gluepy: A Simple Distributed Python Programming Framework for Complex Grid Environments

Ken Hironaka, Hideo Saito, Kei Takahashi, and Kenjiro Taura

The University of Tokyo
{kenny, h_saito, kay, tau}@logos.ic.i.u-tokyo.ac.jp

Abstract. Problem-solving frameworks in large-scale and wide-area environments must handle connectivity issues (NATs and firewalls), maintain scalability with respect to connection management, accommodate dynamic processes joining/leaving at runtime, and provide simple means to tolerate communication/node failures. All of the above must be presented in a simple and flexible programming model. This paper designs and implements such a framework by minimally extending distributed object-oriented models for maximum *generality* and *flexibility*. To make parallelism manageable, we introduce an implicit serialization semantics on objects to relieve programmers from explicit synchronization, while avoiding the recursion deadlock problems from which some models based on active objects suffer. We show how this design nicely incorporate dynamically joining processes. In our implementation, participating nodes automatically construct a TCP overlay so as to address connectivity and scalability issues. We have implemented our framework, gluepy as a library for Python. For evaluation, we show on over 900 cores across 9 clusters with complex networks (involving NATs and firewalls) and process managements (involving SSH, torque, and SGE) configurations, how a simple branch-and-bound search application can be expressed simply and executed easily.

1 Introduction

Grid environment is a complex environment in which to program. Resources are large scale and distributed across multiple clusters. Connectivity among them is restricted by NAT and firewalls. Sites run different resource management software with different policies, and available resources change constantly during computation. Problem-solving environments on the Grid must be designed so the above complexities are made manageable in a simple and uniform framework that alleviates the burden of the users. Thus, the following characteristics (among others) must be addressed:

– Allow processes to simply and seamlessly communicate across sites despite the complexity of underlying networks
– Incorporate dynamically joining processes into the computation while providing means handling of node and network failures

– A programming model to handle parallelism simply and concisely

A common approach has been to design a framework that *hides* these issues. Good examples are systems and frameworks for embarrassingly parallel applications [1–3] and their extensions to express dependencies among tasks [4, 5]. They normally require no programming effort for coordination. A slightly more general approach has been to design programming frameworks specific to certain application domains [6] and coordination models [7]. These frameworks are perfect when the problem at hand naturally fit their model. On the other hand however, they usually provide no or limited means for individual subtasks to communicate with each other. Thus, implementing the coordination among subtasks must frequently resort to using "out-of-band" means (e.g., file transfer, ad-hoc CGI, etc.), making the code awkward and error-prone.

Another approach, which we pursue in this paper, is to leverage a general programming language and conventional parallel/distributed programming concepts, and minimally extend them for issues in large-scale wide-area environments. Object-oriented languages are particularly suitable for this purpose, as they have a general and an accepted model of communication (i.e., remote method invocation or RMI in short). Yet existing parallel RMI-enabled frameworks[8, 9] do not sufficiently address the aforementioned issues (connectivity, scalability, and dynamic processes). Furthermore, managing parallelism and asynchroneity with distributed objects is still not trivial, and race-conditions and deadlocks are commonplace.

We have implemented gluepy, a Python framework with addressing these issues as its primary objective and with the following contributions:

– Objects have implicit serialization semantics. Parallelism is expressed via asynchronous procedure/method calls, using primitives commonly known as *futures* [10]. Yet the semantics eliminate the need for explicit synchronization code. The underlying execution model is still based on passive objects + threads familiar to many programmers. Our design thus does not suffer from the self-recursion deadlocks that some active object-based models do [11, 9].
– An object signalling mechanism that provides simple means by which asynchronous events, such as new process joins, may be handled. It is designed so as to retain the comfortable programming style of using asynchronous method invocations to manage and schedule tasks without resorting to low level event-handling loops.
– A TCP overlay network is built among participating nodes to realize scalable and seamless communication among all participating nodes. To incorporate resources reachable via SSH, it can use SSH port forwarding where specified.

Our experimental platform includes nine clusters with over 900 CPU cores. Many of them perform IP filtering to various degrees, and some only have private addresses. It also includes an almost completely confined cluster that can be reached only by first logging in to its gateway via SSH, then logging in to a cluster frontend via SSH, and finally submitting jobs via a batch scheduler.

The main result of this work is a simple scripting environment that can coordinate processes spread across such complex environments. We implemented and deployed, with little effort, a combination optimization problem solver on our platform of over 900 CPU cores. The core of the solver is an optimized sequential program written in C++ that won the 3rd Grid Plugtest Contest [12]. Python code in our framework merely schedules the underlying C++ processes and exchanges solutions found among participating processes. The glue code is very concise and has only 250 lines, the core of which is shown in Section 3.

