pFun: A semi-explicit parallel purely functional language

André Du Bois, Adenauer Yamin, Maurício Pilla
Escola de Informática,
Universidade Católica de Pelotas
CEP: 96010-000 , Pelotas-RS, Brazil
{dubois, adenauer, pilla}@ucpel.tche.br

Gerson Cavalheiro
Universidade Federal de Pelotas
gerson.cavalheiro@ufpel.edu.br

Abstract

In this paper we present the design and implementation of pFun, a semi-explicit parallel purely functional language. Parallelism is introduced in pFun through simple annotations in the code. Task creation, synchronization and scheduling of computations on remote hosts are managed automatically by pFun's distributed runtime system. pFun's programming model and runtime system are described, and preliminary measurements of the current prototype implementation are presented on an SMP-machine and on a Beowulf Cluster.

1. Introduction

With the arrival of multi-core chips for domestic computers, parallel programming is becoming a mandatory feature in programming languages. Furthermore, networks are increasingly pervasive, and big companies are using clusters or grids to solve large-scale computation problems. Parallel programming is hard, and it is difficult to find programmers that really understand concurrency [7], even more considering the diversity of parallel architectures.

To address these challenges, we advocate the need of a programming platform that provides a simple programming model in which parallelism can be easily expressed and is readily available. Such a language must be supported by an advanced runtime system that provides automatic creation, synchronization and scheduling of computations on remote hosts.

Functional languages have the interesting property that subexpressions of a purely functional program can be evaluated in parallel at any order, always delivering the same final result for the whole expression. Subexpressions of a program will never have implicit control dependencies between them, as the ones introduced by assignment [5].

*p*Fun is a semi-explicit parallel functional language that provides a simple programming model through parallel annotations. The highest level of the *p*Fun architecture provides a semi-explicit parallel language offering two parallel constructs: `par` and `sync` that are used to indicate which expressions in the program could be evaluated in parallel. Task creation, distribution and synchronization are left to the implementation of the language, i.e., its runtime system in bottom levels. Furthermore, the parallel primitives of the language can be used to implement higher-level coordination primitives, such as algorithmic skeletons, i.e., higher-order functions that encapsulate common patterns of parallel computing.

The second layer guarantees execution on heterogeneous environment by compiling programs into architecture-independent byte-code, and *p*Fun's runtime system provides ways for serializing and communicating computations between different processes or hosts. *p*Fun's runtime system is implemented using standard C and TCP/IP sockets for communication, maintaining a high degree of portability. Hence, it allows code to be distributed on demand if executing on grid environments, providing code mobility.

Finally, the third level is responsible for achieving efficient execution of programs in multiprocessors and multicomputers. This level implements a scheduling strategy able to execute efficiently the nested fork-join structure of a parallel functional language.

In this paper, we present the design and implementation of *p*Fun. Our preliminary results validate *p*Fun's runtime system and show that the work-stealing mechanism used for load balancing performs better for data parallel programs.

2. The *p*Fun Language

*p*Fun is a *strict* parallel purely functional language with a syntax similar to Haskell [4]. *p*Fun's syntax, semantics and parallel primitives will be presented through examples in the following Sections. The reader should be aware that *p*Fun *is not Haskell.* Its syntax is similar to Haskell only for convenience.

2.1. The *p*Fun primitives for expressing parallelism

The *p*Fun language provides two basic primitives for expressing parallelism: `par` and `sync`. The `par` primitive is used to express potential parallelism. It takes as an argument a *p*Fun expression of any type and returns a reference to a `Par` value that represents an expression that *could* be evaluated in parallel:

```
par :: a -> Par a
```

The `par` primitive only indicates potential parallelism in the program and it does not guarantee that the expression will be evaluated in parallel with the rest of the program. Task creation, scheduling, distribution and synchronization are left to the implementation of the language as described in Section 3.

The `sync` primitive receives as an argument a reference to a `Par` value and returns the result of the evaluation of that expression:

```
sync :: Par a -> a
```

The operational behavior of the `sync` primitive is to block in its argument if it is being evaluated (by a different processor or remote location) or to create a local thread to evaluate it. The `sync` primitive will only proceed once its argument is evaluated to normal form.

2.1.1. Properties of `par` and `sync` *p*Fun is a *purely functional language.* For any purely functional expression `exp` written in *p*Fun, the following property must hold:

```
sync (par exp) == exp
```

It does not matter if a `Par` value is evaluated locally or on a remote host, it will always return the same result.

In the following Section, some examples using `par` and `sync` are given.

