32 TaskSpaces: A Software Framework for Parallel Bioinformatics on Computational Grids

HANS DE STERCK[1], ROB MARKEL[2], AND ROB KNIGHT[3]

[1]Department of Applied Mathematics, University of Waterloo, Ontario, Canada
[2]Scientific Computing Division, National Center for Atmospheric Research, Boulder, Colorado, USA
[3]Department of Chemistry and Biochemistry, University of Colorado at Boulder, USA

Due to ever-increasing data sizes and the high computational complexity of many algorithms, there is a natural drive towards applying parallel and distributed computing to bioinformatics problems. Grid computing techniques can provide flexible, portable and scalable software solutions for parallel bioinformatics. Here we describe the TaskSpaces software framework for grid computing. TaskSpaces is characterized by two major design choices: decentralization, provided by an underlying tuple space concept, and platform independence, provided by implementation in Java. We discuss advantages and disadvantages of this approach, and demonstrate seamless performance on an ad-hoc grid composed of a wide variety of hardware for a real-life parallel bioinformatics problem. Specifically, we performed virtual experiments in RNA folding on computational grids composed of fast supercomputers, in order to estimate the smallest pool of random RNA molecules that would contain enough catalytic motifs for starting a primitive metabolism. These experiments may establish one of the missing links in the chain of events that led to the origin of life.

—

Note: To appear as a Chapter in the textbook *Parallel Computing in Bioinformatics and Computational Biology*, A. Zomaya, editor, John Wiley and Sons, 2005.

Contents

32.1 INTRODUCTION

In recent years, parallel and distributed computing techniques have steadily been gaining popularity for tackling difficult bioinformatics problems. Two important reasons for the use of parallel techniques can be identified easily. First, bioinformatics problems involve increasingly large datasets. For example, the 2003 release of Genbank contained 36.6 billion basepairs and 118,689 different species, and in routine proteomics experiments, examination of a single sample may easily produce millions of peptide spectra to be processed. Second, the algorithms used in many bioinformatics applications can be computationally prohibitive. For example, the computational complexity of algorithms for phylogenetics typically scales cubically to exponentially in the number of species, and in proteomics data processing applications, the algorithms used to identify proteins and genes from peptide spectra involve searches with high complexity. For many bioinformatics problems it is therefore clear that computations limited to a single CPU cannot deliver the required computing power, and that parallel and distributed computing approaches are therefore necessary. A large class of bioinformatics problems can be parallelized easily, with minimal or no interprocess communication. These types of problems are called loosely coupled, and they are especially suitable for distributed processing. More tightly coupled problems require intensive interprocess-communication. Efficient parallel and distributed computing is typically more challenging for this type of problems. In the present Chapter we discuss both types of problems.

Consider, for example, the case of a university researcher who is confronted with a complex bioinformatics problem. The researcher typically has access to a wide range of computational resources on different scales. These resources include desktop machines that may be available in the researcher's own lab (of the order of 10 CPUs or so), PC clusters that may be available at the department level (order 100 CPUs), parallel computers that may be available in the university's computer center (order 100-1000 CPUs), and large parallel supercomputers (up to several thousand CPUs) that can be accessed at national supercomputer centers such as the US National Center for Supercomputing Applications (NCSA) and the San Diego Supercomputer Center (SDSC). In this Chapter, we propose an approach to parallel bioinformatics that, ideally, allows the researcher to develop the bioinformatics software locally on a single PC. Then, depending on the size of the problem at hand, the task can be distributed seamlesly over any or all of the wide variety of machines available.

This 'universal computing dream' is hard to realize for several reasons. The hardware, operating systems (and versions of operating systems), supporting software, and queueing systems may all vary among available machines. The researcher will wonder how to install and maintain code on all these machines, how to distribute tasks and data, how the results will be collected and centralized, and so forth. Scripts that automate some of these tasks will be brittle when software is upgraded, or machines are added or removed. In the light of these obstacles, the 'universal computing dream' seems little more than an ever-receding mirage.

However, in this Chapter we describe TaskSpaces, a system we developed that demonstrates that many components of the ‘universal computing dream’ can be realized on today’s infrastructure using grid computing. The grid computing concept can be easily understood by considering the analogy with a power grid. A power grid user accesses the grid in order to obtain electrical power, which is an interchangeable commodity. Indeed, the user’s machines do not care where or how the power they use is produced (the user may have ethical concerns that affect the desirability of particular power sources, but, to the hardware, all electricity is equivalent).

Two crucial properties make the power grid work:

1. The grid can be accessed through a standard interface.
   In the case of a power grid, the standard interface is simply the electrical plug, which gives access to the power grid that operates at standard voltages and frequencies.

2. The grid is scalable.
   This scalability works both from the user’s side (the user can access more power as needed), and from the power producer’s side (the grid operator can switch in additional power generators as demand rises).

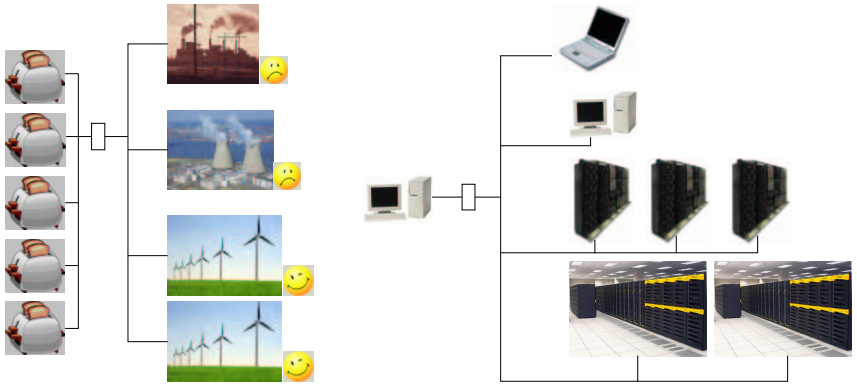

Fig. 32.1 Analogy between a power grid (left) and a computational grid (right). Both exhibit scalability from the user’s and the producer’s sides, and need to be accessible through a standard interface.

