Parallel Programming with the Pool Model

S Maclean, H Glaser and D DeRoure

Abstract

A parallel programming model called the Pool Model is described. The Pool Model supports a parallel programming paradigm which is independent of shared memory threaded programs and of message-based network programming. The pool is merely an environment for constructing concurrent computations. The runtime system of the environment is sketched; highlighted are task location constraints and task racing.

Introduction

A crucial aspect of any programming system is the paradigm or "model" of programming exposed to the programmer. The choice of programming model has an effect on many aspects of both program development and program execution: maintainability, readability, extensibility, correctness, space and time complexity. In the world of sequential programming, familiar models are imperative programming (encompassing procedural and object-oriented), functional programming and logic programming.

When multiple processing units are available, we would like to write programs that take advantage of this increase in processing power. The desired result is thus a program which executes faster, but that has unaltered semantics over the sequential version. As in the case of sequential programming, we require a model of programming for this parallel programming style. The focus of this paper is one such parallel programming model, the Pool Model.

Note that we view issues of shared-memory parallel processing and distributed memory concurrent processing as implementation issues rather than model issues. A discussion on parallel programming models thus embodies all forms of subsequent concurrent execution.

Background

The work described here forms part of the GraphIcsla* project, investigating the use of graphical programming and distributed threads of execution.

The target programming environment of GraphIcsla is the Prograph CPX programming system [5]. The current CPX system supports sequential computation only. Further, at present language enhancements to the integrated programming environment are not possible. The Pool Model described in this paper is thus exported to the user on top of the CPX system via a simple API — the current API consists of two operations: `pool it` and `get pooled value`. Figure 1 shows both a sequential and "pooling" version of the quicksort algorithm for sorting a list, but not in situ (see below).

For the sequential version (the left half of the figure), a list object flows from the input bar (first horizontal bar) through detach and partitioning operations. Pre-partition "match" operations check for empty and singletons lists — a successful match causes the algorithm to halt and return the input list. After the partitioning, the two sublists are fed to the quicksort algorithm recursively. Finally, the now sorted lists are joined, integrating the pivot element at the correct position.

The pool annotated version (the right half of the figure) takes advantage of the fact that the two quicksorts may be performed in parallel, since they share no data (they work on distinct sublists generated by the partition operation). The left most `quicksort` operation is replaced by the `pool it` operation, with inputs being the name of the original operation and its arguments. The result of the pooled quicksort is retrieved using the `get pooled value` primitive. By pooling one of the quicksort operations we have the chance to compute the other (right most call) locally, letting the pooling system locate a separate processor which will compute the pooled computation. The desired result is thus a possible decrease in the time taken to reach the `(join)` operation.

The suitability of the graphical approach to parallel programming is highlighted by this simple example; the visual nature of the algorithm implementation gives some leverage for possible concurrent execution. The pool is a mechanism for realising this leverage.

The Pool Programming Model

Parallel execution of programs involves a model of parallel programming. Well known models include message passing (e.g. MPI[2], PVM[4], Nexus[3]), bulk synchronous parallel programming[6], futures (e.g. in Lisp[7]) and the tuple space concept of Linda[1].

The programming model developed for this project is the Pool Model. The Pool Model can be seen as a "constrained Linda". In the Pool Model, as in Linda, computations may be pooled (Linda uses `out`). Computations may also be consumed for execution (implicit to the runtime system in the Pool Model, Linda uses `in`). Computation results are also implicitly returned to the pool (Linda would use a second explicit `out`). The results are located by the main program using `get pooled value` (Linda would use a second `in`). The process (thread) which pools any computation is always the *same* thread as that which performs the result-obtaining `get pooled value`. Further, due to the graphical nature of the target implementation, the `out` and `in` operations are constrained visually — a `pool it`/`get pooled value` pair are never separated by an output bar (final horizontal bar in the diagram). This visual constraint results in the equivalent of a short-lived tuple or future.

For the Pool Model, the primary computation is hosted on the "console" process or machine. The pool (a data structure supporting concurrent access) is situated within this console.

