Relentless Computing: Enabling fault-tolerant, numerically intensive computation in distributed environments

Lucas A. Wilson and John A. Lockman III
Texas Advanced Computing Center, The University of Texas at Austin, Austin, Texas, U.S.A.

Abstract— *This paper suggests a novel computational paradigm for solving numerically intensive problems on a distributed infrastructure. We detail the basic functionality of this new paradigm, its ability to recover from host loss without requiring a complete restart of the code, and how it could allow for many heterogeneous participants to solve a single, large-scale computational problem. We provide results from a small demonstration run as well as provide avenues for future research.*

Keywords: Volunteer Computing, Distributed Computing, Fault Tolerance, Distributed Hash Tables

1. Introduction

Computer-based simulation and modeling is becoming critical for driving scientific breakthrough and discovery. As the sensitivity and scale of simulations increase, the computational requirements and time-to-solution also rises. Unfortunately modern hardware – although much improved over technologies of several years ago – does not provide researchers with a stable execution platform for simulations requiring weeks or months of computation to complete, and is extremely expensive to deploy in large-scale, tightly-coupled environments. As a result, computer-based simulation for scientific discovery has remained limited to those researchers who have access to high-performance systems at Universities and National Laboratories.

Fully-distributed volunteer computing models, such as the Berkeley Open Infrastructure for Network Computing (BOINC)[1], have provided a means of performing massive-scale computation on a limited subset of problems involving limited-to-no data sharing among participants. While BOINC and similar volunteer computing models have successfully computed millions of hours of user code, the types of problems that can be executed in such an environment are limited and cannot fully enable new scientific breakthroughs.

Distributed Hash Table (DHT) implementations, such as Chord[2], Pastry[3], Kademlia[4], and others[5][6][7][8], provide mechanisms for storing key/value pairs in a decentralized fashion, preventing the failure of any single participant from killing the entire hash table. DHTs are commonly used in distributed file systems[9][10], peer-to-peer file sharing systems[11], and domain name services[12][13].

Volunteer computing models can harness the untapped computing potential of millions of part-time citizen scientists. We propose a system that would couple this potential with the innately fault-tolerant nature of DHTs, allowing for the execution of programs for extremely long periods of time, with built-in failure recovery in the event any set of participants was unable or unwilling to continue contributing. Additionally, proper data partitioning would allow for problems requiring more onerous data sharing among participants to be executed, increasing the potential for more scientific discoveries.

This work describes a new computational paradigm: Relentless computing. With Relentless Computing, traditionally tightly-coupled, numerically-intensive parallel computations can be performed in a decentralized, distributed environment with high fault-tolerance. So long as any single participant and the initial data are present to the system, computation will continue. We will provide a basic description of Relentless Computing, how code is generated and managed, and how global shared memory is implemented through DHTs. We will also provide results from a test case solving a partial differential equation (PDE) using finite differences, as well as outline avenues for future research.

2. Related Work

While existing distributed computing systems, such as BOINC[1], have been extremely useful for executing completely data-parallel computations, such as Monte Carlo and parametric sweeps, they are ill-suited for handling single, large-scale computations that require data sharing among participants. On the other side of the spectrum, Adaptive-MPI (AMPI)[14], which is based on the Charm++ framework[15], allows for the creation of medium-grained virtualized processes which can be overloaded on a single physical processor in order to overlap computation and communication. AMPI does provide many facilities similar to Relentless Computing, including the ability to shrink/expand the number of computational participants and checkpointing of virtual processors to disk. However, AMPI is not designed to handle code written in different languages, to recover from near-catastrophic node failure without the use of a restart file, or to allow very fine-grained parallelism that enables contributions from low-power participants (e.g. "smart" phones, tablets, netbooks, portable computers) without these devices adversely effecting the overall performance of the system.

GRID-GUM[16] implements Glasgow Parallel Haskell[17] on top of the Globus Toolkit[18], making

use of MPICH-G2[19] for handling the underlying process communications. While this method helps to abstract the parallel computation away from the programmer, the use of MPICH-G2 makes participant shrink/expand nearly impossible, and does not allow for fault-tolerance in the event of near-catastrophic node failure.