Ideally, grid computing would work in exactly the same way: a user accesses the geographically distributed grid in order to obtain CPU cycles, which are considered an interchangeable commodity (the user does not care where the computing cycles are produced) (Fig. 32.1). Unfortunately, accessibility through a standard interface (the first of the two essential properties of a grid) can be difficult to achieve with computers. In TaskSpaces, the standard interface is provided by the Java virtual machine, which is almost universally available. Java behaves almost exactly in the same way on all those machines, and Java’s ‘executable byte-code’ is, in theory,

fully interchangeable between machines. In TaskSpaces, the second essential grid property, scalability, is realized through the concepts of 'bag-of-tasks' computing and tuple spaces. Consequently, each user can submit many tasks concurrently, and the grid operator can switch in additional compute farms when demand is high.

The analogy between computational and power grids is not perfect: computing cycles and data are more complex than electrical power units. We can identify the following additional requirements for computing grids, some essential and others pragmatic.

3. Information is not interchangeable, and must often be kept confidential (unlike electrical power). The grid must allow secure resource sharing.

4. Information is not easily replaceable (unlike electrical power). The grid must provide fault-tolerance mechanisms such as transactions.

5. Parallel computers use many different queueing systems. The grid must provide resource allocation and scheduling.

6. Large problems may require deployment on heterogeneous hardware and software. The grid must provide a mechanism for distributing the application code transparently to the machines on which calculations are ultimately performed.

7. Many problems require interprocess communication. The grid must allow efficient communication between processes.

8. Computing resources are expensive. The grid must allow users to be billed according to cycle usage.

9. Some problems require specific turnaround time, data transfer bandwidth, fault-tolerance, etc. The grid may need to provide quality-of-service guarantees.

10. Problems must be connected with computing resources. Either the grid must allow the user to discover resources, or the grid must be able to discover tasks as they are presented (TaskSpaces uses the latter approach, resembling a real power grid).

Many efforts to realize the concepts of grid computing are now underway. Some projects, such as Globus, try to define standards for what eventually may become a worldwide, unified, computational grid ('The Grid'), very much along the lines of 'The Internet' and 'The World Wide Web'. However, many of the difficulties summarized above are still far from being resolved in a general, satisfactory way, and it is not clear that generally usable standards for grid computing will become available and accepted soon. Therefore, we have developed TaskSpaces, a software framework for a smaller-scale computational grid. TaskSpaces is based on the design criteria of decentralization, provided by an underlying tuple space concept, and platform independence, provided by implementation in Java. Our goal was to produce a lightweight grid environment that is easy to install and operate, and to demonstrate that it can be used efficiently for real-world parallel bioinformatics problems. In this

effort, we have attempted to deal with some, but not all, of the challenges listed above. Besides providing an environment for solving real bioinformatics problems on small, 'privately operated' grids, we hope that our experiences may reveal some methods of overcoming the challenges mentioned above, and that these methods may become more generally useful in guiding standards adopted for larger grids. At present, many different approaches are being tested on small-scale, privately operated grids, both in research and commercial settings. The successful approaches will survive, and, driven by demand and cost savings through efficiency gains, these privately run grids may eventually become connected to form a World Wide Grid, very much like national power grids are presently connected to neighboring grids throughout most of the world.

The rest of this Chapter is organized as follows. The next Section describes TaskSpaces, our prototype software framework for grid computing, which we based on tuple space concepts and implemented in Java. Section 32.3 describes a loosely coupled parallel bioinformatics application that we investigated on a computational grid, namely the problem of finding correctly folded RNA motifs in sequence space. Section 32.4 describes our experience with operating the software framework on a computational grid composed of local workstations and parallel clusters at supercomputer centers. Brief results for the RNA motif problem are presented in Section 32.5. The Chapter concludes with a Section on future work, and a Chapter summary.

32.2 THE TASKSPACES FRAMEWORK

TaskSpaces is a prototype lightweight grid computing framework for scientific computing characterized by two major design choices: decentralization, provided by an underlying tuple space concept, and object-orientation and platform-independence, provided by implementation in Java. The TaskSpaces framework has been described in full detail in [9]; in this Section we summarize its main properties.

Tuple spaces were pioneered in the late 1970s, and were first realized in the Linda system and language [2]. In a tuple space distributed computing environment, processes communicate solely by adding tuples to and taking them from a tuple space, a form of independent associative memory. A tuple is a sequence of fields, each of which has a type and contains a value. Fig. 32.2 shows conceptually how distributed computation works in a tuple space environment. An application program places subtasks resulting from the partitioning of a large computational problem into a tuple space (which in the Bag-of-Tasks paradigm is called a 'task bag' [1]), in which each subtask is represented as a tuple. 'Worker processes' take the task objects from the task bag, execute the tasks, and place the result in a 'result bag' as another tuple. The tuple space concept allows tasks to be decoupled both in space and time. The distributed computing process is decoupled in space, as the application, task and results bags, and the various worker processes may reside on a heterogeneous collection of machines that are connected by a network but that are otherwise widely geographically distributed. This decoupling allows flexible topology for the

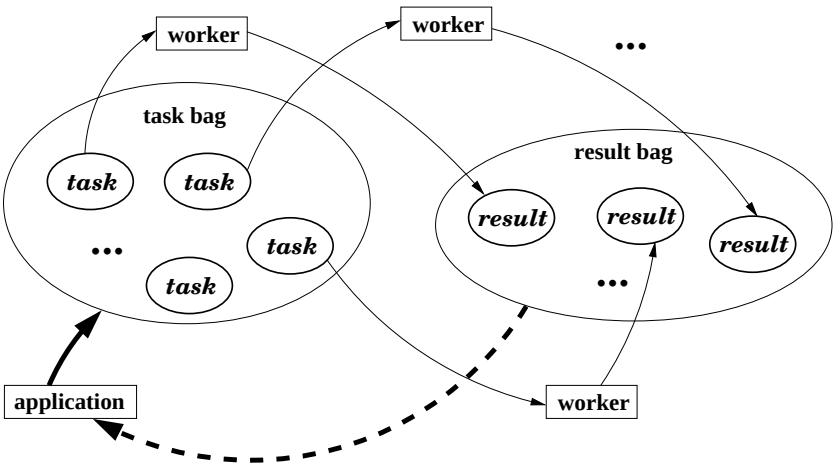

Fig. 32.2 The tuple space paradigm for distributed computing.

computation, permitting automatic configuration based on the availability of worker processes. The distributed computing process is also decoupled in time: since spaces are persistent, tuples are persistent while resident in the space, and processes can access tuples long after the depositing process has completed execution.