2.2. Example 1: Parallel Fibonacci

In this section we present a naive implementation of a parallel fibonacci function, just to demonstrate the use of the `par` and `sync`. A simple parallel version fibonacci function can be implemented as:

```
parFib n = if (n<=1) then 1
           else let
                  fib2 = par (parFib (n-2));
                  fib1 = par (parFib (n-1))
                in (sync fib2) + (sync fib1);
```

In the definition of `parFib`, the two recursive calls are *marked* with the `par` primitive to be computed in parallel. It can be very inefficient to create parallel tasks to evaluate every recursive call to `parFib`, since calculating fibonacci of small numbers is a fine grained task. A threshold could be used in the definition to limit the number of parallel tasks created.

2.3. Example 2: The `parMap` Skeleton

Using the `par` and `sync` primitives it is possible to write more powerful constructors for the *p*Fun language, such as Algorithmic Skeletons [3]. Algorithmic skeletons are higher-order functions that encapsulate common patterns of parallel computation.

These functions can be used by application programmers to write parallel software easily. For example, the function `map`, present in every functional language, is a higher-order function that takes two arguments, a function and a list, and applies the function to every element of the list generating a new list.

A parallel `map` is a function that has the same type as the sequential `map` but applies the function argument to every element of the list in parallel.

To implement a parallel `map` in *p*Fun, we first need a function to create parallel tasks to evaluate the application of a function to every element of a list:

```
parList:: (a->b) -> [a] -> [Par b]
parList f l = case l of
        [] -> [];
        (x:xs) -> (par (f x)) : (parList f xs);
```

The function `parList` creates a list of possible parallel tasks, and we need a way of accessing the values computed by these tasks:

```
syncList :: [Par a] -> [a]
syncList list = map sync list;
```

Finally, the `parMap` skeleton can be implemented using `parList` and `syncList`:

```
parMap :: (a->b) -> [a] -> [b]
parMap f l = syncList (parList f l);
```

Notice that `parMap` first uses `parList` to create all the parallel tasks, and only when all tasks are created it uses `syncList` to collect the results.

3. Distributed Scheduling

The *p*Fun runtime runs a dynamic scheduler implementing a work stealing strategy. This scheduling is particularly interesting to exploit the inherent nested fork-join program structure obtained by the `par/sync` parallel constructors. In this section the term *thread* represents a sequence of *tasks*, i.e., a `Par` object being executed. Tasks are bounded in the context of threads by `par/sync` calls.

Work stealing is a general denomination of a receiver-initiated class of distributed load balancing schedulers. The basic algorithm assumes that the responsibility for managing and executing the set of tasks generated by a running application is shared among processors of a parallel machine. The number of tasks executing simultaneously on each processor is limited in order to allow each processor to maintain a reserve of work represented by a ready tasks queue. The scheduler uses the length of ready queues as the load information. Also distributed among the processors is the control for scheduling decisions. Depending on the size of such queues a processor can start a scheduling operation. A processor sends a request for new ready tasks (a work steal) to another randomly chosen processor when its local ready queue reaches a value below of a certain limit. When a processor receives a steal message it will answer with a task taken from its local queue if the amount of work in reserve is above a certain limit; otherwise the message is forwarded to another randomly chosen processor.

The current prototype was developed for multicomputers, where communication costs are high. In order to avoid large number of messages to achieve load balancing, a hierarchical implementation of the runtime scheduler in terms of *work server* and *slaves* allows to reduce communication when compared to the standard work stealing algorithm. The slave nodes are dedicated to execute threads while work servers can also answer steal requests. The `par` primitive adds a task to the ready queue of a work server. Slaves are hosts that connect to a work server asking for computations to execute. A slave receives work, executes it, and sends the result back to its server.

4. Preliminary results

The first set of experiments in this section were performed on 8 computers, each with an AMD Athlon(tm) XP 2400+ processor and 192MB of RAM, using one as a work server and the other 7 as slaves. Table 1 shows the runtimes for 4 different programs and

	1 Proc (sec)	2 Proc (sec)	4 Proc (sec)	6 Proc (sec)	8 Proc (sec)
`parFib`	58.7	27.7	31.8	15.9	21.0
`parMapFib35`	103.1	51.3	31.2	26.8	15.6
`pMaze`	46.7	23.2	13.9	9.3	9.3
`pCoins`	48.5	35.9	36.1	36.2	37.5

Table 1. *p*Fun on 8 nodes (1 work server and 7 slaves)

The speedups reported here and throughout this section are *relative*, i.e., improvement over the single-processor parallel execution. `parFib` is the program given in section 2.2. The second program is `parMapFib35`, which calculates 8 times the `seqFib` of 35. As `parMap` is used, this program generates 8 threads that can be evaluated in parallel, one for each element of the list. The `pMaze` program searches for an exit in a maze. The maze is represented as a tree and we use depth first search to find the exit. Parallelism is introduced with `parMap`. `pCoins` is a more realistic program: given a collection of coins and an amount to be paid, it computes the number of possible ways to pay it. It uses a divide-and-conquer algorithm and parallelism is again introduced with the `parMap` skeleton.

