# Multiscale Scheduling: Integrating Competitive and Cooperative Scheduling in Theory and in Practice

G. E. Blelloch, L. Blum, M. Harchol-Balter, R. Harper
Computer Science Department, Carnegie Mellon University

## Overview

A chief characteristic of next-generation computing systems is the prevalence of parallelism at multiple levels of granularity. From the instruction level to the chip level to server level to the grid level, parallelism is the dominant method of improving performance relative to cost. While the characteristics of the fabric such as the granularity or the interconnect differ at each level, the common theme is parallel computing. Building applications that take full advantage of parallelism remains a significant challenge, even when exclusive access to the computing fabric is assumed.

But a chief characteristic of next-generation computing is simultaneous access to shared computing resources by millions of users. For example, an Internet search engine must serve multiple simultaneous queries, each of which is decomposed into multiple parallel tasks whose results are combined to produce a response to a query. The need for parallelism at the level of the individual query arises not only to improve response time, but also to permit decomposition of the search space into fragments that can be managed by individual processors. With the emergence of increasingly sophisticated Web services, one can readily envision greater demand for multiscale parallelism. For example, electronic commerce applications may service many purchase transactions simultaneously, each of which can be broken up into sub-transactions that can be executed in parallel. Or one may envision computationally-intensive services such as a biological database that can be searched for sequence or expression patterns homologous to some user input, or a biological simulation service that performs protein folding simulations on demand.

Our view is that the computing systems of the future exhibit parallelism multiple levels of granularity, and that these resources must be shared among many users simultaneously. These combination of these two aspects, which we call *multiscale parallelism* (see Figure 1), poses significant challenges for implementing next-generation systems. Conventional parallel programming decomposes a problem into tasks that are executed *cooperatively* on a shared computing fabric. Conventional job scheduling considers jobs to be atomic units of work that are to be executed *competitively* on a shared fabric. Multiscale parallelism integrates the competitive with the cooperative aspects of multiscale parallelism at all levels of granularity.

Our contention is that *scheduling* is a major component of any successful approach to managing multiscale parallelism. Broadly speaking, by scheduling we mean the placement over time of components of a computation on processing elements. Scheduling is an area that has been studied extensively in many different communities, including the real-time community [28], the supercomputing community [31], the parallel algorithms community [25, 12], the operating systems community, the measurement and modeling community [38], and the job scheduling community [29]. This work spans from very theoretical to very applied. Unfortunately these communities have mostly worked independently with somewhat different concerns, and with little coordination. We believe, however, any successful long-term solution to multiscale parallelism will require strong ties among the different forms and levels of scheduling. These ties will greatly improve performance, flexibility, reliability and usability. We propose a new concept, called *multiscale scheduling*, that synthesizes and generalizes existing scheduling techniques.

To motivate the concept of multiscale scheduling, and to establish terminology, let us first review the state of the art in cooperative and competitive parallel computing. At the highest

level, we envision a model in which the goal is to process a collection of *jobs*, which are complete, independent units of work. Conventional *competitive parallelism* is concerned with processing these jobs by assigning them to one of many possible servers for execution (see Figure 2). It is assumed that there are many jobs to be executed on relatively few servers shared among the jobs, and that each job is assigned to a single server. The *job assignment policy* determines how jobs are assigned to servers. The goal is to minimize the expected response time (ERT) of the jobs; different assignment policies can lead to dramatically different performance [34, 36].

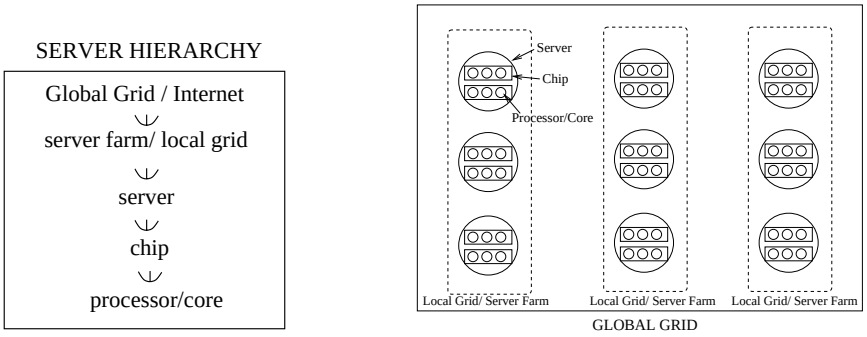
Conventional *cooperative parallelism* focuses on the decomposition of a single job into a collection of *tasks*, all of which must be completed in order to complete the overall job. The execution of the tasks is constrained by a (dynamically determined) *dependency graph*, which specifies the opportunities for simultaneous execution of the tasks (see Figure 3) and possibly also data dependences with communication requirements. It is assumed that there are many more tasks than processors. The overall goal of the scheduler is to map tasks to processors so that dependencies in the graph are not violated and execution time and/or space is minimized. The particular schedule can have a large effect on execution time, as well as on space usage and memory traffic. There are many examples where the scheduler can change the memory usage exponentially [12], or can have catastrophic effects on communication costs [1]. The scheduler assumes exclusive access to the computing fabric in order to achieve its goals.