This paper is organized as follows. We discuss existing problem solving frameworks on the Grid in Section 2. In Section 3 and 4, we present the model and the implementation of our framework. We evaluate our work in Section 5 and conclude in Section 6.

2 Related Work

2.1 Flexible Inter-Process Interactions

In order to address a large range of applications on the Grid, programming frameworks need to provide a simple yet flexible means by which processes may interact. In the realm of Grid enabled bash schedulers [2, 3], users can express inter-task dependencies in a simple script file [4]. However, inter-task interaction and communication means are limited to passing intermediate files among tasks.

The master-worker model allows close interaction between the master and its workers in the form of tasks, and is an accepted paradigm for its simplicity [13]. As such, many frameworks specialize in this type of application [14–17]. However, if the master and worker require frequent interaction, the assigned tasks must be artificially broken into smaller sub-tasks. Some frameworks [14, 16] enable messages to be sent between the two parties, yet this often results in cluttered code.

Satin [18] and distributed-Cilk [19] are frameworks for distributed divide-and-conquer computation. However, the model's applicable problem set is relatively small, and inter-task interaction is limited to times when tasks divide and merge.

In our proposal, we argue in favor of distributed object oriented programming. Communication is made transparent in the form of RMIs, and objects may invoke upon each other. By utilizing asynchronous invocations with futures, processes may freely interact with each other while taking advantage of the simplicity of method invocations.

2.2 Managing Parallelism for distributed objects

Extending an exisiting object oriented programming language for distributed computing is a popular approach. With Java, Ibis RMI [8] and ProActive [9], with Python, DisPyte [20] are notable examples. In these frameworks, parallel is expressed via asynchronous RMIs. However, this often results in race-conditions, making it necessary to use mutual exclusion. The active object [9] model takes an approach such that each object has a dedicated thread for execution, yet this can easily induce deadlocks (e.g.: recursive calls).

We propose a novel synchronization semantics for objects where *at most* 1 thread can operate on an object at any given time. When a thread performs an invocation on the object, it must first acquire its ownership. This implicitly achieves mutual exclusion. However, when the owner thread blocks in the object's context, the ownership is temporarily released, thus preventing deadlocks.

2.3 Handling process joins/failures with ease

On Grid scale computing, where resources fluctuate constantly, it is paramount that handling of process joins and leaves is made simple for the user. Satin [18], which utilizes the divide-and-conquer model, transparently re-executed lost subtasks while handling load balancing via Random Work-Stealing [21]. However, it is difficult to extend this approach beyond the divide-and-conquer model.

Master-worker frameworks like Jojo2 [14] handle worker joins/leaves via event handlers. This necessitates low-level event-driven loops with explicit mutual exclusions; thus cluttering the code and rendering it hard to understand.

We propose, in conjunction with our synchronization semantics, a signaling mechanism for each object, which unblocks an arbitrary thread blocking in the object's context. Thus it is possible to handle asynchronous events such as node joins while adhering to our implicit synchronization semantics. Additionally, process failures are abstracted as exceptions to method invocations. This widely accepted semantics nicely integrates failures into conventional programming.

2.4 Resolving Connectivity on the Grid

Realizing communication on the Grid, where connectivity is limited by NATs and firewalls, in a scalable manner is a difficult task. PadicoTM [22] enables distributed computing using CORBA on various network hardware, yet does not take connectivity issues into account. ProActive [9] requires users to hand write descriptor files that specify connectable points, yet this is a high burden on the user. SmartSockets [23] attempts to establish connectivity on the Grid by transparently attempting various methods to establish TCP connection(s). However, none of the above take connection scalability into account, which becomes increasingly important with hundreds of coordinating processes.

We propose to construct an overlay network over which communication is routed transparently. Each processes establishes a small number of connections, and address the connection scalability issue. Connectivity is achieved by a high probability via our overlay construction scheme.

3 The Programming Model

In this section, we present a distributed object-oriented framework that operates in a NAT/firewall-prone environment with dynamically joining/leaving nodes.

To the user, we provide a seamless view of the underlying environment, and present a set of simple interfaces by which nodes may communicate via RMIs, new nodes may join, and failing nodes are detected.

Like other distributed object-oriented frameworks, our model provides remote objects and communication among them are abstracted in the form of RMIs [9, 8]. However, we address a number of topics important to Grid-enabled programming that were not, or only partially discussed before.

