# PLinda User Manual *

Thomas Brown  Karpjoo Jeong  Bin Li  Suren Talla  Peter Wyckoff  Dennis Shasha

Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

December 13, 1996

## Contents



*This work was supported, in part, by the National Science Foundation under Grants IRI-9224601.

# 1 Introduction

Persistent Linda (PLinda) is a programming environment for writing fault-tolerant distributed/parallel programs that may be run on networks of workstations (NOWs). PLinda is a set of extensions to the Linda parallel programming model[16] and PLinda/C++ (Fortran77 respectively) is an implementation combined with the sequential language C++ (Fortran77 respectively).

For the reader who is not familiar with parallel programming and/or the Linda model we introduce these in sections 2 and 3 respectively.

The rest of this document is organized as follows: section 4 introduces the PLinda system architecture. Section 5 introduces the basic concepts of the PLinda model. Section 6 concerns tuning the execution of PLinda programs. Section 7 shows how the PLinda model can be used to harness idle cycles on NOWs. Section 8 gives the semantics of each PLinda operation. Sections 9, 11, and 12 explain how to write PLinda code, how to compile PLinda code, and how to use the PLinda user interface respectively. In section 10, we explain programming in PLinda/Fortran77. Section 13 explains how to obtain and install PLinda. Before we start explaining the technical details of using the PLinda system, we explain the motivation for the PLinda system model and define the types of applications for which PLinda is well suited.

PLinda supports shared memory-based parallel processing on NOW. Its design focuses on the following issues:

- **Fault tolerance**. The probability of failure grows as execution time or the number of processors increases. Although a single processor failure is in fact rare, the possibility of failure cannot be ignored if execution takes a couple of months or involves a few hundred workstations.

  The PLinda system allows the programmer to build a parallel application in such a way that the application can survive failure during execution (in other words, it can continue execution without restarting from scratch) and finish its task correctly. A "failure-experienced" execution is considered to be correct if its resulting effect is the same as that of a possible "failure-free" execution.

- **Effective utilization of workstations**. Unlike supercomputers, workstations are not reserved solely for parallel computation on NOW[1]. The PLinda system enables the user to use NOW as one parallel virtual shared memory machine whose processors are "idle" workstations. The virtual machine configures itself automatically to use only idle processors. That is, new processors are added to the virtual machine as they become idle, and processors on the virtual machine are deleted when they fail or their owners resume working on them. This dynamic reconfiguration is transparent to the user (except for performance).

- **High performance**. The first concern and main reason for parallel applications is performance. Therefore, the facility for fault tolerance and effective workstation utilization should not degrade performance significantly. Most existing fault tolerant techniques are designed for mission-critical applications which require high availability or distributed applications which require high data reliability. In general, they entail high overhead either during normal execution or on recovery after failure.

  The PLinda system is designed to allow the programmer to customize fault tolerance for each application (i.e., take advantage of each application's characteristics) for better performance.

[1]We may dedicate workstations to parallel computation, but only by sacrificing cost-effectiveness.

## 2 Parallel Programming

Parallel programming works by spreading a computation over a number of processors. For example, a parallel matrix multiplication might replicate one matrix over a set of processors, break the other matrix into pieces assigning each piece to a processor, have each processor do the multiplication for its assigned piece, and then combine the results. This example brings up three fundamental issues of parallel programming: how are concurrent threads of execution created on remote machines, how do these threads communicate, and how do these threads coordinate? In this section, we address these issues and introduce a popular form of parallel programming, the bag-of-tasks model.

This is a very brief description of parallel programming; the interested reader may refer to [5] for a more comprehensive treatment.

### 2.1 Process creation

In the PLinda model parallel tasks are created at the file executable level. The programmer may create an instance of an executable by making a PLinda system call with the file name and its arguments. This is very similar to the method used in PVM [14]. This type of parallelism is know as *task parallelism.* In High Performance Fortran (HPF) [10] parallelism is achieved by writing parallel loops that operate on vectors. This is known as *data parallelism.*

### 2.2 Communication

In the PLinda model communication is facilitated through shared memory. Virtual distributed shared memory (DSM) provides the programmer with a shared address space for all processes, even though the hardware of the system may not physically provide shared memory. With DSM, the programmer is removed from the details of shuffling data back and forth between processors. For the matrix multiplication example, both matrices would be written to the shared memory and then all the processes involved in the computation would read the data they need.

### 2.3 Synchronization

The PLinda shared memory actually consists of typed tuples that are created/destroyed by processes during a parallel computation. A distributed semaphore could be implemented in the shared memory by creating a tuple representing the semaphore. In a message passing environment a semaphore could be implemented by having one process be the controller of the semaphore and then other processes could send messages requesting the use of the semaphore. The process would queue requests that cannot be immediately satisfied. Even in this simple example, it should be obvious that the PLinda approach is somewhat simpler. We are not trying to espouse the view that shared memory is a panacea for the complications involved in parallel processing, but it does relieve some of the problems. Again, there is a cost involved, but for medium to coarse grained parallel problems this can be amortized over the time spent doing computation.

### 2.4 Bag-of-tasks-model

In the bag-of-tasks model a problem is broken into T tasks (this is not always possible, but in many cases it is easy). Each processor involved in the computation performs an infinite loop of getting a task, performing computation for that task and outputting the result. If the number of processes is P and $T \gg P$, faster machines will do more tasks than slower machines and the computation will be load balanced. This model is often used in the master-worker paradigm where a master process creates multiple identical worker processes.

## 3 Brief review of Linda

Linda is a parallel programming model based on virtual shared memory called *tuple space*. Processes communicate and synchronize by creating data objects called tuples in the tuple space and retrieving them from the tuple space *associatively*. A tuple contains a sequence of typed data elements where the data types are basic types such as integers, floats, characters, and arrays of these.

Linda provides four basic operations: `out` for tuple creation, `eval` for process creation, `in` for destructive retrieval and `rd` for non-destructive retrieval. In the destructive case, data objects are removed from tuple space on retrieval. Destructive retrieval is often used for a "point-to-point" style of communication; in contrast, non-destructive retrieval is used for a "broadcast" style of communication. There are also two asynchronous operators **rdp/inp** which return true if a matching tuple was found in tuple space and false otherwise. There use is discouraged due to the phantom tuple problem [2]. Here is a brief description of the four basic operations:

1. **out**. Takes a sequence of typed expressions as arguments. It evaluates them, constructs a tuple from them, and inserts the tuple into tuple space.

2. **eval**. Like `out`, `eval` creates a tuple from its arguments, but a new process is created to evaluate the arguments. In Linda, this is the only way to create a new process.

3. **in** and **rd**. Take a typed pattern for a tuple as their argument and retrieve a tuple to match the pattern in an associative manner. A pattern is a series of typed fields; some are values and others are typed place-holders. A place-holder is prefixed with a question mark. For example,

   ```
   ("string", ?f, ?i, y).
   ```

   The first and last fields are values; the middle two fields are place-holders. On retrieval, place-holders are set to the values in the the corresponding fields of the matching tuple, respectively. If there are multiple matching tuples, then one of them is randomly selected and retrieved. If no matching tuple is found, then `in` and `rd` block until a matching tuple is inserted. The difference is that `in` is destructive (i.e. removes the tuple) while `rd` is not.

Figure 1 shows a sample "producer/consumer" program in Linda. In the code, the main function creates two processes using the `eval` operation: "producer" and "consumer". Then, the two processes communicate and synchronize using tuples.

For a more detailed description of Linda, refer to[2, 4, 5].