Fig. 32.3 shows a conceptual deployment diagram of the TaskSpaces framework. TaskSpaces uses an event-driven model. On startup, worker processes register with a task bag. The application process sends subtask objects to the task bag, and the task bag sends those task objects to available workers. The task bag acts as a 'superqueue', and thus alleviates the problem of scheduling when multiple supercomputers with different unsynchronized queueing systems are used. Scalability is inherent because users may put several applications in the task bag at the same time, and the grid operator can add 'worker farms' when needed. After a task is processed, the worker puts a result object in the result bag, from which the result objects are collected for final assembly by the application. TaskSpaces is implemented in Java, providing a standard, platform-independent interface to the grid system and exploiting Java's built-in networking and security features.

The TaskSpaces code consists of several classes. All classes, except for the Runner class, are served to participant machines via HTTP servers. Configurable properties files which contain system information and parameters, described further below, are also served by HTTP servers.

The Runner class is the driver of the system; it is a 2KB Java bytecode executable which contains a main() method, and it is executed from the command line (or from a queue script). This is the only file whichÄ must be installed on a machine to enable the machine to participate in the grid system in any of the possible functions such as running an application, running a bag, or running a compute node. The Runner accepts as a command line argument the URL identifier of a Java properties file which

contains any number of resource URLs. These resource URLs contain compiled Java classes or JAR format archive files, with methods that can be used during operation. The built-in security features of Java, such as digitally signed JAR files and tools for creating a configurable sand box for the JVM interpreter which can restrict access to local and remote resources for the downloaded code, can be incorporated at this level. The final argument passed to a Runner process is the name of the class to run. This is typically either Node (for a compute node), Space (for a tuple space bag), or the name of a TaskSpaces application. The Runner downloads the appropriate class from the input argument set of URL resources, casts it to an instance of the Java class Runnable, and begins execution by calling its run() method. Any additional classes required for execution are automatically downloaded, provided they are present in the set of URL resources.

The Node class represents a compute node which participates in the system. The properties file, also identified by command line argument, contains a property named "spaces" which identifies the IP address and port number of any existing Space servers. The Node registers with these Space server objects on startup. Other properties define the maximum run time for the Node, after which it automatically terminates, a default port number on which to begin to run the Node, and the maximum number of Task objects to process before termination. Upon startup, the Node creates an instance of the Server class and an instance of the Worker class. The Server registers with the remote task Space object that is identified in the properties file, and then waits for

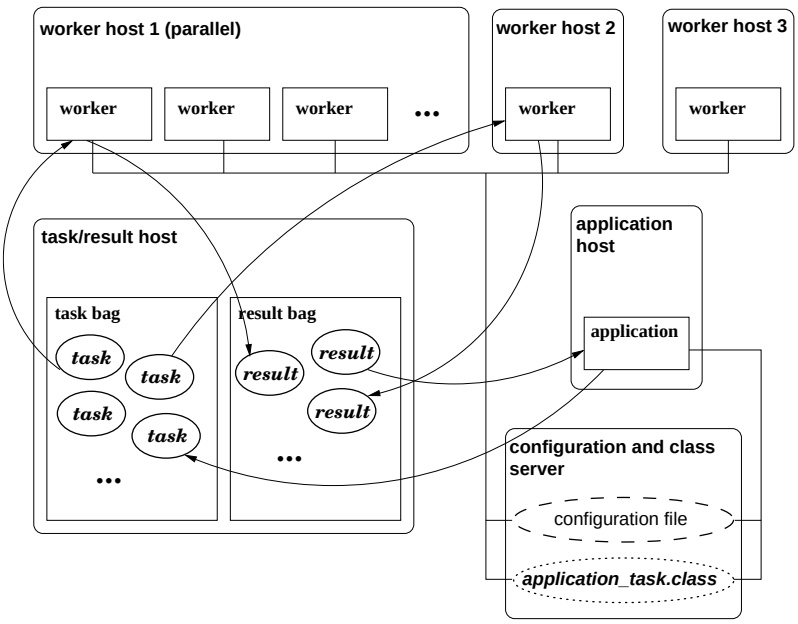

Fig. 32.3 TaskSpaces framework deployment diagram.

incoming requests. Incoming requests may be of three possible object types, Task, Agent, and Message. Task objects are applications or application components to be executed. These objects are passed to a TaskStore object which was created by the Node on startup, and is monitored by the Worker thread. The Worker executes the Task, which may, for applications with interprocessor communication, contain code to monitor the local MessageStore object, also initiated in a separate Thread by the Node at startup. When received, Message objects are placed by the Server into the MessageStore. The MessageStore is monitored by the Task, and the Message objects are pulled into the Task and the data is extracted and used by the Task during execution. Agents are executed immediately by the Server upon arrival, and may be used for a variety of system functions, including shutting down the Node or extracting data held in a Node data structure which is available to the application for storing data, or system or state information.

The Space class contains several data structures (mainly synchronized ArrayLists) and methods for accepting and storing the addresses of registering Nodes, and Task Objects. IP addresses and port numbers of registered Nodes are stored in two structures, one which is permanent, and one from which address and port identifiers are deleted as individual Tasks are sent to the registered Nodes in FIFO order. The permanent addresses can be used by Agents to identify all Nodes which may be participating in the system. This information can be used by Agents to shut down running Nodes or perform other application-dependent functions. Space objects are created in the same manner as starting a Node. The system property file mentioned above specifies the IP address and port number of Spaces in the system that act as task or result bags. On startup, a Space creates a server on a port specified in the properties file. It is expected that the Spaces will typically be started once and then left running as long as worker nodes may be active, similarly to HTTP servers. Spaces are multi-threaded and create a new thread for each incoming request. Each new thread is an instance of the SpaceConnection class, within which the nature of the request is identified and internal processing is performed. When a Task object arrives, it is sent to a worker node if any Nodes are currently registered as available Nodes. If not, the Task is temporarily stored until worker Nodes register with the task Space to indicate their availability. Spaces can also transmit messages to Nodes running applications, and can be used as an intermediary messaging store, or as the medium for Nodes to exchange addressing information in case applications require direct Task to Task communication (see [9]).

