Gardens: High Performance Objects, Tasking and Migration for Cluster Computing

Paul Roe and Clemens Szyperski

School of Computing Science
Queensland University of Technology
Brisbane, Queensland, 4001
tel: 07 3864 {1942, 5222}, fax: 07 3864 1801
{*p.roe,c.szyperski*} *@qut.edu.au*

Abstract

Gardens is an integrated programming language and system which supports efficient parallel computation across workstation clusters. In particular it addresses the three goals of: high performance, adaptive parallelism and abstraction. High performance is the goal of parallel computing, and abstraction simplifies programming. Adaptive parallelism entails a program adapting during its execution to utilise a changing set of otherwise idle workstations. Tasks are used as units of work, and task migration to realise adaptive parallelism. Tasking is non-preemptive; compared to preemptive tasking this leads to simpler programming and greater efficiency. Global objects are used for inter-task communication. These support abstraction and importantly directly map to active messages: a very efficient messaging system. These features of Gardens are tightly integrated yet orthogonal.

1 Introduction

Gardens is an integrated programming language and system which supports efficient parallel computing across networks of workstations (cluster computing). Idle workstations represent a considerable computational resource – as yet untapped. There have been many proposals for utilising idle workstations; Gardens is distinguished in that it concentrates on supporting high performance adaptive parallel computation. Adaptive parallelism refers to parallel computation which must adapt to utilise a changing set of available workstations.

Gardens is predicated upon clusters of high performance workstations connected via modern high performance networks such as ATM and Myrinet. Such systems have the potential to provide supercomputer levels of performance.

The goals of Gardens are:

- high performance
- support for adaptive parallelism

- support for abstraction

To achieve these goals the core of Gardens consists of a tightly integrated system which supports: *efficient communications*, *light weight tasks* (including *fast task migration*), and *global objects* which support abstraction. These facets of Gardens are orthogonal; for example it is possible to use global objects without task migration, if desired.

Note that strictly respecting abstraction, enforced by a suitable language and compiler, makes the system safe. This in turn allows for cheap and straightforward mechanisms to protect the security of the workstation's owner.

Gardens was designed with a minimalist philosophy, inspired by the Oberon programming language and system. It provides the basic building blocks required for safe and extensible adaptive parallel computation. Furthermore our programming language is based on Oberon.

The remainder of this paper is organised as follows, the next two sections describe the communications and tasking model of Gardens. Section 4 summarises the model and its invariants. Section 5 presents some preliminary performance figures. The final two sections discuss related and future work.

2 Communication

To achieve high performance communications (crucial for parallel computing) a very lightweight and efficient messaging software layer is required. Active Messages (AM) is one such system; upon which our global objects are built. The whole of Gardens has been heavily influenced by AM; however similar systems such as, Fast Messages, UNET etc. could equally well be utilised as a base.

Our global objects, supported by our programming language, raise the level of abstraction of AM but without incurring any significant overhead. These simplify programming by supporting the construction and use of communications abstractions (libraries). Our programming language uses type annotations to enforce safety and improve programs' readability.

2.1 Active Messages

Active Messages represents the state-of-the-art for fast messaging on parallel computers [14]. In essence AM is a form of lightweight asynchronous remote procedure call. Request operations may be issued which asynchronously send a message to a remote processor, consisting of a handler (function pointer) and some data. On receipt messages are queued until a poll operation is performed. "Poll" processes messages by invoking handlers on associated data; this gives the recipient control over when handlers are invoked, ie control over mutual exclusion. Note "poll" may process any message; it is not possible to filter particular messages. "Poll" does guarantee that messages are processed atomically, and in order, if from the same sender. A handler may invoke a reply operation (similar to a request) to return a message to the original caller; however reply handlers may perform no communication. Thus communication operations cannot be arbitrarily nested via handlers.

If the destination of a request is the local processor the addressed handler will be called immediately. Thus all request operations also perform a poll on completion; this ensures the semantics of local requests is consistent with that of non-local requests.

A credit counting scheme is used to control network deadlock (as opposed to application deadlock). Credits are lost by sending messages to a host, and gained by receiving reply/acknowledgement messages from hosts. Therefore, no protocol is needed to handle buffer overflow at the receiving end; in general, AM's performance is largely a result of trimming back traditional network and transport protocol overheads. If a request is issued when the credit count is zero the request operation will poll until credit is available; after which the request will be performed. Thus, a request operation may cause a poll before and after its operation.