## 4 The PLinda model

Processes cooperating to solve a problem by communicating through shared memory is the essence of the Linda and PLinda models. When PLinda is used in a fault tolerant manner, two major concepts are added to the model: fault tolerant tuple space and resilient processes.

```
real_main {
  eval(producer());
  eval(consumer());
  out("start"); // broadcast
}

int producer() {
  rd("start");
  while(1) {
    out("job",produce_job()); // produce a job
  }
}

int consume() {
  int job;
  rd("start");
  while(1){
    in("job", ?job);
    compute_on_job(job);
  }
}
```

Figure 1: A Producer/Consumer Program in Linda

```
gid ts_handle;
ts_handle = create_group("my tuple space");
pl_out[ts_handle]("sample tuple", 1(int),2.5(float));
int ivar;
float fvar;
pl_in[ts_handle]("sample tuple",?  ivar(int),?  fvar(float));
destroy_group(ts_handle);
```

Figure 2: Creation and Destruction of a Tuple Space in PLinda

We first describe the non fault tolerant aspects of PLinda: communication/coordination through tuple space and process creation. We then describe the issues involved in making an application fault tolerant and how PLinda's transactions and continuations, fault tolerant tuple space, and continuation recovery address these issues.

### 4.1 Communication - multiple tuple spaces

Linda does not support mechanisms to group tuples or partition tuple space. All the tuples are always accessible to every process[2]. Thus, processes may conflict with each other by accident if they happen to use the same pattern of tuples for different purposes. Such cases are likely in large scale applications which include a large number of components or are developed over a long period of time.

PLinda allows an application to use multiple tuple spaces, called *tuple groups*. Every access to a tuple group requires a handle. We explain how to create tuple groups, how to use tuple groups to communicate, and briefly describe how transactions impact communicating through tuple groups.

Processes create a tuple group explicitly using the `create_group` operation as shown in Figure 2. The operation returns a handle to the new tuple group (the data type of the handle is called `gid`). There is also a `destroy_group` function which takes a handle to a tuple group and destroys the tuple group. Figure 2 also shows how to use the `destroy_group` function.

In PLinda, tuple group handles are first class objects which may be stored in tuple space, and thus, processes can pass `gid`s to other processes. A common programming paradigm in PLinda is for subgroups of processes which are working on the same task to communicate through a tuple group private to them.

The tuple group communication operations are `pl_in`, `pl_rd`, `pl_inp`, `pl_rdp`, and `pl_out`. They are exactly the same as in Linda, except that a `gid` is a required parameter and that array length is not considered during tuple matching. See sections 3 and 8 for more details on these operations.

PLinda transactions cause the `pl_out` and `pl_proc_eval` operation effects to be delayed until an `xcommit` is executed. This reduces concurrency, but is necessary so as to avoid a running process being dependent on a failed process's uncommitted operations. The system needs to abort those operations so that it may restart the failed process, but if other processes were able to access those operations, the system would have to abort the dependent process(es) — a *cascading abort*.

[2] For this reason, communication in Linda is sometimes described as "helium balloon"-style.

### 4.2 Process creation (pl_proc_eval)

Regarding process management, PLinda and Linda differ in the following ways:

- A different unit of parallelism: language expressions in Linda and an OS-level executable file in PLinda.
- Automatic failure detection and backup process restart in PLinda.
- Transactional process creation in PLinda.

First, PLinda uses an OS-level executable file (like `a.out` generated by the `cc` compiler) as the unit of parallelism; that is, a process is invoked to execute an executable file. This design allows the runtime system to exploit the process management facility provided by the underlying OS such as UNIX, instead of providing customized mechanisms. Thus, the design facilitates portability and heterogeneous processing.

For process invocation PLinda supports the `pl_proc_eval` and `pl_arg_rdp` operations. Like `eval` in Linda, the `pl_proc_eval` operation takes a series of expressions to be converted to a tuple (which is called the *argument tuple*). The first field of an argument tuple must be the name of an executable file. The `pl_proc_eval` operation constructs the argument tuple first, and then creates a process to run the specified executable. Using the `pl_arg_rdp` operation, each child process can retrieve the argument tuple associated with the `pl_proc_eval` operation that created it. The `pl_arg_rdp` operation is used exactly in the same way as `pl_rdp`. The argument tuple is private to the process; it is guaranteed that other processes cannot access it even with the `pl_arg_rdp` operation.

Automatic failure detection, process restarting, and transactional process creation will be discussed in the next subsection.

### 4.3 Fault tolerance

In this section, we explain what is needed to make a program fault tolerant and the constructs PLinda provides for fault-tolerance.

There are basically two approaches for fault tolerant languages: transparent and tool based. PLinda uses the latter approach by providing mechanisms whose semantics (called *failure semantics*) are always guaranteed in spite of failure. Using them, the programmer designs applications to be resilient to failure. Compared to transparent approaches to fault tolerance, this kind of tool-based approach usually enables the programmer to develop more efficient code by taking advantage of characteristics specific to the application. A drawback of such approaches is an additional programming burden on the programmer, but in PLinda, this is often negligible.

Fault tolerance for distributed/parallel applications can be divided into three issues:

- Consistent local states for each process— If a failure occurs in any process, the system must be able to restore the process's state which includes its variables and program counter. The system need not restore the state immediately prior to failure, but it must restore the process to a state that is consistent with any external interactions the process may have had. Thus, this must be done in conjunction with restoring a consistent global state.
- Consistent global state— even "fail-stop" processor failure (in which failed processors stop silently without executing malicious operations) usually causes the system state to be inconsistent. For example, suppose process A updated shared memory and failed. Then, the state

of shared memory reflects the updates, but the process that made them is lost. A backup process for the failed process may not remember those updates.

Therefore, the system must be restored to a consistent state. The recovered state should be either a previous consistent state or a state reachable from the beginning of execution.

- Process failure detection and restarting— failures need to be detected and backup process(es) spawned on other machines. Failure detection in some cases must be done by timeout. If the system incorrectly determines that a process P has failed due to timeout (possibly a network partition or a slow machine), the system will need to stop P from interacting with other processes in the future.

### 4.3.1 Transactions and continuations

Transactions are a sequence of operations that are always executed *atomically* regardless of failure. That is, all the operations take effect (called *commit*) or none of them do (called *abort*). No partial effect of a transaction is accessible to other transactions (or, processes) until the transaction commits.

In PLinda if a process fails while executing a transaction, the runtime server detects the failure and aborts the transaction automatically. Runtime mechanisms ensure that no other processes have accessed intermediate updates to the tuple space and the updates are removed from the tuple space on abort. Thus, the resulting effect appears as if the transaction had never occurred. The user will not notice the transaction, provided that there is no user interaction within the transaction (which is usually the case for parallel applications).

In PLinda the `xstart` operation starts a transaction and the `xcommit` operation commits it. The `xcommit` operation takes a tuple as a parameter. This tuple is used to save local state — *continuation*. The runtime system guarantees atomic execution of all the PLinda operations between `xstart` and `xcommit`. Transactions include tuple space access, process spawning, tuple group creation/destruction operations, and local computation. Processes are allowed to run multiple transactions. However, the current design of PLinda does not support nested transactions.

In PLinda a commonly used programming style is to divide the computation of each process into a series of subcomputations which will be atomically (in an "all-or-nothing" manner) executed at runtime[3] Also, local state is saved in private store at the end of each atomic action. On recovery from failure, a process restores the last saved state and resumes execution from after the last committed transaction.