An additional optimization can be made for applications for which the subtasks do not have a large input data set, and have a limited number of input parameters that vary in a systematic way. In this case, it is beneficial to generate the (potentially many) Task objects within the task Space, rather than have the Application process generate all the separate Task objects, and then send them to the task Space one by one. The Application process can define an object of class TaskAgent. The TaskAgent is sent to the the task Space, and there a method is called on the TaskAgent to create a new application task. The Space advances the data by calling the next() method of the TaskAgent which calls the internally held application Task object to advance its state

(whatever this may mean in the context of the application) to create a new application Task. The new Task is returned to the Space and the Space sends this Task to an available Node.

Application processes can choose to have Nodes send result data to a result Space, or directly back to the application. In the first case, the Application registers with the result Space in order to receive any arriving results which are sub-classes of the Result class. The application can then store the data locally on the file system, send it to a database, or process the data and display output on a connected display device.

The final class of the TaskSpaces system is the Communicator. The Communicator contains methods and a protocol definition for all of the other components to communicate with each other. All the components sub-class Communicator to enable remote communication with the other components of the system. The Communicator class centralizes communication in a single object for simplified error tracking and modification which are complex problems in such widely distributed systems. The communication is performed using integer identifiers and serialized Java objects over sockets.

Application code need not be installed and maintained on workers, because it is downloaded from a central server when task objects arrive at each worker. Installing and executing a Java bytecode executable of size $<$ 2kB allows any worker host to participate in the grid. Thus, installation and maintenance of TaskSpaces is extremely lightweight and easy. In fact, the complete TaskSpaces codebase is extremely small and compact, due to the simplicity of the design, and the availability of Java's built-in networking and object manipulation capabilities.

Blue Horizon, SDSC, San Diego, CA (4 workers/processor)	64	128	240
P4 Linux, CU Boulder, CO (2 workers/processor)	4	4	4
Itanium Linux, CU Boulder, CO (2 workers/processor)	4	4	4
forseti1, NCSA, Urbana, IL (1 worker/processor)	16	16	16
hermod, NCSA, Urbana, IL (1 worker/processor)	16	16	16
Total number of workers	104	168	280
Total execution time	105s	103s	101s

Table 32.1 High-throughput grid experiment for a tightly coupled numerical linear algebra scientific computing problem with 500^2 grid points per worker. The number of worker processes and the total execution times are shown. The problem size is constant per worker process, so the nearly constant total execution times indicate almost perfect scalability.

TaskSpaces can be used in taskfarming mode for problems that do not require interprocess communication, such as independent folding of many RNA sequences

(see below). It can also be used for other applications that do require interprocess communication, handling such communication in a scalable way by transmitting serialized Java objects over sockets. Table 32.1 demonstrates that TaskSpaces scales well on large grids composed of supercomputers at NCSA, SDSC, and other supercomputer centers, connected over the internet, for a parallel computing problem in numerical linear algebra [9]. This problem requires neighbor-neighbor interprocess communication, and it is thus surprising that the scalability for this problem in the heterogeneous grid environment is so good.

Looking back at the prerequisites we set out in the previous Section for the 'universal computing dream' we pursue, it is instructive to consider how our prototype grid implementation performs with respect to our aspirations. Some of the functionality is only present in a rudimentary way in our prototype implementation, but more sophisticated versions based on the general concepts presented can easily be imagined.

1. Standard interface: YES.
   Through implementation in Java. In the strict sense this limits the applications to code written in Java, but, with limited sacrifices in generality, application code in other languages can be used as well (see below).

2. Scalable: YES.
   Through the tuple space concept. Scalability from the producer side is currently performed 'by hand', but automated strategies can easily be imagined. Also, bags can in principle be replicated when access loads become high and bottlenecks arise, and automatic strategies to this end can be considered as well.

3. Secure resource sharing: not implemented yet in TaskSpaces.
   But definitely feasible using Java's built-in mechanisms of digital signatures and public-private key cryptography.

4. Fault-tolerance: not implemented yet in TaskSpaces.
   But, for instance, automatic duplication of bags for backup reasons could easily be achieved via simple cloning of Java objects.

5. Resource allocation and scheduling: YES.
   The task bag acts as a 'superqueue'.

6. Automatic distribution of application code to worker machines: YES.
   By downloading Java objects from the task bags. The objects contain both the data and references to the application code, which is downloaded automatically from the class server upon first use by a worker.

7. Scalable interprocess communication: YES.
   Through direct exchange of serialized Java objects over sockets between workers, see also [9]. Efficient collective communications would require additional features such as multi-level communication schemes (see below).

8. User charging algorithms: not implemented yet in TaskSpaces.
   Simple charging strategies are straightforward to implement.

9. Quality-of-service: not implemented yet in TaskSpaces.
   This may require thorough study of the particular grid environments considered, and instrumentation of objects and worker machines with performance measures and priority mechanisms.

10. Resource discovery: YES.
    Computing resources discover tasks by making themselves available to the task bags, rather than the other way around. Compute farms are presently assigned to task bags by hand, but automatic, multi-level assignment strategies are feasible.

The overview above shows that the TaskSpaces design, despite its simplicity, is quite effective in realizing many of the conceptual aspirations of the 'universal computing dream'. In the following Sections, we illustrate how the framework, with minimal effort, can be used for a practical, real-life parallel bioinformatics application on ad-hoc computational grids composed of a variety of widely available hardware types.

32.3 APPLICATION: FINDING CORRECTLY FOLDED RNA MOTIFS IN SEQUENCE SPACE

We have applied the TaskSpaces framework to the following problem, which is relevant both to natural evolution and to a process of artificial evolution called SELEX that has been widely used to select new molecular functions from random pools of RNA. Given a pool of random RNA molecules of a specified length (typically 50-200 bases), what is the probability that the random pool contains molecules that have the right sequence and are folded into the right structure needed for a particular chemical function? This question is critical for the RNA world hypothesis: if molecules that can catalyze a particular reaction are especially common, the idea that the tiny amounts of RNA that would be produced by prebiotic synthesis could produce an RNA metabolism becomes more plausible (Fig. 32.4) [5]. Chemically active RNA molecules are also routinely synthesized, in SELEX laboratory experiments, from intially random pools of RNA molecules [6]. Specifically, we have focussed on determining the abundance of isoleucine and hammerhead RNA motifs in random molecules [6]. The isoleucine motif is the shortest RNA motif capable of binding specifically to the amino acid isoleucine, while the hammerhead motif cuts RNA at specific locations and has been found both in cells and through SELEX. It has been determined experimentally that chemical function of a certain type appears when the random RNA molecule contains a prescribed motif, which is composed of several modules with partially specified sequence, and has a prescribed folding structure (see Fig. 32.5). The probability that a random molecule matches both the prescribed sequence and the structure, $P(seq, struct)$, is calculated in two steps

