J Grid Computing (2006)
DOI 10.1007/s10723-006-9033-9

ORIGINAL PAPER

A Decentralized Computational Infrastructure for Grid-Based Parallel Asynchronous Iterative Applications

Zhen Li · Manish Parashar

Received: 10 September 2005 / Accepted: 5 February 2006

Abstract Parallel asynchronous iterative algorithms relax synchronization and communication requirements, and can potentially extend Desktop Grids beyond *embarrassingly parallel* applications to support a broader class of parallel iterative applications. This paper presents the design and implementation of CometG, a decentralized (peer-to-peer) computational infrastructure that extends Desktop Grid environments to support these applications. CometG provides a decentralized and scalable tuple space, efficient communication and coordination support, and application-level abstractions that can be used to implement Desktop Grid applications based on parallel asynchronous iterative algorithms using the master-worker/BOT paradigm. The deployment and evaluations of CometG and a CometG-based application in a wide-area environment using the PlanetLab [7] test bed, as well as a campus network are presented.

Key words decentralized (peer-to-peer) tuple space · Desktop Grids · parallel asynchronous iterative algorithms

Z. Li (✉) · M. Parashar
The Applied Software Systems Laboratory,
Department of Electrical and Computer Engineering,
Rutgers University, Piscataway, NJ 08854, USA
e-mail: zhljenny@caip.rutgers.edu

M. Parashar
e-mail: parashar@caip.rutgers.edu

Abbreviations
BOT bag of task
PDE partial differential equation

1. Introduction

Grid computing, based on the aggregation of large numbers of independent hardware, software and information resources spanning multiple organizations, is rapidly emerging as the dominant paradigm for distributed problem solving for a wide range of application domains. Complementary to Grid virtual organizations, Desktop Grids [25] leverage Internet connected computers to support large computations. Desktop Grid systems have been successfully used to address large applications in science and engineering with significant computational requirements, including global climate predication (Climatprediction.net) [2], molecular sequence analysis (Folding@Home) [3], protein structure prediction (Predictor@Home) [5], search for extraterrestrial intelligence (SETI@Home) [8], gravitational wave detection (Einstein@Home), and cosmic rays study (XtremWeb) [9].

While the successes of the above applications do demonstrate the potential of Desktop Grids, current implementations are limited to *embarrassingly parallel* [38] applications based on the Bag-Of-Task (BOT) paradigm, where the individual tasks are independent and do not require inter-

task communications. As a result, these implementations cannot support more general scientific and engineering applications, such as those based on parallel iterative computations, as the parallel formulations of these applications require synchronization and inter-task communications. While some Java-based platform independent communication libraries, such as mpiJava [4] and JavaPVM [40] and been developed to support parallel Grid applications, these libraries have targeted relatively tightly coupled, similarly configured, and simultaneously available Grid environments such as multi-site inter-connected clusters [24, 29]. Consequently, supporting the synchronization and communication requirements of general scientific application in heterogeneous, dynamic and unreliable wide-area environment continues to present significant difficulties.

Parallel asynchronous formulations of iterative algorithms [17, 22] relax synchronization and communication requirements, and can tolerate heterogeneous computation powers and unreliable communication channels. These formulations have been proposed to extend Desktop Grids beyond *embarrassingly parallel* applications and support parallel iterative applications, such as computing the lowest eigenvalue and eigenvector of stochastic matrices for Google pageranks [33] and solving linear systems [18]. However, current implementations of these algorithms are limited to tightly coupled clusters and local area networks, and scalable wide-area implementations remain a challenge.

This paper presents the design and implementation of CometG, a decentralized (peer-to-peer) computational infrastructure that extends Desktop Grid environments to support parallel asynchronous iterative applications. CometG provides a decentralized and scalable tuple space [27] that can be associatively accessed by all peer nodes without knowledge of the physical location of the tuples or the identifiers of hosts over which the space is distributed. The CometG tuple space is built on top of a resilient self-organizing overlay, and provides efficient and scalable communication and coordination abstractions. The communication abstraction provides associative content-based messaging and manages system heterogeneity and dynamism, and the coordination abstraction provides Linda-like [19] coordination primitives. CometG also provides application-level abstractions that can be used to implement applications based on parallel asynchronous iterative algorithms using the master-worker/BOT paradigm.

This paper also presents the implementation of a Grid-based PDE application using the CometG computational infrastructure. The application uses parallel asynchronous Jacobi iterations to solve the heat distribution problem [17]. CometG abstractions and mechanisms are used to construct services for dynamic and anonymous task distribution, task execution, decoupled communication and data exchange required by the application. CometG and the PDE application have been deployed on a wide-area environment using the PlanetLab [7] test bed, as well a campus network at Rutgers University. An experimental evaluation using these deployments is presented. The evaluations demonstrate both, the efficiency/scalability of CometG and its ability to support wide-area deployments of Desktop Grid applications based on parallel asynchronous iterative algorithms.

The rest of the paper is organized as follows. Section 2 presents a brief introduction to parallel asynchronous iterative algorithms and applications, outlines requirements for their implementations in Desktop Grid environments, and describes related work. Section 3 presents CometG and describes its design and implementation. In Section 4, the implementation and operation of a parallel asynchronous iterative application using CometG is described. Section 5 presents an experimental evaluation on a campus network at Rutgers as well as a wide-area (using PlanetLab) test bed. Section 6 presents concluding remarks.

2. Parallel Iterative Computations in Grid Environments

2.1. Parallel Asynchronous Iterative Algorithms and Applications

Iterative algorithms are generally of the form: $x^{k+1} = f(x^k), k = 0, 1, \ldots$, where x^0 is given, x^k is an n-dimensional vector, and f is a function from

$R^n \rightarrow R^n$. The sequence x^k generated by the above iteration converges to some x^*, and if f is continuous then x^* is a fixed point of f. These algorithms are typically parallelized using the block-decomposition paradigm, where the x^k is decomposed as m components and f is partitioned conformally. The entire problem can be solved in parallel by m processors and the iteration vector at each step is $x^k = [x_1^k, x_2^k, ..., x_m^k]$, each component of which can be processed by a single processor.

Iterative algorithms can be categorized as *synchronous* or *asynchronous* based on their requirements for global data synchronization. *Synchronous* iterative algorithms have an implicit barrier at the end of each iteration step, and require that all communications be completed and all messages become available before the next iteration starts. *Asynchronous* iterative algorithms relax this requirement for global synchronization, and allow processors to continue computing using only partial information from other processors. This allows these algorithms to tolerate variances in computational power and communication delay, which are typical in Grid environments. Note that, as expected, the convergence of asynchronous iterative algorithms is delayed due to the unsynchronized data. However, in spite of this, these algorithms have the potential of outperforming synchronous algorithms as they avoid synchronization overheads, which can be significant in Grid environments.

Potential applications of parallel asynchronous iterative computation span a range of scientific and engineering disciplines, such as high-performance linear algebra and optimization problems. Examples include: (1) Computation of eigen-systems, which are used in the study of nuclear reactor dynamics, dynamic finite element analysis of structural models, and the next generation particle accelerators [41]; (2) solution of large sparse linear systems of equations obtained from the discretization of partial differential equations (PDE) [26], which are used for aircraft simulation, computer graphics, weather prediction, fluid flow, gravitational fields, and electromagnetic field description; and (3) variational inequalities that can be viewed as generalization of both constrained optimization problems and systems of equations, which are used as models for equilibrium studies ranging from economics to traffic engineering [17].

Note that while there has been significant work on parallel asynchronous iterative computations in recent years, these efforts have focused on algorithmic and implementation issues such as convergence rate, termination detection, and load balancing [11, 12, 15–17]. The research presented in this paper leverages these efforts and focuses on the development and execution of applications based on these algorithms on Desktop Grid environments with Internet-scale connectivity.