Figure 3 shows an example of PLinda code which is for the master process in the bag-of-tasks programming model. A master process in the programming model usually: (1) creates worker processes; (2) creates task tuples in tuple space which will be grabbed by worker processes; (3) collects result tuples generated by worker processes. In the code the computation of the master process is divided into two transactions.

Suppose that the master process fails while creating task tuples, that is, in the first transaction. Then, the runtime system automatically aborts the transaction. More specifically, it deletes the task tuples that were so far inserted, and destroys the tuple group and all the worker processes which have already been created.

Likewise, if the process fails in the last `for` loop, which is collecting results, the runtime system aborts the second transaction. Then, it inserts any deleted result tuples back into tuple space.

[3] One may wonder why not make the entire computation atomic. Unfortunately, such atomic actions restrict communication with other processes within their execution.

```
gid ts_handle;

xstart;
  ts_handle = create_group("workspace");
  for(i=0; i<num_workers; ++i) {
    pl_proc_eval("worker", ts_handle(gid), i(int));
  }
  for(i=0; i<num_tasks; ++i) {
    pl_out[ts_handle]("task",task_desc(taskDescStruct));
  }
xcommit();

xstart;
  for(i=0; i<num_tasks; ++i) {
    pl_in[ts_handle]("result",?result(resStruct));
  }
xcommit();
```

Figure 3: Master Process in PLinda

However, the abortion does not affect the effect of the first transaction.

In summary, the runtime system guarantees that the master process of Figure 3 can *logically* experience failure only at one of three points during execution: before the first transaction, between the first and second transactions, and after the second transaction.

#### 4.3.2 Consistent global state (fault tolerant tuple space)

In the current implementation the runtime server manages the entire tuple space and saves it to disk periodically — **checkpointing**. That is, the runtime server plays the role of a tuple space server. If the server fails, it restores the latest checkpointed state from disk on recovery and resumes execution from that state — **rollback recovery**.

The checkpointing/rollback-recovery operations and checkpointing interval are application-independent. The runtime system masks failure in the tuple space from applications transparently. Thus, the programmer can ignore tuple space failure when constructing applications.

To avoid restoring an inconsistent global state, the runtime system only checkpoints committed tuple space data. That is, partially completed transaction updates are not reflected in a checkpoint. Thus, on failure the tuple space reflects a state where all the processes had finished a transaction or had not not started any transaction. In the event of tuple space failure, the system must kill all active processes and then let its process resilience mechanisms respawn them. The respawned processes will continue from a state in which they had just completed their last transaction which was checkpointed to disk. Thus, the tuple space and all processes will be consistent with one another.

```
gid ts_handle;

int TransactionNumber = 0;
xrecover(?TransactionNumber(int));
if(TransactionNumber == 0){
  xstart;
    ts_handle = create_group("workspace");
    for(i=0; i<num_workers; ++i) {
      pl_proc_eval("worker",ts_handle(gid), i(int));
    }
  for(i=0; i<num_tasks; ++i) {
    pl_out[ts_handle]("task",task_desc(taskDescStruct));
  }
xcommit(++TransactionNumber(int));
}
if(TransactionNumber == 1){
  xstart;
    for(i=0; i<num_tasks; ++i) {
      pl_in[ts_handle]"result",?result(resStruct));
    }
  xcommit(++TransactionNumber(int));
}
```

Figure 4: Fault Tolerant Master Process in PLinda

#### 4.3.3 Backup processes (continuation recovery)

The PLinda runtime system detects the failure of a process and then respawns the process (we call the new process the **backup process**) to take over the task of the failed process automatically. The backup process will restore the state of the failed process from the last committed transaction and resume execution from after that transaction. The `xrecover` system call restores continuation and returns a boolean indicating whether this process is a backup or the original process.

We now show how to make the master process of the previous subsection fully fault tolerant by using `xrecover`, transactions, and continuations.

In figure 4, the process saves the variable `TransactionNumber` at the end of each transaction. This is the only state needed in this example. But, more variables could have easily been saved. If a failure occurs, the runtime kernel will re-execute the process on another machine. In this case, the `xrecover` call will restore `TransactionNumber` to the value it had at the last commit.

# 5 Implementation

The current prototype PLinda system[4] is based on the client-server architecture model: application processes (compute engines) are clients and the runtime kernel is a centralized server. The system

[4] The public release is available at `http://merv.cs.nyu.edu:8001/~binli/plinda/`.

supports multiple applications at the same time. The prototype server and client library are written in C++ and use TCP/IP for communication and standard UNIX system functions.

The runtime system consists of four major components:

- **The server**. The server (1) detects failures, (2) schedule processes and migrates processes due to daemon status changes as well as failure detection of machines, (3) manages tuple space in a transactional manner, and (4) checkpoints tuple space.

- **Daemon processes**. A daemon runs on each workstation. The daemon monitors the idleness status of the local machine and informs the server of status changes. The daemon currently uses keyboard, mouse and console idle times and CPU load averages as idleness criteria[5].

  The daemon also manages processes on its machine; that is, it creates application processes upon request from the server and kills them when the machine becomes busy. When a machine is busy, the daemon intermittently checks the status of the machine and sleeps the rest of the time so as not to disturb the owner of the machine.

- **Clients**. The PLinda translator converts PLinda operations in application programs to PLinda library calls[6].

- **The administration program**. The administration program allows the PLinda system administrator (1) to monitor the status of every machine, (2) to add and delete hosts from the system, (3) to monitor application process behaviors, (4) to start and kill applications, and (5) to change the tuning parameters (e.g., execution methods) of the server as discussed in Section 6.

Figure 5 illustrates the runtime structure of the PLinda system.

The architecture/OS types supported by the current prototype are SGIs running IRIX, Sparc machines running SunOS and Solaris, DEC Alphas running DigitalUnix (also known as AIX/OSF1), and IBM Compatible PCs running Linux.

To accommodate these different architectures, the runtime system differentiates between executables for the different architectures and converts data that is transfered between machines that have different internal representations, e.g., big endian and little endian machines.

A different subdirectory is created for the executables of each architecture type and the runtime system chooses the correct executable when spawning new processes. This allows the system the flexibility to run in an environment where some architectures share a common file system while other do not. In the case of no common file systems, we could just keep all the executables in the same directory path. This kind of a scheme has already been proposed and used in PVM [18, 8].

Data conversion between machines of different types can be done either by using a common external data representation, as in Sun's XDR routines, or by converting data in place. We have implemented both schemes. In our data-in-place implementation, the client library routines always convert data to and from the format of the machine which the server is running on. We have found that the performance of the latter scheme is slightly better than the former. This is mainly because it minimizes the work of the server which is especially important in the current centralized design.

[5]These criteria have already been used by other work-stealing systems and demonstrated to be effective[1, 6, 3, 7, 9, 17, 19, 20, 21].

[6]The current PLinda translator is only a preprocessor written in Perl and requires the programmer to annotate PLinda operations with variable type information explicitly.

