Crystal-Growth-Inspired Algorithms for Computational Grids

Yuriy Brun and Nenad Medvidovic
Computer Science Department University of Southern California
Los Angeles, CA 90089-0781, USA
{ybrun,neno}@usc.edu

ABSTRACT

Biological systems surpass man-made systems in many important ways. Most notably, systems found in nature are typically self-adaptive and self-managing, capable of surviving drastic changes in their environments, such as internal failures and malicious attacks on their components. Large distributed software systems have requirements common to those of some biological systems, particularly in the number and power of individual components and in the qualities of service of the system. However, it is not immediately clear how engineers can extract useful properties from natural systems and inject them into software systems.

In this paper, we explore the nature's process of crystal growth and develop mechanisms inspired by that process for designing large distributed computational grid systems. The result is the tile architectural style, a set of design principles for building distributed software systems that solve complex computational problems. Systems developed using the tile style scale well to large computations, tolerate faults and malicious attacks, and preserve the privacy of the data.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Reliability, Security

Keywords

Nature-Inspired Software, Software Architectural Style, Self-Assembly, Privacy, Fault Tolerance, Computational Grid

1. BIOLOGY'S INSPIRATION

Biological systems are in many ways superior to man-made software and hardware systems. The human body alone has orders of magnitude more complexity than our most intricate designed systems. Further, biological systems are decentralized in such a way that allows them to benefit from built-in error correction, fault tolerance, and scalability. Despite added complexity, human beings are more resilient to failures of individual components and injections of malicious bacteria and viruses than engineered software systems are to component failure and computer virus infection. Other biological systems, for example worms and some sea stars, are capable of recovering from such serious hardware failures as being cut in half (both worms and some sea stars are capable of regrowing the missing pieces to form two nearly identical organisms), yet we envision neither a functioning desktop, half of which was crushed by a car, nor a machine that can recover from being installed with only half of an operating system. It follows that if we can extract certain properties of biological systems and inject them into our software design process, we may be able to build complex self-adaptive software systems.

Distributed Internet-sized systems is one area that is likely to benefit from biological inspiration. While we have a lot of experience and knowledge in how to build software to be executed on a single processor, and even on controlled networks of fixed sizes, the notions of "programming the Internet" or "running an operating system or a virtual machine in a distributed fashion on an ultra-large network" are fairly new. Such distributed systems will likely require a great deal of collaboration, while the large size of the network is likely to require that collaboration to scale well. Further, since no single entity controls the Internet, and the nodes may join or leave the network at any time, the collaboration must be fault-tolerant and resilient to dynamic node addition, removal, and failure. Finally, if these systems are to perform important computations, they must be resilient to malicious attacks from the network's nodes. In nature, systems often deal with constant component birth, death, and failure, as well as attacks from malicious components within the system, while allowing well-scaling collaboration between the components. The need for the development of design and implementation tools for these large distributed software systems together with the apparent similarities in requirements between these systems and large-scale natural systems make large distributed software systems an ideal target for research into nature-inspired algorithms.

While the notion that software and hardware systems

Figure 1: Termite mounds, such as those found in Australia, can reach several meters in size, but are built by millimeter-sized termites. These mounds have intricate internal structure that individual termites neither understand nor internally represent.

can benefit from biologically inspired paradigms has been around for some time [1], it has been unclear how one might leverage the mechanisms we see in nature to actually engineer software systems. Suggested methods include exploring automated solution-generation techniques by leveraging ideas from evolution [17], replicating the control flow of biological systems in software [12], and developing high-level design principles [18]. These methods may each prove fruitful in developing biologically inspired software and hardware; however, in this paper we argue that another method, leveraging the algorithms that individual biological components of a system follow to produce system-level emergent behavior, should also be a focus of research. Studying the algorithms of social insects and other simple components in nature, and the way their behavior combines to produce emergent system-level behavior, has already been an area of research in robotics for some time [21].

Let us consider an example biological system and explore how it can influence our engineering process. Termites, whose size is on the order of millimeters and who have fewer than 10^5 neurons, construct complex and comparatively giant termite mounds of the size on the order of meters (see Figure 1). These mounds have intricate and functional internal structure that individual termites neither understand nor internally represent. This structure involves careful control over air circulation for temperature and atmospheric regulation, placement of fungus combs, gardens, nurseries, and royal chambers. As an example of the mounds' functionality, large colonies are capable of keeping the mound's internal temperature at a constant 30°C, while the ambient air temperature changes from 0°C to 40°C, throughout the day [14].