2.2. Requirements for Grid-Based Parallel Asynchronous Iterative Algorithms and Applications

Parallel asynchronous iterative applications can definitely benefit from the potentially large numbers of processors available on Grid. However, developing and executing Grid-based implementations requires addressing the complexity of the Grid environment, including its heterogeneity in computational, storage and communication capabilities, its dynamism and its unreliability. Clearly, this complexity must be abstracted from the application scientists/engineers and effectively addressed by a computational infrastructure. Such an infrastructure should support *dynamic* and *anonymous* task management, allowing application execution to be independent of system configuration and promoting the simplicity and convenience of the BOT paradigm. Further, it should provide appropriate coordination and communication mechanisms to support dynamic dependencies and interactions.

Specifically, Section 2 the task coordination and communication mechanisms should be: (1) *asynchronous* to enable decoupled (in time and space) and dynamic task allocation and inter-processor communication; (2) *associative* to allow interactions to be anonymous and based on content rather than defined in terms of addresses or names of end-points', since maintaining common knowledge about names and addresses in dynamic Grid environments is infeasible and can pose security risks [18]; (3) *scalable* to address increas-

ing system size (number of nodes) and application problem size; and (4) *failure-resilient* to reduce the loss of application computational effort when system or application failures occur. The tuple space paradigm, which supports an asynchronous associative communication model and provides simple programming abstractions, presents an attractive approach for addressing the issues outlined above.

2.3. The Tuple Space Paradigm

The tuple space paradigm, made popular by Linda [23], addresses many of the requirements outlined above. Its key features include: asynchronous communication that decouples senders and receivers in space and time; an associative multicast medium through which multiple receivers can read a tuple written by a single sender using pattern-matching mechanisms instead of names and locations; and a small set of operators (write, read, and remove) providing a simple and uniform interface to the tuple space. Additionally, resilience to process failures can be simply provided by a stable tuple space [14] where failed processes can be recovered on any host. Further, tuple spaces naturally support BOT solutions for parallel applications using the master worker model – the master inserts task tuples into the space and collects result tuples, and the workers extract task tuples from the space and insert result tuples. While sufficiently scalable distributed tuple space implementation, where the tuple retrieval performance is proportional to at least the logarithm of the system size [30], can effectively address the requirements outlined above, such implementations in Grid environments remain a challenge.

The original Linda model must be enhanced and customized to support asynchronous iterative algorithms. First, tuple insertion and retrieval are unordered and non-deterministic. As a result, the programmer must implement "latest version" retrieval semantics (e.g., by adding a sequence number field to the tuple) and guarantee processing of all tasks (e.g., by using a global counter tuple). Second, associative communications implemented using the pattern-matching mechanism are inherently inefficient for large data transfers [35]. This inefficiency is further amplified if the tuple delivery requires multiple routing steps as large message sizes increase transmission time as well as probability of failure at each step.

2.4. Related Work

Related research efforts that focus on supporting asynchronous parallel applications in peer-to-peer systems include P^3 [31], Jace [13], and parallel iterative computing using associative broadcast [18]. P^3 proposes a peer-to-peer network platform for high performance parallel computing in an Internet-based environment. It uses a distributed file system for inter-process communication and synchronization. Scalability in P^3 is achieved using dynamic load balancing between computing nodes, P2P communication and dynamically changing sets of manager nodes. However, the P^3 network implementation is still ongoing research to the best of our knowledge.

Jace [13], is a Java based distributed programming environment designed specifically for distributed asynchronous iterative computations. It provides a parallel virtual machine to implement computing tasks using message passing. However, it does not allow nodes to dynamically join and/or leave the system, and the application data is statically partitioned across and stored at the participating nodes. Further, fault-tolerance issues are not addressed by Jace.

Parallel iterative computing using associative broadcast [18] is most closely related to the research presented in this paper. In [18], the programming models and implementation issues for executing parallel computations on Desktop Grids are discussed, and combining associative interactions with parallel asynchronous iterative algorithms are proposed as an effective approach. Specifically, asynchronous data communications between the parallel computation tasks is achieved using the associative broadcast mechanism. The implementation of associative broadcast, however, does not currently address scalability to Grid environments. Further, this system does not support dynamic task distribution. CometG implements a scalable tuple space to support the associative communication model, and also provides support for dynamic task distribution and fault-tolerance.

3. A Decentralized Computational Infrastructure for Grid-Based Asynchronous Iterative Computations

The CometG computational infrastructure presented in this paper builds on a scalable, decentralized tuple space [27] that spans the nodes of the Desktop Grid. The tuple space is essentially a global virtual shared-space constructed from the semantic information space used by entities for coordination and communication. This information space is deterministically mapped, using a locality preserving mapping, onto the dynamic set of peer nodes in the Grid system. The resulting structure is a locality preserving semantic distributed hash table (DHT) built on top of a self-organizing structured overlay.

A schematic overview of the CometG architecture is shown in Figure 1 and consists of three key layers. The communication layer provides scalable content-based messaging services as well as channels for direct communication, and manages system heterogeneity and dynamism. The coordination layer provides Linda-like primitives and supports the tuple space coordination model. The application layer provides abstractions and services for asynchronous iterative computations, which are implemented using the communication and coordination layers.

3.1. Tuples and Tuple Distribution

As mentioned above, the CometG tuple space is a global virtual semantic shared-space constructed from the semantic information space used by entities for coordination and communication. A

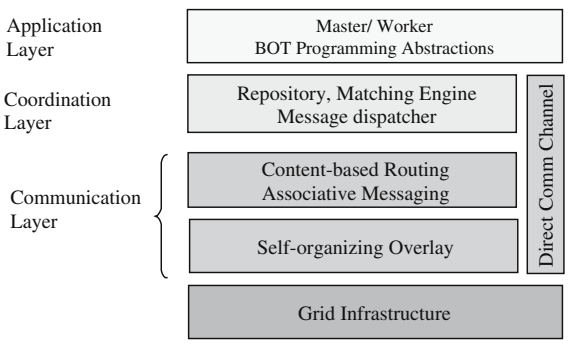

Figure 1. A schematic overview of CometG

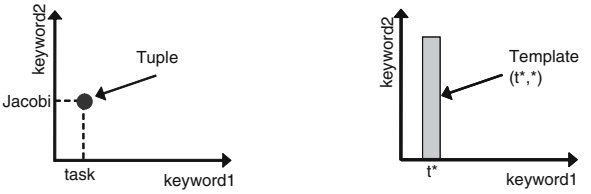

Figure 2. Example of tuples, templates and the semantic information space in CometG

tuple in CometG is associated with k keywords that are selected from its tag and fields. Using these keywords, a tuple can be viewed as a point in a k-dimensional (kD) space where each keyword represents an axis of the space. The possible sets of keywords used to describe tuples collectively define the semantic information space. A template can similarly be associated with keywords, however, in this case it may include partial keywords, wildcards, or ranges. Figure 2 illustrates examples of tuples and templates in a 2D semantic space. A tuple described by complete keywords is mapped to a point in the information space. A template described by partial keywords and wildcards identifies a region in the information space. In CometG, it is assumed that the information space is known to participating nodes.

In CometG, a tuple is implemented as a simple XML string, where the first element is the tuple's tag and is followed by an ordered list of elements containing the tuple's fields. Each field has a name followed by its value. The tag, field name, and value must be data value for a tuple and may contain wildcard ('*') for a template tuple. This lightweight format is flexible enough to represent information for a wide range of applications and can support rich matching relationships [37]. It is suitable for efficient information exchange in distributed and heterogeneous environments. A tuple is retrieved if it exactly or approximately matches the template [27]. Figure 3 show an example of tuples that match exactly. The task tuple in Figure 3(a), tagged "Task," has fields *BlockID, TotalBlocks, Partition, Solver, Precision, MaxIteration, MasterNetName and DataPort with values 5, 10, strips, Jacobi, 0.0001, Inf, foo.cs.bar.edu, 9914*, respectively, and can be retrieved using the template in Figure 3(b).