The Partitioned Global Address Space (PGAS) model, on top of which languages such as X10[20], Chapel[21] UPC[22] and others[23][24][25] are built, provides a method of abstracting a virtual shared memory platform on top of distributed memory architectures. Currently, PGAS languages running on distributed memory clusters rely on existing message-passing methods, such as MPI[26], to handle the cross-node communication that synchronizes virtual global memory and migrates process threads. Because of this, the fault-tolerance and dynamic capabilities of PGAS languages are limited to the capabilities of the underlying communications framework, of which little currently exists.

Global Arrays[27] provide yet another way of abstracting shared memory on top of a distributed memory architecture. However, memory synchronization is once again dependent on an existing communication framework, either MPI or ARMCI[28], which provide little-to-no capability to shrink/expand the participant pool dynamically at runtime, seamlessly recover from near-catastrophic host failure, or efficiently function over high-latency, low-bandwidth networks.

TStreams (also called Concurrent Collections (CnC)[29]) provide a model of describing computation in terms of serial execution components and data-flow specifications[30], in much the same way that Relentless Computing does. While TStreams provides facilities for creating static checkpoints, we are not aware of any particular implementation of this model that provides fault tolerance that enables continuing execution in the face of hardware failures, or that is designed with high-latency, low-bandwidth interconnects in mind. TStreams is also not specifically designed to incorporate low power consumer devices into the participant pool, or to enable execution on fickle participants that may only allow for the use of a fraction of total cycles to be consumed, for limited amounts of time.

The Linda coordination language allows for the separation of coordinate and computation by placing information into an external tuple-space data store, allowing computation from multiple languages to interact[31]. Fault tolerance mechanisms have also been proposed for Linda[32]. Much of the published work found by the authors on Linda is over a decade old, and many of the principles in Linda are incorporated into Relentless Computing. Relentless computing has been designed from the start to work on highly distributed, Internet-connected devices of varying computation capability with the ability to shrink/expand the participant pool at will, as well as recover from near-catastrophic failures.

3. The Relentless Computing Model

The Relentless Computing model seeks to leverage the untapped computing potential of various hardware resources, all connected to the Internet by some mechanism (hardwire, wireless, cellular). The use of high-latency, low-bandwidth network connections requires that computation remain limited to highly partitioned, small pieces of data to allow for reasonable read/write from/to the DHT. Programs are written as codelets (self-contained pieces of code) chained together by data dependency. When a hardware resource volunteers to participate in the solving of a particular problem, the solution is sought in a bottom-up fashion, with the participant seeking to complete the result first. If a participant is unable to build the result, it steps up the dependency chain until a data part that can successfully be computed is found. Code written for this environment are built in two pieces: (1) A set of multi-language codelets, which can interact with each other through global shared memory implemented in the DHT, and (2) a descriptive framework that determines the order in which these codelets are to run, and the data dependencies that chain them together.

3.1 Codelets

Each codelet is a self-contained piece of code that performs a set of sequential load/compute/store operations. Because there is no direct interaction between codelets (i.e. each codelet is independent of each other, with interaction performed through data sharing), codelets can be constructed in different languages. This allows for development teams to be able to work in the languages that members are most comfortable with, without worrying about the issues involved in coupling different languages together in traditional software development. Additionally, reducing the codelet size and it's associated data dependencies allows for out of order execution of codelets to occur, provided the input data is available.

Codelets can be precompiled in a compilable language, so long as they are capable of executing on the participant hardware. Scripting languages can also be utilized, assuming a mechanism exists to execute the script code from the compute daemon. Runtime environments for some scripting languages (e.g. JavaScript via SpiderMonkey[33], Python[34], Lua[35], LISP via ECL[36], etc.) could be embedded directly into the compute daemon to allow the daemon to compute results directly.