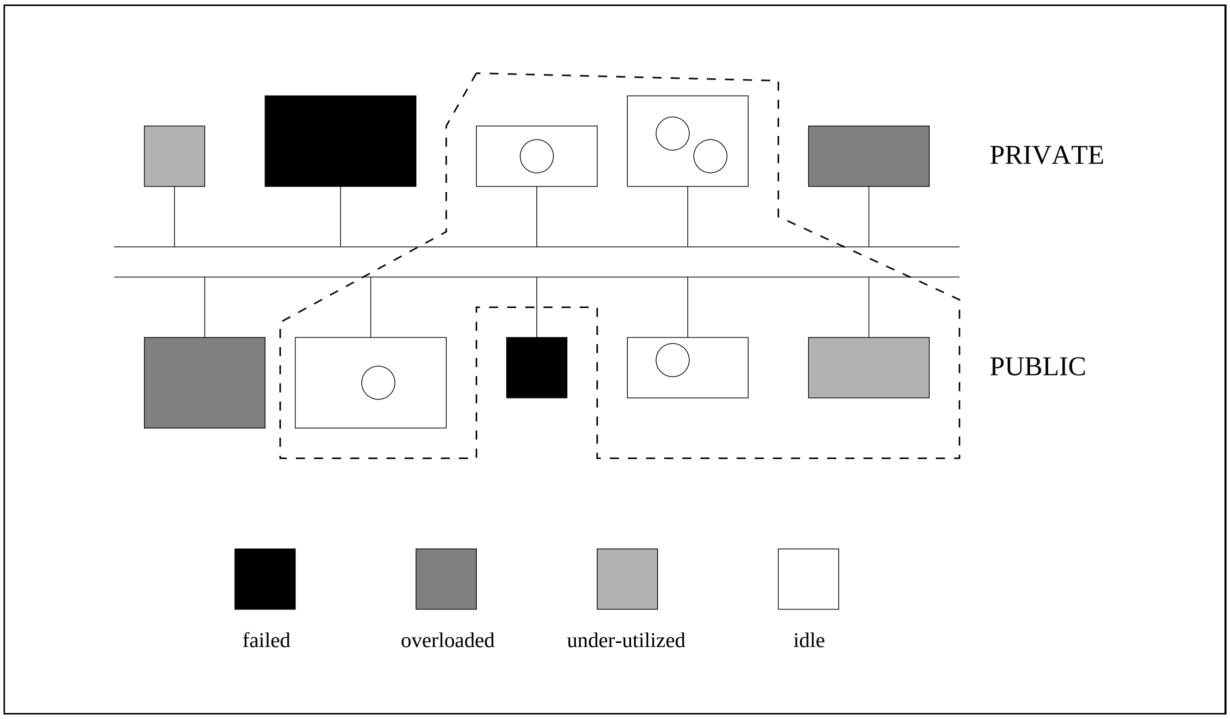


Figure 5: PLinda System Runtime Structure

## 6 Tuning the execution of PLinda programs

Recall that PLinda allows the programmer to control the amount of shared data, the size of data for replication (i.e., contents of continuation), and the frequency of replication operations (i.e., transaction commit points) when designing fault-tolerant processes. The only constraint on the programmer is that commits are required for communication: a process must commit before any of the tuples it has produced can be read by other processes.

The frequency at which the continuations are written to disk can have a substantial impact on runtime performance. Continuations are sent via the communication network to the PLinda server where they are processed. Thus, continuation committing uses two valuable resources: communication bandwidth and server processing cycles. Also, communication is slow on a network of workstations.

The current design of PLinda allows the user to tune execution of the continuation committing operations in three ways [7]:

- **Commit-consistent execution**. A process saves the continuation to tuple space at every commit as specified by the programmer. On failure, the process recovers its state from the continuation of the last commit, independently of other processes.

  This works well when continuations are small.

- **Coordinated checkpointing**. Processes do not save continuations to tuple space independently. Periodically, the runtime system forces all the processes to save continuations to tuple space and then checkpoints tuple space to disk. On any failure, tuple space is first restored

[7] Message logging and replay is no longer supported,but the PLinda interface may still reflect it.

| Commit-consistent execution | |
|---|---|
| 1 | Only uncommitted transactions |
| 2 | Continuation committing at each commit |
| 3 | Yes |
| 4 | With small continuations |

| Coordinated checkpointing | |
|---|---|
| 1 | The entire system rolls back to the last checkpoint |
| 2 | Only periodic continuation committing |
| 3 | No |
| 4 | With large continuations and a modest amount of access to tuple space |

1: Work lost on failure
2: Overhead during normal execution
3: Independent failure recovery
4: Suitable Applications

Table 1: Comparison of the Two Execution Methods

to the last checkpointed state on disk and processes recover their states from the restored tuple space. That is, a single failure leads to massive rollback.

This method is fastest in the failure-free case.

- **Message Logging and Replay**. This method is not supported in the current release of PLinda.

All three methods are guaranteed to be correct. The interested reader may see [15] for a proof.

The default in PLinda is to use the commit consistent execution method. This method is currently called *Process Private snapshot* in the user interface (this will be changed to continuation committing in the near future). To change the method, click on the "settings" button of the user interface and then click on the method of choice.

For all three methods, the frequency at which tuple space is checkpointed may be tuned. Again, click on the "settings" button of the user interface and there is an entry for checkpoint interval which is measured in seconds. Remember, the frequency at which continuations are sent to the server is orthogonal to the frequency at which the server checkpoints the tuple space (which includes continuations).

Once the setting have been changed, click on the apply button to apply the change(s). The change(s) will affect the PLinda processes which are currently executing.

Table 1 summarizes the the first two methods.

# 7 Unobtrusively using idle workstation cycles

In this section, we explain how the PLinda system may use otherwise unused idle workstation cycles for parallel computations. The issues involved in using idle workstations are: idleness detection, scheduling and process migration from busy to idle machines. In this section, we explain how these issues are addressed in PLinda and describe how the PLinda runtime system uses idle machines.

### 7.1 Issues

We discuss the issues involved in detecting whether a machine is idle or busy and how to do scheduling. Then, we describe how the current design of PLinda addresses these issues.

Idleness criterion are a set of conditions by which we can determine the idleness of a machine at runtime. Common idleness criterion are:

- *Keyboard, mouse and console idle times.*
- *Load average.*
- *The number of users logged on.*
- *Remote-login-session idle time.*

Besides these, there are other criterion such as time of day which are used relatively less often. See [17] for a comprehensive description of idleness criterion. Currently, the PLinda system uses keyboard, mouse, and console idle times, but we plan to add other criterion in the near future.

There are two approaches to scheduling on distributed systems: centralized and decentralized. The advantage of a centralized scheduler is that it is simple and easy to implement because there is only one scheduler process which maintains all necessary information. A centralized scheduler works efficiently for small and medium scale systems. For these reasons, most systems designed to utilize idle workstations employ centralized schedulers. However, a centralized scheduler generally has a scalability problem; that is, the scheduler becomes a bottleneck as the numbers of processes and processors increase. The PLinda system is currently based on a centralized scheduler.

Regarding scheduling parallel applications over networks of workstations, another important issue is supporting multiple applications. Since many users share networks of workstations, there are likely to be multiple parallel applications running concurrently. Therefore, scheduling multiple parallel applications is crucial to effectively utilizing idle workstations. Also, both throughput and turnaround time can benefit because most parallel applications do not exhibit linear speedup; that is, they are more efficient on a smaller number of processors[17]. The PLinda system is designed to support multiple applications.

Process migration is intended to move processes from overloaded or busy machines or to under-utilized or idle machines. In PLinda, process migration and process failure-resiliency are treated uniformly. More specifically, we use the fault tolerance mechanisms to migrate processes from busy to idle machines by treating owner activity as processor failure. The fault tolerance mechanisms can be used for process migration because they are light-weight and allow a process to save its state and recover from failure independently of other processes. That is, a process can be migrated efficiently and independently, using the fault tolerance mechanisms. Also, since the PLinda fault tolerance mechanisms can be designed to support heterogeneous processing, processes can be migrated over heterogeneous machines.