The CometG decentralized tuple space is essentially an associative Distributed Hash Table

Figure 3. An example of a tuple and a template: **(a)** A task tuple. **(b)** A task template

```
<Task>
   <BlockID> 5 </BlockID>
   <TotalBlocks> 10 </TotalBlocks>
   <Partition> strips </Partition>
   <Solver> Jacobi </Solver>
   <Precision> 0.0001 </Precision>
   <MaxIteration> Inf </MaxIteration>
   <MasterNetName> foo.cs.bar.edu
   </MasterNetName>
   <DataPort> 9914 </DataPort>
</Task>
```

(a)

```
<Task>
   <BlockID> * </BlockID>
   <TotalBlocks> * </TotalBlocks>
   <Partition> * </Partition>
   <Solver> * </Solver>
   <Precision>* </Precision>
   <MaxIteration> * </MaxIteration>
   <MasterNetName> * </MasterNetName>
   <DataPort> * </DataPort>
</Task>
```

(b)

(DHT). The nodes in the Desktop Grid form a one-dimensional self-organizing overlay. The Hilbert Space Filling Curve (SFC) [28] is used to construct the index space of the DHT from the information space, and to map tuples/templates from the information space to peer indices in the one-dimensional overlay. The Hilbert SFC is a locality preserving continuous and recursive mapping from a k-dimensional space to a one-dimensional space. It is locality preserving in that points that are close on the curve are mapped from points that are close in the k-dimensional space. The Hilbert curve readily extends to any number of dimensions. Further, its locality preserving and recursive nature enables the index space to maintain content locality and efficiently resolve content-based lookups [34]. The SFC-based index space is mapped to the overlay such that each node in the overlay stores the keys that map to the segment of the curve between itself and its predecessor node. A tuple described by complete keywords and mapped to a point in the information space is located on at most one node. A template described by partial keywords, wildcards, or ranges and defining a region in the information space may be mapped to a collection of segments on the SFC and correspondingly, to a set of nodes in the overlay. While the CometG architecture can support scalable tuple distribution, failure of nodes can result in tuple loss. This is addressed by the CometG application layer using timeout regeneration and checkpointing-restart mechanisms, as described in Section 3.4.

3.2. The Communication Layer

The CometG communication layer provides an associative communication service and guarantees that content-based messages, specified using flexible content descriptors, are served with bounded cost. This layer also provides a direct communication channel to efficiently support large volume data transfers between peer nodes. The communication channel is implemented using a thread pool mechanism and TCP/IP sockets.

The major components of the associative messaging service include a content-based routing engine and the one-dimensional structured self-organizing overlay. The routing engine implements the Hilbert SFC mapping and supports flexible content-based routing and complex querying using partial keywords, wildcards, or ranges. It also guarantees that all peer nodes with data elements that match a query/message will be located. The routing engine has a single operator for associative messaging, *post(keys, data)*, where *keys* form the semantic selector and *data* is the message payload. The overlay is composed of peer nodes, which may be any node in the Desktop Grid system (e.g., end-user computers, servers, or message relay nodes). The peer nodes can join or leave the network at any time. While the CometG architecture is based on a structured overlay, it is not tied to any specific overlay topology. In the current implementation, we use Chord [36], which has a ring topology, primarily due to its guaranteed performance, efficient adaptation as nodes join and leave the system, and the simplicity of its implementation. In principle, this overlay could be replaced by other structured overlays. The overlay provides the *lookup(identifier)* operator. Given an identifier, this operation locates the node that is responsible for it, i.e., the node with an identifier that is the closest identifier greater than or equal to the queried identifier. The lookup algorithm in Chord enables the efficient data routing with cost

bounded at O(Log N) [36], where N is the number of nodes in the system.

3.3. The Coordination Layer

The coordination layer provides the following primitives to support the tuple space coordination model.

- *Out(ts, t)*: A non-blocking operation that inserts tuple *t* into space *ts*.
- *In(ts, $\bar{t}$, timeout)*: A blocking operation that removes a tuple *t* matching template $\bar{t}$ from the space *ts* and returns it. If no matching tuple is found, the calling process blocks until a matching tuple is inserted or the specified *timeout* expires. In the latter case, *null* is returned.
- *Rd(ts, $\bar{t}$, timeout)*: A blocking operation that returns a tuple *t* matching template $\bar{t}$ from the space *ts*. If no matching tuple is found, the calling process blocks until a matching tuple is inserted or the specified *timeout* expires. In the latter case, *null* is returned. This method performs exactly like the *In* operation except that the tuple is not removed from the space.

The main components of the coordination layer include a data repository for storing tuples and templates, a local matching engine, and a message dispatcher that interfaces with the communication layer to translate the *Out, Rd* and *In* coordination primitives to content-based routing operations at communication layer and *vice versa*. As mentioned above, tuples are represented as simple XML strings as they provide small-sized flexible formats that are suitable for efficient information exchange in distributed heterogeneous environments. The data repository stores XML string tuples as DOM level 2 objects [39]. Further, it employs a hash structure to perform pattern-matching in constant time in memory.

The tuple distribution and retrieval operations are implemented using the content-based messaging abstraction and mechanisms provided by the communication layer. Using the keywords associated with a tuple, a tuple is routed to the appropriate peer node in the overlay, and a template tuple is routed to the set of peer nodes that contain matching tuples. The tuple insertion and retrieval processes are illustrated in Figures 4 and 5, respectively.

The exact tuple/template matching process consists of the following steps. (1) Keywords are extracted from the tuple or template and used to generate keys for the *post* operation. The payload of the message includes the tuple data and the coordination operation. (2) The routing engine uses the SFC mapping to identify the indices corresponding to the keys and the corresponding peer id. (3) The overlay *lookup* operation is used to route the tuple/template to the appropriate peer node. The *Out* operation returns after receiving a response from the destination peer to guarantee tuple delivery. In the case of exact *Rd* and *In* operations, templates are routed to the appropriate peer node in a similar manner. The *In* and *Rd* operations block until a matching tuple is returned by the destination or a timeout occurs. Tuple and template insertion are guaranteed using acknowledgements and timeout-retry mechanisms.

The approximate retrieval process is similar. A retrieval request may be sent to multiple nodes in this case, and each of them may return a matching tuple. However, the *In* and *Rd* oper-

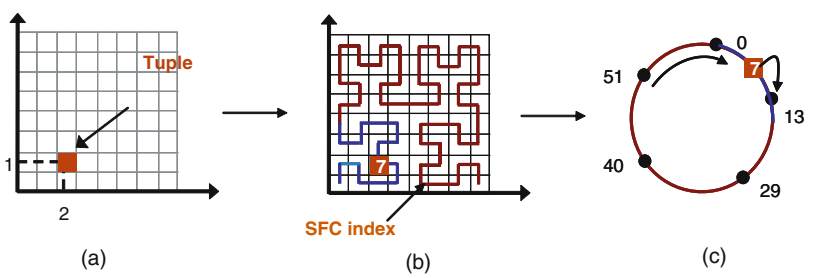

Figure 4. Example of tuple insertion in CometG: **(a)** a tuple is represented in a 2D keyword space, as the point (2, 1); **(b)** the point (2, 1) is mapped to index 7 using the Hilbert SFC; **(c)** the tuple is inserted at node 13 (the successor of SFC index 7)