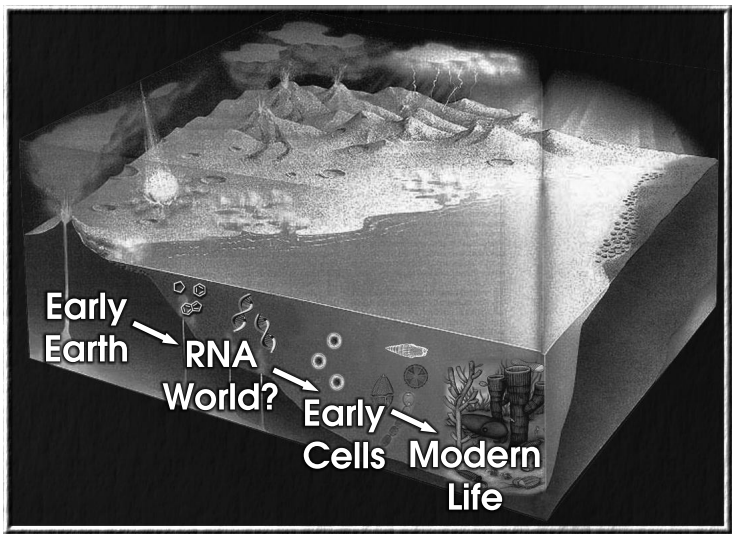

Fig. 32.4 The RNA world hypothesis: if a small number of random RNA molecules, say a pool of 10^6 to 10^9 sequences, has a reasonable probability of containing molecules with various chemical functions, then primitive metabolisms would be expected to have arisen many times on the early Earth.

as $P(seq, struct) = P(seq)\ P(struct|seq)$. The sequence probability $P(seq)$ can be approximated accurately by combinatorial formulas [5, 6]. The conditional probability of obtaining the right folding structure, given a partially random molecule that contains the right sequence, cannot be approximated analytically. In stead, we approximate this probability by computational folding of large samples of RNA molecules (a sample size of 10,000 is typically used): the probability is approximated by the number of partially random molecules that fold into the correct structure, divided by the total number of molecules in the sample. One important question of interest is the variation of the probability $P(seq, struct)$ as the composition of the random pool changes, since the composition of RNA pools may have varied widely on the primitive earth and since modern genomes vary widely in composition, possibly affecting the evolution of specific functions. We set out to investigate whether specific kinds of chemical function arise more often in pools with overall composition biases in particular directions. This required the computational folding of many samples in $\{A, C, G, U\}$ composition space. We used 5% intervals in composition space, leading to 969 different compositions to be tested. Varying the length of the random molecules (we have considered lengths of 50, 100, and 150 nucleotides), further increased the number of foldings required. For the results to be discussed briefly below (see [6] for a more detailed discussion), we performed about hundred million computational foldings. This constitutes a computational problem of moderately large size, which would require weeks to months on a single fast workstation. We

decided to use a grid computing approach, mainly for flexibility, portability and scalability reasons.

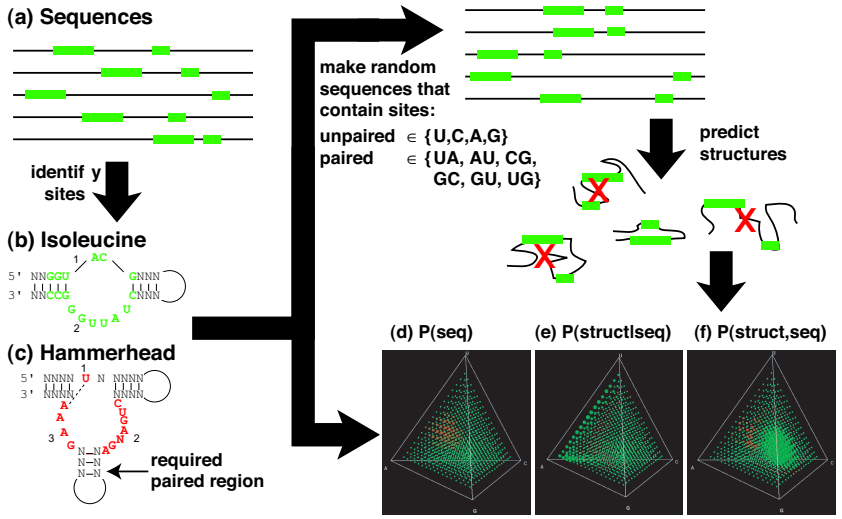

Fig. 32.5 Procedure for determining the effects of folding and sequence composition on motif abundance. (a) the motifs are identified by comparing sequences with the same function. The isoleucine aptamer (b) and the hammerhead ribozyme (c) both consist of modules that must have an exact sequence, and flanking helices that must base pair but need meet no other constraints. These diagrams show the exact sequence and structure requirements that were used in the calculations: base pairs are indicated by connecting lines. We calculate Pr(sequence) (d) from the sequence requirements, and Pr(structure|sequence) (e) by constructing large samples of random sequences that contain the motif and computationally predicting their structures. The overall probability of finding a correctly folded sequence (f) is obtained by multiplying the probabilities from (d) and (e).

We used the Vienna RNA folding package [3], which is written in C, for folding individual sequences. The RNAfold executable is called by the Java application on each worker node as needed. Non-Java executables must be compiled in advance for each worker architecture, and can be downloaded from the code server by the workers upon first use. Thus, although reliance on code written in other languages increases the effort required for cross-platform operation, it is still feasible.

32.4 CASE STUDY: OPERATING THE FRAMEWORK ON A COMPUTATIONAL GRID

We simulated the RNA function probability problem on a grid composed of the NCSA IA32 Linux Platinum Supercluster and various P4 Linux workstations at CU Boulder, Colorado. The Platinum machine features 968 P3 compute processors (1GHz). For code development and execution of some smaller subproblems, only the local workstations were used, while for larger problems the local workstations were combined with up to 200 Platinum processors concurrently. The total computing time used for this project so far, including extensive initial runs for exploring the problem and determining the right approach and questions to be answered, amounts to approximately 10,000 Platinum processor hours.