The termite mound is a complex structure, but let us consider a simplified structure that is only concerned with the outer shape of the mound. Figure 2 presents an algorithm that termites can use to build such structures. Each termite executes this algorithm locally and assists in building a mound, even though neither the termite nor the algorithm have an internal description of the mound. This algorithm allows simple termites to build hill structures far larger than themselves, though without the added functional complexity of actual termite mounds. The algorithm real termites use is more complex than this one.

```
Hill-Build:
   if (not carrying dirt)
      if (see small patch of dirt)
         pick up dirt
   else
      if (see dirt)
         place dirt on top
      else
         take a few random steps
   repeat
```

Figure 2: An algorithm for self-organizing creatures to build simplified termite mounds.

The `Hill-Build` algorithm demonstrates how simple creatures can build structures without an internal representation of those structures anywhere within the creatures or the algorithm they follow. When thousands of termites, each executing `Hill-Build`, arrive at a location with some dirt, they move around, picking up the small patches of dirt and building a hill. Note that the algorithm never mentions or describes a hill or a mound and does not contain or use any global view of the world. We can use termite-inspired ideas to develop new algorithms for construction and self-organization of software systems.

In the remainder of this paper, we will, in Section 2, describe the nature's process of crystal growth and how scientists have leveraged crystals to compute mathematical functions, in Section 3, present ideas for a way to build large distributed computational software systems based on crystal-growth-inspired ideas, and, in Section 4, demonstrate our plans for the tile style's theoretical and empirical evaluations.

2. CRYSTALS AND COMPUTATION

In nature, crystals play a role that is integral to life. Crystals are composed of simple monomers and grow into distinct complex structures based on the environmental conditions of the growth process and several, typically few, dimensions of freedom in the attachments between the monomers.

With some careful control over the growth conditions and the degrees of freedom of the monomers, scientists have been able to produce crystals out of DNA that display information, such as the Sierpinski triangle [20], and compute simple functions, such as copying an input and counting in binary [4]. The ability to control the parameters of crystal growth and to encode the monomers to compute mathematical functions has given researchers in the field of self-assembly good reason to be optimistic about the future uses of crystal growth for computation [2], fabrication of nanoelectronic devices [19], and development of nanomachines [26].

Self-assembly and crystal growth are exciting to computer scientists because they offer computation mechanisms that are easily susceptible to (1) parallelization, with billions of computations easily happening in parallel in a test tube, (2) data mobility, with computation and data represented within the same medium, and (3) fault-tolerance techniques, detecting and correcting possible errors as they happen [25]. Further, self-assembly and crystal growth are also exciting to software engineers, as the mechanisms and algorithms monomers follow to assemble complex crystals can lead to novel paradigms for building distributed software systems that (1) scale well, (2) preserve the privacy of the data, and (3) tolerate node faults and malicious attacks [11, 10]. In order to demonstrate how software engineers can leverage the desirable properties of crystal growth in software systems, we must first formalize a model of crystal growth.

2.1 Tile Assembly Model

The tile assembly model [24] is a formal mathematical model of self-assembly and crystal growth. It allows modeling of the crystal-growth process on assemblies that can, among other things, build complex shapes and compute functions.

In the tile assembly model, individual components are square tiles with special labels on their four sides. Tiles are not allowed to rotate but they can stick together under certain conditions when their abutting sides' labels match. It is possible to encode inputs using tiles and design sets of tiles that attach to those input tiles to compute functions [6]. Figure 3 shows a very simple example of adding the numbers 34 and 27, which we describe in Section 2.2.

A tile assembly computes by growing a crystal. A crystal starts from a seed configuration (this configuration can encode an input to the computation) and tiles attach, autonomously, to that seed to grow a larger crystal. The attachments are autonomous but follow straightforward rules based on a set temperature and the sides of the tiles. In a simplified version of tile assembly, tiles can attach to a configuration when the number of sides that match the attachments meets of exceed the temperature. For example, if the assembly's temperature is 3, a tile can attach to a configuration only if at least 3 of its side labels match the labels of the tiles to which it is attaching. Eventually, the attachment process can terminate, producing a final configuration. If the assembly is properly designed, the final configuration can encode the output of a mathematical function. We now illustrate the process of crystal growth with an example.