### 7.2 Parallel Virtual Machine in PLinda

The current PLinda prototype allows the user to use networks of workstations as one fault-tolerant parallel virtual machine whose processors are only idle workstations. However, the current implementation assumes that the user runs the server and user interface processes on machines dedicated to the parallel application. That is, the server and user interface processes are not migrated during execution.

In the prototype system, such a virtual machine is implemented as follows:

1. *Processor pool.* The parallel virtual machine consists of a pool of processors which are running or failed. Idle workstations are treated as running processors and busy or failed workstations are treated as failed processors. Only running processors participate in computation.

2. *Registering a processor.* Using the user interface, the user can register a workstation in the PLinda runtime system at any point. This causes the user interface to create a daemon on the workstation.

3. *Process creation.* If the server needs to create a client process, it selects one of the running processors which executes the smallest number of processes (i.e., a least loaded one) and spawns the client process there.

4. *Process migration.* When a running processor becomes failed (because of either owner activity or a real processor failure), its daemon or the failure immediately destroys all the running client processes on the machine. Then, the server is either informed of the simulated failure or it detects the real failure.

5. *Load balancing.* Periodically, the server performs load balancing. It migrates processes from overloaded processors to less loaded ones. Since process migration is expensive, load balancing is performed only periodically. The frequency is a tuning parameter.

# 8 PLinda Operation Semantics

In this section, we describe the five PLinda extensions to Linda, and the PLinda modifications to the six Linda functions.

Please take note of section 8.3 which describes the augmented PLinda/C++ language which must be used to tell the PLinda translator the types of all tuple fields. This must be used for any operator which takes a tuple as a parameter. For specific PLinda/Fortran77 syntactic issues, users should refer to section 10.

## 8.1 Tuple Group Creation and Destruction

- **gid** — the type of a tuple group handle.

- **create_group** — this operator creates a new tuple group (space) and returns a handle to it.

$$gid\, ts\_handle = create\_group("Some\, string");$$

- **destroy_group** — take a tuple group handle as a parameter and destroy the group.

$$destroy\_group(ts\_handle);$$

## 8.2 Linda tuple access operators

The tuple creation and retrieval functions are augmented to take a tuple group handle as a parameter, and they are all prefixed with pl_.

$$pl\_in[ts\_handle]("foo", ?A(int) : 30, ?B(int));$$

### 8.3 Augmented PLinda/C++

The programmer must specify the basic type of each tuple field. The translator determines if a field is an actual or a formal based on whether it is prefixed with a ?, and it determines if a field is an array based on whether there is a : and then an expression after the field.

The *type tag* for each field must immediately follow the field's actual or formal expression and it must be enclosed in parentheses. The allowable types are int, long, float, double, char, gid, str, and structTypeName. For constant strings (not const char*, but strings enclosed in parentheses) the type should NOT be given. Two examples of this are:

$$pl\_in[ts\_handle]("foo", x + 12 * 3(int), y(float) : 10);$$

$$pl\_out[ts\_handle]("foo", z(str), dataStuff(dataStuffStruct) : 3);$$

Where x is an int, y is an array of at least 10 floats, z is a char* array terminated by a null, and dataStuff is an array of at least three dataStuffStructs.

Please see the sample code that is distributed with PLinda for more examples.

For the details of either the PLinda/C++ translator or PLinda/Fortran77 preprocessor, study the installation instructions and available source code.

### 8.4 Transactions

- **xstart** — this starts a new transaction. The effects of `out` operations are not accessible to other processes until this transaction commits. The processes created by `proc_eval` operations are not started until this transaction commits. For example, `xstart`;
- **xcommit** — this commits a transaction. A tuple is an optional parameter that can be used to save continuation. See section 4 for more on continuations.

$$xcommit(TranNumber, localVariable1, localVariable2);$$

### 8.5 Process creation and reading command line arguments

- **pl_proc_eval** — Linda's `eval` is replaced by the `pl_proc_eval` function. `pl_proc_eval` takes an executable file name [8] and command line arguments as its parameter.

$$pl\_proc\_eval("slave.exe", 2(int), 2.3(float), "hello", a(int));$$

- **pl_arg_rdp** — this is used to read arguments passed with the `pl_proc_eval` function. It returns a boolean like the other asynchronous operators in Linda.

$$if\ (!pl\_arg\_rdp("slave.exe", ?myInt(int), ?myFloat(float), ?myString(char) : 20, ?myA(int))$$

$$cout\ <<\ "Error\ did\ not\ get\ my\ arguments..."; $$

[8] The executable file must reside in the `~/plinda/lib/$ARCHTYPE` directory.

### 8.6 Restoring state

The `xrecover` command is used to restore state and determine if a process is starting for the first time or recovering from a failure. It should take as its parameter a tuple which consists of all the local state variables as formals. It should be noted that the type of this tuple and all the `xcommit` tuples used throughout a process should be the same.

$$if\ (xrecover(?TranNumber, ?localVariable1, ?localVariable2))$$

$$cout\ <<\ "Recovering\, from\, failure\, to\, transaction"\ <<\ TranNumber;$$

## 9 Building a Simple Program in PLinda

In this section, we illustrate fault tolerant parallel programming in PLinda by a matrix multiply example. It should be noted that PLinda can be used in a non fault tolerant manner by not using transactions or `xrecover`. In this case, PLinda is exactly like Linda except for the advantage of having multiple tuple spaces and the disadvantage of a prototype implementation.

The PLinda programmer designs processes to be resilient by writing them as a series of atomic (all or nothing transactional) actions using the PLinda constructs `xstart` and `xcommit`. The PLinda system guarantees that when a failure [9] occurs, it will appear as if the failure happened between two transactions. That is, the PLinda system cleans up the intermediate actions performed on tuple space during the transaction in which failure occurred.

The programmer saves each task's continuation at the end of each transaction by placing live variables in the `xcommit` tuple. The programmer must also save an integer indicating the last transaction which was successfully executed. Using this integer and making an initial PLinda system call to ascertain if the process is recovering from failure (and if so restoring the state variables), the programmer makes each process resilient. It should be obvious that the pattern of the `xcommit` tuple must be identical throughout each process.

It is important to note that tasks cannot access the intermediate results of other task's transactions, thus avoiding cascading aborts. Hence, the programmer must be careful to commit a transaction if he expects to collect results that depend on that transaction, thereby avoiding deadlock.

Here is an annotated example of a fault tolerant parallel matrix multiply.

