Scaling Work Queue for the Cloud with Hierarchy

Michael Albrecht, Dinesh Rajan, Douglas Thain

UNIVERSITY OF NOTRE DAME

Cooperative Computing Lab
http://nd.edu/~ccl/software/workqueue

Conventional wisdom suggests that a hierarchical architecture for computational middleware is inherently more scalable than a centralized system. When properly implemented, hierarchy can allow some systems to scale well beyond what would be possible with a flat organizational structure. However, implementing hierarchy within a system often imposes an additional overhead cost, and to our knowledge there has been no comprehensive study of under what conditions a hierarchical structure can help. We have attempted to rectify some of that oversight by identifying conditions under which the addition of hierarchy provides more benefit than the loss incurred due to the overhead.

Work Queue

In order to evaluate the benefits of hierarchy we used the Work Queue system, a lightweight master-worker framework consisting of a library and a worker executable. The library is linked into a master program via a C, Python or Perl API. The programmer composes tasks that consist of individual applications, each annotated with the input files that they require and the output files they are expected to produce.

```
while ( not done ) {
  for( each new task ) {
    task = work_queue_task_create( command );
    /* specify files used by task here */
    work_queue_task_submit( queue, task );
  }
  task = work_queue_wait( queue );
  /* process the result of this task */
  work_queue_task_delete(task);
}
```

Figure 1: A typical Work Queue application

The workers are then run by hand or submitted as jobs to whatever batch, grid or cloud system is available. Workers connect to the master process via TCP and receive and execute tasks. We've extended Work Queue to allow for intermediate sub-master or 'foreman' processes to test the effects of hierarchy.

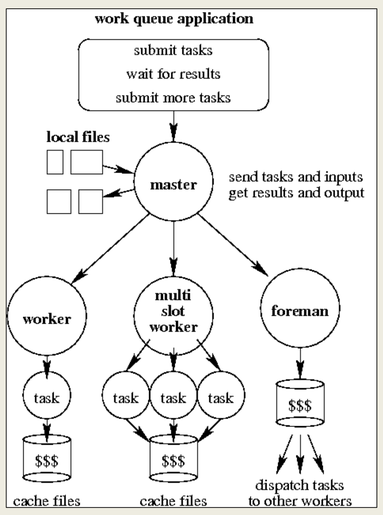
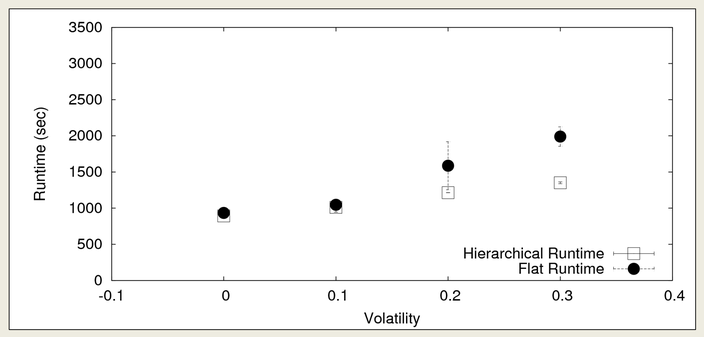
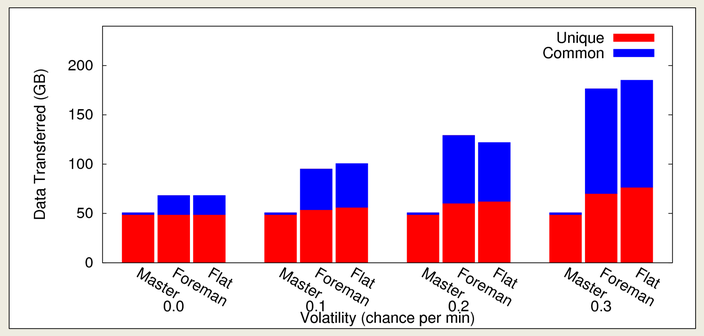