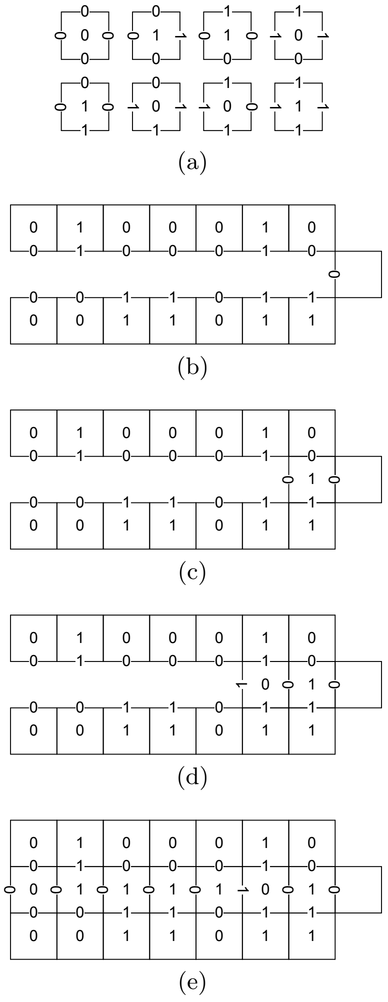

Figure 3: A sample tile assembly that adds numbers. The assembly has eight computation tiles (a) carefully designed with addition logic. A seed configuration (b) encodes the inputs, 34 = 100010$_2$ and 27 = 11011$_2$. At temperature 3, a tile can attach only when three of its sides match the already attached tiles, so a single tile attaches on the east side (c), creating the ability for a second tile to attach (d). This process continues until no more tiles can attach and the final configuration (e) displays the sum of 34 + 27 = 61 = 111101$_2$.

2.2 Adding Tile Assembly

The adding tile assembly depicted in Figure 3 has eight computational tile types. These eight tiles are shown in Figure 3(a). The eight computational tiles encode the logic of addition: each of the tile's labels (number in the center of the tile) is the binary sum of its north, east, and south sides (e.g., $1+0+1 = 0_2$) and each of the tile's number on its west side is the carry bit of the sum of the north, east, and south sides (e.g., carry bit of $1 + 0 + 1$ is 1). Figure 3(b) shows a seed configuration that encodes two numbers, in binary: $34 = 100010_2$ and $27 = 11011_2$, in the top and bottom rows of the seed, respectively. This tile assembly executes at temperature 3, so tiles can only attach when they match on three of their sides. A single tile, with the north, east, and south labels 0, 0, and 1, respectively, can attach to the seed, as shown in Figure 3(c). Note that this tile, in its center, displays the binary sum of the least significant bit of 34 and the least significant bit of 27 ($0 + 1 = 1$) and, on its west side, displays the carry bit associated with adding those least significant bits (carry bit of $0 + 1$ is 0). This process now allows a second tile to attach, as shown in Figure 3(d). This tile adds the carry bit from the previous tile and the second-least-significant bits of 34 and 27. This process continues until no more tiles can attach, and the bits in the centers of the tiles in the center row spell out the solution: $111101_2 = 61 = 34 + 27$.

While our explanation should give some intuition to how a simple tile assembly, combined with the process of crystal growth, can solve mathematical functions, we refer the reader to [6] for the full proof that the tile assembly shown in Figure 3 is an adding assembly.

2.3 Pushing Tiles Further

The addition example illustrates that tiles can compute mathematical functions. The key to the computation is the eight tiles from Figure 3(a) whose logic encodes that of addition. An interesting question is whether tile assemblies that compute more complex functions can be designed and how large and complex would those assemblies have to be. Research revealed that the tile assembly model, and thus crystal growth, is Turing universal [23], implying that crystals can compute all the functions that a traditional computer program can compute. However, it remains unclear just how complex tile assemblies would have to be in order to solve complex computational problems.