### 9.1 Matrix multiplication example in PLinda/C++

```
#include <plinda.h>

const int size = 100;
const int Grain = 10;
const int NumSlaves = 10;
typedef float Matrix[size][size];

// Master code for matrix multiply
int real_main(int argc, char *argv[],char **env)
```

[9] Failures include: processor power supply, operating system crashes, and communication omission/corruption errors

```
{
   Matrix A, B, C;
   int tranNumber = 0;
   gid ts_handle;

   // xrecover() to save variable state for fault tolerance, needed for recovery.
   xrecover(? tranNumber(int), ?A[0](float) : size * size ,
            ?B[0](float) : size * size,
            ?C[0](float) : size * size, ? ts_handle(gid), ? NumSlaves(int));
   if(tranNumber == 0) {
      xstart;
      transpose(B, size);
      ts_handle = create_group("mailbox");
      for(int i = 0 ; i < NumSlaves ; i++) {
// pl_proc_eval() to execute slave process on each machine.
        pl_proc_eval("matrix.slave", ts_handle(gid));
      }
      // pl_out() to out the B matrix into tuple-space for each slave to pl_rd()
      pl_out[ts_handle]( B[0](float) : size * size);
      // xcommit() to save state of all local variables for task continuation.
      xcommit(++tranNumber(int), A[0](float) : size * size ,
             B[0](float) : size * size,
             C[0](float) : size * size, ts_handle(gid), NumSlaves(int));
   }
   if(tranNumber == 1) {
      xstart;
      for(int i = 0; i < size ; i+= Grain) {
        // pl_out() Grain*Size chunks of the A matrix for a slave to pl_in()
        pl_out[ts_handle](i(int), A[i](float) : Grain * size);
      }
      xcommit(++tranNumber(int), A[0](float) : size * size ,
              B[0](float) : size * size,
              C[0](float) : size * size, ts_handle(gid), NumSlaves(int));
   }
   if(tranNumber == 2) {
      xstart;
      for(int i = 0 ; i < size ; i += Grain) {
        // pl_in() Grain*Size resultant chunks from the slaves.
        pl_in[ts_handle]("res", i(int), ? C[i](float) : Grain * size);
      }
      xcommit(++tranNumber(int), A[0](float) : size * size ,
              B[0](float) : size * size,
              C[0](float) : size * size, ts_handle(gid), NumSlaves(int));
   }
}

#include <plinda.h>
```

```
const int size = 100;
const int Grain = 10;

// Slave code for matrix multiply
int real_main(int argc,char**,char**)
{
   float A[Grain][size];
   float C[Grain][size];
   float B[size][size];

   gid ts_handle;
   int index,j,k;
   char name[100];

   // pl_arg_rdp() to accept the master´s corresponding pl_proc_eval()
   pl_arg_rdp( ? name(char) : 100, ? ts_handle(gid));
   // pl_rd() to let all slaves read the master´s Size*Size matrix.
   pl_rd[ts_handle]( ? B[0](float) : size * size);

   while(1) {
     xstart;
     // pl_in() a Grain*Size chunk of the A matrix.
     pl_in[ts_handle](? index(int), ? A[0] (float): Grain*size);

     for(j = 0 ; j < size ; j++) {
       for(int i = 0 ; i < Grain ; i ++) {
         C[i][j] = 0.0;
         for(k = 0 ; k < size ; k ++) {
           C[i][j] += A[i][k] * B[j][k];
         }
       }
     }
     // pl_out() the computed Grain*Size resultant chunk - the C matrix.
     pl_out[ts_handle]("res", index(int), C[0](float) : Grain*size);
     xcommit();
   }
}
```

**The master process:** The master process initially makes a call to `xrecover` to see if it is recovering from a failure. If so, the variables `tshandle, A, B, C`, and `tranNumber` are restored to the values they had at the last completed transaction. If not, the two matrices, `A` and `B`, are filled with random numbers.

Next, if `tranNumber == 0`, indicating that the first transaction has not yet taken place, a tuple group is created, the slaves are created, matrix `B` is put into tuple space, and matrix `A` is divided and put into tuple space. The first transaction then commits so that the slaves will start executing and so that the A and B matrices will be accessible to other processes — namely the slaves. If there is a failure during this transaction, the PLinda system will automatically clean up the intermediate results of this transaction in the tuple groups that the transaction has modified.

Next, if `tranNumber == 1`, indicating that the second transaction has not yet taken place, the master collects results from the workers. If there is a failure during this transaction, the PLinda system will put back all the result tuples that were collected prior to the failure so that the backup master will have access to them.

**The slave process:** The slave processes are stateless and therefore do not need to make a call to `xrecover` initially (although they could just to tell the user that they are recovering).

The slaves all read the B matrix. Then, they perform an infinite loop of getting GrainSize rows of A, performing the multiplication for those rows, and outputting the result to tuple space. Each iteration of the loop is done in a transaction so that if a slave fails during computation, the work tuple (the rows of A) will automatically be put back into tuple space by the PLinda system.

In the example, the slaves are left executing even after the matrix multiplication is completed. To avoid this, the master could output special work tuples signifying that the slaves should stop executing — *poison pill tuples*. One such tuple should be output for each slave after the master collects all the results. The slaves should compare the value of their work tuple to see if it is a poison pill and if so, they should commit the transaction and execute a return (if they do an exit, the PLinda system would detect this as a failure and restart the slave). In the example, a poison pill tuple could look like `("A", -1, JunkDataHere)`. The -1 in the field where the index should be indicates that this is not a normal work tuple.

## 10 Programming in PLinda/Fortran77

The PLinda system has been extended to support application development with the Fortran77 programming language. Although the PLinda kernel is written in C++, the Fortran77 interface to PLinda is transparent to the programmer.

The following is an illustration of PLinda/Fortran77 programming. The example follows the matrix multiply example shown in section 9.1

C Master code for matrix multiply

```
      subroutine freal_main()
        integer GRAIN, SIZE
        parameter (GRAIN = 2, SIZE = 10)
        REAL A(SIZE,SIZE)
        REAL B(SIZE,SIZE)
        REAL C(SIZE,SIZE)
        gid ts_handle
        integer numSlaves, tranNumber, i, j, k, ct, temp, error
        numSlaves = 5
        error =  0
        tranNumber = 0

C     Initialize the matrix entries to random numbers.
        DO i=1,SIZE
           DO j=1,SIZE
```

```
            A(i,j) = RAND(0)
            B(i,j) = RAND(0)
         END DO
      END DO

C     xrecover() to save variable state for fault tolerance, needed for recovery.
      xrecover(? tranNumber, ? ts_handle)
      IF (tranNumber .EQ. 0) THEN
         xstart()
         create_group(ts_handle, "mailbox")
         DO j=1,numSlaves
C           pl_proc_eval() to execute slave process on each machine.
            pl_proc_eval("slave_fortran_MM", ts_handle)
         END DO
         tranNumber = tranNumber + 1
         xcommit(tranNumber, ts_handle)
      END IF

C     Transpose the B matrix
      DO i=1,SIZE
         DO j=1,SIZE
            temp = B(i,j)
            B(i,j) = B(j,i)
            B(j,i) = temp
         END DO
      END DO

      IF (tranNumber .EQ. 1) THEN
         xstart()
C        pl_out() entire A matrix into tuple space for all slaves to pl_rd()
         pl_out[ts_handle](A:SIZE * SIZE)
         ct = 1
         DO WHILE (ct .LE. SIZE)
C           pl_out() B matrix in GRAIN*SIZE chunks for a slave to pl_in()
            pl_out[ts_handle](ct,B(1,ct):GRAIN * SIZE)
            ct = ct + GRAIN
         END DO
         tranNumber = tranNumber + 1
         xcommit(tranNumber, ts_handle)
      END IF

C     Collect Results from slaves
      IF (tranNumber.eq.2) THEN
         xstart()
         ct = 1
         DO WHILE (ct .LE. SIZE)
```

```
C               pl_in() GRAIN*SIZE resultants from slaves into C result matrix.
                pl_in[ts_handle]("result", ct, ?C(1,ct):GRAIN * SIZE)
                ct = ct + GRAIN
             END DO
             tranNumber = tranNumber + 1
             xcommit(tranNumber, ts_handle)
          END IF

C         Clean up tuple space
          IF (tranNumber .EQ. 3) THEN
             xstart()
C            pl_in() to remove pandemic matrix from tuple-space
             pl_in[ts_handle](?A(1,1):SIZE * SIZE)

C         pl_out() poison pill tuples to shut slaves off
             DO j=1,numSlaves
                pl_out[ts_handle](-1,B(1,1):SIZE)
             END DO
             tranNumber = tranNumber + 1
             xcommit(tranNumber, ts_handle)
          END IF
        END
```