2.2 Global Objects

Active messages is rather difficult to use since all communications occur through shared memory. Thus typically programs use global variables for communication, which do not support abstraction – one of our goals. Furthermore there are restrictions on message handlers, which cannot be enforced by a traditional language such as C. Finally AM does not support the addressing of mobile tasks (see Section 3.3). The key goal of our global objects is to support abstraction of communications, and to do so without sacrificing the performance AM provides.

Tasks may perform point to point communication with other tasks via global objects (GOs). In general a task will manage several GOs, which act as communication ports for that task. Global objects support asynchronous remote dynamic dispatch; that is a task may invoke a method on an object which is located on a different processor. This is implemented by AM's request operation[1]. GOs are ordinary objects created within the heap of a particular task, and are managed by that task. The only difference between ordinary local objects and GOs is that GOs are globally contactable. The task owning a GO may access it as a normal object (a record in Oberon), and it is indistinguishable from any other heap allocated record. A GO reference, accessible by other tasks, is created by the `@` operator. Any object can be made globally contactable by handing out a global reference to it. GO references cannot be dereferenced, and they only support a subset of the original type's methods, in particular those methods labelled `@`. GO methods (those labelled `@`) are required to have a restricted interface. In particular normal pointer types (including ones nested inside an argument), var parameters, and return values are disallowed. These restrictions prevent local (ie non-GO) references escaping from tasks, including implicit ones created by var parameters.

A simple example is shown below:

```
TYPE Object = POINTER TO RECORD count: INTEGER; sum: REAL END;

VAR o: Object;
    go: @Object;

PROCEDURE (self: Object) @Add (s: REAL);
```

[1] As an aside: requests are limited to small packets. For larger transfers, modified AM store operations are used instead, the ramifications of which are beyond the scope of this paper.

```
BEGIN
  self.sum := self.sum + s;
  DEC(self.count)
END Add;

PROCEDURE Foo(go: @Object);
VAR localsum: REAL;
BEGIN
  ...                  (* expensive calculation of local sum *)
  go.Add(localsum)     (* global method invocation *)
END Foo;

BEGIN
  NEW(o);
  o.sum := 0;
  o.count := NTasks;
  go := @o;
  ...                          (* create NTask tasks performing Foo(go) *)
  WHILE o.count#0 DO Poll END; (* wait for results *)
  ...
END
```

The poll operation is an explicit transfer point. It enables global method invocations to occur. Therefore code which cannot perform a poll explicitly or implicitly is *atomic*. Remote method invocations may implicitly cause one or more polls before or after invocation. Furthermore GO method invocations may not be nested. To control this, procedures, including methods, may be labelled `ATOMIC` if they contain no calls to non-atomic code. This is statically checked by the compiler. Shared object methods must be `ATOMIC`, and this is implied, by labelling a procedure `@`. However the *invocation* of a global object method is non-atomic, as is a poll. Thus a GO method may not invoke other GO methods or call poll.

These `ATOMIC` labels are also useful for the programmer. In particular, if a library routine is labelled `ATOMIC` the programmer knows that no poll operations (implicit or explicit) will be performed by the library. Thus objects which have global references need not be in a consistent state when the library is called. However if a procedure is not labelled `ATOMIC`, poll operations may occur, and hence any globally referenced objects must be prepared for global method invocations.

The type system also enforces that `ATOMIC` methods may only be overridden with `ATOMIC` methods and likewise that `@` methods may only be overridden with `@` methods.

Note, unlike other parallel object-based systems, there is no possibility of deadlock as the result of recursive or cyclic invocations, since nested invocations are statically prohibited. (The restrictions required by AM are automatically and naturally met.) Finally, the owning task may invoke a GO method on one of its own objects with no detrimental effects.

As described here, GOs can only be used indirectly for performing a read: a read request must be posted to a GO and that GO's managing task must then reply by performing a request with the read value on another GO. A more direct method is clearly desirable and currently under investigation.

3 Tasks

Tasks are our unit of work, they are used for load balancing. That is for efficiently mapping the potentially irregular and dynamic parallel computation across the changing set of available workstations. This is accomplished by over-decomposing a problem into more tasks than there are workstations. To adapt to a changing set of workstations, tasks may be migrated. Task migration is primarily targeted at supporting the release and acquisition of workstations; however it may also be used for some coarse-grained load balancing of tasks across a stable processor set. The key goal of our tasking system is to support adaptive parallelism, and to do so efficiently without sacrificing the performance provided by the underlying workstation.

The programmers model of computation is a network of tasks; these are mapped onto the network of workstations. Tasks are created dynamically; they communicate with other tasks, using GOs, independent of which processor they occupy. Note that there is no concept of task identifiers, since such values tend to break abstraction. Instead we rely on GOs to support communication between tasks. Thus tasks are rather intangible entities which cannot be directly referenced.