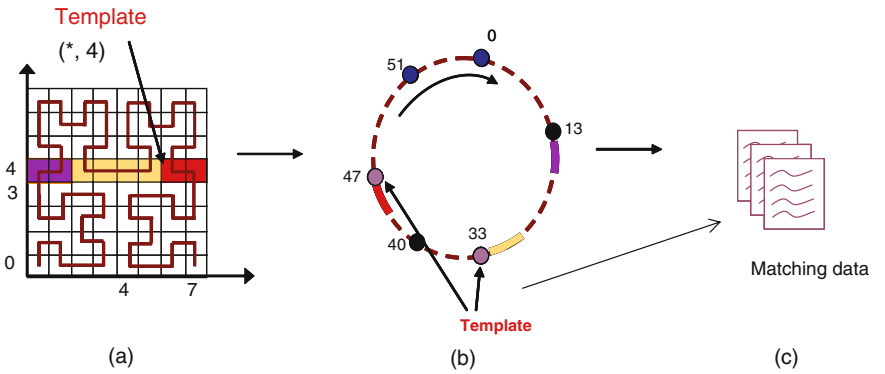

Figure 5. Example of tuple retrieval in CometG: **(a)** the template defines a rectangular region in the 2D space consisting of three clusters; **(b)** the nodes that store the clusters are queried; **(c)** results of the query are sent to the requesting node

ations are implemented differently. In case of *Rd*, the first tuple that is returned is accepted and forwarded to the application, and subsequent tuples returned are ignored. In case of an *In* operation, one of the matching tuple must be deleted and this is coordinated by the requesting node. For each matching tuple found, the node with the matching tuple sends it to the requesting node and waits for a delete confirmation. The requesting node responds with a delete confirmation to the first matching tuple that it receives and responds with an ignore message to all other returned tuples.

3.4. The Application Layer

The CometG application layer provides coordination space abstractions and programming modules to support master-worker/BOT parallel formulations of asynchronous iterative computations. Specifically, two customized coordination spaces, *TaskSpace* and *BorderSpace*, are defined and implemented separately. *TaskSpace* stores task tuples representing application tasks and specifying the masters that are responsible for the tasks. This space implements First-In-First-Out (FIFO) semantics for tuple and template operations, and provides a queue abstraction for task distribution and management. An example of a task tuple is shown in Figure 3. *BorderSpace* is used for exchanging border data tuples between neighboring tasks. This space enforces *over-write* semantics during tuple insertion, where tuples in the space always store the latest content, resulting the latest messaging semantics. A border tuple has a border id field and an associated binary data block. The data block is not used for content-based distribution, lookup, and pattern-matching.

The programming modules include masters and workers. A worker module contains an application-specific computational component that can locally compute a retrieved task. The worker uses the tuple space abstractions to retrieve tasks and exchange borders. Task retrieval consists of two steps – removing a task description from the *TaskSpace* and downloading the task data from the corresponding master. A master module is responsible for partitioning the application data, generating tasks, collecting results, and terminating the application when it completes. CometG provides single master mode as well as multiple master mode. In multiple master mode, hierarchical or decentralized termination algorithms [12] are supported based on the organization of the masters. A master module has five components:

- The *configuration manager thread*, which reads the application configuration (including whether it is a single or a member of multiple master organization) and the data partitioning strategy.
- The *task generator thread*, which generates application tasks based on the partitioning strategy, encapsulates task descriptions as tuples and inserts the task tuples into *TaskSpace*.
- The *data transfer thread*, which uses the direct communication channel to process requests for task data retrieval and for result submission from workers, as well as coordination messages (e.g., 'convergence' message) between masters.
- The *terminator thread*, which checks for convergence among tasks that the master is responsible for, monitors convergence messages from other masters, and terminates when overall convergence is achieved.

– The *task monitor*, which maintains a table of tasks the master is responsible for, and records the current state of the tasks in this table. The state of a task can be *generated, retrieved, computing, submitting* or *completed*.

3.5. Supporting Large Application/System Scales

CometG supports large application/system scales using multiple coordination groups. A coordination group includes one *TaskSpace*, one *BorderSpace*, and a group of masters and workers. A group can support multiple applications with logically separate semantic spaces. An application can also span multiple groups, each of which handles a part of the application. The application is hierarchically partitioned, first across coordination groups, and then across masters within each coordination group. Tasks with communication dependencies should be mapped to the same coordination group if possible as communications across groups can be expensive. Workers within a coordination group communicate using the shared *BorderSpace*. Masters within and across coordination group communicate using direct communication channels.

Using coordination groups thus distributes the load of *TaskSpace* and reduces the size of *BorderSpace*, effectively improving the scalability of the system. Nevertheless, it may not always be possible to partition the application to eliminate inter-group communications. However, as the number of these communications is relatively small, these communications can simply be ignored in the case of asynchronous applications. While ignoring them will affect convergence, we have observed that the improvement in overall application performance using this approach outweighs these effects. In cases where the number of inter-group communications is large, or when task dependencies are complex, data exchange can be coordinated through a single node in each group [32].

3.6. Addressing the Unreliability of the Grid

The CometG computational infrastructure provides application level fault tolerance mechanisms to address the unreliability inherent in Grid environments. These mechanisms assume a fail-stop failure model and timed communication behavior [20, 21]. Under these assumptions, possible failures include border tuple communication failure, master failure, and task loss. These failures are addressed below:

Border tuple communication failures are simply handled by *Rd* timeouts, due to the resilient nature of asynchronous algorithms. *Master failures* are handled using checkpoint-restart. The runtime system periodically checkpoints the local state of each master, including its task table and current intermediate results, to a stable storage. Users are currently responsible for the detecting the failure of a master node. When a master fails, users can recover its state from the stable storage and resume the computation. Finally, *task loss* is handled using timeout-regeneration and a retrieval–submission protocol. It is well known that detecting this kind of failure in tuple spaces is very difficult because there can be multiple reasons for the failure, including *TaskSpace* crashes, message losses, communication link failures, failures of workers with unfinished tasks, etc. In CometG, the loss of un-retrieved and retrieved tasks, are handled separately as follows.

Un-retrieved task loss occurs only when the relevant *TaskSpace* node crashes since task tuple insertions are guaranteed. Masters can detect this failure using a keep-alive mechanism, and can handle it by regenerating unfinished tasks. The regenerated tasks will be deterministically routed to an operational *TaskSpace* node on the DHT due to the resilience of the overlay (e.g., the Chord routing around failure functionality [36]).

Retrieved task loss is detected using the task tables at the masters. Each task in the table is associated with a timer which is initialized when the task is retrieved by a worker. If the results for a task are not returned before the timer expires, the task is considered as lost. The master regenerates the lost task and updates the task table. The value of the task timer depends on the computational requirements of the specific application as well as the current performance of the system. In CometG, this value is dynamically determined based on a user specified threshold and the observed maximum task processing time,

which is the time interval from when a task is retrieved to when the corresponding results are returned.

Note that task regeneration can lead to the problem of *duplicated tasks* where the same task may be allocated to multiple workers. This can be addressed using a simple retrieval–submission protocol where the master refuses all data transfer requests and result submissions for a task that it has tagged as completed in its task table.

3.7. System Implementation and Operation

The current prototype of CometG has been implemented on Project JXTA [6], a platform independent peer-to-peer framework. The JXTA platform provides a virtual network for applications, which can cross barriers such as firewalls/NATs to establish peer communities spanning any part of the physical network. JXTA peers can discover peer resources, communicate with each other, and self-organize into peergroups. A JXTA peergroup provides a scoping mechanism, using which messages are only propagated among group members. JXTA also provides security features that can be used by applications.

Each CometG node operates as a JXTA peer identified by a *JxtaID*. Each node in CometG can support multiple masters and/or workers associated with different applications. Further, CometG coordination groups are implemented as JXTA peergroups. Nodes in CometG organize using *JXTA Discovery Protocol* to form the ring overlay. The overlay *lookup* operator of the CometG communication layer maps the logical overlay peer identifier to the node's *JxtaID*, and uses the *JXTA Resolver Protocol* for communication. The implementation of the CometG tuple space primitives are illustrated in Figure 6.

