Message–passing Implementation for Process Functional Language[1]

Miroslav Vidiščak

Miroslav.Vidiscak@tuke.sk

Abstract: In this paper we introduce the concept an implementation of distributed programming in PFL – a process functional programming language and describe implementation of concurrency in other Haskell–like parallel functional languages – Concurrent Haskell and GpH. The process of writing parallel program is complicated by the need to specify both the parallel behaviour of the program and the algorithm that is to be used to compute its result. Main ideas about implementation of parallel execution in PFL are make PFL able to handle huge set of problems that are effective running only in distributed environments.

Keywords: functional languages, process functional language, message–passing interface, distributed execution

Introduction

Functional languages are programming languages, which are radically different from imperative languages. A program is expressed as a function from its input to its output. This differ functional languages from many of real systems, because they are not functional. The programs are executed using I/O, exceptions, interrupt handling, communication, etc. depending on the application to which they are proposed.

Functional languages have at least two benefits: the first is that they are mathematically tractable and hence they can be reasoned about more easily than conventional languages. This also makes program derivation much easier. The second benefit is that functional programs are amenable to parallel evaluation.

Concurrency is necessary in nearly all applications nowadays. A concurrent program does several things at once and consists of a number of execution units

[1] This work was supported by VEGA Grant No. 1/1065/04 – Specification and implementation of aspects in programming.

(either processes or threads), which are supposed to be executed independently at the same time on the same processor or computer.

In past, concurrency was based only on processes that were statically defined from within the program for efficiency. Later, the evolution of software required a more flexible approach to concurrency, where concurrent systems were designed without the assumption on how many processes will be needed. This led to the inclusion of mechanisms for dynamic process creation in procedural languages. Still, processes were too monolithic to meet the requirements of today's software since several activities need to take place within each process. Many programming languages and systems are designed or specifically adapted to support multi–threading [2].

Process functional language is an experimental imperative functional programming language [4, 5, 6, 7]. PFL incorporates all purely functional features. In addition, new special constructs are used in process functional language for increasing the run–time performance. Process functional language reduces a gap between a purely functional language and an imperative language. It is also proved that all imperative programs can be transformed to process functional programs [6]. Currently we have three generators from PFL – a generator to Java, Haskell and C++ with support for Message–Passing Interface [8] and extend PFL to work in distributed environment.

The goal of MPI is to develop a widely used standard for writing message–passing programs. As such the interface attempts to establish a practical, portable, efficient and flexible standard for message passing [8]. Nowdays, there are several implementation of MPI, including versions for networks of workstations, clusters of personal computers, distributed–memory multiprocessors, and shared–memory machines.

1 Parallel Functional Languages

At the present time, there are many implementations of parallel functional languages, which used different models for implementation parallel execution. Each of them has own advantages and disadvantages.

1.1 Concurrent Haskell

Concurrent Haskell is a concurrent extension to the functional language Haskell [3]. It adds to Haskell only four new primitive operations, which are both expressive and easy to implement.

Concurrent Haskell provides a new primitive *forkIO*, which starts a concurrent processes:

$$forkIO :: IO () \rightarrow IO ()$$

$$forkIO\ m\ s = s' \text{ `par` } return\ r\ s$$

$$where\ (r, s') = m\ s$$

forkIO is an action that takes an action as its argument and spawns a concurrent process to perform that action. The I/O and other side effects performed by *m s* are interleaved in an unspecified fashion with those that follow the *forkIO*.

Concurrent Haskell run as a single Unix process, performing its own scheduling internally. Each use of *forkIO* creates new process. The scheduler can be told to run either pre-emptively (timeslicing among runnable processes) or non-pre-emptively (running each process until it blocks). The scheduler only switches processes at well defined points at the beginning of basic blocks.

Synchronisation in Concurent Haskell is based on mutable variables (*MVars*)[1]. Mutable variable is similar to a lock variable and represents a mutable location that is either empty or contains a value of type *t*. The following primitives have been provided to create, access and alter *MVars*:

$$newMVar :: IO\ (MVar\ t)$$

$$takeMVar :: MVar\ t \rightarrow IO\ t$$

$$putMVar\ MVar\ t \rightarrow t \rightarrow IO\ ()$$

Primitive *newMVar* creates a new *MVar*, *takeMVar* blocks until the location is nonempty, then reads and returns the value of an *MVar*, *putMVar* writes a value to *MVar*.