We envision a synthesis of these two models, as illustrated in Figure 4. Jobs are decomposed into tasks that are scheduled onto a shared computing fabric, consistently with the dependencies among them, but independently of the jobs from which they arose. As in the cooperative parallelism model, each job is not considered complete until its constituent tasks are all complete, and the goal is to minimize job completion time by maximizing the parallel execution of tasks. As in the competitive model, the overall goal of the scheduler is to minimize the expected response time of the high-level *jobs*, not just the tasks that comprise them. The added ingredient here is that tasks are not complete units of work, but rather parts of a larger whole, the overall job whose response time is to be minimized.

In this scenario it is not enough to simply minimize the ERT of the individual tasks independently from their relationship to high-level jobs. For even if we minimize the ERT of the tasks considered independently, we may even maximize the ERT of the overall jobs! For example, if jobs A and B are each broken into two independent tasks, then a schedule in which the tasks from A and the tasks from B are fully interleaved results in a larger mean response time for the two jobs as compared to a schedule in which both of A's tasks are run before both of B's. If a job scheduler is to minimize average job response time, then it must take account of the *cohesion* among the tasks comprising a job in order to arrive at the latter schedule.

When viewed from the perspective of cooperative parallelism, a task scheduler can no longer assume exclusive access to a slice of the parallel computing fabric. The scheduler for the tasks of job A must take account of the possibility that the processing elements are in use by job B, and *vice versa*; it cannot simply assume that there is a fixed number of processors, all of which are devoted to a single job. Thus a cooperative task scheduler must be cognizant of the job level. Otherwise, the task scheduler for job A may assign tasks to processors that are in use for long-running tasks of job B, resulting in poor completion time compared to a schedule that takes account of the overall workload when assigning tasks to processors.

The idea of multiscale scheduling, then, is to *integrate cooperative and competitive scheduling methods into a unified framework* that takes account of both levels to minimize ERT of competitively scheduled jobs while permitting their decomposition into cooperatively scheduled tasks. This unified framework exposes many interesting questions in regards to the effectiveness of vari-




(a) Component Hierarchy (b) System Hierarchy

Figure 1: Multilevel Parallelism

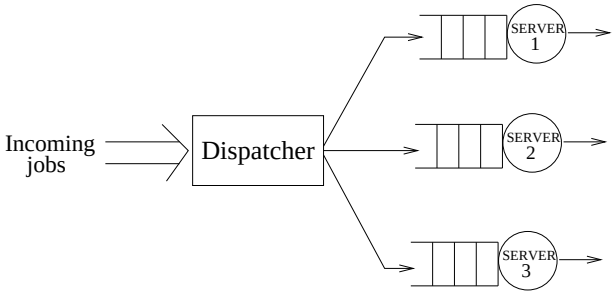



Figure 2: Job Assignment Model

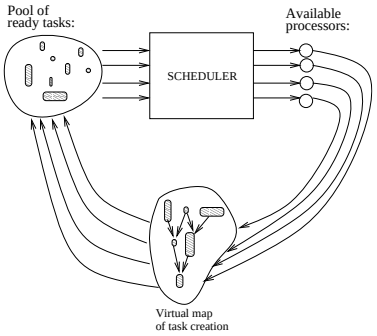



Figure 3: Parallel Computing Model

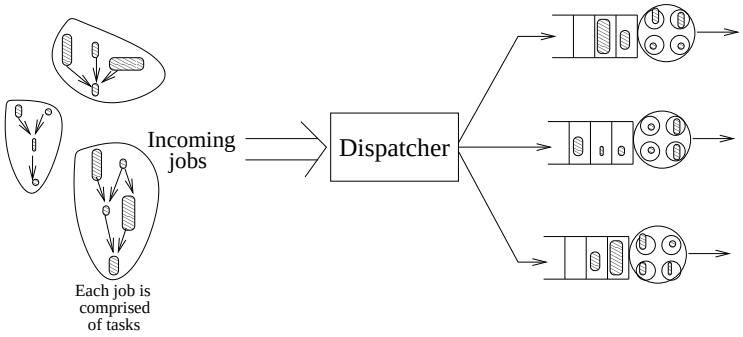



Figure 4: Multiscale Model

ous scheduling policies under various conditions and criteria. In addition the simple scenario just sketched can then be generalized in a number of directions. For example,