3.8. System Operation

The overall operation of CometG consists of two phases: *bootstrap* and *running*. The *bootstrap* phase is used to setup a coordination group. During this phase, peer nodes join the CometG JXTA peergroup and exchange messages with the rest of the group. Each joining peer attempts to discover an existing peer in the system and to construct the overlay and setup its routing table. It also sends discovery messages to the group. If the message is unanswered after a pre-defined time interval (in the order of seconds), the peer assumes that it is the first one in the system. If a peer responds to the message, the joining peer queries this bootstrapping peer according to the join protocol of the overlay, and updates routing tables in the overlay to reflect the join.

The *running* phase consists of *stabilization* and *user* modes. In the *stabilization* mode peer nodes manage the structure of the overlay. In this mode, peer nodes respond to periodic queries from other peers to ensure that routing tables are up-to-date and to verify that other peer nodes in the group have not failed or left the system. In the user mode, peer nodes participate in user applications. In this mode, application developers can configure the system, setup application parameters such as coordination groups, relevant semantic spaces, master configurations, and initiate the master processes.

4. Grid-Based Parallel Asynchronous Iterative Applications Using CometG

This section illustrates the use of the CometG computational infrastructure to implement and execute a Grid-based PDE application. The application uses parallel asynchronous Jacobi iterations for

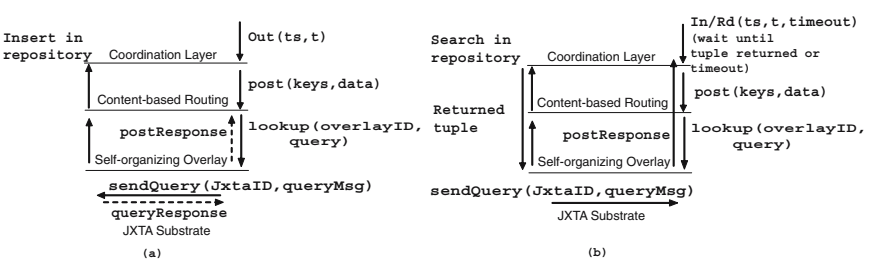

Figure 6. CometG tuple space operation: **(a)** Tuple distribution using the Out operator. **(b)** Exact tuple retrieval using the In/Rd operator

solving the heat distribution problem [17]. In this illustrative application, the temperature at the edges of a square sheet are known, and the temperature at a point in the interior surface of the sheet is computed based on the temperatures around it. The square sheet is discretized as a two-dimensional Grid and represented as a two-dimensional array of points. In each iteration, the value of each point in the interior of the array is computed as an average of four points around it. The computation is repeated until the stop criterion is satisfied, i.e., the difference in temperature values at a point between iterations is less than a prescribed threshold, or the bound on the number of iterations is reached.

Assuming that the application uses strip partitioning, the Grid points are divided into blocks of rows. Each block defines a task and is processed by one worker. Since each point needs its four immediate neighbors, each worker needs to exchange data in the rows at the top and bottom of the block with workers processing neighboring blocks. The workers assigned the top most and bottom most rows are exceptions and need to exchange data in only one row. A conceptual overview of the CometG based implementation of this application is shown in Figure 7. Flow charts for the operation of master and worker nodes are presented in Figure 8, and are described below.

Once a worker is initiated, it repeats the following steps until explicitly terminated: (1) Extract a task tuple from *TaskSpace*, (2) read the required top and/or bottom border rows from *BorderSpace*, (3) locally compute temperature, (4) insert updated border rows into *BorderSpace*, (5) repeat steps (2)–(4) until the stop criterion specified in the task tuple is reached, and (6) send results to the master corresponding to the task using a direct communication channel.

When the master is launched, it uses user inputs to configure the application (e.g., setup the number of coordination group and master organization, etc.) and initiates the *BorderSpace*. If a single master is used, that master is responsible for the entire Grid. The master first partitions the Grid into blocks and inserts corresponding tasks into *TaskSpace*. When a task is assigned to a worker, the worker obtains task data from the master using the direct communication channel. When the task completes, the work submits the results to the master also using the direct communication channel. After all its tasks have completed, the master checks if the stop criterion is satisfied by the computed data, since the overall application may not satisfy the stop criterion even though each task locally satisfies its stop criterion. If the overall stop criterion is not satisfied, the master repartitions the Grid to create new tasks and inserts them into *TaskSpace*.

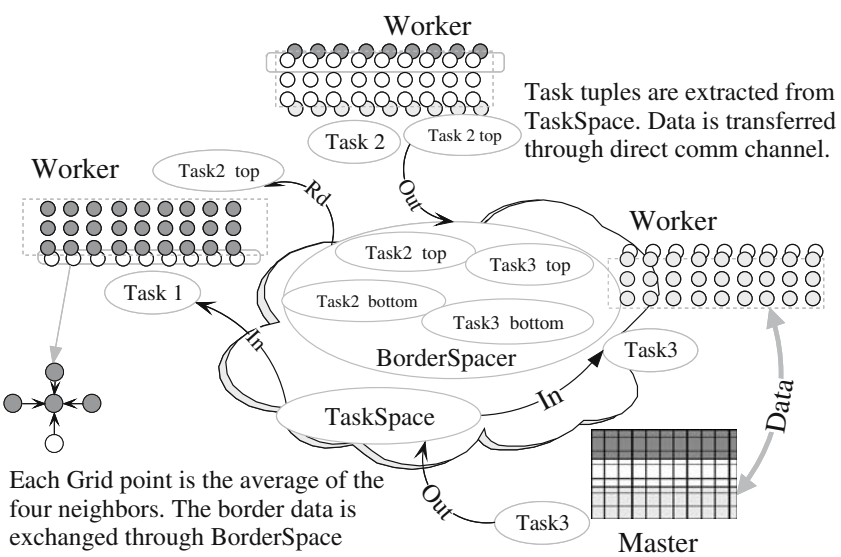

Figure 7. CometG-based implementation of the heat distribution problem using parallel asynchronous Jacobi iterations

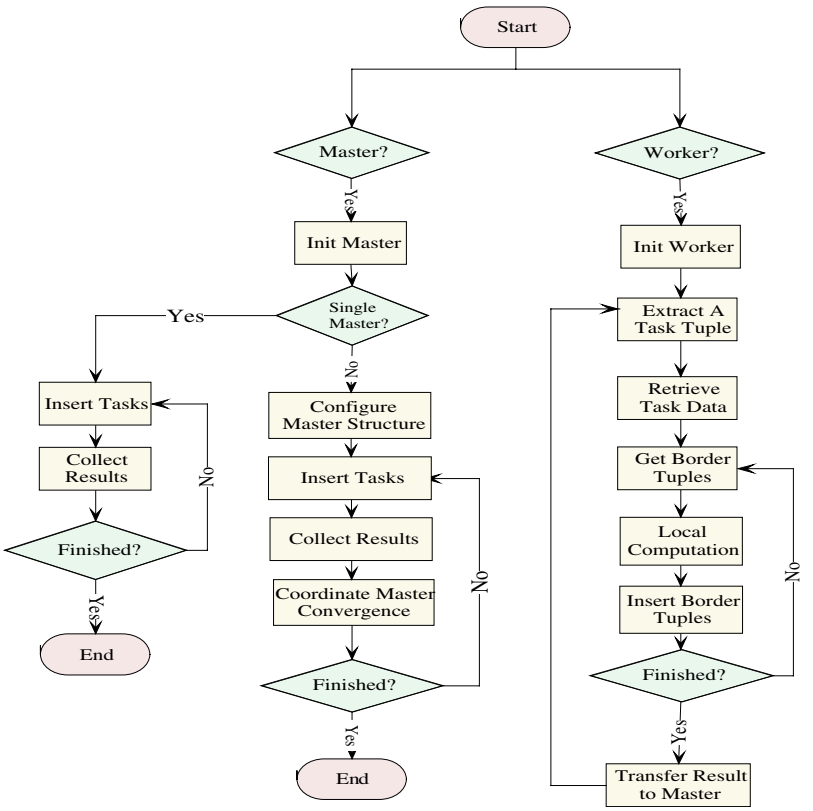

Figure 8. Operation of master and worker nodes for the CometG-based implementation of the heat distribution problem

This process constitutes one global application iteration, and is repeated until the overall stop criterion is satisfied, at which point, the master terminates the application.