To achieve any form of true parallel processing requires

*`http://www.dsse.ecs.soton.ac.uk/dsse/graphicsla.html`

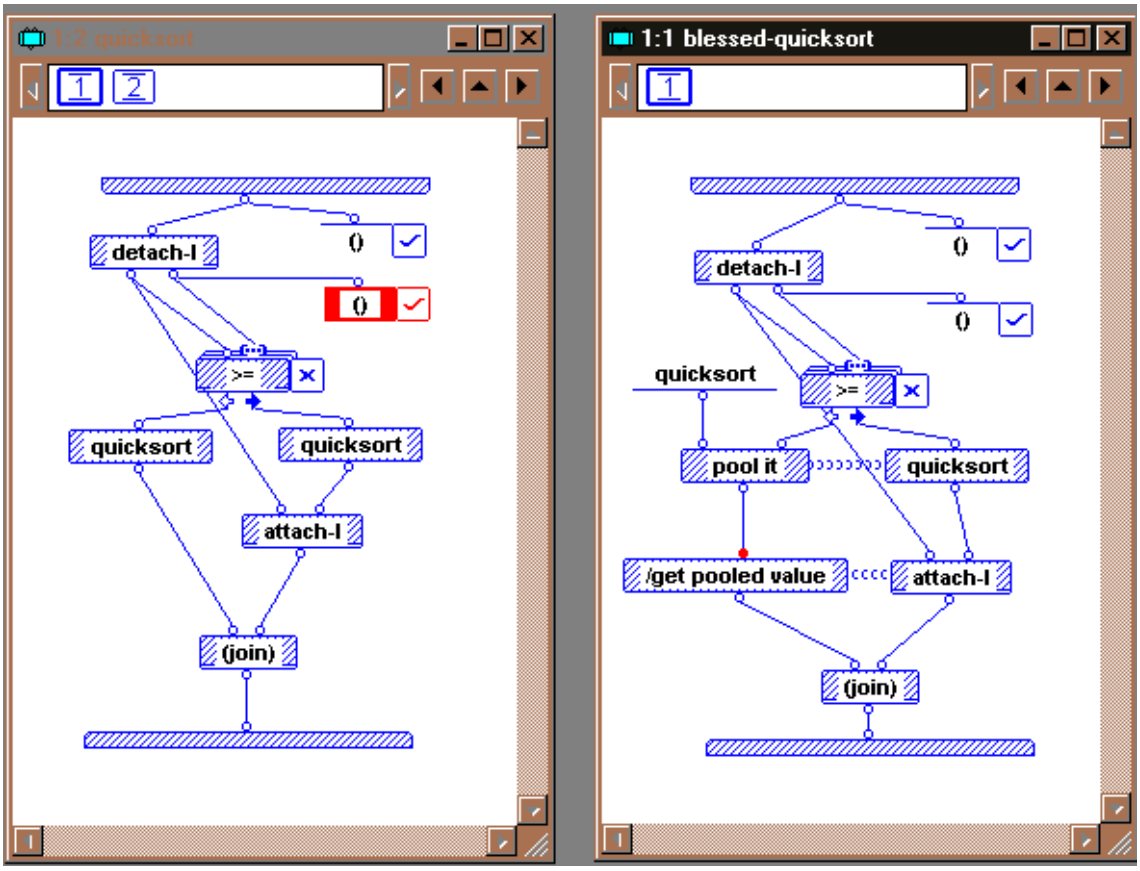

Figure 1: Pooling Annotations for Quicksort in CPX; sequential on the left and "pooled" on the right

that other processing units be made available to the console. The approach taken in the current system is that "remote" or "server" workstations locate the machine on which the "main" computation (and hence the pool) are running. These workstations then take tasks from the pool, compute the tasks, and return the result to the pool.

The union of the console and all the servers is the set of "nodes" in the parallel machine. The topology of the node set comprising this "parallel machine" is shown in Figure 2.

The semantics of the primary computation must not be dependent on the presence or otherwise of the server stations. Thus a sequential program to which pool annotations have been added must return an equivalent result to the original sequential program when only the console processing unit is available. Currently this is achieved only for referentially transparent programs. For these programs, no data is shared between nodes, so non-deterministic execution is avoided. Extension of the pool programming environment to support global data is currently in progress. This is a non-trivial extension to the set of all supported programs, requiring program analysis tools and / or further interaction with the programmer through the program development environment.

Bound and Unbound Tasks

Some computations must be performed on the console. For example, any I / O based operations, such as error message reporting, cannot be exported (since the messages would be written to the console of the remote machine.

To provide location constraints on pooled computations, we introduce the notion of bound and unbound tasks. A *bound* task is one which must be executed on the console. An *unbound* task can be executed on any machine in the parallel machine.

The "boundness" of the pooled task is specified at time of pooling, via an additional annotation to the `pool it` operation. Textual equivalents would be of the form

```
pool it( taskName, args, Unbound );

pool it( taskName, args, Bound );
```

Implementation Issues

The simple and uncluttered view of the pool model as seen by the user is supported by a complex run-time system to which the user-program is linked. This section highlights some properties of this run-time system.

Target Platforms

The run-time system in the console program has the capability to manage the user program, the pool, network communications and local concurrent processing. This