We have studied computation in the tile assembly model and found that fairly simple assemblies can compute fairly complex functions. Most notably, each of the following systems uses $\Theta(1)$ computational tile types, so the complexity of the assembly does not grow with input size: an assembly with 28 computational tile types can multiply two numbers [6], as assembly with 49 computational tile types can nondeterministically solve the NP-complete problem *SubsetSum* [8], an assembly with 50 computational tile types can nondeterministically factor integers [7], and an assembly with 64 computational tile types can nondeterministically solve the well-known NP-complete problem 3-*SAT* [9]. Of particular interest to software engineers are the tile assemblies that solve NP-complete problems. We will now describe one such assembly, and in Section 3 describe how to leverage this assembly and the algorithms inherent to crystal growth to build distributed software systems for solving NP-complete problems.

2.4 SubsetSum-Solving Tile Assembly

SubsetSum is a well-known NP-complete problem that consists of determining whether the sum of a subset of numbers adds up to a given target number. The input to the problem is a set of natural numbers and a natural target number, and the output is 1 if the sum of some subset of those numbers is equal to the target number, and 0 otherwise. For example, the answer to the question whether some subset of $\{1, 2, 3\}$ adds up to 5 is "yes" (or 1), because $2 + 3 = 5$.

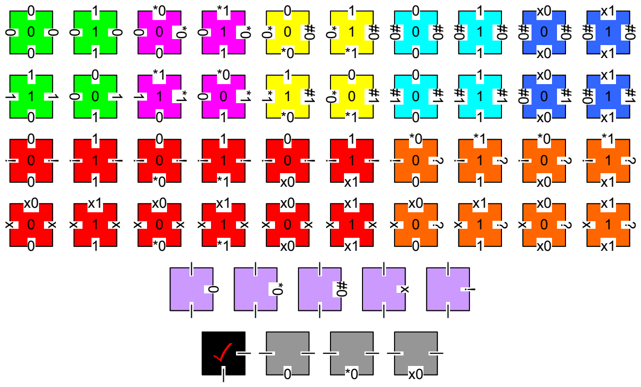

Figure 4: The *SubsetSum*-solving tile assembly has 49 computational tile types.

The nature of NP-complete problems is that if one can solve one such problem quickly, then one can solve all such problems quickly. For example, if one finds a polynomial-time algorithm to solve *SubsetSum*, one can now solve the traveling salesman, 3-*SAT*, protein folding, and all other NP problems in polynomial time. Thus, designing a tile assembly that solves one NP-complete problem is in some sense sufficient to solve all other NP-complete problems.

The tile assembly we present to solve *SubsetSum* uses 49 computational tile types, shown in Figure 4. (There are also 7 tile types used to encode the input.) These tile types are carefully designed to solve this problem, as we describe in [8].

The *SubsetSum* tile assembly is really a combination of four tile assemblies, designed to work together. Each of the four assemblies, respectively, (1) nondeterministically selects whether or not to subtract the next number, either (2) subtracts or (3) does not subtract the next number, and (4) checks if all the subtractions completed correctly and if the final result is 0. In combination, these assemblies nondeterministically select a subset of the input set of numbers, subtract each of the numbers in the subset from the target number, and determine if the result is 0. If the result is, in fact, 0, the assembly has found a subset of numbers that adds up to the target number and a special ✓ tile attaches to indicate success.

Figure 5(a) shows a sample seed configuration that encodes, in binary, the set of numbers $\{11, 25, 37, 39\}$ and the target number 75. Because $75 = 11+25+39$, one nondeterministic execution of the tile system finds the proper selection of numbers and attaches the special ✓ tile. Figure 5(b) shows one such execution with the ✓ tile attached in the northwest corner. If there were no subset of $\{11, 25, 37, 39\}$ whose sum equaled 75, the attaching tiles would create an incomplete configuration to which neither the ✓ tile nor any other tile could attach. Therefore, the ✓ tile's ability to attach to at least one of the seeds indicates a positive answer to the problem, whereas the inability to attach indicates a