If multiple masters are used, the user must define the organization of the masters, e.g., a hierarchical structure [13], and the termination detection algorithm to be used. The Grid is uniformly partitioned across the masters in this case, and each master locally partitions its sub-Grid into blocks and inserts corresponding tasks into *TaskSpace*. The operation at each master then proceeds as in the single master case described above. When a master detects local termination, it coordinates with the other masters to establish global convergence. In case of a hierarchical master organization, it sends a 'converge' message up the hierarchy to the root node. If the master stays in a 'converged' state, no further messages are sent, otherwise, a 'diverge' message is sent to the root. The root node checks the messages received from all the masters end of each iteration, and if all of them are in the converged state for a specified number of iterations, it broadcasts a 'stop' message to the masters, which causes them to terminate the application.

5. Experimental Evaluation

CometG and the PDE application have been deployed on a wide-area environment using PlanetLab [7] test bed, as well as a campus network at Rutgers. The objective of the experiments presented in this section is to evaluate and demonstrate system performance and scalability, its ability to tolerate faults, and its ability to support wide-area deployments of parallel asynchronous iterative applications. The experiments use a horizontal block partitioning strategy and vary the size of the problem as listed in Table 1. In the multiple master mode, a hierarchical organization of the masters was used and measurements were made at the root node. The different experiments and the results obtained are described below.

Table 1 Problem sizes used in the experimental evaluation

Problem size	Partitions	Block size	Border tuple size
2,000 × 2,000	100	0.32 M	16.026 K
3,000 × 3,000	100	0.7 M	24.026 K
8,000 × 2,000	200	0.64 M	16.026 K

5.1. Experiments Using the Campus Network at Rutgers

These experiments were conducted on a Grid consisting of 70 heterogeneous Linux-based computers on the Rutgers campus network. Each machine was a peer node in CometG overlay and the machines formed a single CometG group.

The first set of experiments evaluated the costs of basic tuple insertion and exact retrieval operations. The tuples in the experiments were fixed at 200 bytes, which is roughly equal to the size of a task tuple. A ping-pong like process was used in the experiments, in which an application process inserted a tuple into the space using the *Out* operator, read the same tuple using the *Rd* operator, and deleted it using the *In* operator. In these experiments, the *Out* and exact matching *Rd/In* operations used a 3D information space. For an *Out* operation, the measured time corresponded to the time interval between when the tuple was posted into the space and when the response from the destination was received, i.e., the time between Post and PostResponse in Figure 6(a). For a *Rd/In* operation, the measured time was the time interval between when the template was posted into the space and when the matching tuple was returned to the application, assuming that a matching tuple existed in the space, i.e., the time between Post and receiving the tuple in Figure 6(b). This time included the time for routing the template, matching tuples in the repository, and returning the matching tuple. Note that the time for in-memory template matching is very small compared to the communication time for this ping-pong test.

The average performance of the operators were measured for different system sizes. Figure 9 plots the average measured performance and shows that the system scales well with increasing number of peer nodes. When the number of peer nodes increases 32 times, i.e., from 2 to 64, the average round trip time increases only about 1.5 times. This is due to the logarithmic complexity of the routing algorithm used by the Chord overlay. *Rd* and *In* operations exhibit similar performance, as seen in the figure. Further, increasing the size of border tuples can cause message transmission delays. However, as expected, the message routing time remains the dominant factor as the system size increases. Note that the JXTA 2.3 Resolver Protocol used to implement CometG has been shown to effectively transfer message of size up to 128 KB [10], which is sufficient for supporting border tuple communications for the current application.

The next set of experiments measured overall application performance using a problem of size 3,000 × 3,000 Grid points and precision thresholds

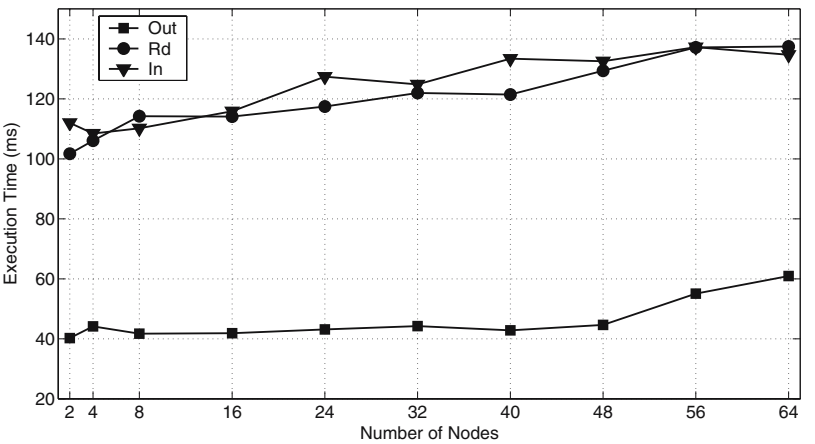

Figure 9. Average time for *Out*, *In*, and *Rd* operations for increasing system sizes on the Rutgers campus network

of 10^{-3}, 10^{-5}, and 10^{-7}. The Grid was partitioned into 100 blocks and uniformly distributed across 10 master nodes. The masters were organized as a hierarchy with one root using the algorithm in [13]. All other nodes served as worker nodes, each hosting two worker instances. The total execution time is plotted in Figure 10. In this plot, the *X*-axis represents the number of workers plotted using a logarithmic scale with base 10. The plots show the overall application performance improvements and demonstrate that, as expected, the improvements are more significant when there is more computation (e.g., when the precision threshold is smaller).

The third experiment demonstrates the CometG fault tolerance mechanisms for handling task losses due to worker dynamism. The experiment was conducted on 32 machines and used a problem of size 2,000 × 2,000 Grid points and a precision threshold of 10^{-5}. The Grid was partitioned into 100 tasks distributed across four master nodes. The user defined task timeout threshhold was set to 50 s. All the other nodes served as workers and hosted multiple worker instances. Tuple losses were simulated by having workers that have retrieved a task tuple fail with a probability of 25%. A global monitor process was used to calculate the number of alive workers in the system each time worker was started or failed. In the experiment, 20 workers were initially started on randomly selected nodes. As the application progressed, workers failed and the lost task tuples were regenerated. Meanwhile, 20 new workers were started at 285 and 439 s after the start of the application, at the rate of one worker every 3s. The results of this experiment are plotted in Figure 11. Figure 11(a) plots the fluctuations in the number of workers during the lifetime of the application. Of the 100 total tasks in the application, 22% were regenerated once and 3% were regenerated twice due to worker failures. Figure 11(b) illustrates the life-cycles of tasks including timeouts and the resulting task regenerations. For clarity, this figure only shows a subset of tasks with id between 80 and 90. Plots for other tasks are similar.

5.2. Experiments Using the PlanetLab Wide-Area Test Bed

This section presents the experiment on the wide-area PlanetLab [7] test bed. PlanetLab is a large scale heterogeneous distributed environment composed of inter-connected sites on a global scale. The goal of the experiment is to demonstrate the ability of CometG to support application even in an unreliable and highly dynamic environments such as PlanetLab, which essentially represents an extreme case for a Desktop Grid environment. The CometG is currently deployed on 234 machines on PlanetLab, which have been used in the experiment presented below. In the experiment, each machine ran an instance of the CometG stack, randomly joined the CometG overlay during bootstrap phase, and served as a master or worker node with one worker instance per node.