- One may envision assigning priorities to jobs based on some extrinsic considerations (such as payment for improved service) that influence the expected response time in proportion to the priority. How can the multiscale scheduler take account of job priorities, while remaining cognizant of the task structure?
- One may also envision assigning other characteristics to jobs, such as deadline requirements, or affinities for certain resources, that influence the job schedule. How can we take account of these in a multilevel environment?
- Or one may wish to maximize other properties, such as the utilization of the processing elements, by permitting jobs or tasks to be re-assigned from one processor to another by the scheduler. How can this be done in a multilevel environment?
- Or, in another direction, one may envision a further decomposition of tasks into sub-tasks, in rough correspondence to the availability of parallelism at the hardware level. For example, one might consider a global grid-level application whose jobs are decomposed into tasks for a local grid, which themselves are decomposed into sub-tasks for each server, and into sub-sub-tasks for each processor, and so forth.

We believe that the theoretical issues of multiscale scheduling cannot be properly addressed without carefully considering how these issues will work with particular applications and how they coordinate with the programming languages used to express the parallelism. We therefore have PIs in these areas and plan to study application and language issues as they relate to multiscale scheduling.

**Applications.** The multiscale scheduling model opens up new possibilities for building applications that take advantage of multiscale parallelism. One class of applications, such as the biological self-assembly simulator described below, seeks to exploit parallelism to enable faithful simulations of complex biochemical processes. One possible strategy for building such an application is to divide the workload into *speculative* and *non-speculative* tasks, where non-speculative tasks must be completed as part of the overall computation, while speculative tasks explore possible future computations that may or may not be needed. We may consider a speculative task to be of lower priority than the non-speculative tasks, weighted by an estimate of the probability it will need to be executed. The speculative computation is thus carried out only on an as-available basis, avoiding preemption of resources needed by non-speculative tasks. We propose to validate our research by considering this problem in detail. It affords ample opportunities for parallel computing at various levels of granularity and will therefore provide a useful testbed for our ideas.

Another class of applications is to electronic commerce. Consider a Web-based vendor that sells both expensive and inexpensive commodities in very high volume. The vendor might wish to assign priorities to purchase transactions in proportion to their value, allowing purchasers of more expensive commodities to jump the queue ahead of those buying less expensive commodities. To achieve this using current technology, the vendor is required to treat each purchase as an atomic transaction so as to ensure that prioritized scheduling provides better response to more valuable purchases. However, purchasing transactions are often naturally decomposed into many sub-transactions such as gaining approval from a credit agency, receiving payment authorization, removing the purchased goods from the inventory, confirming the purchase, and arranging shipping. Multilevel scheduling permits the purchase transaction to be decomposed into a collection

of such sub-transactions that can be scheduled independently on multiple servers, while ensuring that expensive purchases are processed more quickly, on average, than inexpensive ones.

One application of multiscale scheduling is to grid computing. There has been significant work on batch scheduling on parallel machines [31]. The model in this context is that a user submits a job which specifies a number of processors and a maximum running time. When scheduled, the user's job receives the specified number of processors as a group and keeps an equal number of processors for the duration of the job. In this framework it is up to the user to determine how to schedule tasks within their processors. Grid scheduling as realized, for example, in the Globus Toolkit [32], builds on batch scheduling to provide grid-level services for applications, including finding and allocating resources from various administrative domains. These approaches differ from our work in that they do not attempt to integrate job scheduling with task scheduling; it is entirely up to the application to manage the resources assigned to it by the scheduler. Moreover, these frameworks are intended to work with legacy applications. Our work, by contrast, seeks to develop a theoretical framework for integrating competitive and cooperative parallelism, without necessarily accommodating legacy software systems.

A more closely related application of multiscale scheduling is to Harper's previous work on the ConCert trustless grid computing framework [27]. The ConCert Project designed, implemented, and deployed a general framework for deploying applications on a volunteer network of personal computers. This framework included a functional programming language for programming grid applications, a peer-to-peer network infrastructure for dynamically configuring the grid, a type-based code certification infrastructure for ensuring safe execution of foreign code, a distributed hash table mechanism for propagating results on the grid, and a work-stealing task scheduler derived from Leiserson's Cilk system [24]. Functional programming is particularly natural for grid computing, because it avoids the need for shared state among tasks on the grid. This greatly simplifies handling of failures (a common occurrence on a large grid), and avoids the need for transactions to ensure atomicity. However, the simple scheduling algorithms used in ConCert do not scale to account for multiple jobs on the grid, nor for multiscale parallelism of the kind considered here. We propose to extend the ConCert framework with multiscale scheduling, and to use this as a platform for experimenting with applications.