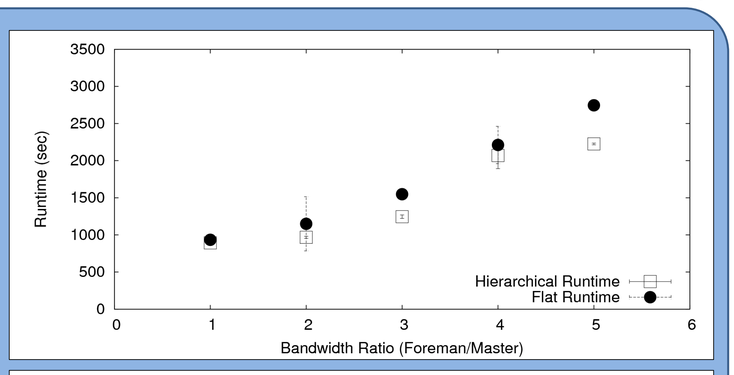
Figure 2: Overall Work Queue Architecture

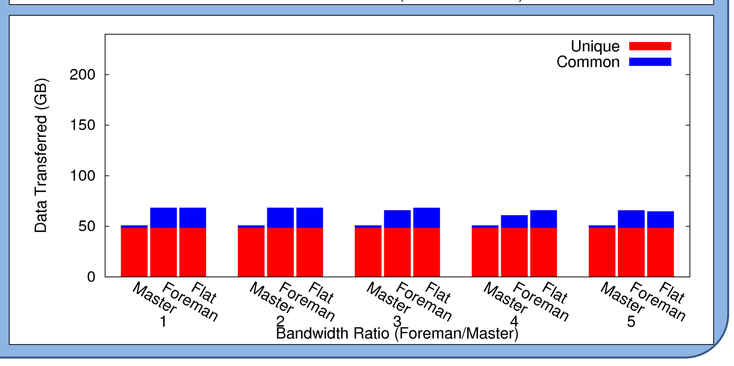
We investigated the behavior of the system with varying levels of **worker volatility**. In master-worker the connection of a new worker means the master must transfer over both the unique data for a task and any common data, increasing that task's data transfer time and reducing the master's task dispatch rate. A hierarchical architecture isolates these disruptions at the foreman instead of impacting the whole system.
For this experiment we varied the volatility of each worker from a 0% chance of disconnection up to a 30% chance of disconnection per minute, and compared the overall runtime of our hierarchical setup to that of a flat architecture. We found that the isolation provided by the foremen allow us to **reduce runtime despite occasionally transferring more total data**.