C Slave code for matrix multiply

```
        subroutine freal_main()
          integer GRAIN, SIZE
          parameter (GRAIN = 2, SIZE = 10)
          REAL A(GRAIN, SIZE)
          REAL B(SIZE, SIZE)
          REAL C(GRAIN, SIZE)
          REAL T(SIZE, GRAIN)
          gid ts_handle
          integer tranNumber, i, j, k, x, y, index, infinite_loop, ret
          character name(100)

C         pl_arg_rdp() to accept the master´s corresponding pl_proc_eval()
          ret = pl_arg_rdp(? name:100, ? ts_handle)
          tranNumber = 0

C         pl_rd() to let all slaves read the master´s Size*Size matrix.
          pl_rd[ts_handle](? B:SIZE * SIZE)

          infinite_loop = 1
          DO WHILE (infinite_loop .EQ. 1)
             xstart()
C            pl_in() GRAIN*SIZE chunks of the whole matrix from master.
```

```
          pl_in[ts_handle](? index,? A(1,1):GRAIN * SIZE)

C       If poison pill tuple read, then commit the data and exit.
          IF(index.eq.-1) THEN
             xcommit()
             return
          END IF

C       In order to avoid complication of the matrix mult. algorithm, flip the
C       GRAIN*SIZE matrix read to avoid Fortran column major architecture.
          x = 1
          y = 1
          DO i=1,SIZE
              DO j=1,GRAIN
                  T(x,y) = A(j,i)
                  x = x + 1
              END DO
              IF (x .EQ. (SIZE + 1)) THEN
                  y = y + 1
                  x = 1
              END IF
          END DO

C       Compute GRAIN*SIZE resultant.
          DO i=1,GRAIN
             DO j=1,SIZE
                C(i,j) = 0
                DO k=1,SIZE
                   C(i,j) = C(i,j) + (T(k,i) * B(j,k))
                END DO
             END DO
          END DO

C       In order to avoid complication of the matrix mult. algorithm, flip the
C       GRAIN*SIZE resultant to avoid Fortran column major architecture.
          x = 1
          y = 1
          DO j=1,GRAIN
             DO i=1,SIZE
                  A(x,y) = C(j,i)
                  x = x + 1
             END DO
             IF (x .EQ. (GRAIN + 1)) THEN
                  y = y + 1
                  x = 1
             END IF
```

```
            END DO

C       pl_out() the resultant GRAIN*SIZE chunk back to the master.
            pl_out[ts_handle]("result", index, A:GRAIN * SIZE)
            xcommit()
          END DO
        END
```

The most salient difference in the PLinda/Fortran77 environment is the lack of the need for the user to give type hints for tuple-space arguments as required in the Augmented PLinda/C++ interface, see section 8.3. The PLinda/Fortran77 preprocessor parses the program body, extracts type definitions, and makes the appropriate PLinda kernel calls, thus eliminating the need for explicit user defined type hints.

Another minor difference in the above example is found in the `create_group` function call. Recall that in PLinda/C++, the `create_group` function returns the tuple space handle. PLinda/Fortran77 runtime system changes the tuple space handle in a call by reference manner.

$(PLINDA_HOME)/code/samples/fortran77 is the location of all sample PLinda/Fortran77 programs included with the public release of the PLinda system.

The only hindrance toward algorithm transference between PLinda/C++ and PLinda/Fortran77 programs lies in the architectural differences between the Fortran77 and C++ languages. For instance, Fortran77 has a column major order array layout in memory while C++ has a row major array layout. The layout of array memory affects the PLinda/Fortran77 programmer when passing arrays through tuple-space.

In the PLinda/Fortran77 matrix multiplication example provided with the PLinda system, a decision was made to accommodate the column major architectural difference by adjusting the array on the slave side before and after the matrix computation. Adjusting arrays to fit a notion of correctness is in fact quite simple. In this case, the alternative would be to alter and possibly obfuscate the simple matrix multiplication algorithm. As in all application programming, there may exist a tradeoff between performance and structural intelligibility[13]. The PLinda/Fortran77 implementation allows the programmer to make that decision.

# 11 Compiling a PLinda Program

Compiling a PLinda program written in C++ or Fortran77 follows similar guidelines. This section outlines how to successfully compile a PLinda program into an executable. An important note to remember is that the PLinda system is set up for individual executables for each different platform. Each platform, below referred to as variable `ARCHTYPE`, contains its own executables pertinent to the user's development work.

## 11.1 Compiling a PLinda/C++ Program

The PLinda system should be installed in the directory `~/plinda`. Throughout this section we refer to this directory as PLINDA_HOME. That is, `plinda-2.x`, `lib`, and `include` are all in the directory PLINDA_HOME.

PLinda source files need to include the PLinda client header file. This file defines the PLinda interface just as the PVM header files define the PVM message passing interface[14]. The source files should include the file `plinda.h`. Make sure the include directory directive is used when compiling, `-I$(PLINDA_HOME)/include`.

The suffix of a PLinda/C++ source file should be `.plc`. Compilation of a PLinda program `foo.plc` consists of three steps:

1. Use the PLinda translator, `plc`, to generate a preprocessed file `foo.C` by issuing the following command:
   `plc foo.plc`.

2. Use `g++ -c` to compile `foo.C`.
   The PLinda include files path must be given to the compiler.
   `g++ -c -I$(PLINDA_HOME)/include foo.C`

3. Link the object(s) with the PLinda library to create an executable.
   `$PLINDA_HOME/lib/$ARCHTYPE/libplinda.a` is the location of the PLinda library. Use the library directory directive to specify the library's path and the library directive to specify the library. For example:

   ```
   g++ -L$(PLINDA_HOME)/$(ARCHTYPE)/lib -lplinda foo.o
   -o $PLINDA_HOME/lib/$ARCHTYPE/foo.exe
   ```

   PLinda application executables must be placed in the directory `$PLINDA_HOME/lib/$ARCHTYPE` so that the PLinda runtime kernel can find them. `$ARCHTYPE` is the type of machine on which you compile the program (e.g., `$ARCHTYPE=sunos` when a SunOS machine is used and `$ARCHTYPE =solaris` when a Solaris machine is used).

**Note:** You must compile source code for each of the types of architectures you plan to use in your virtual machine host pool. The runtime system assumes you have compiled code for all those types of machines.

Please look at the sample Makefiles in the subdirectories of $(PLINDA_HOME)/code/samples.

### 11.2 Compiling a PLinda/Fortran77 Program

Compilation of a PLinda program `foo.plf` consists of three steps. The suffix of a PLinda/Fortran77 source file is constrained to be `.plf`.

1. Use the PLinda/Fortran77 preprocessor `plf`, found in `$PLINDA_HOME/bin/$ARCHTYPE`, to generate the preprocessed target file `foo.f` by issuing the following command:
   `plf foo.plf`.

2. Use `f77` to compile `foo.f` into object file `foo.o`:
   `f77 -c foo.f`.

3. Generate the Plinda/Fortran77 executable by linking in the PLinda system library and the PLinda/Fortran77 library and including the PLinda header files:

   ```
   g++ foo.o -L$(PLINDA_HOME)/$(ARCHTYPE)/lib -lplinda -lfplinda $(FLDFLAGS)
   -o $(PLINDA_HOME)/$(ARCHTYPE)/lib/foo.exe
   ```

   FLDFLAGS should contain the Fortran77 system libraries. This is very system and compiler dependent. For example, this could be `-L/usr/lib -lF77 -lm`.