**Programming Languages.** Building applications that exploit parallelism using conventional programming techniques is notoriously difficult and error-prone (see, for example, Edward Lee's recent critique [41]). Based on our earlier experience with designing programming languages for parallel computing [9] and building an infrastructure for grid computing [27], we propose to investigate the design of programming languages suitable for multiscale parallel computing. The NESL language is based on the data parallel model in which parallelism arises from purely functional computation on data structures such as tuples and vectors. This provides an elegant model of parallel programming in which the compiler takes care of the decomposition of a program into tasks that may be scheduled in parallel. This simple model may be extended with concepts such as speculative parallelism that mesh well with multilevel scheduling.

**Contributions.** In Figure 5 we summarize the overall contributions of the proposed research. In short we propose a clean-slate theoretical analysis of multiscale parallelism for next-generation computer systems and applications. This includes the design and analysis of new multiscale scheduling algorithms using methods from queueing theory and parallel and distributed algorithms, the development of a cost semantics for a multiscale parallel programming language, and the development of a significant application, specifically, a biological self-assembly simulator, to validate the proposed model. We note that this is a relatively exploratory proposal in the sense

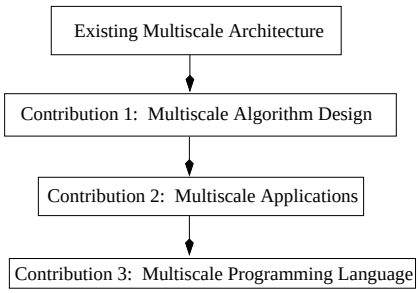



Figure 5: Contributions of the Proposed Research

that it is a new area. As such it is hard to identify up front what the specific outcomes of the work will be (*e.g.*, titles of papers). Two of the PIs have significant experience in two very different areas of scheduling: cooperative scheduling [13, 12, 42, 1, 11], and competitive scheduling [37, 3, 4, 34, 38, 35]. In both cases the PIs have worked both on theoretical and applied aspects. Two of the PIs also have significant experience on programming language issues. As such we believe we have an excellent team to study the issues at hand.

**Outline of Research Plan.** Over the first year we plan to work mostly on theoretical issues in developing a model that spans both competitive and cooperative scheduling. Specific issues we plan to study include:

- How the scheduler can affect overall memory usage in an environment with both competitive and cooperative tasks, and where the number of processors assigned to a job can change. This will extend previous work on space-efficient schedulers [25, 42].
- How the scheduler can affect cache and memory locality in an environment with both competitive and cooperative tasks and memory hierarchies with both shared and distributed caches. This will extend previous work on the data locality of various schedules [1, 11].
- How priorities associated with competitive jobs might be used in an environment with varying and controllable number of cooperative tasks within the jobs. Also understating how the job priorities might be combined with task priorities within a cooperative job.
- Theoretical models for the user to understand the performance or cost of their jobs in a shared environment. This will extend previous work [2, 8, 26] on understanding runtime in an environment where the amount of parallelism is dynamic.
- How the job assignment policy (JAP) should be designed when both the overall size of competitive jobs varies significantly, but each job is broken into tasks that can scheduled individually and cooperatively. How will the individual tasks be assigned and interleaved?
- How can the significant recent theory on competitive job assignment be applied in the context of cooperative task assignment. Do ideas like separating tasks by size make sense? This will extend previous work on server job assignment [36, 34].
- Programming language constructs that can be integrated with a multiscale scheduler, possibly allowing prioritization of tasks, and specification of soft real-time constraints, such as deadlines.

We also plan to hold a workshop that brings together researchers from different scheduling communities to better understand common themes and the differing concerns.

Over the following two years we plan to continue the work from year one, and also to follow up on any promising directions derived from the workshop. By the third year we plan to implement prototypes of the approaches we have developed, and implement the Web clustering server and biological self-assembly simulator described later in the proposal.

# Proposed Work

## From Multiple Jobs to Multiple Tasks

The server farm architecture is prevalent throughout computer science. When applications require more processing power, rather than buying a more expensive server, it is common to instead employ a farm of slow servers, which together function as a more powerful server. The server farm architecture has the advantage of price (many slow servers are cheaper than a single expensive server) as well as the advantage of scalability (servers can be added or removed easily, allowing us to scale the combined processing power up or down.

An example of the typical server farm architecture is shown in Figure 2. Every arriving job is fed into a front-end dispatcher (high-speed router), which immediately dispatches the job to one of the (identical) servers in the server farm. The front-end dispatcher (also known as a "load balancer"), may employ any job assignment policy (algorithm) for assigning jobs to hosts. The mean response time performance varies widely depending on the Job Assignment Policy (JAP) used. A common research goal is to discover better JAP's and prove performance bounds on existing JAP's. Real-world examples of server farms include the Cisco Content Switching Module[1] and the IBM Network Dispatcher product[2].