3.1 Synchronization and Asynchronous Event Handling

In the context of parallel computing, parallel and asynchronous RMIs are crucial. Yet, manipulating asynchronous RMIs is still a non-trivial task. Some implementations allow the users to define callbacks for when the results are available. However, this requires using locks to handle critical data. To resolve this issue, there are future primitives that allow the invoking thread to block for results when they become necessary; the invoking thread may perform other computation in the mean time. Its advantage is that a single thread is in control of the entire flow, and the transition from sequential programming is natural.

This does not resolve the issue on the RMI handler's standpoint. Since a remote object may receive an incoming invocation handled by an independent thread at any time, the programmer must use locks for objects that *might* receive incoming RMIs. To resolve this problem, some models [9] have implemented *active objects* where there is a dedicated thread for each object. The dedicated thread handles all incoming RMIs sequentially. However, this model easily creates deadlocks when an RMI handler also is an RMI invoker.

Yet another issue arises when the program has to handle asynchronous events, such as new node joins. The RMI-based model alone cannot handle these events. One possible approach is to handle RMI callbacks, node failures, and node joins all in one single event driven loop, like in many other master-worker frameworks. The obvious advantage is that a single control thread does all operations, eliminating locks and conditional variables. However the programmer must take care so that event handlers do not block, and the natural flow derived from sequential programming is completely lost.

We summarize the qualities favorable in a distributed object-oriented model.

– provides future primitives for expressing parallelism
– allows objects to be accessed mutually exclusively without explicit locks
– avoids unpleasant deadlocks induced by implicit serialization semantics above
– may handle asynchronous events without low-level event handling models

In the proposed model, at most one thread may run on an object at a time; the thread implicitly acquires the object's ownership for the duration. However, if this thread blocks while in the scope of a method, it temporarily releases the ownership, and another pending thread is permitted to run. When there are more than one pending thread, an arbitrary thread is scheduled, and acquires the ownership. We supply future primitives by which threads may block for results.

Finally, we provide a signaling mechanism for each object, by which a thread blocking on future primitives in the object's context is made to unblock.

When an asynchronous RMI is performed, the invocation returns immediately with a future object. To do an RMI `fib` on an object `foo`, do the following.

```
future = foo.fib.future(args)
```

In order to obtain the results for the asynchronous RMI, we do

```
result = future.get()
```

If the results are not available, the call will block until they are. For scheduling, it is also very common that one waits for an array of future objects simultaneously, until any result becomes ready. To this end, we provide a global `wait` primitive that takes a list of future objects, blocks until at least one of the futures' results are available, and returns a list of futures whose results are available.

```
ready = wait(futures)
```

Finally, each object is provided with a signaling primitive invoked by

```
obj.signal()
```

This forcefully unblocks a thread that is blocking on `wait` in the object's context. If no threads are blocking at that time, the *next* thread that calls on `wait` in its context will be woken. The woken thread will contest for and reaquire the object ownership, after which the unblocked `wait` primitive will return None.

This serialization semantics eliminates the need for explicit locking. This is similar to that of active objects, adopted in some object-based languages [9]. Active objects, however, easily lead to an unpleasant behavior called self-recursion deadlocks. In contrast, our model allows another thread to run on an object when the current thread blocks in the midst of a method execution (a synchronous RMI, call on `wait`, or `future.get`). This property prevents deadlocks due to such recursive calls. With respect to atomicity, a thread is guaranteed to have exclusive control in between potentially blocking operations. Thus, the object's state may be modified without worrying about races.

Furthermore, a special method `signal` allows to unblock a thread currently waiting on the object. This is similar to the semantics of UNIX signals, which unblocks threads blocking on some I/O system calls (e.g., read). This can be used to wake a blocking thread for handling of some event(s).

3.2 Failure Semantics and Bootstrapping Nodes

For RMI failures and object lookups, we utilize the semantics widely accepted in existing RMI frameworks. Node failures are commonplace on the Grid, and simple means for handling them must be presented to the programmer. To this end, node failures are abstracted as exceptions for RMIs to objects on failed nodes. The user may catch such exceptions for failure handling. Aside from uncaught application-level exceptions in an RMI, when any of the below failures occur during an RMI, a `RemoteException` is raised on the caller side. In all cases, they may be handled by the invoker to perform recovery or evasive measures in the regular Python semantics.

1. The communication route between the caller and the callee nodes is broken
2. The callee node fails during execution

Another crucial issue for Grid-computing with dynamically joining nodes is bootstrapping a node. In distributed object-oriented models, this translates to obtaining the *first reference to a remote object*. In our framework, any remote object may be *published* with a human readable string name as follows:

```
object.register("any_name")
```

A process may obtain a reference to a published object using the name.

```
ref = RemoteRef("any_name")
```

This way, the newly joining node may obtain the reference to an existing object and thus join the computation by notifying existing members of its joining.