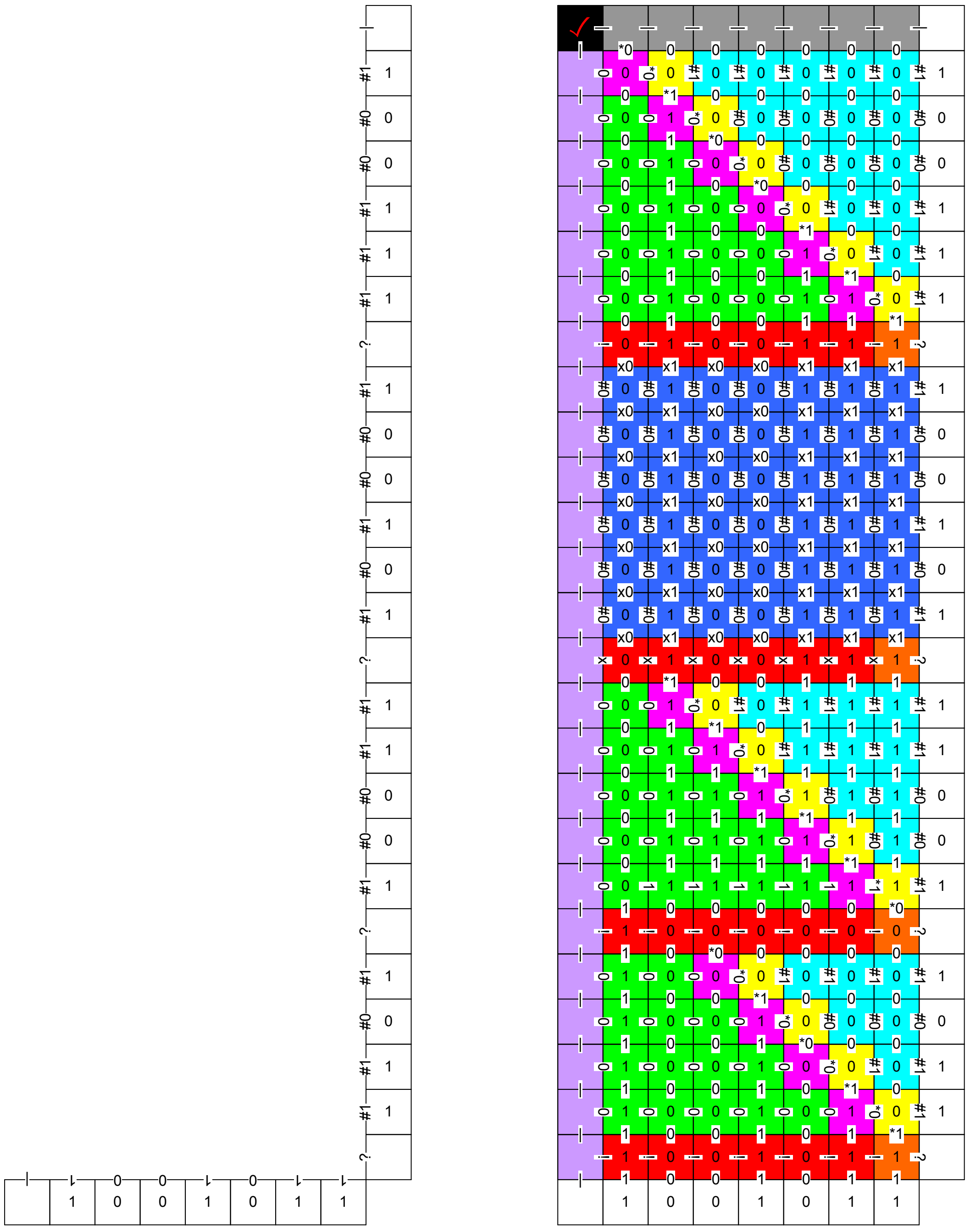
(a) (b)

Figure 5: An example execution of the tile assembly that solves *SubsetSum*. The seed configuration (a) encodes the input in binary: a set of numbers: $\{11 = 1011_2, 25 = 11001_2, 37 = 100101_2, 39 = 100111_2\}$ along the right column, and a target number $75 = 1001011_2$ along the bottom row. Because $75 = 11 + 25 + 39$, one nondeterministic execution (b) of the tile assembly finds the proper selection of numbers and attaches the special ✓ tile in the northwest corner. If there were no subset of numbers whose sum equaled 75, no such tile could attach.

negative answer. We refer the reader to [8] for the full description and proof of the *SubsetSum*-solving tile assembly.