The prior work on JAP analysis and design is vast, particularly in the ACM Sigmetrics community. The "best" JAP depends, interestingly, on the scheduling policy used at the individual servers. If First-come-first-served (FCFS) scheduling is used at the servers, as is common in supercomputing applications where jobs are not time-shared, and are run to completion, then the research of Harchol-Balter has shown that one desires a JAP which splits jobs up by size (service requirement), so that short jobs are sent to one server, medium-length jobs are sent to a second server, long jobs are sent to the third server, etc. [36]. Some particularly counter-intuitive results include the fact that load-UNbalancing is far superior to load-balancing in this setup, and that even without knowing the size of jobs, one can still achieve performance comparable to that obtained when one does know the size of jobs [34, 33]. If, on the other hand, Processor-Sharing (PS) scheduling is employed at the servers, as is common for Web applications where Web jobs are time-shared, the best JAP looks very different. Here the Shortest-Queue JAP is very effective, however its analysis has eluded researchers for decades. Prof. Harchol-Balter is currently working with Prof. Ward Whitt at Columbia University to provide the first analysis of the Shortest-Queue JAP for these models.

All the above work on designing server farms with good JAPs is based on the notion of a "job" as some *indivisible* quantity of work. By "indivisible" we mean that the entire job is run on some one server and the size of that job is the amount of processing required by the server to complete all the work involved in that job.

Now consider the parallel computing community. This community doesn't deal with finding good JAP's. Rather the community asks a different question: Consider a single job, where the

[1] `http://www.cisco.com/go/csm`

[2] `http://www-306.ibm.com/software/webservers/edgeserver/library.html`

job is viewed as comprised of a bunch of tasks, with possible dependencies between the tasks. Some tasks are large and some are small. Given some number of processors, the question is how to decide which tasks are run on which processors, as shown in Figure 3.

Figure 4 represents the combination of the approaches. Our goal is still to minimize mean per-job response time, but now we have the added flexibility of being able to schedule different tasks at different processors on different servers. As we've discussed before, simply minimizing per-task response time *will not* lead to low per-job response time. Therefore we need new ideas. This multi-scale optimization problem has never been explored in the prior literature, to the best of our knowledge. We are not sure what the right answer is, however here are some (often contradictory) principles that we would like to adhere to based on our prior research. First, we'd still like to isolate short *jobs* from long jobs, so as to minimize overall mean per-job response time. On the other hand, we would like to maximally exploit parallelism by allowing some short tasks of the long jobs to run on the servers "reserved" for short jobs only. Second, we'd like to give *every* job, short or long, a chance to complete initial tasks which expose further parallelism that we can exploit. On the other hand, we don't necessarily want to start up too many jobs, because the memory needed to support the many parallel tasks is limited.

To bring together expertise from both the competitive job assignment community and the cooperative task scheduling community, we plan to host workshops which will be organized by Prof. Harchol-Balter (distributed analysis), Prof. Blelloch (parallel analysis) and Prof. Harper (programming methodologies for parallel and distributed).

### From Multiple Tasks to Multiple Jobs

The previous section considered extending ideas from the area of competitive job assignment to include cooperative task scheduling. In this section, we consider extending ideas from task scheduling to job scheduling. Ultimately our goal is to produce a united approach.

There are many models that have been proposed to represent parallel computations, from the Parallel Random Access Machine model to the bulk synchronous model, and from various fixed network topologies, to systolic arrays (see [40, 15]). Unfortunately most of these models assume a fixed set of processors, an assumption that is not well suited for the dynamic multi-job environment we are considering.

An alternative class of models consider a computation somewhat more abstractly in terms of work (total number of operations) and depth (critical path of dependences) without a particular mapping onto processors. The parallel circuit model was an early example of such an approach. It served as a useful theoretical tool that was used, among other purposes, to define and analyze an important complexity class, NC [43, 30]. However circuits have not been successful as a practical model for defining parallel algorithms because their static nature is not suited for describing many types of parallel algorithms. Other classes of models based on work and depth include vector models [7], models based on various programming abstractions [14, 8], and multi-threaded models [25]. At some level all the models based on work and depth view a computation as a DAG, where the nodes represent sequential tasks, and the edges represent dependences between the tasks. The DAG can either be given offline [29], or revealed online [26, 24, 16, 12].

What is common with all these DAG models is that they require some form of scheduling to map a computation onto actual physical hardware. The scheduler must assign tasks to processors with the constraint that a task can only be scheduled at a time after all of its ancestors have completed, but otherwise has a lot of freedom in the order in which tasks are selected. The particular schedule can make a huge difference in performance. Separate theoretical results are therefore given that state the effectiveness of a particular scheduling algorithm in regards to runtime, space

usage or even cache effectiveness. For example, research on work stealing schedulers [26] has shown that a reasonably broad class of multi-threaded computations with $W$ work and $D$ depth, and $S_1$ sequential space can be mapped onto $P$ processors such that it will run in $W/P + O(D)$ time and $O(S_1 P)$ space. Most of the theoretical results presented for various schedulers have been shown to map well into practice.