3.2 Memory Management

Relentless computing environments (RCEs) create a global shared memory space from which codelets can read data and to which codelets can write data. This global shared memory space is implemented as key/value tuples to the DHT, with the key replacing the variable name/address, and the value representing the stored data. This is similar to the tuple-space data storage methods employed in

the coordination language Linda[31]. In order to eliminate possible side-effects associated with uncoordinated writes to memory each codelet must be deterministic, with one set of inputs guaranteed to produce the same output. This allows for the possibility of multiple resources executing the same codelet instance (perhaps because both attempted to solve it simultaneously, or knowledge that a particular data point had already been computed was temporarily lost). In order to reduce memory bloat, entries in the DHT will be given a lifetime (e.g. 24 or 48 hours). During the lifetime of the data, codelets can use that information as input for other computation. After that time the data will be deleted, and any participant requiring that particular key/value pair will be required to recompute it.

3.3 Problem Description Framework

In order to chain together multiple codelets and have them interact with the DHT-implemented global shared memory space, a problem description framework must exist that allows Relentless Computing daemons to determine which codelets to run and with which data to run them. This framework must provide the ability to easily define inputs and outputs, as well as specify the particular codelet to execute for each input/output set. Additionally, a result component must be specified so that participants know which data elements are considered final, providing a starting point from which execution can begin. One choice for this is to create an extensible framework language with the Extensible Markup Language (XML)[37]. Not only does XML provide the extensibility to add new features and constructs easily, it is well accepted and understood by the community and is easily compressible, allowing for faster transmission between compute daemons.

A potential problem description for solving the 1-dimensional heat equation using the Forward in Time, Central in Space (FTCS) method may look like Figure 1. In this case, we have assumed that the compute daemon can natively interpret JavaScript code and that the boundary values have already been inserted into the system. Each time the codelet is executed, it requires three inputs (denoted by the `depends-on` tags): the (l)eft, (m)iddle, and (r)ight values from the previous timestep and outputs a single value $u[x][t]$. Function parameters are linked to values in the `depends-on` tags in order from top to bottom, so parameter `l` is associated with $u[x-1][t-1]$, `m` is associated with $u[x][t-1]$, and `r` is associated with $u[x+1][t-1]$.

When a user submits a job containing a problem description and codelets to an RCE, that problem description will then spread across the network to various participants using a gossip protocol [38]. Once other participants are made aware of the new problem, they can begin solving it as well. Each participant – responsible for both starting codelets as well as participating in the DHT – has no advance knowledge about the current state of the problem. In order to determine which codelet to run while avoiding duplication of effort, the computing daemon parses the work-flow in a bottom-up fashion, beginning with the result codelet and working its way back up the dependency chain until a codelet that is capable of being executed (but that has yet to be executed) is discovered. Once a codelet has been executed and the resulting data stored in the DHT, the compute daemon begins again with the result codelet, working back up the dependency chain in order to take advantage of other more recent dependencies that may have been computed in parallel. The process diagram of an RCE daemon is shown in Figure 2.

```
<problem name=heat_transfer>
  <codelet name='finite_diff'>
    <result/>
    <source lang='javascript'>
      <![CDATA[
        function finDiff(l, m, r) {
          return m + 0.25*(l - 2*m + r);
        }
      ]]>
    <parameter name='x' range='0..99'/>
    <parameter name='t' range='1..99'/>
    <depends-on name='u[x-1][t-1]'/>
    <depends-on name='u[x][t-1]'/>
    <depends-on name='u[x+1][t-1]'/>
    <output name='u[x][t]'/>
  </codelet>
</problem>
```