3. TILE ARCHITECTURAL STYLE

The tile assembly model has turned out to be a fruitful environment for research in theoretical computer science and biological-agent computing, such as DNA computation. In this section, we describe how the algorithms and paradigms inspired by crystal growth and made formal by the tile assembly model can be used to develop distributed software systems that solve complex computational problems, such as NP-complete problems. We will first, in Section 3.1, describe a domain of systems and problems that can benefit from crystal-growth algorithms, and then, in Section 3.2, describe how particular aspects of crystal growth can benefit distributed software systems.

3.1 Computational Grids

Solving computationally intensive problems is integral to modern research in artificial intelligence, physics, astrophysics, bioinformatics, disease and drug design, economics, networking, neuroscience, system biology, and a number of other fields [5, 15]. To satisfy this extensive need for fast computation, researchers have turned to *computational grid systems*, which pool hardware and software resources to deliver computation as a standardized service. For example, computational grids can leverage public networks to combine machines with idle cycles into single-interface distributed computing systems [3]. The organization of loosely coupled networked devices achieved by computational grids has permitted advances in science and engineering that would otherwise not be possible [16].

There are many existing solutions to the problem of aggregating the computational power of machines into a single computational grid. For example, BOINC is an autonomic grid platform currently deployed on over a million computers. Examples of BOINC applications include SETI@home, LHC@home, Folding@home, Quantum Monte Carlo at Home, Malariacontrol.net, Climateprediction.net, PrimeGrid, and many others [3]. BIONC applications allow users around the world to volunteer their computers' idle cycles to solve small parts of large problems and use somewhat centralized (though independent for each application) servers to distribute that computation. Another example of a successful computational grid technology is Google's MapReduce [13]. Developers who use MapReduce to create applications design two functions: `Map` and `Reduce`. `Map` divides a problem into a small number of subproblems that can be solved in parallel and `Reduce` takes the solutions to several subproblems and combines them into a solution to the original problem. The MapReduce infrastructure then handles taking a single problem, distributing it by recursively dividing it into smaller and smaller subproblems, and recursively combining the solutions to the subproblems into the final answer. Commercial uses of MapReduce include computing Google's PageRank. As many as a thousand MapReduce jobs are executed on Google's clusters daily [13].

3.2 Crystal-Growth Benefits

Computational grids coordinate distributed federated resources on a network and combine them into a single-interface computational resource. This allows users to submit a computation to a single place and have it distributed onto a large number of computing devices. The exiting methods typically rely on the trustworthiness of the underlying computers by either using nodes located on a single trusted cluster (e.g., a cluster owned by Google [13]) or replicating and performing the same computation on several independent nodes to ensure quality [3]. These methods often require a single central node to have contact with every other node on the network [3] and can withstand only easily detectable failures. We propose a fundamentally different approach to distributing computation, one based on the nature's process of crystal growth, that ensures scalability, privacy of data, and tolerance of a wide range of faults and malicious attacks.

Suppose a user wishes to solve an NP-complete problem[1]. The user could first convert, using a standard reduction [22], the problem to a different NP-complete problem for which there exists a tile assembly, such as the *SubsetSum* assembly from Section 2.4 or the 3-SAT assembly from [9].

Armed with a tile assembly for solving a problem, one could imagine a virtual beaker that contains many copies of each tile type from that tile assembly. The user could drop a single seed into this virtual beaker, initiating two processes: replication and recruitment. *Replication* is the autonomous copying of a seed, whereas *recruitment* is the attachment to a growing seed of a tile that matches its neighbors on a sufficient number of sides, as we explain further in Sections 3.2.1 and 3.2.2. After some time, the imaginary beaker would contain many copies of the seed (if each replication takes some constant amount of time, then the number of replicas would be exponential in the elapsed time), and each seed would have recruited some tile attachments, exploring one nondeterministic execution of the tile-assembly algorithm. Eventually, either some replica finds the positive solution or enough replicas exist and have not found the positive solution that, with high probability, a positive solution does not exist.

The beaker thought experiment is one way to solve the user's problem that provides the possibility of interesting properties that traditional software approaches lack. First, each of the contributing tiles knows very little about the entire structure of the problem; at most, a tile might know a single bit of the input or some bit internal to the computation, and perhaps which neighbors it is attached to, but would know neither a significant part of the input nor the significance of the data it does know. We call this property *privacy preservation*. Second, if some of the tiles were faulty and misattached, or even if some tiles were intelligent, malicious, collaborating agents, existing techniques for error correction developed for the tile assembly model [25] could quickly reduce the probability that those faults and attacks would break the system. We call this property *fault and malice tolerance*. Third, because each of the seeds acts independently of the others, under the proper conditions (e.g., no competition for tiles), the system scales as well as a system can scale for large inputs. We call this property *scalability*.