The framework was easy to install on candidate worker machines. Even though Java is normally not thought of very much as a language for supercomputing, it is actually available on all machines we obtained access to, even the largest parallel supercomputers. In fact, Java is catching up fast in execution speed with other languages, and the advantages in ease of use and portability may actually give it a good future in scientific computing. Locating the Java executable (which is typically not included in the standard path), copying the Java worker bytecode to the worker machine, and starting the workers, was typically very fast: for most machines it did not take more than 15 minutes to make them participate in the grid. On parallel computes, the standard queueing systems were used. Varying queue delays on concurrently participating machines did not cause a problem, because the taskbag (typically located on a workstation in Boulder) acts as a superqueue, and the RNA folding tasks are loosely coupled and do not require any interprocess communication and synchronization. The Internet was used as network connection between the grid machines, and network performance was adequate at all times.

A major obstacle in constructing ad-hoc grids like this is security, which will become increasingly important as research networks and institutions are increasingly targeted by malicious intruders. Under pressure from malicious attacks, potential worker machines will often be protected by firewalls. Participation in a grid then requires additional firewall configuration, as our framework requires at present worker nodes with externally accessible IP addresses. Security is another reason why we expect, as argued before, that grids will develop as ‘islands’ for the foreseeable future, further delaying the concept of a ‘World Wide Grid’. Another inconvenience in operating a grid is the variety of queueing systems operating on parallel computers and clusters. If machines were available where TaskSpaces workers would be the only, continuously running processes, then much of the queueing considerations could be dealt with in more efficient ways that decrease turnaround times, for instance by extending the use and functionality of task bags as superqueues.

We can summarize our experiences with operating the grid framework for a real problem on a real moderately sized grid, by saying that the framework mostly delivered the promised flexibility, portability and scalability.

32.5 RESULTS FOR THE RNA MOTIF PROBLEM

We estimated the abundance of two motifs, the hammerhead ribozyme and the isoleucine aptamer, in random-sequence pools of many compositions and several lengths. These two well-studied motifs provide test cases for our code on the TaskSpaces framework, with which we plan to analyze dozens to hundreds of motifs. Knowing where particular kinds of RNA sequences are most likely to be found in the space of possible compositions, and where these sequences are most likely to fold into the correct structure if they are found, will provide striking new insight into the conditions under which particular RNA activities can evolve.

To test the effects of nucleotide composition on the probability of meeting the sequence requirements and the probability of correct folding, we generated 10,000 random sequences at each of the 969 possible 5% intervals of sequence composition. The sequences were of total length 50, 100, and 150 nucleotides, meeting the sequence requirements for each of the hammerhead and isoleucine motifs. We repeated the analysis for sequence length 50 allowing G-U base pairs (a weaker type of pairing than the more familiar 'Watson-Crick' G-C and A-U base pairs, which are found at a small but not negligible frequency in biological RNA structures). Thus we folded a total of 77,520,000 sequences for this experiment.

We found that the composition of the randomized sequences had a striking effect on both the probability of finding each motif and the probability of correct folding. Figure 32.6 shows the probability of meeting the sequence requirements, the probability of correct folding given that the sequence requirements were met, and the overall probability of finding the motif, for each of the 969 5% compositions that include at least some of each of the four nucleotides U, C, A, and G. The patterns in the different diagrams are strikingly different, indicating that folding and sequence abundance can actually have antagonistic effects on the overall probability of finding a correctly folded motif.

The probability of correct folding ranged over many orders of magnitude. Figure 32.6 shows, for all 5% intervals of nucleotide composition in the space of possible compositions, the probability of meeting the sequence requirements in a completely random sequence for the hammerhead and isoleucine motifs (left and right respectively; Figure 32.6a and 32.6b), the probability of correct folding in partially random sequences that already meet the sequence requirements (Figure 32.6c and 32.6d), and the combined probability of finding the correctly folded motif. In sequences of total length 100, he probability of finding the isoleucine motif ranged from 1.44×10^{-21} to 5.71×10^{-10} with a mean of 3.62×10^{-11}, reaching a value of 1.71×10^{-10} at unbiased nucleotide frequency and a maximum at the coordinates 15%A, 25%C, 35%G, and 25%U. the probability of finding the hammerhead motif ranged from 0 to 4.58×10^{-10} with a mean of 7.37×10^{-12}, reaching a value of 3.38×10^{-11} at unbiased nucleotide frequency and a maximum at the coordinates 35%A, 10%C, 25%G, and 30% U.

Sequence length also had a substantial effect on the probability of correct folding. As expected [8, 10, 5], longer sequences had a large combinatorial advantage over