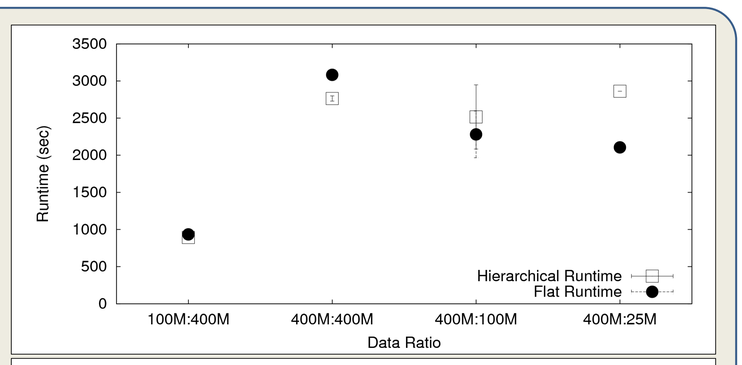


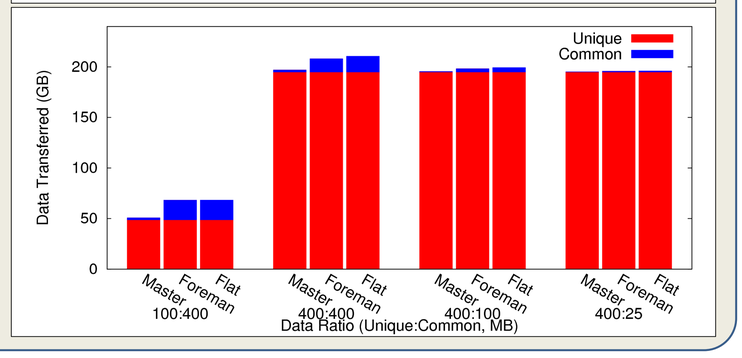
We looked at what happened when there are **bandwidth** constraints between the master and the foremen/workers. It is common in distributed systems to have a fast connection available within a cluster of workers but a slow connection from that cluster to the master. A hierarchy can help by reducing the number of times common data gets transferred over the slower link.
We ran a set of experiments with different ratios of bandwidth across the two types of link. We compared the runtime of our hierarchical architecture to that of a flat setup, and **found a moderate improvement in performance** despite having **no change in the amount of data transferred.**



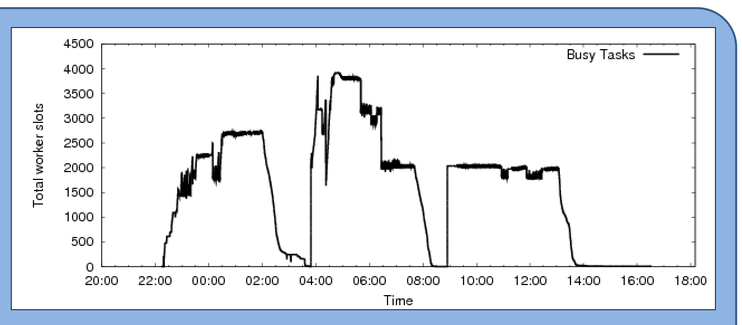


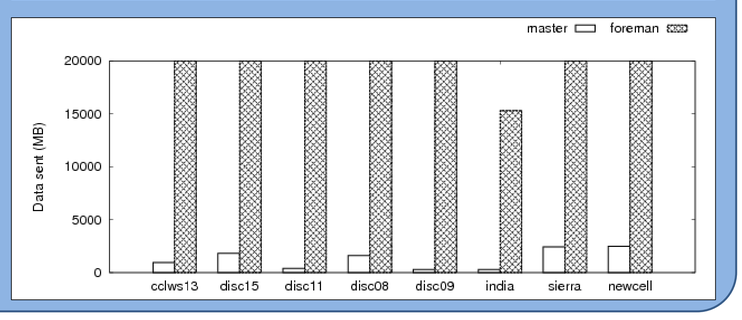
We also looked at the impact of the **ratio of common data to unique data** necessary for each task. When common data is large the startup phase of the application benefits from parallel data transfer by the foremen.
On the other hand, when the common data is small compared to the unique data, the foremen add an extra transfer step to every data transfer. This increases latency without benefiting from parallelism. In the absence of volatile workers or major communications bottlenecks we **saw a substantial decrease in performance for the hierarchical system** as the **ratio tilted in favor of unique over common data.**





Accelerated Weighted Ensemble or AWE is a method for enhancing the sampling accuracy of the molecular dynamics simulations of protein systems. It partitions the conformational space of a protein into cells and creates a fixed number of simulation tasks or "walkers" in each cell. Each walker simulation can then be run in short time intervals as an independent task, the results of which are recycled into providing starting data for the next set of walkers.
AWE's profile (small unique data, large common data, many tasks) and ability to scale almost arbitrarily large provided an excellent platform to illustrate the benefits of adding hierarchy.
We were able to run an AWE experiment consisting of **more than 33,500 tasks** on up to **3500 cores** simultaneously, across **3 different distributed systems**, with just **10 GB** transmitted by the master to the foremen and **875 GB** transmitted by the foremen to the workers.





This work was supported in part by the Department of Energy and the National Science Foundation via grants OCI-1148330 and CBET-0941565