Fig. 1: Potential Problem Description

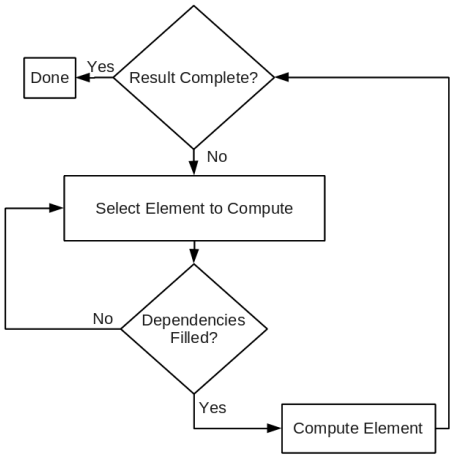
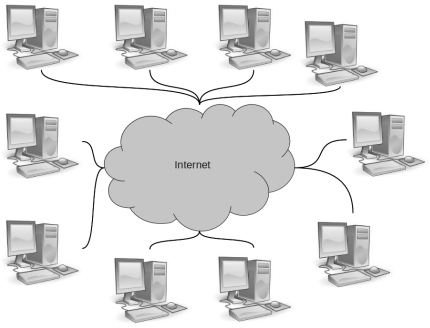
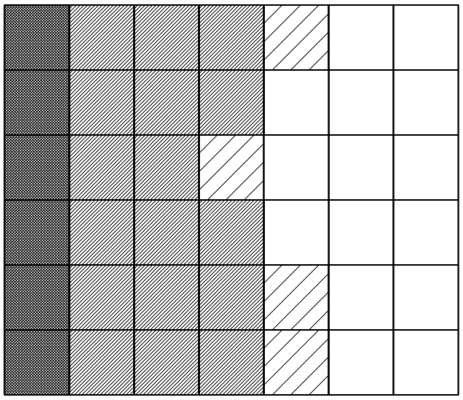


Fig. 2: Process Diagram for RCE daemon

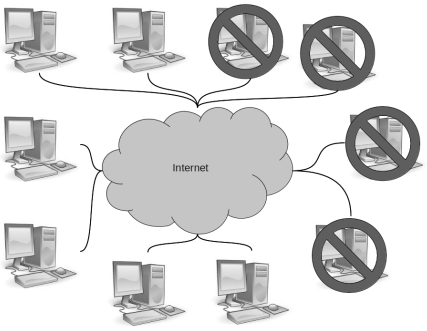


(a) Participants

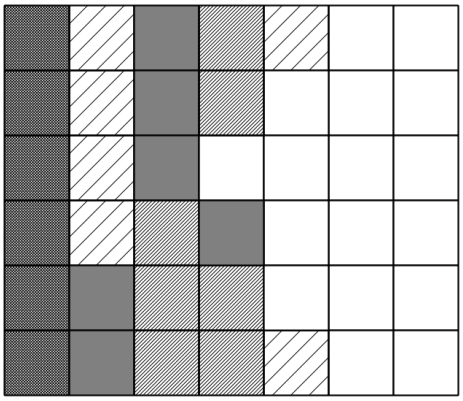


(b) Distributed Hash Table

Fig. 3: Active Compute Collective



(a) Participants



(b) Distributed Hash Table

Fig. 4: Compute Collective After Node Failures

3.4 Managing Node Failure

In any large-scale distributed environment, node failure is a constant risk that cannot be ignored. In most high-performance machines and with typical multi-thread/process computing paradigms (Pthreads, OpenMP[39], MPI[26], Parallel Virtual Machine (PVM)[40], PGAS languages[20][21][22][23][24][25]), the loss of a host or process results in the simultaneous aborting of all processes associated with a problem. This error handling method may be sufficient in environments with relatively high uptime guarantees. However, distributed volunteer computing environments provide no such guarantee. As a result, more effective fault-tolerance mechanisms must be employed.

DHTs are by their nature relatively fault-tolerant. So long as any single host remains, part of the DHT still exists. The loss of any single participant does not destroy the entire table. The compute daemons of the proposed system would also be the DHT participants, each locally storing part of the hash table in addition to volunteering computational cycles.

Figure 3 shows an example of a volunteer collective working on a particular problem, for example a finite difference problem with a 3-point central difference in the space dimension (vertical axis) and a forward difference in the time dimension (horizontal axis). In this example, many participants (Figure 3(a)) are working on various parts of the problem (represented by the logical matrix in Figure 3(b)). The leftmost column values are the initial data, the densely striped values are already computed pieces, and the sparsely striped values are those pieces that can currently be computed with the data that already exists. The boundary values are not shown in this illustration, but can be considered initial data if the boundaries are constant.

If, before the next step can be computed, several of the participants fall offline (Figure 4(a)), some of the data stored in the DHT would be lost (Figure 4(b), gray values). In this case, participants would be unable to compute all of the values that were previously possible due to loss of input data. Instead, they would continue on as though those values never existed, in some cases needing to return to the input data in order to compute the necessary intermediate values.