Thus to implementation as single Unix thread, Concurrent Haskell is well–suited for running on single machine. Enhancement to run in distributed environment is hard and need more low–level programming.

1.2 GpH/GUM

GpH (Glasgow Parallel Haskell), extends Haskell with modest parallelism primitives, GUM (Graph Reduction on a Unified Model) is the parallel system that supports GpH [12].

GUM is architecture–independent runtime systems for Glasgow Parallel Haskell 12, a parallel variant of the Haskell lazy functional language. It can be used on both shared–memory and distributed–memory architectures. It used standard PVM message–passing library [13].

GpH is a small extension to the standard Haskell lazy functional language. The model of parallelism in GpH is mainly implicit with dynamic resource allocation. All work for parallel execution, like mapping of threads to processors, communication among threads and thread synchronization, is done by runtime system (GUM).

Simple parallel programming model is providing operator 'par'. The expression

$$p \text{ 'par' } e$$

has the same value as e and indicate that p could be evaluated by a new parallel thread, with the parent thread continuing evaluation of e. Since control of sequencing can be important in parallel functional languages, a sequential composition operator 'seq' is used

$$e_1 \text{ 'seq' } e_2$$

The expression has the same value as e_2, but e_2 cannot be evaluated before e_1 is evaluated.

Better parallel performance in GpH is done with using lazy higher–order functions to separate the two concerns: specifying the algorithm and specifying the program's dynamic behaviour. A function definition is split into two parts, the algorithm and the strategy, with values defined in the former being manipulated in the latter. The algorithmic code is consequently uncluttered by details relating only to the dynamic behaviour. In fact the driving philosophy behind evaluation strategies is that it should be possible to understand the semantics of a function without considering its dynamic behaviour.

A strategy is a function that specifies the dynamic behaviour required when computing a value of a given type. The simplest strategies introduce no parallelism: they specify only the evaluation degree.

2 PFL

Process functional language PFL [7] is language with positive properties imperative and functional languages. New PFL elements – environment variables, loop comprehension and spatial types, differ the process functional language from a purely functional one. The approach to programming is the same like in functional language (with function/process application). In addition, using the environment variables we can manipulate with states like in imperative languages. PFL also provides ways to profiling functional languages [5, 9, 10] and abstract type definitions [14, 15].

2.1 Computation Strategies

Any parallel programming language must be based on certain assumptions about the underlying machine. The philosophy to parallel programming with functional languages has been to find the minimum necessary to write efficient parallel functional programs for the target machine. In particular it was desired to relieve the programmer from as much parallel organization as possible. The programmer must devise a parallel program and annotate it to indicate which expressions are suitable for parallel evaluation. Thus the programmer is responsible for addressing the question *what to spark?* but not where or when execute tasks.

We can recognize these types of idealized parallelism types:

- Farm parallelism
    - Farm parallelism paradigm consists of the entities master and multiple slaves. The master is responsible for decomposing the problem into small tasks (and distributes these tasks among a farm slave processes), as well as for gathering the partial results in order to produce the final result of the computation. The slave processes execute in a very simple cycle: get a message with the task, process the task, and send the result to the master. The computation terminates when all processes terminate. Usually, the communication takes place only between the master and the slaves.
- Pipeline parallelism
    - This is a more fine–grained parallelism, which is based on a functional decomposition approach: the tasks of the algorithm, which are capable of concurrent operations, are identified and each processor executes a small part of the total algorithm. The pipeline is one of the simplest and most popular functional decomposition paradigms. Processes are organized in a pipeline – each process corresponds to a stage of the pipeline and is responsible for a particular task. The efficiency of this paradigm is directly dependent on the ability to balance the load across the stages of the pipeline. This paradigm is often used in data reduction or image processing applications.
    - The tasks (processes) are executed to solve different problems P_i, each processing the same processing the same type of data. Therefore this decomposition is sometimes called functional decomposition. In general, it is hardly decompose a problem to a sufficiently long pipeline of subproblems, which would utilize high number of processors.
- Expansive parallelism
    - Expansive parallelism is method where a problem P is suggested to be solving set of n same problems recursively. For $n=2$ method is called *divide–and–conquer*. The divide and conquer approach is well known in

sequential algorithm development. A problem is divided up into two or more subproblems. Each of these subproblems is solved independently and their results are combined to give the final result. We can identify three generic computational operations for divide and conquer: split, compute and join. The application is organized in a sort of virtual tree: some of the processes create subtasks and have combined the results of those to produce an aggregate result.