3.3 Sample Code

Using our programming model, one can easily implement applications using dynamic processes. We show one of such examples in Figure 1(a), a simple yet complete template for master-worker applications. The master initiates work on each worker, and results are collected using futures. The code can accommodate node joins using the `signal` primitive. Node failures are handled by catching exceptions. It is noteworthy that locks are not necessary, and that by using futures, the master's flow resembles that of a sequential program. With the active object model, where locks are also unnecessary, maintaining this flow is impossible as the master object is in method `run` the entire time; workers will never have a chance to run `nodeJoin`. In comparsion, Figure 1(b) shows the code for a typical master-worker framework. A handler, independently invoked on each event, has to branch of each event type, and the loop must perform explicit mutual exclusion. The event-driven code also destroys natural sequential flow of control.

3.4 Discussion

In our object semantics, atomic blocks do not encompass an entire method block, but rather between potentially blocking operations within a method. Yet we believe this is acceptable semantics. Atomic sections in real life applications are very short (e.g., checking if a given value exists in a map before insertion). Moreover, users are aware of blocking operations in advance (synchronous RMIs, access to futures, calling `wait`). Also, as common practice, it is not favorable to design atomic blocks such that they encompass blocking operations.

In the semantics however, livelocks may still occur, like in cases where a thread infinitely loops in a method without blocking. This is arguably as hard to debug as deadlocks. Currently, we defer this as future work.

```
class Master:
  def __init__(self, jobs):
    self.nodes = []
    self.jobs = jobs

  def nodeJoin(self, node):
    self.nodes.append(node)
    self.signal() # notify join

  def run(self):
    assigned = {}
    while True:
      # dispatch work to available workers
      while len(self.nodes)>0
            and len(self.jobs)>0:
        node = self.nodes.pop()
        job = self.jobs.pop()
        # asynchrous RMI to worker
        f = node.work.future(job)
        assigned[f] = (node, job)

      # wait for any results
      readys = wait(assigned.keys())

      # if got signal, loop back
      if readys == None: continue

      #read ready results
      for f in readys:
        node, job = assigned.pop(f)
        try:
          print "done:", f.get()
          self.nodes.append(node)
        except RemoteException, e:
          # in case of a fault, rerun job
          self.jobs.append(job)

class Worker:
  def work(self, job):
    #do work on job...
    return results

  def run(self, mastername):
    #obtain reference to master and join
    master = RemoteRef(mastername)
    master.nodeJoin(self)
```

a. The core for a simple Master-Worker Program. Classes for the Master and the Worker are shown.

```
class Master:
  def __init__(self, jobs):
      self.jobs = jobs
      self.workers = []
      self.tabs = {}
      self.lock = Lock() # for mutex

  #invoked on new event with arg. e
  def handleEvent(self, e):
      #need mutual exclusion
      self.lock.acquire()
      try:
        #give job to new node
        if e.type == NEW_NODE:
            node = e.node
            #give new job, if any
            if len(self.jobs) > 0:
               job = self.jobs.pop()
               self.tabs[node] = job
               self.giveJob(node, job)
            else:
               self.workers.append(node)

        #handle result and give-out a new job
        elif e.type == JOB_DONE:
            print "done:", e.result
            node = e.node
            #give new job, if any
            if len(self.jobs) > 0:
               job = self.jobs.pop()
               self.tabs[node] = job
               self.giveJob(node, job)
            else:
               self.workers.append(node)

        #re-enque lost job on node failure
        #do not re-enqueue worker
        elif e.type == FAILURE:
            node = e.node
            job = self.tabs.pop(node)
            self.jobs.append(job)
      finally:
        self.lock.release()
```

b. Master-Program template in a typical master-worker framework

Fig. 1. Sample Code and Comparison with typical master-worker framework

Our signal mechanism sends the signal to *objects* rather than to *threads*. This design decision was motivated by the fact that in a distributed non-active object model, threads are ephemeral existences only used to gain parallelism. Thus, the programmer is more concerned about interacting with objects, than threads.

Finally, our model can address a wide range of applications beyond the master-worker model shown in the sample code. For example, more P2P-like applications like distributed island-GA applications, where each node performs GA and periodically do crossovers with other peer nodes, can also be easily expressed using RMIs to each other's object.