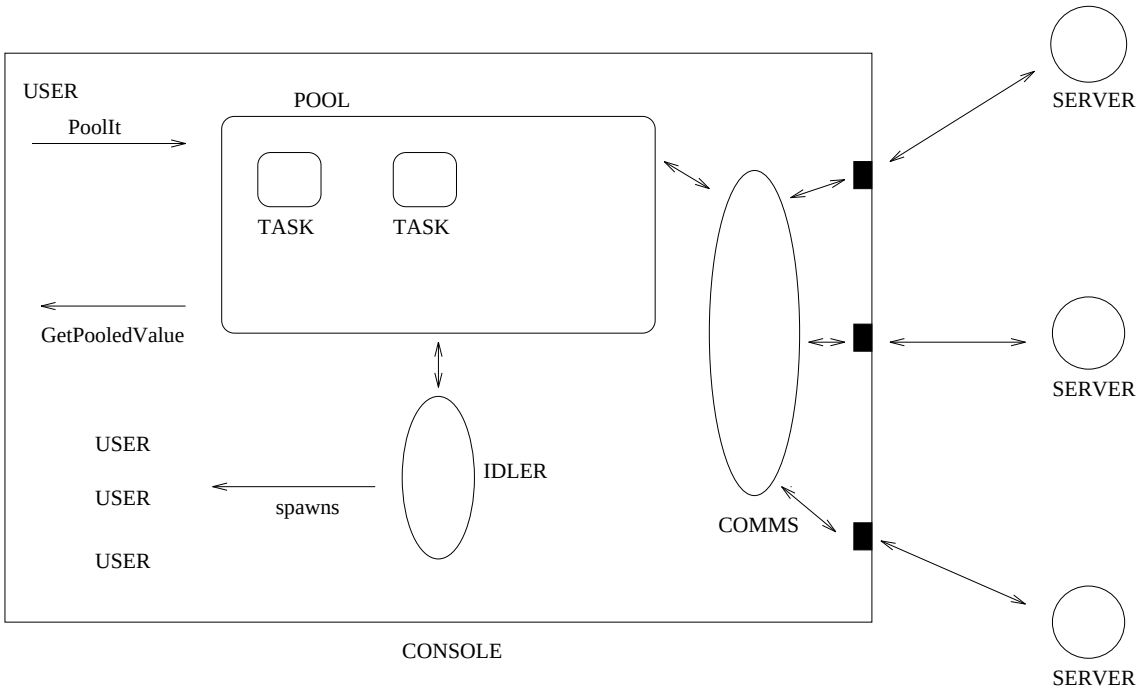

Figure 2: Parallel Machine Topology

means that the same model of programming (and in consequence the same user-program source code) can be used across a variety of parallel implementations: shared memory parallel processing (e.g. a 4 processor Windows box with thread capabilities built into the operating system) and distributed memory parallel processing (a network of Unix workstations). Of course the pool run-time system must be made available for each of these platforms.

Code Availability

We assume of form a SPMD processing for the remote server stations — servers are setup so that will be able to locate the required code corresponding to *any* task they wish to compute. In the case of locally spawned tasks, code availability is not an issue.

Transport Strategies

A number of middleware communication packages are available e.g. PVM, MPI. Our motivation for *not* using such a package is a lack of software port of any of these systems to the Apple Macintosh, currently the sole supported platform for the CPX system.

The communication strategy used in the pool system described is thus sockets, available on a variety of platforms including Unix, Windows, Apple Macintosh.

Thread Strategies

The console program (see above) runs both the primary computation and manages the pool, spawning local computations, exporting operations and collecting results. It is therefore highly desirable that the console be a multithreaded process. The alternative would be a heavyweight and therefore expensive IPC mechanism. This requirement implies the use of a programming environment supporting threads. Amongst the hardware currently available, threads are supported by the Solaris and Linux flavours of Unix — the console program must be run on such a machine.

All shared data in the run-time system, notably the pool and each task object, is protected by traditional concurrent programming primitives: semaphores and mutexes. Note however that the run-time system is unaware of the contents of the user program; it is currently the programmer's responsibility to maintain program data integrity. This issue has not arisen in our experiments with purely functional programs.

Task Racing

The run-time system supports task racing. When a node in the machine becomes idle, it requests a task from the pool. If the pool has no as yet unstarted tasks, a task already executing elsewhere may be returned to satisfy the work request. It is then the case that multiple nodes will be computing the same task, thus racing. The first result returned becomes the result of the task; further results are discarded. No thread suspension or killing is available to clean up task racing, so redundant computations are possible.

Conclusions

We have introduced a parallel programming system based on the Pool Model. This model supports a heavyweight parallel programming model, and is thus suited to coarse grain parallelism. We have discussed issues of task placement in the presence of constraints due to

for example I/O, and have shown that the system supports task racing, whereby multiple processors execute the same computation and one "wins" by returning the result.

Work is now focusing on supporting programs with shared or global data. Problems involved here include access to distributed global data, non-determinism due to race conditions and deadlock.

Acknowledgements

This work is supported by UK EPSRC project GR/K41526.

References

1 Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

2 Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

3 Ian Foster et al. The Nexus task-parallel runtime system. In *Proceedings of the 1st International Workshop on Parallel Processing*, pages 457–462. Tata McGraw Hill, 1994.

4 Al Geist et al. *PVM 3 User's Guide and Reference Manual*, sep 1994.

5 F.R. Giles, P.T. Cox, and T. Pietrzykowski. Prograph: A step towards liberating programming from textual conditioning. In *Proceedings of IEEE Workshop on Visual Languages*, pages 150–156, 1989.

6 W. F. McColl. BSP programming. In *Specification of Parallel Algorithms. Proc. DIMACS Workshop*, pages 21–35, Princeton, may 1994.

7 Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *acm Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.