We now explain how nodes on a distributed network can benefit from the ideas of our beaker thought experiment to self-organize into a distributed computational grid. We call the set of principles for designing distributed software systems using these ideas the *tile architectural style* [11].

Each computer on the network will be responsible for de-

[1]The approach we describe should extend to other computationally-intensive problems as well, but we limit our discussion to NP-complete problems for now.

ploying copies of a single tile type. Thus, for example, for the *SubsetSum* tile assembly, for each of the $49 + 7 = 56$ tile types, every 56th computer will be deploying tiles of that type. At the start of the computation, the user will create a *tile type map* that maps the IPs to tile types and distribute that map to all the nodes by broadcasting it to its neighbors, who will then broadcast it to their neighbors, and so on. The user will then create a single seed on the network that encodes the input by asking computers that deploy the appropriate tile types to deploy tiles and attach to their appropriate neighbors.

Next, the distributed system takes over the distribution and execution of the computation through the processes of replication and recruitment.

3.2.1 Replication

For each tile within a seed, the computer that deploys that tile will participate in the replication procedure. That computer, called the parent computer, will use the tile type map to select a different computer on the network, called the child computer, that deploys the same type of tile. The parent will ask the child to deploy a tile. If the child has adequate resources and agrees, the new tile will represent the replica of the parent's tile.

Once the parent's neighbors perform the same procedure and obtain replicas, these replicas must be connected to form a standalone seed. The parents relay the identities of their children to the children of their neighbors, completing the replication procedure.

Each of the standalone seeds can replicate in parallel, growing the number of independent seeds exponentially, until the computing network's resources are used up. At the same time, each of the seeds performs recruitment, as we describe next. Whenever seeds finish recruiting, the resources they use can be freed to allow more replication.

3.2.2 Recruitment

Computers deploying adjacent tiles without appropriate neighbors will attempt to recruit other computers to deploy those neighbors. These computers deploying adjacent tiles will survey computers that deploy tiles of various types to find one that may attach. That computer will deploy a tile and attach to its new neighbors. The seeds, thus, grow independently to execute the computation and determine the solution, which the node deploying the special ✓ tile reports to the user.

4. EVALUATING THE TILE STYLE

The tile style can be evaluated along three dimensions: privacy preservation, fault tolerance, and efficiency. Because the tile style is based on a formal mathematical model of crystal growth, desirable properties of tile-style systems can be proven formally, as well as demonstrated empirically. For example, demonstrating what fraction of the network a spy must compromise in order to learn a private input and what fraction of the network an adversary must compromise to break the computation indicate privacy preservation and fault tolerance. Further, a tile-style implementation can empirically determine the speed and efficiency in a real-world environment.

We anticipate the tile style to be highly efficient and scalable. One interesting and common intuition about tile-style systems that contradicts our expectation of efficiency is that these systems will be significantly slower than traditional distributed systems because data that typically is internal to a single processor must be sent over the network to other nodes, and network communication is far slower than communication internal to a single computer. However, this intuition is incorrect. Each node in a tile-style system deploys many tiles (of the same tile type) at once, and thus participates in many of the nondeterministic executions in parallel. Because best known algorithms for NP-complete problems require an exponential number of executions (either parallel or sequential) and because tiles are lightweight, each node can easily deploy tens of thousands of tiles. Thus, while some tiles have to wait for network communication, other tiles can be executing. The result is that no node is ever idly waiting and the system executes virtually as fast as it would if network communication were instantaneous (with the exception of latency, which is insignificant for reasonably sized computations).

5. SUMMARY

We have given a high-level description of the nature-inspired algorithms that allow the distribution of computation onto distributed networks in a scalable, privacy-preserving, and fault- and malice-tolerant manner. We have only provided the intuition of how the system will work here and have omitted numerous details that go along with turning these ideas into an actual running software system. These details, along with empirical data on the executions of these systems and their efficiency, as well as comparisons with traditional computational grid technologies, can be found in [11].