4 Implementation

In the following section, we will discuss how we implemented our framework to cope with the physical issues of a Grid environment. In particular, we had to resolve three issues: point-to-point communication in a WAN setting (NATs, firewalls), allowing nodes to join with ease, tolerating abrupt node failures.

4.1 Overlay Network Construction

In our framework, to realize point-to-point communication among all nodes, we automatically construct an overlay using TCP connections. Each participating node establishes connections with a small number of nodes chosen at random (about 10 connections). Analysis has shown that such a scheme will create a connected graph of all participating nodes with high probability [24]. However, some cluster completely filter incoming and outgoing packets, and thus there are no means by which these resources may be connected. For these exceptional cases, we automatically perform SSH portforwarding over which TCP connections are forcefully established. Only for these cases, we require the user to specify the points between which SSH portforwarding is done. However, we view this as a very rare case and only requires one line in a configuration file.

Over this overlay, we implement a routing layer adapted from a reactive routing protocol, AODV [25]. It adapts well to dynamic graph changes and fits our setting where nodes join and leave at will; the lazy routing path construction does not entail broadcast storms in face of high churn. The routing metric is the RTT latency for each TCP link.

4.2 Dynamic node join

For nodes to join the computation, it must first become connected with the overlay. In order to do so, it needs a set of bootstrap peer node information, or *endpoints*, with which it will first connect. In a TCP overlay, this entails obtaining a set of initial (IP, port) pairs. We implemented an *endpoint server* that all nodes access before joining. Each node obtains a set of endpoints. It then adds its *own* endpoint to the server so that other nodes may connect to itself. We provide a number of options for this server. One is an HTTP server. The other is a server built on top of GXP [1][26], a Grid shell. Using GXP, one may log into hundreds of remote servers via SSH and execute commands on them in parallel. It also provides a mechanism with which all nodes may communicate via SSH tunnels. Because all communication is done via SSH, this mechanism can be used even for resources that are not accessible by any other means.

4.3 RMI Fault Detection

In our context of an overlay network, a communication route between 2 points may constitute more than 1 TCP connection, and thus is not trivial to detect RMI faults. We assume that when a node fails, it closes all established TCP connections. For our implementation, an RMI is realized by two protocol messages, the RMI Request and the RMI Return message. Additionally, each RMI is identified by a globally unique id, an *RMIID*. In the implementation, we define an RMI to have failed if *either the RMI handler node fails, or if any of the TCP connections that the RMI Request message has traversed fails.*

[1] http://www.logos.ic.i.u-tokyo.ac.jp/gxp/

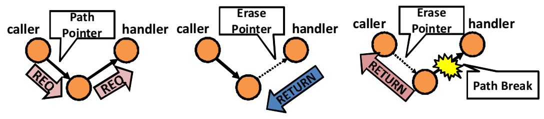


Fig. 2. The left and right figures show how an RMI Request and Return messages are sent along connections. The center figure depicts what would happen if an intermediate connection is lost.

On method invocation, an RMI Request Message is sent towards the object hosting node. As a node forwards the message, it creates a *path pointer* for the RMIID along the connection to the forwarded node.

After the method invocation has been handled by the object hosting node, an RMI Return Message is returned towards the invoker. The message is forwarded along the path on which the RMI Request message came. As it follows the path in reverse, all intermediate nodes erase the path pointer for the RMIID.

When a node fails, all nodes connected to it will detect a connection failure. Each node finds out if any RMI path pointer exists along the failed connection. If such a pointer exists, the node deletes the pointer and sends an RMI Return message carrying a failure exception, back along the stored path. By doing so, the intermediate node can notify the RMI invoker of the RMI failure, and the path created by the RMI Request message is effectively torn down. An image of the entire process is shown in Figure 2.

5 Evaluation

By using the program shown in Figure 1(a) as the base, we have implemented a number of master-worker type applications. In this section, we provide some micro-benchmarks to evaluate our overlay performance. We also evaluate its ability and effectiveness to run in a Grid environment with dynamic resources fluctuations. First, we explain the setup of our experimental environment. In Figure 3, we show the clusters used for our evaluations. Most cluster nodes are Core2Duo 2.13GHz, except for `kototoi` and `mirai` and `hiro` (Xeon 2.33GHz), `istbs` (Xeon 2.4GHz), and `tsubame` (Dual Core Opteron 2.4GHz). It is noteworthy that each cluster has different network administration settings. For example, due to the NAT configuration, most nodes in cluster `kyoto` and `imade` are not accessible from outside. Clusters `kototoi` have global IPs, yet due to the firewall at the gateway router, no incoming connections can be accepted. However, we were able to utilize all nodes in all clusters without any manual configuration; our bootstrapping scheme in Section 4.2 automatically bootstrapped all nodes to the peer-to-peer overlay. Exceptions are cluster `istbs` and `tsubame`, in which all its incoming and outgoing packets on most ports are filtered for security reasons. In order to utilize the two clusters, we enabled configurations for ssh forwarding from one gateway node in each cluster to a node in cluster `hongo`; all other nodes in the cluster connected to the gateway node within its cluster.