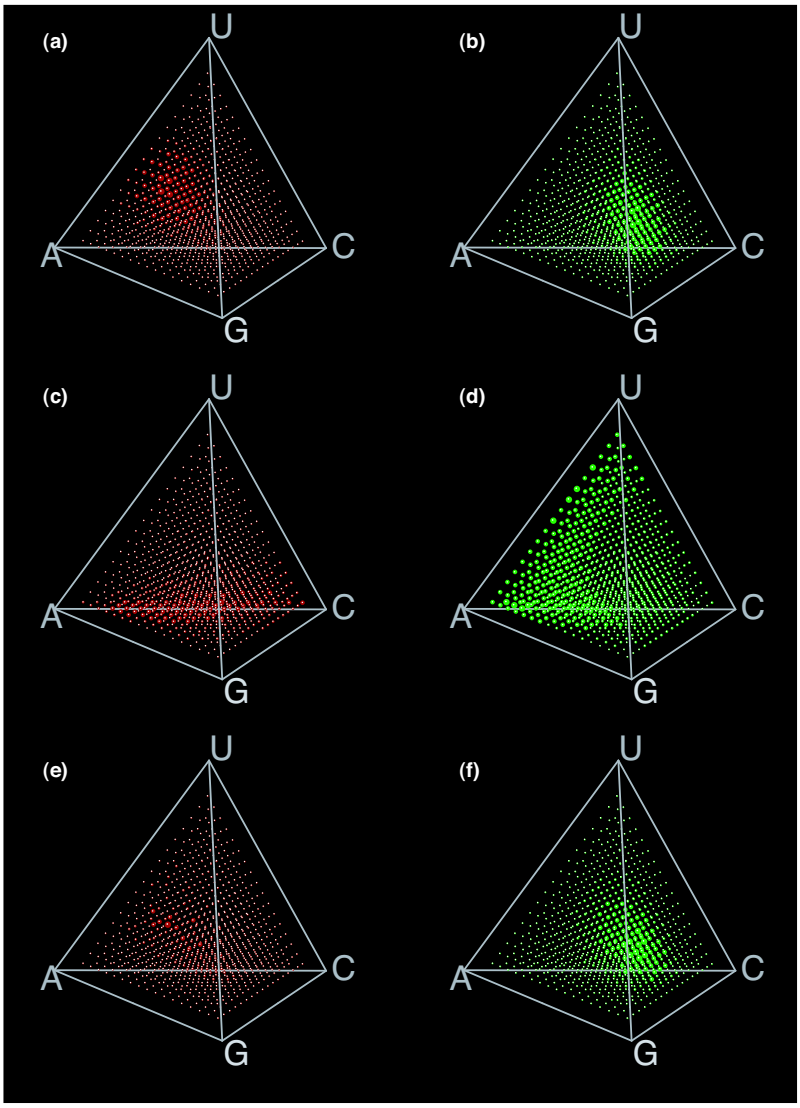

Fig. 32.6 Folding results for the hammerhead (left) and isoleucine (right) motifs. Probability of finding the required sequence elements (a and b), probability of folding correctly given that the required sequence elements were present (c and d), and overall probability of having the required sequence elements and folding correctly (e and f). Volume of each sphere is proportional to the probability at each of the 969 internal 5% intervals in the space of possible compositions. Radii are scaled such that the maximum radius in each diagram is set to 0.01 composition unit. These results are for sequence length 100.

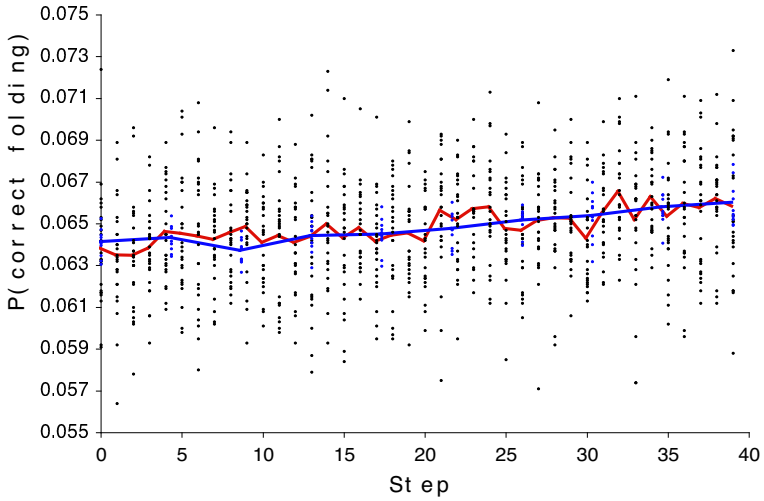

Fig. 32.7 Fine-grained analysis of regions between the two most probable points for isoleucine aptamer folding, which were ACGU = [10,30,35,25] and ACGU = [10,30,40,20] with a total sequence length of 50 nucleotides. We made ten independent samples (dots), each of 100,000 sequences, at each of ten equal intervals between the two most probable points (smoother line shows the mean), and made twenty-five independent samples (dots), each of 10,000 sequences, at each of forty equal intervals between these same two points (more wiggly line shows the mean). Both series are shown at the same scale in terms of absolute composition. The lines for the means are smooth in both cases, although (as expected) the scatter is lower for the points at the larger sample size.

short sequences in meeting the sequence requirements (maximum probabilities of 1.74×10^{-8}, 1.42×10^{-6}, and 7.87×10^{-6} for 50, 100, and 150 nucleotides for the hammerhead motif, and 3.46×10^{-9}, 3.20×10^{-8} and 8.94×10^{-8} for isoleucine: the probability for the isoleucine aptamer changes more slowly because it has two modules instead of three for the hammerhead). However, this combinatorial advantage was offset somewhat by substantially worse folding at greater sequence lengths (maximum probabilities of 5.64×10^{-2}, 2.49×10^{-2}, and 1.08×10^{-2} for 50, 100, and 150 nucleotides for the hammerhead motif, and 3.17×10^{-1}, 1.78×10^{-1} and 1.29×10^{-1} for isoleucine). The maximum overall probabilities for the two sites were 4.27×10^{-12}, 4.57×10^{-10}, and 8.61×10^{-10} for 50, 100, and 150 nucleotides for the hammerhead motif, and 1.88×10^{-10}, 5.71×10^{-10} and 1.06×10^{-9} for isoleucine (note that these are not the products of the best probabilities for finding the sequence requirements and for folding, because the optima occurred at different compositions). These findings are difficult to reconcile with experiments that show that motifs are much more difficult to find in longer random regions [4, 7]. One possibility is that the computational folding systematically overestimates the probability of a correct fold in longer sequences; another is that other effects of sequence length, notably

amplification efficiency, outweigh the effects of function at the RNA level. We plan to test these effects directly by synthesizing sequences that are computationally predicted to fold into one motif or the other. We will then use chemical and enzymatic probing to test the structural predictions around each motif, and assay the relevant catalytic and binding parameters to determine whether the molecules perform the predicted function.

To test whether the compositional grid was sufficiently fine to locate the region of maximum probability, we performed a more detailed analysis of the transect between the two best-folding points for the isoleucine aptamer using a larger sample size of 100,000 sequences per point to reduce the effects of sampling error. Fig. 32.7 shows the folding probabilities at 10 intervals between the two best points at sequence length 50: 10%A, 30%C, 35%G, 25%U and 10%A, 30%C, 40%G, 20%U. Interestingly, the region of maximum probability was insensitive to the length of the sequence, although (as seen above) the length of the sequence changed the probability at each point by orders of magnitude.