6. REFERENCES

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. F. Knight, Jr., R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss. Amorphous computing. *Communications of the ACM*, 43(5):74–82, May 2000.

[2] L. Adleman. Towards a mathematical theory of self-assembly. Technical Report 00-722, Department of Computer Science, University of Southern California, Los Angeles, CA, 2000.

[3] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID04)*, pages 4–10, Pittsburgh, PA, USA, 2004.

[4] R. Barish, P. W. K. Rothemund, and E. Winfree. Two computational primitives for algorithmic self-assembly: Copying and counting. *Nano Letters*, 5(12):2586–2592, 2005.

[5] B. Berger and T. Leighton. Protein folding in the hydrophobic-hydrophilic (HP) is NP-complete. In *Proceedings of the 2nd Annual International Conference on Computational Molecular Biology (RECOMB98)*, pages 30–39, New York, NY, USA, March 1998.

[6] Y. Brun. Arithmetic computation in the tile assembly model: Addition and multiplication. *Theoretical Computer Science*, 378(1):17–31, June 2007.

[7] Y. Brun. Nondeterministic polynomial time factoring in the tile assembly model. *Theoretical Computer Science*, 395(1):3–23, 2008.

[8] Y. Brun. Solving NP-complete problems in the tile assembly model. *Theoretical Computer Science*, 395(1):31–46, 2008.

[9] Y. Brun. Solving satisfiability in the tile assembly model with a constant-size tileset. *Journal of Algorithms*, 63(4):151–166, 2008.

[10] Y. Brun and N. Medvidovic. Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In *Proceedings of the 2nd International Workshop on Engineering Fault Tolerant Systems (EFTS07)*, pages 38–43, Dubrovnik, Croatia, September 2007.

[11] Y. Brun and N. Medvidovic. Preserving privacy in distributed computation via self-assembly. Technical Report USC-CSSE-2008-819, Center for Software Engineering, University of Southern California, 2008.

[12] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litiou, H. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems. In B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*. Lecture Notes in Computer Science, In Press, 2009.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI04)*, San Francisco, CA, USA, December 2004.

[14] J. Korb and K. E. Linsenmair. Thermoregulation of termite mounds: what role does ambient temperature and metabolism of the colony play? *Journal Insectes Sociaux*, 47(4):357–363, November 2000.

[15] M. Lamanna. The LHC computing grid project at CERN. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 534(1-2):1–6, 2004.

[16] D. Lucent, V. Vishal, and V. S. Pande. Protein folding under confinement: A role for solvent. *Proceedings of the National Academy of Sciences*, 104(25):10430–10434, 2007.

[17] P. McKinley, B. H. Cheng, C. Ofria, D. Knoester, B. Beckmann, and H. Goldsby. Harnessing digital evolution. *IEEE Computer*, 41(1):54–63, 2008.

[18] S. Raudys and M. Tamosiunaite. Biologically inspired architecture of feedforward networks for signal classification. In *Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition*, pages 727–736, Alicante, Spain, 2000. Springer-Verlag.

[19] D. Reishus. Design of a self-assembled memory circuit. In *Proceedings of the 5th Foundations of Nanoscience: Self-Assembled Architectures and Devices (FNANO08)*, pages 239–246, Snowbird, UT, USA, April 2008.

[20] P. W. K. Rothemund, N. Papadakis, and E. Winfree. Algorithmic self-assembly of DNA Sierpinski triangles. *PLoS Biology*, 2(12):e424, 2004.

[21] W.-M. Shen, M. Krivokon, H. Chiu, J. Everist, M. Rubenstein, and J. Venkatesh. Multimode locomotion for reconfigurable robots. *Autonomous Robots*, 20(2):165–177, 2006.

[22] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[23] E. Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, Pasadena, CA, USA, June 1998.

[24] E. Winfree. Simulations of computing by self-assembly of DNA. Technical Report CS-TR:1998:22, California Institute of Technology, Pasadena, CA, USA, 1998.

[25] E. Winfree and R. Bekbolatov. Proofreading tile sets: Error correction for algorithmic self-assembly. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS02)*, volume 2943, pages 126–144, Madison, WI, USA, June 2003.

[26] B. Yurke, A. Turberfield, A. Mills, Jr., F. Simmel, and J. Neumann. A DNA-fuelled molecular machine made of DNA. *Nature*, 404:605Ű–608, 2000.