We believe that DAG models are much better suited than processor-based models for the next generation of parallel computing fabric exactly because they do not assume a fixed number of processors. Instead they allow the tasks to be scheduled onto whatever processors are available at any given time during the computation, and even allow tasks to be easily suspended or moved without locking up the whole computation. Arora, et al [2] did some early work on applying the multi-threaded model onto a machine with multiple jobs. This showed that for a computation with W work and D depth can be mapped onto a machine with P processors using a work stealing scheduler in time $O(W/P_a + DP/P_a)$ where $P_a$ is the average number of processors available. This was an important first step in the theory of multiscale scheduling. We believe there are a huge number of open theoretical problems within the area, many of which are likely to have significant practical impact. We outline a few here.

**Hierarchical Scheduling.** A key part of the proposed work is to study how one might design a unified scheduler that works across multiple scales of parallel architectures, from multiple cores on a chip to multiple servers on a network. As it stands, different scheduling policies work well at different scales. This would indicate the use of hybrid policies. Although a fixed hybrid policies might work well for any particular configuration, we believe that the best way to attack this problem is to rethink how scheduling is organized.

For example it is known that work stealing schedulers work well in regards to cache performance when caches are not shared [23, 1], and that parallel depth-first schedulers work well for shared caches [11]. One might therefore consider a hybrid approach of the two scheduling policies that works across the two levels. Without being careful, however, this line of research could lead to a wide variety of ad. hoc. approaches combining various policies each optimized for different measures and system configurations. This problem is exasperated by the fact that many schedulers are described in terms of the particular implementation, instead of the policy the implementation is implementing (e.g. work stealing schedulers). This makes it hard to merge the policies. When combining the cooperative schedulers with competitive schedulers this will become even more complicated.

To avoid the proliferation of hybrid techniques we plan to study approaches that abstract the overall scheduling framework from the particular policies in use. The idea is to be able to parameterize the framework with specific policies. We imagine in addition to allowing hybrid policies, this framework could allow dynamic policies that change based on the particular work load, or on the hardware that is available. Key to this approach is abstracting away the policy from the particular implementation. Interesting theory would include being able to analyze the competitive ratio of the dynamic policy to the best policy based on full prior knowledge of the demand.

**Prioritized Scheduling.** As mentioned in the introduction, it is important to support the ability to have multiple levels of priority on jobs. This is both for the purpose of separating user classes but also because more effective use might be made of the whole system if certain jobs (e.g. small ones) are cleared earlier. It turns out that some of the existing DAG scheduling techniques are based on maintaining the threads in a priority order [12]. These priorities, for example, are maintained so that certain bounds can be proven on the memory usage or cache performance. It is

natural to believe that the same overall mechanism might be used for both purposes—scheduling of jobs and scheduling of individual tasks. The overall priority could be a combination of the priorities.

The simplest approach to combining the priorities would be to use lexicographical order using the job priority and then the task priority. We believe, however, that this would not be a good policy. It could lead to starvation of low priority jobs, delay completion of jobs that are almost completed, require more memory than other policies, and have bad effects on cache usage. We plan to study how these priorities can be effectively used together, with the expectation of proving various properties of the overall system.

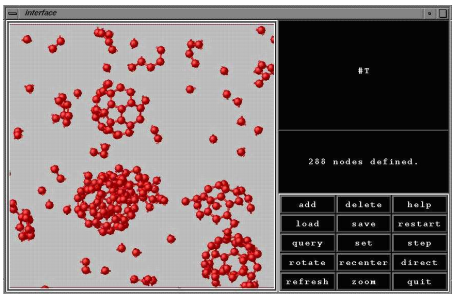
**Language Interface.** We do not believe that multiscale scheduling can be properly addressed without considering programming interface and language issues. To do so would risk generating a framework that has good theoretical characteristics but is not well suited for users. This has been a problem with many parallel computing models in the past. The challenge is presenting a convenient abstraction to the user, while still allowing them to theoretically (and practically) analyze performance characteristics. From a theory point of view the interesting research is in determining abstract cost metrics that can be understood by the programmer and then to map these metrics onto particular machine types with particular schedulers. This idea has been employed in the NESL [8] and CILK [24] programming languages for analyzing runtime as well as space usage, where the abstract metrics are work, depth and sequential space.

A major source of difficulty in implementing parallel algorithms and applications is that conventional programming languages are based on the concept of mutable storage (assignment to variables, modification of shared data structures), which, in the parallel setting, requires complex protocols to ensure coherence and consistency among the various threads that can read and write shared memory. These protocols can, in turn, introduce further complications, such as deadlock, that are not present in the purely sequential setting. Moreover, the mutable storage model requires that threads share state, which is impractical in a competitive setting and places strong demands on the interconnect between processors in a cooperative setting. Furthermore, it is difficult to make such programs tolerant of faults, the normal case at the grid level, where the common case is that servers fail, or otherwise become unavailable or unreachable. We will therefore likely emphasize languages and programming that avoids mutable state, or at least clearly separates it.