These results demonstrate striking relationships between nucleotide composition and the probability of finding specific sequences, and suggest that we may be able to predict which kinds of random-sequence pools (for SELEX or in organisms) might be most able to evolve particular functions. The probability of finding the specific functions we searched for (10^{-8} to 10^{-12}) are rather lower than we had predicted from previous work, demonstrating that the effects of folding are important and cannot be ignored. These figures are consistent with the observation that new RNA activities are routinely isolated in the laboratory from random-sequence pools of 10^{12} to 10^{15} molecules, although they do not provide support for the idea that an RNA metabolism could have arisen from only a few hundred thousand random RNA molecules as might have been present on the prebiotic Earth. Due to the chemical problems in synthesizing large amounts of RNA without enzymes, it has often been suggested that a simpler self-reproducing RNA system preceded the RNA World. However, once RNA was first synthesized (perhaps for an entirely different reason), our results show that catalytic activity would soon be likely to emerge: 10^{15} 100-nucleotide RNA molecules is about 50 micrograms of RNA, less than the amount of RNA found in a single gram of modern tissue.

32.6 FUTURE WORK

As demonstrated above, our prototype grid framework delivers promising flexibility, portability and scalability for real-life applications on ad-hoc grids. However, there are many interesting ways in which the framework can be extended.

First of all, we are planning to build full Python language functionality into the framework to allow researchers familiar with that language to scale their single-CPU tasks easily to the grid. Python is becoming increasingly popular as a language for bioinformatics, mirroring its success for other scientific computing tasks. Second, as indicated in the enumeration in Section 32.2, the framework implementation needs to

be extended with regards to scalability, fault-tolerance, security, charging algorithms, and quality-of-service. For example, fault-tolerance may be enhanced by cloning of objects and bags, and transaction-type communication. Third, we plan to add more extensive functionality, in terms of support for complex parallel workflows (see Fig. 32.8), connection with databases for data furnishing and result collection, and multi-level tree-based collective communication for tightly-coupled parallel applications.

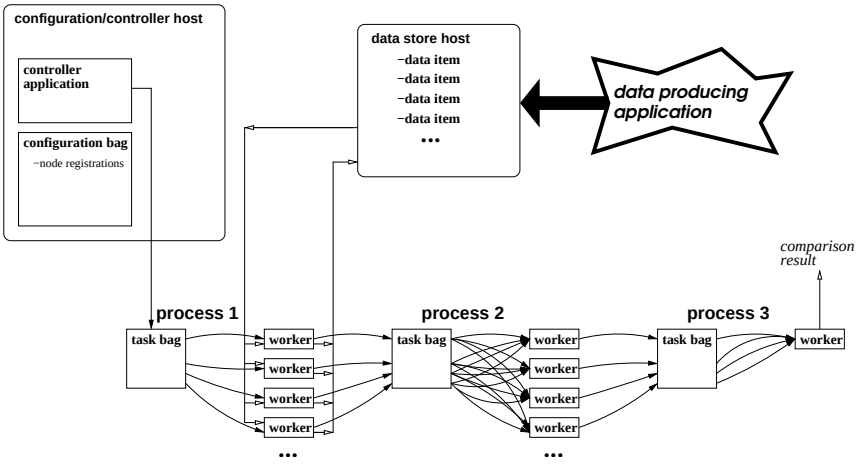

Fig. 32.8 Proposed agent-mediated workflow diagram. In the first phase, a configuration agent sets up a workflow topology for a workflow, consisting of two parallel processes and a serial process in this example. In the second phase, the data are carried through the workflow by execution agents. For fault-tolerance purposes, the workflow could be made self-migrating.

On the parallel bioinformatics application side, additional loosely coupled parallel bioinformatics applications will be studied, including variants of the previously considered RNA folding statistics problem (for instance, investigation of the effect of the length of the molecules on correct folding), and an examination of whether certain compositional features of ribosomal RNA are universal across organisms or across RNA molecules. We are also considering more challenging applications, including proteomics workflows and tightly coupled problems such as building large phylogenies.

32.7 SUMMARY AND CONCLUSION

We have described a software framework for scientific computing on computational grids that is based on tuple-space principles and implemented in Java, and we have demonstrated that seamless simulation on an ad-hoc grid composed of a wide variety of hardware is feasible for real-life parallel bioinformatics problems. The language and general approach we used is most appropriate in cases in which flexibility and ease of configuration outweigh concerns about extracting maximal performance on

a given architecture for a given, fixed, application with fixed, large problem size that must be executed repeatedly. In this latter situation, it is often a good investment to develop specific optimized software solutions of 'high-performance computing' type. In many situations, however, research is dynamic, and research goals and directions change continuously. In such a rapid-prototyping environment with wide variations in problem sizes, with complex changing workflows, and with fast variations in application code, a platform-independent 'high-throughput computing' grid solution of the type proposed in this Chapter may be most appropriate, because of the gains in flexibility, portability and cross-platform scalability.

Acknowledgments

This work was partially supported by the National Computational Science Alliance under grant MCB020011 and utilized the NCSA IA32 Linux Supercluster Platinum.

REFERENCES

1. G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, Boston, 2000.

2. D. Gelertner. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

3. I. Hofacker, W. Fontana, P. Stadler, L. Bonhoeffer, M. Tacker, and P. Schuster. Fast folding and comparison of RNA secondary structures. *Monatshefte Fur Chemie*, 125(2):167–188, 1994.

4. F. Huang, C. W. Bugg, and M. Yarus. RNA-catalyzed CoA, NAD, and FAD synthesis from phosphopantetheine, NMN, and FMN. *Biochemistry*, 39(50):15548–55, 2000.

5. R. Knight and M. Yarus. Finding specific RNA motifs: function in a zeptomole world? *RNA*, 9(2):218–30, 2003.

6. R. Knight, H. De Sterck, R. S. Markel, S. Smit, A. Oshmyansky, and M. Yarus. Finding correctly folded active RNA motifs in sequence space using computational grids. *RNA*, Submitted, 2004.

7. C. Lozupone, S. Changayil, I. Majerfeld, and M. Yarus. Selection of the simplest RNA that binds isoleucine. *RNA*, 9(11):1315–22, 2003.

8. P. C. Sabeti, P. J. Unrau, and D. P. Bartel. Accessing rare activities from random RNA sequences: the importance of the length of molecules in the starting pool. *Chem Biol*, 4(10):767–74, 1997.

9. H. D. Sterck, R. Markel, T. Pohl, and U. Rüde. A lightweight Java TaskSpaces framework for scientific computing on computational grids. In *Proceedings of the ACM Symposium on Applied Computing, Track on Parallel and Distributed Systems and Networking*, pages 1024–1030, 2003.

10. M. Yarus and R. Knight. *Translation Mechanisms*. Landes Bioscience, 2003.