4. Experiment Setup

In order to test the viability of the proposed paradigm in solving a traditional numerically intensive problem, a prototype RCE daemon was written in Python using the Entangled [41] library to perform the base DHT operations. The

Table 1: Breakdown of participant contributions

Participant	Contributions	Percentage
0	2841	27.3
1	2629	25.3
2	2462	23.7
3	2468	23.7
Total	10400	100

prototype daemon was written to solve the one-dimensional heat equation using the FTCS method:

$$u_x^{t+1} = u_x^t + r(u_{x-1}^t - 2u_x^t + u_{x+1}^t, \qquad r = \frac{\alpha \Delta t}{\Delta x^2}$$

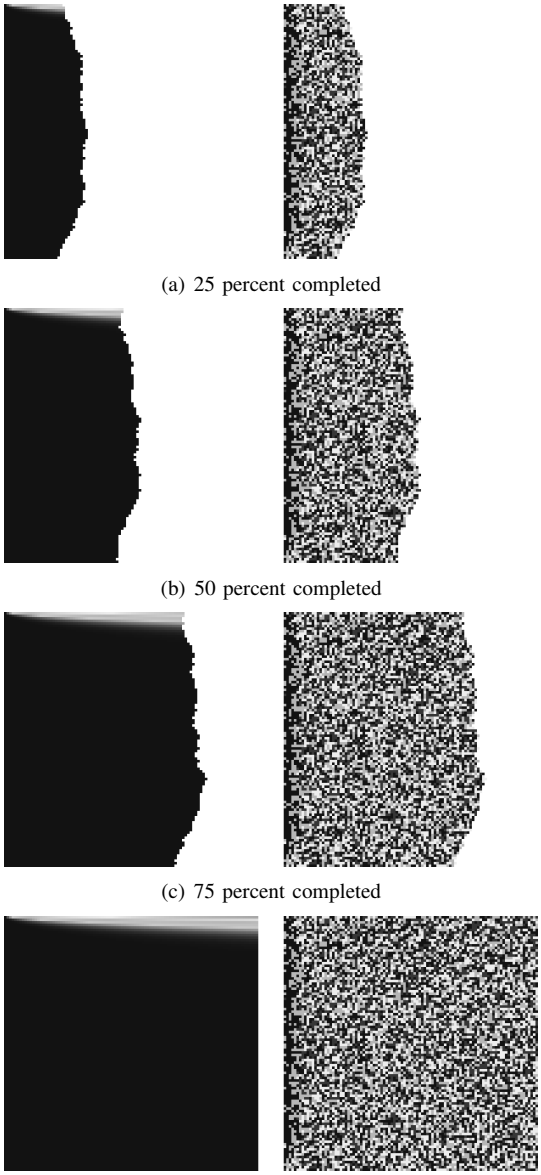
In this case, each element at time $t+1$ is computed by looking at the corresponding values of itself and its neighbors at time t, meaning each element computed needs three inputs from the previous time-step.

For this test, a constant heat source was placed in the leading boundary ($x = -1$), while the trailing boundary was set to 0. The initial ($t = 0$) temperature of the system is set to 0. Experiments were run for a 100x100 case (100 spatial units for 100 time-steps). For this experiment, 4 participants (2 nodes with 2 participants each) were used. In order to test the ability of the system to handle new participants joining mid-computation, participant 0 was initially alone, with participant 1 added next, followed by participants 2 and 3. A communication error (host disconnected from then reconnected to the Internet) was introduced several minutes into the simulation to test the RCE's failure recovery capability.

5. Results

Data collected from a 4 participant run were graphed based on times at which data was stored in the DHT, with both solution values and contributing participant recorded. In Figure 5, the left panels show the solution values to the heat problem, while the right panels show which participant contributed that particular element to the solution. As can be seen, the overall problem was solved in non-linear order, with some sections of the solution growing faster than others. Additionally, the right panels show that work was well distributed, with participant 0 naturally performing more operations than the others, and participants 2 and 3 nearly equal (they joined the computation at the same time) (see Table 1). Table 1 also shows total contributions of 10,400, while the total number of cells in the 100x100 system is 10,000. This means that 400 elements, or 4%, were recomputed for various reasons including simultaneous attempts to calculate a specific element, and the test communication failure that prevented participants from querying the full DHT.