## Validation

**A Web Graph Query and Compute Server.** As a test of both the theoretical and practical aspects of multiscale scheduling we plan to develop a web-graph query and compute engine. The goal will be to store a reasonably large sample of the web on a distributed server. The sample will be stored as a graph representing the links between pages, in addition to an index to access pages with specified terms. It will be made available through an interface that allows a variety of queries and computations on the sample. The queries will allow significantly more complicated operations than permitted on a standard search engine. For example a user could select a subset of pages that match on some boolean expression of terms, and then run an SVD calculation on the resulting subgraph to find Hubs and Authorities. Or a user could ask for all the pages within a given distance (in links) from a given page, and cluster the results. The system will need to schedule both competitive tasks from multiple users, as well as collaborative tasks from the parallelism within each calculation.

We believe this will be an interesting test case for several reasons. From a theoretical standpoint there are many interesting parallel algorithms that will need to be designed and analyzed, including graph algorithms, clustering algorithms, searching algorithms, and algorithms for ac-

A 

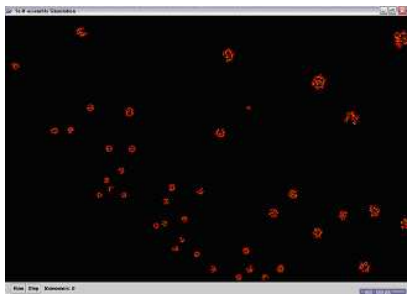
B 

Figure 6: Screen snapshots of a (A) a non-quantitative parallel self-assembly simulator previously developed by Schwartz [44] and (B) the currently serial quantitative discrete event simulator that is the basis of the work proposed in this proposal [48], each modeling virus capsid assembly.

cumulating various forms of statistics. From a scheduling point of view (both theoretical and practical), it will expose a relatively sophisticated use of both competitive and collaborative tasks in a focused domain. From a programming language point of view it will be a test of a relatively broad set of algorithms and applications.

**a Parallel Biological Self-Assembly Simulator** As a second test of the proposed methodology, we will apply it to a real-world problem in biological modeling: discrete event simulation of molecular self-assembly. Russell Schwartz's group will be involved in the project to work on the problem. Schwartz, has extensive experience with the problem domain [47, 44, 5, 45, 46], including past implementation of parallel simulation tools [47, 44].

We propose to implement a parallel variant of a existing continuous-time Markov model for the reaction systems developed by the Schwartz group [39] (Figure 6B). The variant will be designed to exploit parallelism at several scales. At the coarsest scale we will set up the system as a server so that multiple simultaneous experiments can be run in parallel. These might be from one user or from multiple users. These jobs need to be scheduled competitively, and the individual jobs may take very different amounts of time in ways unpredictable to the user. At a finer cooperative scale, each time step of a simulation will be parallelized. The bottleneck in the current serial algorithm is a linear-time operation in which the simulator samples reaction times for all possible single-step reactions involving some assembly. A parallel algorithm could evaluate possible reactions concurrently and since event times would differ the dependences among completions would have to be maintained and scheduled dynamically. This might also involve speculative evaluation of events and roll back if earlier events yield consequences that would have interfered with later events. An even finer scale, each event involves parallelism that can be exploited.

Traditional scheduling techniques consider the competitive and cooperative aspects of the scheduling separately, allocating a fixed number of processors to each job and scheduling them as a batch or gang [31]. Within each job a separate scheduler (*e.g.*, a hand-coded event scheduler) would be used to schedule the individual fine-grained tasks. The main purpose of this test will be integrate the two schedulers and study the advantages and disadvantages.

## References


[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[2] N. S. Arora and C. G. Plaxton R. D. Blumofe. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[3] Nikhil Bansal and Mor Harchol-Balter. Analysis of SRPT scheduling: Investigating unfairness. In *Proceedings of ACM SIGMETRICS*, 2001.

[4] Nikhil Bansal and Mor Harchol-Balter. Approximate analysis of M/G/1/PS and M/G/1/SRPT under transient overload. *Performance Evaluation Review*, 29(3), December 2001.

[5] B. Berger, J. King, R. Schwartz, and P. W. Shor. Local rule mechanism for selecting icosahedral shell geometry. *Discrete Applied Mathematics*, 104:97–111, 2000.

[6] B. Berger, P. W. Shor, L. Tucker-Kellog, and J. King. Local rule-based theory of virus shell assembly. *Proceedings of the National Academy of Sciences USA*, 91:7732–7736, 1994.

[7] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, 1990.

[8] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[9] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[10] G. E. Blelloch, Perry Cheng, and Phillip B. Gibbons. Scalable room synchronizations. *Theory of Computing Systems*, 36(5):397–430, 2003.