Table 1 shows that, for the set-up used in the experiments, the distributed scheduling performed by *p*Fun's runtime system works better for data parallel programs (`parMapFib35` and `pMaze`) than for divide-and-conquer programs (`parFib` and `pCoins`). `parMapFib35` creates only 8 tasks, one for each computer, hence it improves run time for all number of processors measured. The same happens with `pMaze` that creates 10 threads of equal size, one to evaluate each branch of the tree. In `parFib` there is an improvement of performance up to 6 processors, after that there is an increase of communication in the system affecting performance: there are many idle slaves sending messages asking for work, and the work server's ready queue is empty. The `pCoins` program creates 1 large grained thread, and many fine grained threads, therefore the runtime on more than one processor is always the time needed to evaluate the larger thread.

Another interesting result came from a different setup: we used only one laptop computer with a Centrino Duo 1.60GHz processor, an Intel dual core processor for laptops, and only one work server and one slave,

	Speedup
`parfib`	1.75
`pmapFib35`	1.93
`pMaze`	1.96
`pCoins`	1.33

Table 2. Speedup on a Centrino Dual Core

each allocated to a different core of the processor. For all parallel programs some speedup was achieved as can be seen in Table 2.

5. Related Work

The potential of functional programming languages to support parallelism has been recognized for a long time and several extensions for parallel programming in functional languages have been implemented (for a survey on the field, the reader should refer to [5]). Here we discuss the ones that are more closely related to *p*Fun.

GPH [8] is a parallel extension of Haskell for parallel programming. To express parallelism, the programmer uses a `par` combinator (similar to *p*Fun's `par`). Since Haskell is a lazy language, it is difficult to predict the order of evaluation of expressions, thus the `seq` combinator must be used to control sequencing. GUM (Graph reduction for a Unified Machine model) [8] is the distributed runtime system that implements GPH. It is also the core of several other Haskell extensions for parallel and distributed computing like Eden [2] and Distributed Haskell [6]. GUM is designed to run on Beowulf clusters, hence all PEs (Processing Elements) know each other and they all function as work servers: when a PE is idle, it can search for work on other PEs. GUM is also a closed system: once the system is running, no other PE can join the computation, while in *p*Fun's model, slaves can join a work server at anytime. Another difference between GUM and *p*Fun's runtime system is that, in GUM, the machine code for parallel applications must be installed on all the PEs before execution can be started, while in *p*Fun the code can be distributed together with task to be executed.

Grid/ML [1] is an extension of Standard ML for GRID programming. In Grid/ML all nodes maintain a queue of pending work, and they can steal work from other nodes. Grid/ML provides primitives (similar to *p*Fun's primitives) to express parallelism and populate the node's queue of pending work. The Grid/ML system focuses mainly on fault-tolerant distributed programming: GRID applications are written as a series of deterministic functions that can be memorized by the network and restarted at any time. No measurements of their current implementation are given.

6. Conclusions and Future Work

We have presented the design and implementation of *p*Fun, a strict parallel functional language. *p*Fun provides an intuitive parallel programming model for heterogeneous environment doted with an efficient runtime in a layered architecture.

There are a number of issues that could be investigated in the future. *p*Fun's implementation could easily be extended to provide security through authentication: all code being communicated is serialized into strings, and these strings can be easily encoded through cryptography, e.g., using the public key of the sender/receiver. In that way, we can ensure that the code received by a slave comes from a trusted work server.

In the case of failure of one node, purely functional expressions can be re-started at any time as they are free of side effects. *p*Fun's architecture could be extended with a failure recovering system, such as the one provided by Grid/ML [1].

In the language level, we plan to investigate how *p*Fun's programming model could be extended with higher-level abstractions to express parallelism, e.g., using different parallel skeletons.

References

[1] T. Murphy VII. Ml Grid programming with concert. In *The 2006 ACM SIGPLAN Workshop on ML (ML 2006)*. ACM press, 2006.

[2] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden Coordination Model for Distributed Memory Systems. In *HIPS*. IEEE Press, 1997.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.

[4] S. L. P. J. (Editor). Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 1(13), 2003.

[5] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, UK, 1999.

[6] R. Pointon, P. Trinder, and H.-W. Loidl. The design and implementation of Glasgow Distributed Haskell. In *IFL 2000*, LNCS, Volume 2011. Springer-Verlag, 2000.

[7] H. Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[8] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: a portable implementation of Haskell. In *PLDI*, Philadephia, May 1996.