Of the 400 elements that were recomputed, 356 were recomputed only once, while 22 were recomputed twice. Times between recomputations varied widely, with a maximum time between first element computation and final recomputation of 683.62 seconds (see Table 2 and Figure 6). The average recomputation time was 104 seconds, with a standard deviation of 119 seconds.



(a) 25 percent completed

(b) 50 percent completed

(c) 75 percent completed

(d) 100 percent completed

Fig. 5: RCE computing solution to 1-D heat equation

6. Conclusions

We have proposed a novel computational paradigm called Relentless Computing. This new paradigm allows users to develop codes that solve numerically intensive problems on more disparate and distributed resources, as well as provide the ability for dynamic expanding and shrinking of the participant pool. This model of computation provides fault tolerance, in that the code will continue to execute

Table 2: Time Between Recomputations

Metric	Time (secs.)
Max	683.62
Average	104.78
Std. Dev.	119.12
Median	56.82

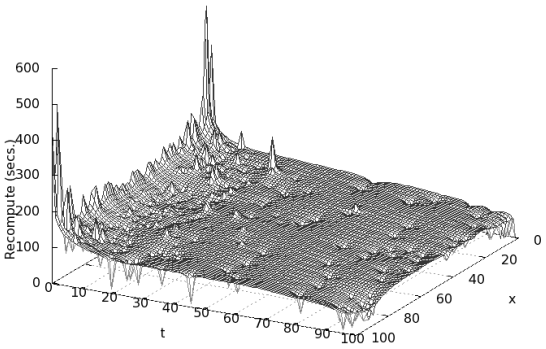


Fig. 6: Times Between Initial and Final Recomputation for Each Element

so long as the initial data and a single participant remain online. Early experimental results have shown that this paradigm provides a relatively simple way for computing numerically intensive problems in a distributed, decentralized, dynamic, and fault-tolerant fashion without requiring excessive work from the programmer. Future work could include development of a standardized problem description framework language, porting of traditional scientific codes to this paradigm, further optimization of the RCE daemon to be more computationally efficient, and inclusion of multiple language runtime environments into the daemon to allow for codelets written in script and interpreted languages to be executed directly by said daemon.

References

[1] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, ser. GRID '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10. [Online]. Available: http://dx.doi.org/10.1109/GRID.2004.14

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160. [Online]. Available: http://doi.acm.org/10.1145/383059.383071

[3] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *MIDDLEWARE 2001*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001, vol. 2218/2001, pp. 329–350.

[4] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. London, UK: Springer-Verlag, 2002, pp. 53–65. [Online]. Available: http://portal.acm.org/citation.cfm?id=646334.687801

[5] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt, "P-Grid: a self-organizing structured P2P system," *SIGMOD Rec.*, vol. 32, pp. 29–33, September 2003.

[6] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *Selected Areas in Communications, IEEE Journal on*, vol. 22, no. 1, pp. 41 – 53, January 2004.

[7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 161–172, August 2001.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, October 2007.

[9] P. Druschel and A. Rowstron, "PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility," *Hot Topics in Operating Systems, Workshop on*, vol. 0, p. 0075, 2001.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 202–215.

[11] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The Bittorrent P2P File-Sharing System: Measurements and Analysis," in *Peer-to-Peer Systems IV*, ser. Lecture Notes in Computer Science, M. Castro and R. van Renesse, Eds. Springer Berlin / Heidelberg, 2005, vol. 3640, pp. 205–216.

[12] V. Ramasubramanian and E. G. Sirer, "The design and implementation of a next generation name service for the internet," in *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '04. New York, NY, USA: ACM, 2004, pp. 331–342. [Online]. Available: http://doi.acm.org/10.1145/1015467.1015504

[13] Y. Doi, "DNS Meets DHT: Treating Massive ID Resolution Using DNS Over DHT," *Applications and the Internet, IEEE/IPSJ International Symposium on*, vol. 0, pp. 9–15, 2005.

[14] C. Huang, O. Lawlor, and L. V. KalÃl', "Adaptive MPI," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, L. Rauchwerger, Ed. Springer Berlin / Heidelberg, 2004, vol. 2958, pp. 306–322.