[11] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *Proc. 16th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 235–244, June 2004.

[12] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[13] G. E. Blelloch, P. B. Gibbons, Y. Matias, and G. J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 12–23, June 1997.

[14] G. E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the Symposium on Functional Programming and Computer Architecture*, pages 226–237, June 1995.

[15] G. E. Blelloch and B. Maggs. Parallel algorithms. In *Algorithms and Theory of Computation Handbook*. CRC Press, Boca Raton, Florida, 1999.

[16] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. *Theory of Computing Systems*, 32(3):213–239, 1999.

[17] L. Blum. Computing over the Reals: Where Turing meets Newton. *Notices of the American Mathematical Society*, 2004.

[18] L. Blum, F. Cucker, M. Shub, and S. Smale. *Algebraic settings for the problem "$P \neq NP$?"*, volume 32 of *Lectures in Applied Mathematics*, pages 125–144. American Mathematical Society, 1996.

[19] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer, New York, 1997.

[20] L. Blum and C. Frieze. In a more balanced computer science environment, similarity is the difference and computer science is the winner. *Computing Research News*, 17(3), May 2005. `http://www.cra.org/CRN/articles/may05/blum.frieze.html`.

[21] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: Np-completeness, recursive functions and universal machines. *AMS Bulletin (New Series)*, 21(1):1–46, 1989.

[22] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[23] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, June 1996.

[24] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiseron, K. H. Randall, and Y. Zhou. CILK: An efficient multithreaded runtime system. In *Proc. 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 207–216, July 1995.

[25] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal of Computing*, 27(1):202–229, 1998.

[26] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[27] B. Chang, K. Crary, M. DeLap, R. Harper, J. Liszka, T. Murphy VII, and F. Pfenning. Trustless grid computing in conCert. In M. Parashar, editor, *Grid Computing – Grid 2002 Third International Workshop*, pages 112–125, Berlin, November 2002. Springer-Verlag.

[28] Albert M. K. Cheng. *Real-Time Systems: Scheduling, Analysis, and Verification*. Wiley, 2002.

[29] E. G. Coffman. *Computer and Job-Shop Scheduling Theory*. John Wiley and Son, New York, 1976.

[30] S. A. Cook. Towards a complexity theory of synchronous parallel computation. *Enseign. Math.*, 27:99–124, 1981.

[31] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling — a status report. In *Proc. 10th International Workshop on Job Scheduling Stategies for Parallel Processing*, pages 1–16, June 2004.

[32] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13, 2005.

[33] Mor Harchol-Balter. Task assignment with unknown duration. In *20th International Conference on Distributed Computing Systems*, Taipei, Taiwan, April 2000.

[34] Mor Harchol-Balter. Task assignment with unknown duration. *Journal of the ACM*, 49(2):260–288, March 2002.

[35] Mor Harchol-Balter and Paul Black. Queueing analysis of oblivious packet-routing networks. In *Proceedings of 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 583–592, Arlington, VA, January 1994.

[36] Mor Harchol-Balter, Mark Crovella, and Cristina Murta. On choosing a task assignment policy for a distributed server system. *Journal of Parallel and Distributed Computing*, 59(2):204–228, November 1999.

[37] Mor Harchol-Balter and Allen Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, August 1997.

[38] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems*, 21(2):207–233, May 2003.

[39] F. Jamalyaria, R. Rohlfs, and R. Schwartz. Queue-based method for efficient simulation of biological self-assembly systems. *Journal of Computational Physics*, 204:100–120, 2005.

[40] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 869–941. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[41] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, California, January 2006.

[42] G. J. Narlikar and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Trans. on Programming Languages and Systems*, 21(1):138–173, 1999.

[43] N. Pippenger. On simultaneous resource bounds. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pages 307–311, 1979.

[44] R. Schwartz. The local rules dynamics model for self-assembly simulation. Technical report, Massachusetts Institute of Technology, 2000. Computer Science Technical Report MIT-LCS-TR-800.

[45] R. Schwartz, R. L. Garcea, and B. Berger. "local rules" theory applied to polyomavirus polymorphic capsid assemblies. *Virology*, 268:461–470, 2000.

[46] R. Schwartz, P. W. Shor, and B. Berger. Local rule simulations of capsid assembly. *Journal of Theoretical Medicine*, 6:81–85, 2005.

[47] R. Schwartz, P. W. Shor, P. E. Prevelige, and B. Berger. Local rules simulation of the kinetics of virus capsid self-assembly. *Biophysical Journal*, 75:2626–2636, 1998.

[48] T. Zhang, R. Rohlfs, and R. Schwartz. Implementation of a discrete-event simulator for biological self-assembly systems. In *Proceedings of the 2005 Winter Simulation Conference*, pages 2223–2231, 2005.