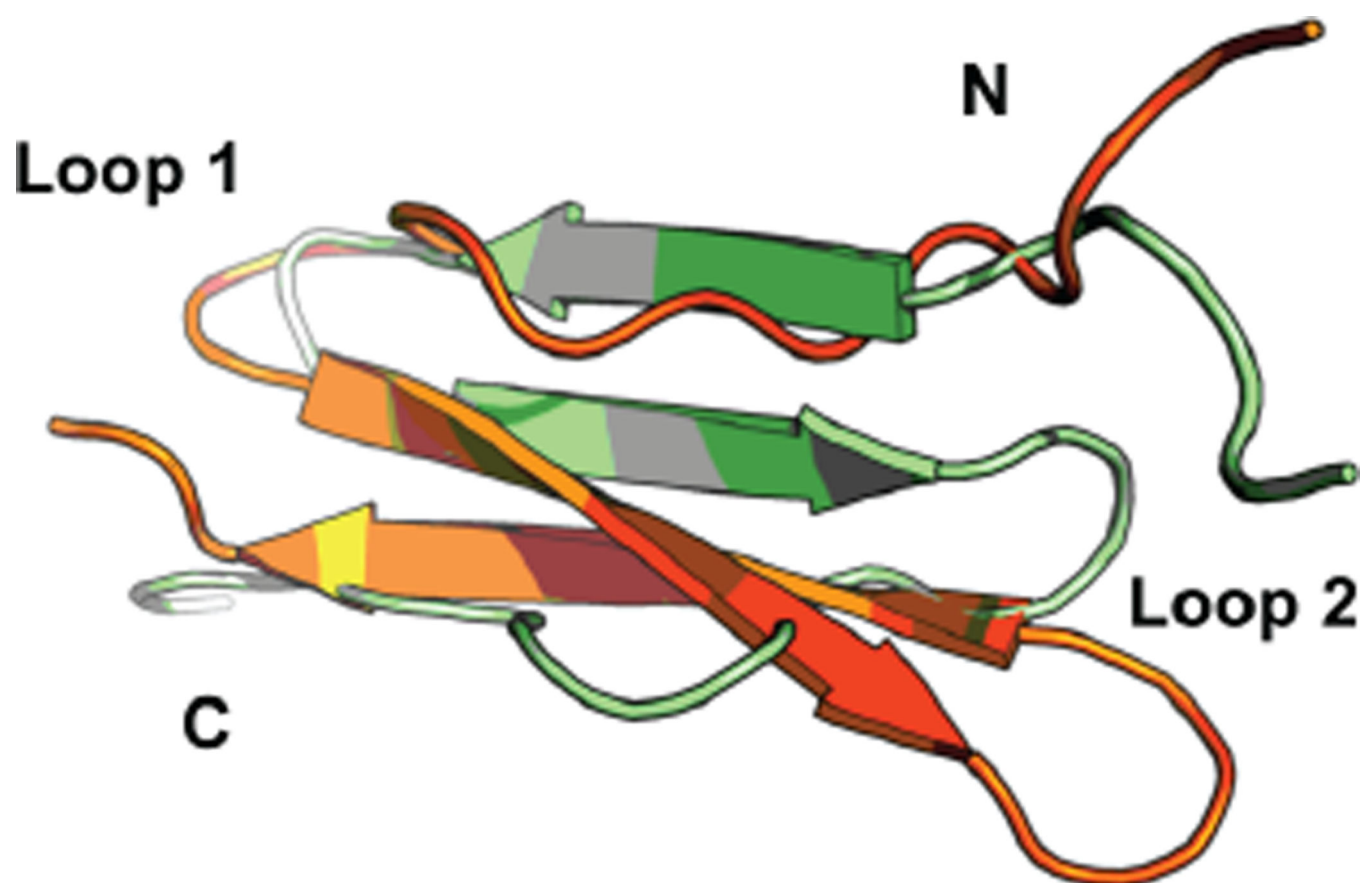
Fig. 8.
The Zipper and Collapse pathway intermediates from AWE (as mentioned in Fig. 7) shown in orange (cell 591) and green (cell 404) respectively. Cell 591 formed Loop 2 first, and Loop 1 is in the process of forming (see the kink at Loop 1). The angle suggests a "zipper"-like mechanism as previously mentioned. Cell 404 has Loop 1 already formed, with Loop 2 in the process of "collapsing" as the third beta-sheet forms.