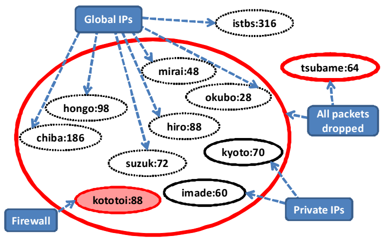


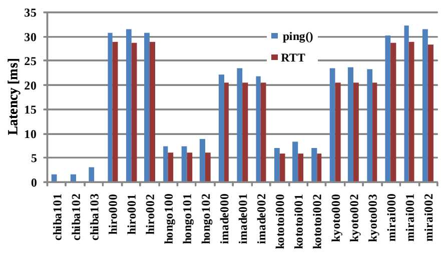
Fig. 3. experiment cluster settings denoted by core counts

5.1 Micro-Benchmark

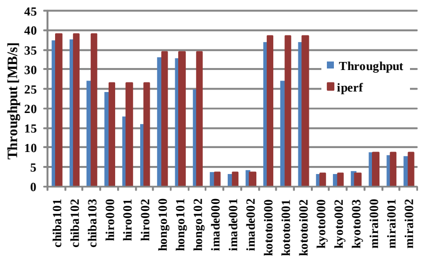
We present our framework's microbenchmarks on our overlay, in particular, its latency overhead and its performance with data-intensive operations. We utilize clusters `chiba`, `hongo`, `kototoi`, `hiro`, `imade`, `kyoto`, and `mirai`. From a node in cluster `chiba`, we made RMIs on objects located on each of the other nodes. We show the latency of a `ping()` method, a no-op, of selected nodes from each cluster (denoted by `clusterXXX`) in Figure 4(a). The actual RTT value is paired to show the ideal minimum. Most nodes were reached within 3 hops on the overlay. The intra-cluster latency for cluster `chiba` was ~150[us]. The hop latency overhead amounts to roughly 1.5[ms], sufficiently small for inter-cluster communication.

We do the same operation using the `send_data()` method, which is a no-op that takes 1 argument, and see the throughput of the data transfer. As its argument, we pass a long string of 100[MB]. The throughput is computed from the method completion time and is shown in Figure 4(b). The arguments have to be serialized (throughput: 78[MB/s]) for communication, and this reduces the maximum throughput for 1Gbit Ethernet links (overlapping serialization and sending would prevent the maximum throughput to drop to 40[MB/s], this remains to be our future work). The maximum possible point to point throughput, that accounts for serialization, calculated from iperf is paired to show the ideal maximum. For nodes where 1Gbit Ethernet is available, a value close to this is obtained. `imade`, `kyoto`, `mirai` have gateway switches of only 100Mbit. `imade` and `kyoto` have particularly anemic bandwidth where even iperf registers 3.5[MB/s]. Within the same cluster, some nodes have much lower throughput (e.g., chiba103, hongo102, kototoi001). This is because these nodes take multiple hops on the overlay, and the store-and-forward routing diminishes the throughput on each hop.

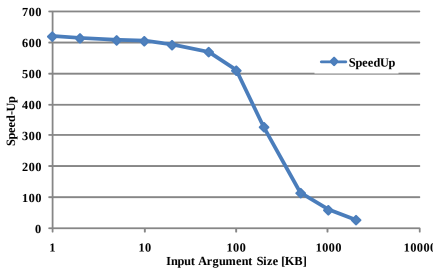
We submit 10,000 invocations of a `send_and_wait()` method, same as `send_data()` but additionally sleeps 1[s], in a master-worker style to see the parallelism throughput when the argument size varies. This measures our framework's tolerance to parallelly handling jobs with large input data. We show the speed-up for 710 cores in Figure 4(c). The speed-up drops dramatically from around 50[KB]. This is expected as the master's maximum bandwidth(40[MB/s]) becomes saturated from the size of 56[KB] with 710 workers.