3.1 Multitasking

How can multitasking and GO's coexist? Answer: by only allowing a context switch to occur while a task is at a poll point and by ensuring new tasks must always be the first to initiate communications between themselves and other tasks.

The Gardens system supports lightweight non-preemptive tasks which inhabit a relatively heavyweight OS process (eg Unix or NT process). Each task has its own stack and heap. Non-preemption simplifies programming, is compatible with our AM/GO programming model and is more efficient than preemptive multitasking. Furthermore no protection or isolation is enforced at run-time; this is enforced statically by our programming language. Thus when a task executes a purely sequential (atomic) portion of code performing no communication or blocking (see next section), that code will be run at maximum speed, incurring no overheads due to parallel execution.

Tasks are created using `FORK`, a procedure similar to the Unix fork operation:

```
PROCEDURE Fork (p: PROCEDURE (o: @ ANYPTR); o: @ ANYPTR);
```

For example the `NTasks` of the previous GO example may be created thus:

```
FOR i:= 1 TO NTasks DO FORK(Foo,go) END;
```

The forked procedure runs asymmetrically from the parent task. A global object is passed to the child task; this enables the child task to communicate with another task, and thereby to inform other tasks of its own global objects. Notice that no task may communicate with the child task until the child task has initiated some communication. Note this does not prevent client server computation from being expressed. This also means that the fork operation is atomic, ie causes no communication or context switching. It also allows simple migration of new tasks which have not yet been activated, see Section 3.3.

3.2 Blocking

Efficient tasking requires support for blocking: even with asynchronous communication, tasks will eventually need to synchronise. Blocking eliminates unnecessary context switches to tasks waiting to synchronise; it is also useful for identifying idle time which is a useful load balancing measurement. The `Block` and `Unblock` routines implement blocking; they take no arguments. Block blocks the currently running task by removing it from the runnable queue and putting it in the blocked pool. Semantically, block is equivalent to a poll and context switch. Unblock, usually performed by a GO's method, unblocks the relevant task in the obvious way. Thus blocking blocks a task but does not prevent global method invocations on that tasks GOs: which may unblock it. (Where no unblocked tasks remain, the kernel continues polling for incoming messages.) Block/unblock operations have direct effect regardless of their nesting level: unlike semaphores they are not counted. This is sufficient for a non-preemptive setting.

For example we may add blocking to our previous example thus:

```
PROCEDURE (self: Object) @Add (s: REAL);
BEGIN
  self.sum := self.sum + s;
  DEC(self.count);
  IF self.count=0 THEN Unblock END (* if last result unblock owning task *)
END Add;

...

BEGIN
  NEW(o);
  o.sum := 0;
  o.count := NTasks;
  go := @o;
  FOR i:= 1 TO NTasks DO FORK(Foo,go) END;
  Block;     (* wait for all results *)
  ...
END
```

Note, in general we expect such detailed coding to be encapsulated in abstractions. Our blocking and unblocking is general and efficient. For example, there is no need for a kernel task to repeatedly test a flag to see whether a task should be unblocked.

3.3 Task migration

To support the acquisition and release of workstations, work must be dynamically reallocated to workstations. Tasks are our unit of work hence reallocation implies migration of tasks. Currently task migration is only supported across homogeneous platforms. The tasking implementation partitions the virtual memory address space of a process across processors. Tasks occupy disjoint regions of virtual memory. Thus task migration can be achieved by copying a tasks heap and stack from one processor to the same regions on another processor. Task migration is initiated by a rudimentary load balancing system (not described here).

Migration is non-preemptive, and can only occur when all tasks are at poll points. Thus general migration requires global synchronisation. This enables all messages in transit to be flushed before migration, hence no message forwarding mechanism is needed. It also means that our model invariants are preserved, see next section.

New tasks which have never been run, support a much more efficient form of migration. Until a task is run no other task can communicate with it and all the state that needs to be kept in such a *task seed* are the arguments to `FORK` (two addresses). Task seeds can be migrated without requiring global synchronisation. In fact one AM request/reply communication can be used for migration of task seeds. Such tasks do not even need a heap or stack until they are first run.

Thus our system understands three types of tasks: runnable tasks, blocked tasks and task seeds.