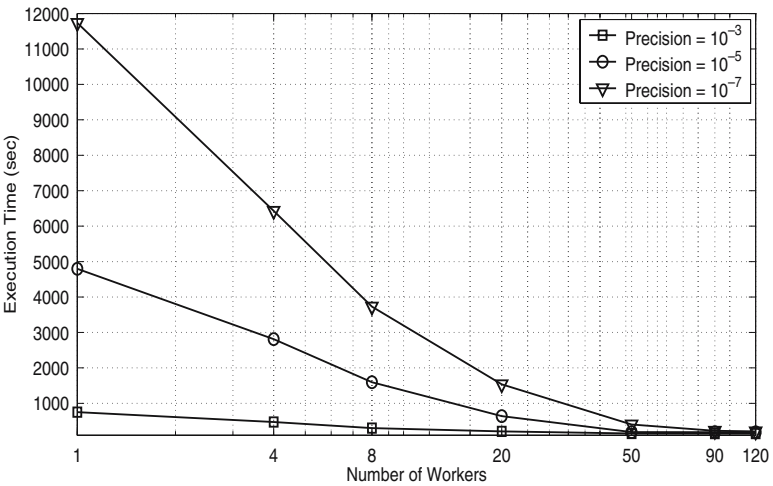

Figure 10. Overall application execution time for a problem size of 3,000 × 3,000 and 100 partitions on the Rutgers campus network

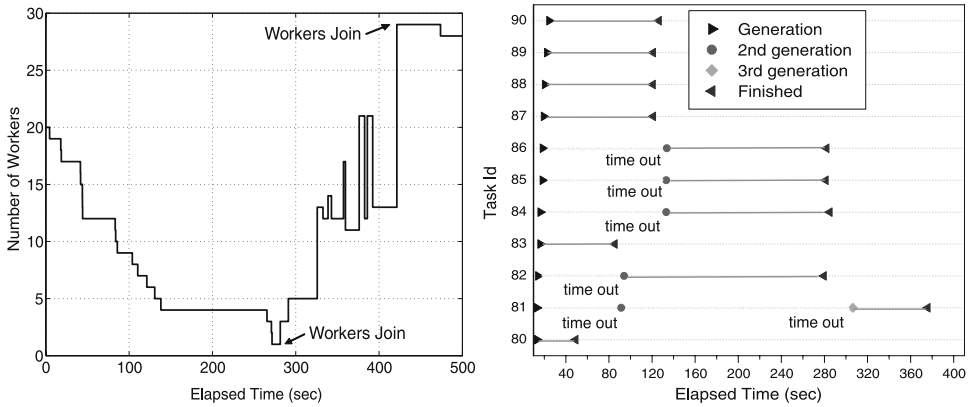

Figure 11. Evaluation of CometG fault tolerance mechanisms for tolerating loss of task tuples. (**a**) Fluctuations in the number of worker due to failures. (**b**) Life-cycles of tasks 80 through 90

The experiment used a problem of size 8,000 × 2,000 and a precision of 10^{-5}. The problem was partitioned into 200 tasks, which were uniformly distributed and across four CometG coordination groups. Each group had about 60 peer nodes, in which five nodes acted as masters and others served as workers. The task timeout threshhold was set to 500 s and the border tuple read timeout was set to 100 s. The experiment was conducted on December 9, 2006, and lasted more than 3 h, including the infrastructure setup, bootstrap, application deployment, configuration, and execution. The application terminated after two global iterations, during which multiple worker nodes left the system or failed and were handled by the CometG fault tolerance mechanisms. One master in coordination group 3 also failed and was restarted manually. The task tables of all the masters were collected and summarized in Figures 12 and 13. Figure 12 separately plots the retrieval, computation, and result submission times for all the tasks for each of the two global iterations. The *X*-axis in these plots represents the task id, and the *Y*-axis represents the execu-

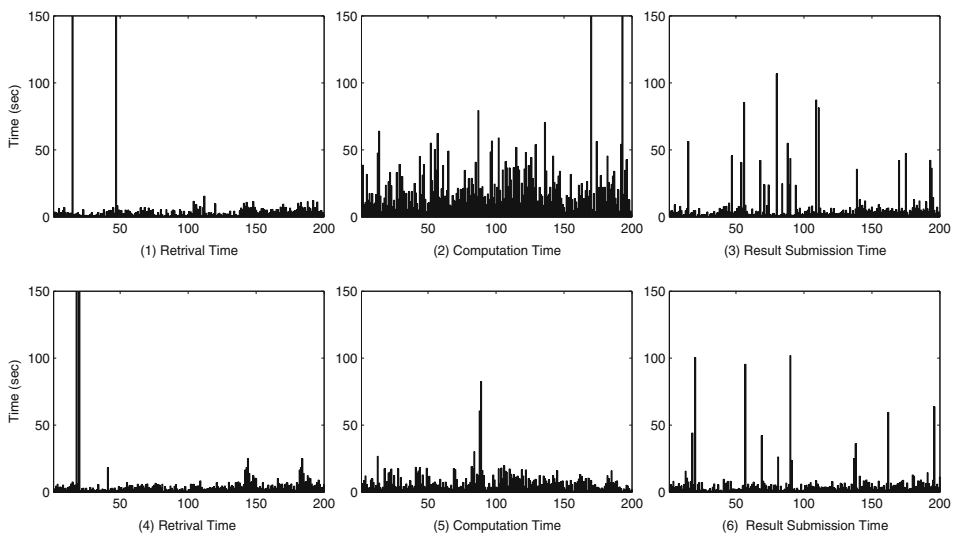

Figure 12. Execution time of each CometG phase on PlanetLab using a problem size of 8,000 × 2,000 and 200 tasks. **(1)–(3)** Phases of the first global iteration. **(4)–(5)** Phases of the second global iteration

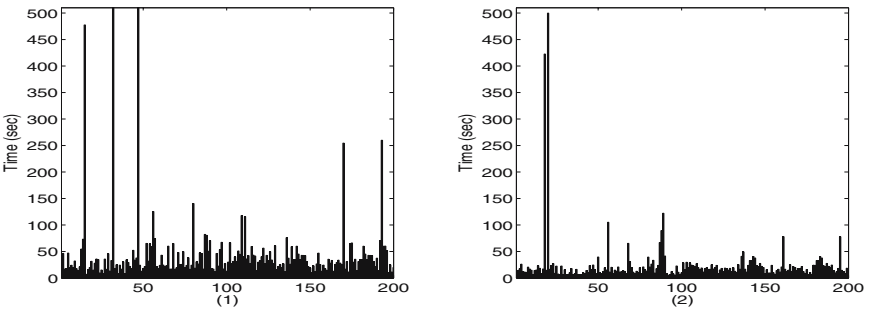

Figure 13. Overall execution time for two global iterations on PlanetLab using a problem size of 8,000 × 2,000 and 200 tasks. (**1**) Total execution time of the first global iteration. (**2**) Total execution time of the second global iteration

tion time of each phase. Note that the computation time for the second iteration is significantly smaller and the first, as expected. Figure 13 plots the total execution time for each iteration. The *X*-axis once again represents the task id. The variation in the execution time for different tasks illustrates the heterogeneity of the workers and the PlanetLab test bed. These experiments demonstrate that CometG system can effectively support parallel asynchronous iterative applications on an extreme case of a wide-area Desktop Grid environment with very high heterogeneity, dynamism, and uncertainty.

The experimental results presented in this section demonstrate both, the efficiency/scalability of CometG and its ability to support wide-area deployments of Desktop Grid applications based on parallel asynchronous iterative algorithms.

6. Conclusion

Desktop Grids leverage Internet connected computers and have been successfully used to tackle large applications in science and engineering with significant computational requirements. However, current Desktop Grid systems have been limited to *embarrassingly parallel* applications since individual tasks in these applications are independent and do not require inter-task synchronization and communications, which can present significant challenges in heterogeneous, dynamic and unreliable Grid environments. Parallel asynchronous iterative computations relax these synchronization and communication requirements and can tolerate heterogeneous computational capabilities and unreliable communication channels, and are emerging as promising approaches for enabling large scale computation problems on Desktop Grids.