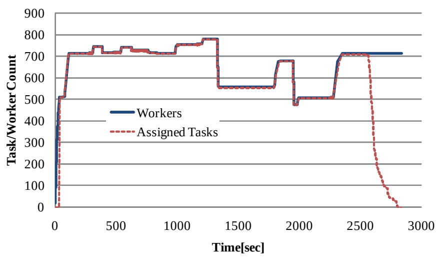
a. Latency of `ping()`



b. Throughput for `send_data()` for 100MB string



c. Speedup for parallel `send_and_wait()`
for varying argument size



d. Active workers/assigned nodes
for FT master-worker

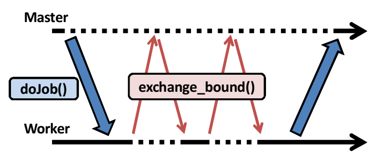
Fig. 4. Benchmark Results

5.2 Fault-Tolerance

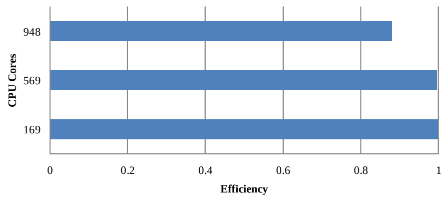
We evaluated our framework's ability to handle dynamic node insertions and failures. A single Master object dynamically distributes 10,000 tasks to Worker objects. For node addition, we used the method described in Section 4.2. For node failures, we simply killed the processes on nodes abruptly without warning. No tasks were lost during this experiment. We give the time series for the number of Workers running, and the number of tasks allocated by the Master to each Worker in Figure 4(d). The figure shows that as the number of workers fluctuates, the master schedules the tasks accordingly. The failure detection latency was in the order of milliseconds. All fault detection is done at the programmer level by detecting faults as exceptions as in Figure 1(a). Moreover, it is noteworthy that all participating nodes are interconnected on a TCP overlay and direct connection is not necessary for fault-detection due to the scheme in Section 4.3.

5.3 Real-life Application

Many real-life distributed applications require parallel tasks to interact with each other. An example of such applications is a problem solver that uses branch-and-bound. In these applications, it is imperative that all parallel solvers share the latest bound information for efficient computation; these are applications that



a. Master-worker RMI interaction



b. Execution efficiency

Fig. 5. Evaluation of the Permutation Flowshop Solver

require periodic communication among nodes. Such applications are impossible to express in programmingless frameworks where inter-task communication is not permitted, or in divide-and-conquer type frameworks where communication is limited to immediate parent and children tasks. As discussed in [13], such applications can be expressed naturally in master-worker models, but there are virtually no frameworks that can handle the hostile network environment (NATs, firewalls, and IP filtering) in our experimental settings.

As our case study, we have taken on one such problem, the Permutation Flowshop Scheduling Problem. (P-FSP). P-FSP is a problem where n jobs have to be processed on m machines in the same order. This problem entails finding a schedule which minimizes the makespan (execution time) with proof. The solver does parallel branch and bound in a master-worker model, where each worker receives a small section of the search space.

When a worker first joins, it first receives a task to solve. The worker and the master periodically (every 60 seconds) exchange bound information used for the branch-and-bound. When a worker finishes or aborts its given task, the master gives it its next task. The flow is expressed in Figure 5(a).

Because the computation would take months(perhaps years), fault-tolerance was a crucial part of the design. The master and the worker programs, which won the 3rd Grid Plugtest Competition [12], had already been implemented in C++. We used our framework to serve as the glue to integrate the two and deploy it on our platform. The code in Python took merely about 250 lines.

We ran the program on three different configurations: 168, 569, and 948 cores. The only necessary network configuration, in the 948 core case, was to specify the SSH portforwarding settings for clusters istbs and tsubame, which took a mere 6 lines. The rest of the deployment was taken care of automatically and successfully created a connected graph of all processes.

We present an evaluation using a relatively small randomly generated problem instance of ($n = 28$, $m = 20$). To measure the performance of our framework, rather than of the algorithm, we calculated the computation efficiency as $\frac{C}{NT}$, where N, T, and C resepctively denote the core count, the completion time, and the cummuative computation time across all cores. The results are shown in Figure 5(b). With 948 cores across 9 sites, 88% efficiency is maintained.

6 Conclusion

We have presented a programming framework that aims at simple and flexible programming in a Grid environment with limited network connectivity, dynamic node joins, and node failures. We provide simplicity, without the loss of generality, by extending a widely accepted object-oriented language, Python for wide-area parallel computing. Parallelism is expressed in the form of RMIs with the aid of futures for a natural transition from sequential programs. Accesses to objects are implicitly serialized without the fear of deadlocks, effectively eliminating locks. We provide simple means to add nodes to ongoing computation, as well as to tolerate node failures without jeopardizing the computation. We automatically construct a TCP overlay and realize transparent communication among nodes even in the face of NATs and firewalls.