[15] L. V. Kale and S. Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 91–108.

[16] G. M. A. D. Al Zain, P.W. Trinder and H.-W. Loidl, "Managing heterogeneity in a grid parallel haskell," *Scalable Computing: Practice and Experience*, vol. 7, pp. 9–25, September 2006.

[17] J. G. Hall, C. Baker-Finch, P. Trinder, and D. J. King, "Towards an operational semantics for a parallel non-strict functional language," in *Proceedings of the International Workshop on the Implementation of Functional Languages (IFL'98)*, September 1998. [Online]. Available: http://mcs.open.ac.uk/djk26/apset/transitionsystem.ps

[18] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Jounral of High Performance Computing Applications*, vol. 11, pp. 115–128, 1997.

[19] N. T. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-enabled implementation of the Message Passing Interface," *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 551 – 563, 2003, special Issue on Computational Grids. [Online]. Available: http://www.sciencedirect.com/science/article/B6WKJ-48BKT9V-1/2/6462834e0f7d0175d57043bbf3df8a80

[20] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented program-*

ming, systems, languages, and applications, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538.

[21] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, August 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1286120.1286123

[22] T. El-Ghazawi and L. Smith, "UPC: unified parallel C," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.

[23] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: a high-performance java dialect," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, pp. 825–836, 1998.

[24] E. Allen, D. Chase, J. Hallett, V. Luchango, J.-W. Maessen, S. Ryu, G. S. Jr., and S. Tobin-Hochstadt, "The Fortress Language Specification," 2008. [Online]. Available: http://labs.oracle.com/projects/plrg/Publications/fortress.1.0.pdf

[25] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, pp. 1–31, August 1998.

[26] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, MA, USA: MIT Press, 1994.

[27] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: a portable "shared-memory" programming model for distributed memory computers," in *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '94. New York, NY, USA: ACM, 1994, pp. 340–349.

[28] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Libray for Ditributed Array Libraries and Compiler Run-Time Systems," in *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*. London, UK: Springer-Verlag, 1999, pp. 533–546. [Online]. Available: http://portal.acm.org/citation.cfm?id=645611.662053

[29] K. Knobe, "Ease of use with concurrent collections (CnC)," in *Proceedings of the First USENIX conference on Hot topics in parallelism*, ser. HotPar'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 17–17. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855591.1855608

[30] K. Knobe and C. D. Offner, "TStreams: How to Write a Parallel Program, Tech. Rep. HPL-2004-193,2004. [Online]. Available: http://www.hpl.hp.com/techreports/2004/HPL-2004-193.pdf

[31] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, vol. 32, pp. 444–458, April 1989. [Online]. Available: http://doi.acm.org/10.1145/63334.63337

[32] D. E. Bakken and R. D. Schlichting, "Supporting fault-tolerant parallel programming in linda," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, pp. 287–302, March 1995. [Online]. Available: http://portal.acm.org/citation.cfm?id=203121.203132

[33] Mozilla.org, "SpikerMonkey (JavaScript-C) Engine," `http://www.mozilla.org/js/spidermonkey/`. [Online]. Available: http://www.mozilla.org/js/spidermonkey/

[34] Python.org, "Python Programming Language - Official Website," `http://python.org/`. [Online]. Available: http://python.org/

[35] Lua.org, "The Programming Language Lua," `http://www.lua.org/`. [Online]. Available: http://www.lua.org/

[36] ECL, "ECL - A Common-Lisp Implementation," `http://ecls.sourceforge.net/`. [Online]. Available: http://ecls.sourceforge.net/

[37] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (XML) 1.0," *W3C recommendation*, vol. 6, 2000.

[38] B. Pittel, "On spreading a rumor," *SIAM J. Appl. Math.*, vol. 47, pp. 213–223, March 1987. [Online]. Available: http://portal.acm.org/citation.cfm?id=37387.37400

[39] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46 –55, Jan.–Mar. 1998.

[40] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315–339, 1990.

[41] "Entangled: DHT and tuple space based on Kademlia." [Online]. Available: http://entangled.sourceforge.net/