This paper presented the design and implementation of CometG, a decentralized (peer-to-peer) computational infrastructure that effectively extends Desktop Grid environments to support parallel asynchronous iterative applications. CometG provides a decentralized scalable tuple space, efficient communication and coordination, and application-level abstractions that can be used to implement Desktop Grid applications based on parallel asynchronous iterative algorithms using the master-worker/BOT paradigm. The deployment and evaluation of CometG and a CometG-based application in a wide-area environment using PlanetLab test bed, as well as a campus network were presented. The evaluations demonstrated both, the efficiency/scalability of CometG and its ability to support wide-area deployments of applications based on parallel asynchronous iterative algorithms.

Currently, CometG can effectively support coordination groups consisting of tens to hundreds of peers. Further, each peer can run multiple instances of masters and/or workers. The scalability of CometG can be potentially extended to thousands or even millions of nodes with the following enhancements: (1) Separating the space nodes from end nodes, where the space nodes provide coordination services and the end nodes host the application program modules; (2) employing relatively powerful peers, i.e., super-peers, with larger memory capacity and network bandwidth, as space nodes and master nodes; and (3) using high-throughput task dispatch implementations such as the task servers popularly used by current Desktop Grid projects [1, 9] to support millions of users.

Acknowledgements The research presented in this paper is supported in part by the National Science Foundation via grants numbers ACI 9984357, EIA 0103674, EIA 0120934, ANI 0335244, CNS 0305495, CNS 0426354 and IIS 0430826.

References

1. 'Boinc.' http://boinc.berkeley.edu/
2. 'Climatprediction.net.' http://climateapps2.oucs.ox.ac.uk/cpdnboinc/download_main.php
3. 'Folding@Home.' http://folding.stanford.edu/
4. 'mpiJava.' http://www.hpjava.org/mpiJava.html
5. 'Predictor@Home.' http://predictor.scripps.edu/
6. 'Project JXTA.' http://www.jxta.org
7. 'Project PlanetLab.' http://www.planet-lab.org
8. 'SETI@Home.' http://setiathome.ssl.berkeley.edu/
9. 'XtremWeb.' http://xw.lri.fr:4330/XtremWeb/
10. Antoniu, G., Hatcher, P., Jan, M., Noblet, D.P.: Evaluation of JXTA communication layers. In: Proceedings of the Fifth International Workshop on Global and Peer-to-Peer Computing (2005)
11. Bahi, J., Contassot-Vivier, S., Couturier, R.: Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms. IEEE Trans. Parallel Distrib. Syst. **16**(4), 289–299 (2005)
12. Bahi, J., Contassot-Vivier, S., Couturier, R. Vernier, F.: A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. IEEE Trans. Parallel Distrib. Syst. **16**(1), 4–13 (2005)
13. Bahi, J.M., Domas, S., Mazouzi, K.: Combination of Java and asynchronism for the Grid: a comparative study based on a parallel power method. In: Proceedings of 18th International Parallel and Distributed Processing Symposium (2004)
14. Bakken, D.E., Schlichting, R.D.: Supporting fault-tolerant parallel programming in Linda. IEEE Trans. Parallel Distrib. Syst. **6**(3), 287–302 (1995)
15. Baudet, G.M.: Asynchronous iterative methods for multiprocessors. J. ACM **25**(2), 226–244 (1978)
16. Bertsekas, D.P., Tsitsiklis, J.N.: Convergence rate and termination of asynchronous iterative algorithms. In: Proceedings of the 3rd International Conference on Supercomputing, pp. 461–470 (1989)
17. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and distributed computation: Numerical methods. Athena Scientific (1997)
18. Browne, J.C., Yalamanchi, M., Kane, K., Sankaralingam, K.: General parallel computations on desktop grid and P2P systems. In: Proceedings of the 7th Workshop on Workshop on Languages, Compilers, and Run-time Support for Scalable Systems, pp. 1–8 (2004)
19. Carriero, N., Gelernter, D.: Linda in context. Commun. ACM **32**(4), 444–459 (1989)
20. Cristian, F.: Understanding fault-tolerant distributed systems. Commun. ACM **34**(2), 56–78 (1991)
21. Cristian, F., Fetzer, C.: The timed asynchronous distributed system model. IEEE Trans. Parallel Distrib. Syst. **10**(6), 642–657 (1999)
22. Frommer, A., Szyld, D.: On asynchronous iterations. J. Comput. Appl. Math. **123**(1), 201–216 (2000)
23. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985)
24. Huet, F., Caromel, D., Bal, H.E.: A high performance Java middleware with a real application. In: Proceedings of the Supercomputing Conference (2004)
25. Kondo, D., Taufer, M., Brooks, C., Casanova, H., Chien, A.: Characterizing and evaluating Desktop Grids: an empirical study. In: Proceedings of the International Parallel and Distributed Processing Symposium (2004)
26. Kreyszig, E.: Advanced Engineering Mathematics. Wiley (1998)
27. Li, Z., Parashar, M.: Comet: a scalable coordination space for decentralized distributed environments. In: Proceedings of the 2nd International Workshop on Hot Topics in Peer-to-Peer Systems, pp. 104–112 (2005)
28. Moon, B., Jagadish, H.V., Faloutsos, C., Saltz, J.H.: Analysis of the clustering properties of the Hilbert space-filling curve. IEEE Trans. Knowl. Data Eng. **13**(1), 124–141 (2001)
29. Nieuwpoort, R.V., Maassen, J., Kielmann, T., Bal, H.E.: Satin: simple and efficient Java-based Grid programming. Scalable Computing: Practice and Experience **6**(3), 19–32 (2005)
30. Obreiter, P.: Extending truple spaces towards a middleware for eCommerce. Thesis, University of Karlsruhe (2000)
31. Oliveira, L., Lopes, L., Silva, F.: P^3: parallel peer-to-peer an internet parallel programming environment. In: Proceedings of the Workshop on Web Engineering and Peer-to-Peer Computing (2002)
32. Plaat, A.: Optimizing parallel applications for wide-area clusters. In: Proceedings of the 12th. International Parallel Processing Symposium, p. 784 (1998)
33. Sankaralingam, K., Yalamanchi, M., Sethumadhavan, S., Browne, J.C.: Pagerank computation and keyword search on distributed systems and P2P networks. J. Grid Computing **1**(3), 291–307 (2003)
34. Schmidt, C., Parashar, M.: Enabling flexible queries with guarantees in P2P systems. IEEE Internet Computing, Special issue on Information Dissemination on the Web (3), 19–26 (2004)
35. Sterck, H.D., Markel, R.S., Phol, T., Rüde, U.: A lightweight Java taskspaces framework for scientific computing on computational Grids. In: Proceedings of the ACM symposium on Applied computing, pp. 1024–1030 (2003)
36. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of the ACM SIGCOMM Conference (2001)
37. Tolksdorf, R., Glaubitz, D.: Coordinating web-based systems with documents in XMLSpaces. In: Proceedings of the 6th International Conference on Cooperative Information Systems (2001)
38. Wilkinson, B., Allen, M.: Parallel programming: Techniques and applications using networked workstations and parallel computers. Prentice Hall (2004)
39. World Wide Web Consortium. Document Object Model

(DOM) Level 2 Core Specification. W3C Recommendation (2000). http://www.w3.org/TR/DOM-Level-2-Core
40. Yalamanchilli, N., Cohen, W.W.: Communication performance of Java-based parallel virtual machines. Concurrency – Practice and Experience **10**(11–13), 1189–1196 (1998)
41. Zhang, Y., Li, X.S., Marques, O.: Towards an automatic and application-based eigensolver selection. In: LACSI Symposium (2005) accepted