Taking a branch-and-bound optimization application as an example, we showed that our framework enables quick and effective development of parallel applications in large Grid environments with 900 cores, despite network hindrances like NATs and firewalls. A prototype for gluepy is currently available from its homepage:

http://www.logos.ic.i.u-tokyo.ac.jp/~kenny/gluepy.

Acknowledgements

This research was partially funded by “New IT Infrastructure for the Information-explosion Era” of the MEXT Grant-in-Aid for Scientific Research on Priority Areas.

References

1. : OpenPBS. http://www-unix.mcs.anl.gov/openpbs/
2. Ayyub, S., Abramson, D., Enticott, C., Garic, S., Tan, J.: Executing Large Parameter Sweep Applications on a Multi-VO Testbed. In: CCGRID ’07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, IEEE Computer Society (2007) 73–82
3. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the Condor experience. Concurrency - Practice and Experience **17**(2-4) (2005) 323–356
4. : DAGMan. http://www.cs.wisc.edu/condor/dagman/
5. : Taverna Project Website. http://taverna.sourceforge.net/
6. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI. (2004) 137–150
7. Egner, M.T., Lorch, M., Biddle, E.: UIMA Grid: Distributed Large-scale Text Analysis. In: CCGRID ’07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, IEEE Computer Society (2007) 317–326
8. van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T., Bal, H.E.: Ibis: a Flexible and Efficient Java-based Grid Programming Environment. Concurrency and Computation: Practice and Experience **17**(7-8) (2005) 1079–1107

9. Huet, F., Caromel, D., Bal, H.E.: A High Performance Java Middleware with a Real Application. In: Proceedings of the Supercomputing conference. (2004)
10. Taura, K., Matsuoka, S., Yonezawa, A.: ABCL/f: A Future-Based Polymorphic Typed Concurrent Object-Oriented Language: Its Design and Implementation. (1994)
11. Agha, G.: Actors: a Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
12. : 3rd Grid Plugtest Report. http://www-sop.inria.fr/oasis/plugtest2006/plugtests_report_2006.pdf
13. Aida, K., Osumi, T.: A Case Study in Running a Parallel Branch and Bound Application on the Grid. In: SAINT '05: Proceedings of the The 2005 Symposium on Applications and the Internet, IEEE Computer Society (2005) 164–173
14. Aoki, H., Nakada, H., Tanaka, K., Matsuoka, S.: A Programming Environment with Dynamic Node Configuration for Hierarchical Grid: Jojo2. In: SACSIS. (2006)
15. Linderoth, J., Goux, J.P., Yoder, M.: Metacomputing and the Master-Worker Paradigm. Technical Report ANL/MCS-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory (February 2000)
16. Nakada, H., Matsuoka, S.: A Java-based programming environment for hierarchical Grid: Jojo. In: CCGRID. (2004) 51–58
17. Shudo, K., Tanaka, Y., Sekiguchi, S.: P3: P2P-based Middleware Enabling Transfer and Aggregation of Computational Resources. In: CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid - Volume 1, IEEE Computer Society (2005) 259–266
18. Wrzesinska, G., van Nieuwpoort, R.V., Maassen, J., Kielmann, T., Bal, H.E.: Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments. International Journal of High Performance Computing Applications (IJHPCA) **20**(1) (2006)
19. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. In: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation. (June 1998) 212–223 Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.
20. Fuhner, Tim, Popp, Stephan, Jung, Thomas: A novel framework for distributing computations dispyte distributing python tasks environment. Journal of Computational Electronics **5**(4) (December 2006) 349–352
21. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient Load Balancing for Wide-area Divide-and-conquer Applications. In: PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming, ACM (2001) 34–43
22. Denis, A., Pérez, C., Priol, T.: Padicotm: an open integration framework for communication middleware and runtimes. Future Gener. Comput. Syst. **19**(4) (2003) 575–585
23. Maassen, J., Bal, H.E.: Smartsockets: Solving the Connectivity Problems in Grid Computing. In: HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing, ACM (2007) 1–10
24. Horita, Y., Taura, K., Chikayama, T.: A Scalable and Efficient Self-Organizing Failure Detector for Grid Applications. In: GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, IEEE Computer Society (2005) 202–210
25. Perkins, C.: Ad-hoc On-demand Distance Vector Routing. In: MILCOM '97 panel on Ad Hoc Networks, Nov. 1997. (1997)
26. Taura, K.: GXP : An Interactive Shell for the Grid Environment. IWIA **00** (2004) 59–67