4 Summary: the Model and its Invariants

Our model of tasking, task migration and communications has the following invariants:

- Tasks can be in one of three states: seed, runnable, or blocked. (A task is blocked either explicitly or implicitly when out of credits.)
- On any one host, at most one runnable task is not at a poll point; all blocked tasks are at poll points.
- Message handlers are called only if all local tasks (excluding seed tasks) are at poll points.
- Migration can only occur if all tasks (excluding seed tasks) on all processors are at poll points, and there are no outstanding messages.
- No "network deadlock" can occur, as is the case with AM.
- Messages are causally ordered, even across migration.
- Task seeds have performed no communication and cannot be communicated with.
- Tasks occupy disjoint regions of virtual memory.

With AM's credit scheme, we replace waiting for credits by setting a task to an "out of credits" state known to the scheduler, causing that task to block.

The invariant of at most one active task per processor ensures that on every call to the communication layer an incoming message could be delivered to the destination task by calling its handler without introducing preemptive semantics for the receiving task.

In principle, preemptive task migration is desirable to swiftly release workstations. However, to maintain our invariant of at most one active task per processor at a time, we can only migrate tasks at a poll point. Furthermore since we do not perform message forwarding, we need to flush all messages before migration: thus all tasks must logically be at a poll point. Since message handlers are atomic, ie, cannot perform remote invocations themselves, message flushing is guaranteed to terminate.

System	Communication	Multitasking	Task mig/adaptation
Split-C	poor abstraction, efficient	no	no
CHARM(++)	many kinds, abstract, efficient	yes	mig: only new tasks
Orca	objects, abstract, inefficient	yes	no
Piranha	Linda, abstract, inefficient	no	prog. adaptation
UPVM	PVM, poor abs, inefficient	yes	task mig, efficient
Cilk(v2)	functional, abstract, efficient	yes	migration, efficient
Gardens	GO, abstract, efficient	yes	task mig, efficient

Table 1: Comparison of Approaches

5 Performance Results

Our current implementation is a simple prototype; we are currently working on optimising it's performance. The test platform consists of four 32Mb 120MHz Sun SparcStation-4s connected via a Myrinet.

The first set of measurements show the overhead of using global objects versus raw active messages.

data size (bytes)	AM (μs)	GO (μs)
16 (short comm)	20	23
1024 (long comm)	156	167

The following figures demonstrate the efficiency of our current tasking system:

fork (seed task creation)	$10\mu s$
fork (eager task creation)	$77\mu s$
block task and context switch	$16\mu s$
unblock task	$5\mu s$

The time taken for task migration depends on the task size and the time required to synchronise processors, which is application dependent. The figures below give times for synchronising all processors and migrating a single task:

Task size (Kb)	4	8	16	32	64	100
Migration time (ms)	1.0	2.7	4.4	7.7	14.5	22.1

The increase in time for a 4K versus 8K task is due to packetisation: the maximum packet size in AM is 4K.

The time to create (via fork) and migrate a task seed, which consequently has no stack or heap data and requires no global synchronisation, is just $41\mu s$!

6 Related Work

There is a quite diverse set of work related to our approach. Table 1 lists key aspects of some of these approaches, comparing them to ours.

Split-C [8] is a parallel extension of ANSI C. It adds a relatively small number of new mechanisms to C, and does not enforce a particular style of parallel programming. It is built on active messages and is very efficient. However, like C, it is not safe and does not easily support abstraction. This is the language used by the NOW project [2]. Split-C assumes a static set of processors and has no associated multitasking, or support for task or work migration. Recent work has looked at very efficient fine grained parallelism/tasking [9].

CHARM is a language and system [12, 10]. The key concept is message-driven execution where context switching is used to hide communication latencies. A *chare* is a small task; contexts are switched on every message send. It has been found that context switching to hide latencies is much faster than blocking sends [12]. However, software pipelining based on asynchronous sends is even faster. CHARM supports several information sharing abstractions, eg, accumulators, lookup tables, read only variables, write once variables, and monotonic variables. CHARM uses sophisticated load balancing strategies and works well for irregular problems. Task migration is only possible for their equivalent of task seeds; once a task is running it cannot be migrated. A particularly interesting concept is that of *branch office chares*. These are replicated exactly ones per processor and allow for programming of processor-local functionality, which is being considered but is not currently possible in Gardens.

Orca is a language for parallel computing on distributed systems [4]. Orca separates processes that are local to individual processors and shared data-objects [13] that are accessible from all processes that have a reference to such an object. While processes are active, shared objects are passive. The actual allocation of shared objects, including possible replication, is hidden from the programmer. All operations on shared objects are user-provided and all are atomic (indivisible). Compared with Gardens GO's Orca's objects are rather more abstract, but less efficient. They rely on compiler and runtime optimisation[3]. In particular, the system decides which objects to replicate and where to place non-replicated objects. Redistribution is not supported but can be handled manually by creating new sets of processes for new computational phases.