The example makefiles for PLinda/Fortran77 given in the sample programs included with the PLinda public release should serve as a good basis for understanding the Fortran77 libraries.

# 12 Running Plinda

## 12.1 How to start the PLinda user interface

The PLinda user interface can be started in the `~/plinda` directory by typing `pl`. The user must supply the file `~/.plinda.hosts` with the names of workstations and their architecture types for each machine he will use. The user should also have a `~/.plindarc` file that has the default settings of the system. Samples of `.plinda.hosts` and `.plindarc` are included in the distribution. Again, all executables including ones that will be spawned from PLinda code should be located in the directory $PLINDA_HOME/lib/$ARCHTYPE. The PLinda kernel will create some files and subdirectories in the directory where it is run from. Therefore, it is important that the system be run from the directory `~/plinda`.

The PLinda kernel will use the Unix `rsh` command to run processes on remote machines. Therefore, it is necessary to be able to access those machines from within an executable. To facilitate this the user needs to have an `.rhosts` file on each remote machine listed in the `.plinda.hosts` file. The file must define all machines and alternate login names from which the PLinda user interface will be run.

## 12.2 How to select a PLinda program to run

1. Start the PLinda user interface — type `pl`.

2. Starting the server: Clicking on the "System" button on the menu bar will bring up a dialog box for you to specify the name and type of the machine on which you want to start a server. The default values filled in are read from `plinda.defaults`. The dialog box also allows you to change the working directory and display host, but normally you do not want to change them. Check the radio button "Boot" and then click the "Apply" button to start the server.

3. Add hosts to the virtual machine: Click on the "Hosts" button on the menu bar and a window with host information will pop up. It lists all the hosts in the `.plinda.hosts` file. Check the radio button beside a host name to select a host and click the "Add" button to add it to the PLinda host pool. You can select multiple hosts and "Add" them together. To delete a host from the PLinda host pool, select the host and click on the "Delete" button.

4. Select a PLinda program to run: Do so by clicking on the "Apps" button on the menu bar. This brings up regular file selection interface that starts with the directory `~/plinda/lib`. You should select either "sunos" or "solaris", whichever directory has the executables. Then select an application to run.

**Note:** You must compile source code for each of the types of architectures you plan to use in your virtual machine host pool. The runtime system assumes you have compiled code for all those types of machines.

### 12.3 How to observe and control Execution Behaviors

When you start an application, two additional windows should pop up. The first one is the "Process Watch" window (you can also get it by clicking on the "Monitor" button on the menu bar). This window shows you all running PLinda processes with process id (pid), the name of the host the process is on, etc. The "Process Watch" window is refreshed every `Update_Interval` seconds, where `Update_Interval` can be defined either in `plinda.defaults` or by changing the value in the dialog box that pops up when you click the "Settings" button on the menu bar. The second window is an xterm used as an I/O shell for the PLinda master.

One important piece of information about a process is its status. When a PLinda process is spawned, its status is "DISPATCHED". When a PLinda process is waiting for a tuple, its status is "BLOCKED"; most of the time its status is "RUNNING". When a process goes from "RUNNING" to "BLOCKED", there is a middle stage called "READY". But "READY" holds for a very short period of time and will be rarely seen in the "Process Watch" window. When the PLinda server respawns a failed process, the status of the process is "FAILURE_HANDLED".

The user can "Kill", "Migrate", "Suspend", or "Resume" processes by clicking on appropriate buttons in the "Process Watch" window. These functions are mostly used to test the fault-tolerance of a PLinda program

## 13 How to get and install PLinda

To receive PLinda (source and/or binaries), send email to `plinda@cs.nyu.edu`.
For installation instructions, see the PLinda web page at:
`http://merv.cs.nyu.edu:8001/~binli/plinda/`

## 14 Suite of Sample Programs

Included with the PLinda package is a suite of sample programs. The sample programs included solve problems from physics and mathematics. PLinda/C++ samples include ping-pong, matrix multiply, particle simulation, and mandelbrot. The PLinda/Fortran77 sample is matrix multiply.

Particle simulation is a parallelization of the Greengard sequential algorithm[12] which models a square computational box containing N point charges, with given charge values, initial positions, and velocities to compute the state of the system (i.e. position and velocities of all particles) after a given time interval. Mandelbrot set[11] computes operations in a complex plane and displays the results graphically.

All samples may be found in $PLINDA_HOME/code/samples.

## 15 Acknowledgments

The following people were instrumental in creating, maintaining, and enhancing the PLinda system:

Brian Anderson, Hansoo Kim, Jihai Qiu, and Zasha Weinberg.

Special thanks to Edmond Schonberg for guidance on PLinda/Fortran77 compilation techniques.

## References


[1] Yeshayahu Artsy and Raphael Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, pages 47–56, September 1989.

[2] Robert Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Department of Computer Science, Yale University, 1992.

[3] Clemens H. Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19:1221–1234, 1993.

[4] N. Carriero and D. Gelernter. Linda in context. *Communication of the ACM*, 32(4):444–458, April 1989.

[5] Nicholas Carriero and David Gelernter. *How to write parallel programs : a first course*. MIT Press Cambridge, 1992.

[6] David Cheriton. The V distributed system. *Communication of the ACM*, pages 314–333, March 1988.

[7] P. Dasgupta, R.J. LeBlanc, M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*, pages 34–44, November 1991.

[8] J. Dongarra, G. A. Geist, R. Mancheck, and V. S. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–175, 1993.

[9] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software–Practice and Experience*, 21(8):757–785, August 1991.

[10] High Performance Fortran Forum. High performance fortran language specification version 1.0. Crpc-tr92225, Center for Research on Parallel Computation, Rice University, 1993.

[11] Vincent W. Freeh. A comparison of implicit and explicit parallel programming. Technical Report TR-93-30, Dept of Computer Science, University of Arizona, December 1993.

[12] Leslie Greengard. *Rapid Evaluation of Potential Fields in Particle Systems*. PhD thesis, M.I.T. Press, 1987.

[13] W. G. Griswold and D. Notkin. Architectural tradeoffs for a meaning-preserving program restructuring tool. *IEEE Transactions of Software Engineering*, 21(4):275–287, April 1995.

[14] Jack Dongarra and others. *A Users' Guide to PVM Parallel Virtual Machine*. Oak Ridge National Laboratory, July 1991.

[15] Karpjoo Jeong. *Fault-Tolerant Parallel Processing Combining Linda, Checkpointing, and Transactions*. PhD thesis, Department of Computer Science, New York University, January 1996.

[16] Karpjoo Jeong and Dennis Shasha. Persistent linda 2: A transactional/checkpointing approach to fault-tolerant linda. In *Proceedings of the 13th Symposium on Fault-Tolerant Distributed Systems*. IEEE, October 1994.

[17] D. Kaminsky. *Adaptive Parallelism with Piranha.* PhD thesis, Department of Computer Science, Yale University, 1994.

[18] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[19] C.A. Waldspurger, T. Hogg, B.A. Huberman, J.O. Kephart, and W.S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

[20] Jingwen Wang, Songnian Zhou, Khalid Ahmed, and Weihong Long. LSBATCH: A distributed load sharing batch system. Technical Report CSRI-286, University of Toronto, April 1993.

[21] Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. Technical Report CSRI-257, University of Toronto, April 1992.