Piranha is one of the very few existing approaches to explicitly address some of the issues of adaptive parallelism [6]. Piranha uses the tuple space abstraction introduced by Linda and adds a special computational model plus a few run-time hooks to support adaptive parallelism. A feeder process formulates subproblems and codes them into tuples placed into the tuple space. Worker processes started on available processors grab subproblems by picking up tuples whenever they are idle. Unlike Gardens, adaptation in Piranha must be explicitly programmed via casting programs into a master worker style and coding "retreat" functions to be invoked when an idle workstation is reclaimed.

UPVM [11] is a system designed to support lightweight tasks and their migration. Like Gardens, UPVM relies on over decomposing a program into tasks. However the system aims at supporting the PVM message passing library, and does not support task blocking. PVM does not support abstraction, unlike our global objects, and can be rather inefficient. A more recent project Mist [7] considers migration of full OS processes, rather than lightweight tasks; however this is very expensive compared with lightweight task migration. Unlike Gardens Mist addresses fault tolerance.

Cilk(v2) [5] is an interesting project which supports efficient adaptive and fault tolerant parallel computation. However Cilk programs must be cast into a very restricted functional model of parallelism.

A number of other attempts have been documented to integrate general process migration facilities into operating systems [1]. All these systems aim at distributed computing with much coarser granularities than we do in supporting high performance parallel computing.

7 Future Work

Currently work is underway to optimise our prototype implementation. The following points are the subject of ongoing work:

- Efficient reading, at present reading a GO is rather clumsy: a more direct approach is being worked on.
- Collective communications are also rather inefficient. A system similar to CHARM's branch offices is under consideration. Some initial ideas have already been implemented.
- It is desirable to control the number of tasks in the system: prevention of task generation can be simply programmed, deleting task seeds and task coalescing are more complex ideas.
- We would like to relax the synchronisation requirements for task migration.
- Our load balancing system is rather rudimentary; we wish to improve this, and in particular to preserve locality under load balancing.
- Our GOs require distributed garbage collection which is not currently implemented.
- We are investigating heterogeneous task migration (funded by an ARC grant).
- I/O is difficult under task migration; parallel I/O is even harder to achieve!

Note, fault tolerance is not in this list; this is orthogonal to the Garden's goals.

Acknowledgements

Our thanks for their efforts in discussing and implementing the model described in this paper go to: Ashley Beitz, Siu-Yuen Chan, Luke Kristenson, Nickolas Kwiatkowski, Chuen-Yuen Suen, and all other members of the Gardens Project group[2].

References

[1] A Barak A, A Braverman, I Gilderman, and O La'adan. The MOSIX multicomputer operating system for scalable now and its dynamic resource sharing algorithms. Technical Report 96-11, Institute of Computer Science, The Hebrew University, July 1996.

[2] Th E Anderson, D E Culler, and D A Patterson *et al.* A case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[2] http://www.fit.qut.edu.au/~szypersk/Gardens

[3] H E Bal and M F Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In *Proc., 8th Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1993.

[4] H E Bal, M F Kaashoek, and A S Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Software Engineering*, 18(3):190–205, March 1992.

[5] R D Blumofe and P A Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems, Anaheim, California*, 1997.

[6] N Carriero, E Freeman, D Gelernter, and D Kraminsky. Adaptive parallelism and piranha. *IEEE Computer*, pages pp40–49, January 1995.

[7] J Casas *et al.* MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2):171–216, Spring 1995.

[8] D E Culler et al. Parallel programming in Split-C. In *Proc., Supercomputing '93 Conf.*, November 1993.

[9] S C Goldstein, D E Culler, and K E Schauser. Enabling primitives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, Rochester, NY*, 1995.

[10] L V Kale and S Krishnan. CHARM++ : A portable concurrent object oriented system based on C++. In *Proc, Conf on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, 1993.

[11] R Konuru, J Casas, S Otto, R Prouty, and J Walpole. A user-level process package for pvm. In *Proceedings of the Scalable High Performance Computing Conference*, pages 48–55, May 1994.

[12] W W Shu and L V Kale. Chare kernel - a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11:198–211, 1990.

[13] A S Tanenbaum, M F Kaashoek, and H E Bal. Parallel programming using shared objects and broadcasting. *IEEE Computer*, 25(8):10–19, August 1992.

[14] T von Eicken, D Culler, S C Goldstein, and K E Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings 19th International Symposium on Computer Architecture*, May 1992.