- Massive parallelism
  - Massive parallelism problem *P* is solved at the same time on a set of data in parallel. The computation is based on independent tasks running on independent data sets also data sets may be overlapped. In a typical case this is fine grained parallelism, that may be utilized on SIMD architectures using data parallel programming model, but also on MIMD architectures using message passing programming model (in a more coarser manner, with the additional software overhead).

Two popular methods for writing parallel programs are *expressing parallelism using combinators* (such as map and scan), and *farm parallel programming* (which is commonly supported by many available parallel systems). These methods offer complementary advantages and are used in implementation of distributed application of PFL.

Parallel programming languages consist of two parts:

- A set of *parallel operations*. For high level programming with farm parallelism, every parallel operation is expressed directly as a combinator. For massive parallelism, different mechanisms are used, depending on whether the parallel operation is purely local or uses interprocessor communication.
- A *coordination language*, which expresses with parallel combinators, the coordination language could comprise the entire the entire functional language (e.g. Haskell), but it could also be restricted to functions written in a particular form. For conventional farm parallel programming method, the coordination language is C or Fortran (if we used MPI).

The *combinator method* is well suited for abstract, high level specifications of algorithms. A family of functions, such as *map*, *fold* and *scan*, is used to express parallel computations. For example, a set of parallel local computations using the same function *f* can be expressed as *par_map f xs*, where the data structure of *xs*, are executed simultaneously in different processors. There is a rich set of mathematical laws relating the combinators, making this approach well suited for formal reasoning as well as a variety of optimisations and program transformations.

The *farm parallelism programming method* is another popular model for programming parallel computers. Many available parallel systems support this

style (SPMD – Single program, multiple data), and it offers relatively good program portability. The idea behind SPMD is simply that the programmer writes a program that will run on one processor, but the parallel operating system executes it concurently. The term SPMD is apt because the processors all use the same program, but they normaly have different data in their local memories.

Both of these methods are used in implementation of distributed computation in PFL.

2.2 Generator Architecture

Generator for distributed implementation of PFL use existing implementation of PFL, but it extends to generate code to C++ with or without MPI support. In this way, we can measure effectivity of implementation PFL in distributed environment. Other aspect of implementation, like profiling, use existing base of PFL implementation [9].

As a base for generating distributed computation was choosen a C++ language for easily binding to MPI. As a mid–layer between pure C++ and MPI is used object–oriented implementation of MPI called 11 [11]. It allows using full features of C++ – overloading operators, polymorphism, inheritance, abstract datatyping in easy way. We can use C++ extension of MPI–2, but in present time, there is only few implementation of MPI–2.

2.2.1 Implementation of Basic Send/Receive Functions

Primitive Send/Receive functions are defined explicitly. Primitive function type definition is placed in source code of PFL program:

primitive mpiSend :: Integer → List a → ()

and function definition in a file Primitives.ctp written in a C++ language:

```
static Object* mpiSend( Object* a, Object* b ){
    return (((new Constructor( "()", 0)) → init())→ apply(
        11_COMM_WORLD[*((Long*)a)].Send((Object*)b))
        );
}
```

2.2.2 Implementation of Collective Operations

Collective operations, like *map*, *foldl*, etc. are implemented using different method.

Here is a generated code for par_map that is generated throught PFL generator with MPI support switch enabled (source code in PFL is *par_map plusInteger 3 lista*, where *lista* is is generic list, contains integer number):

```
class pfl_par_map : public Function{

virtual Object* expression( void ) {

    Object *oro_19184575 = NULL, *oro_33409388 = NULL, *oMsg;

    int nSize=11_COMM_WORLD.Size();

    int nRank=11_COMM_WORLD.Rank();

    oro_33409388 = ((Function*) ((Function*) new pfl_lista()) → init()) → expression();

    int iCount = oro_33409388→args→size() / nSize;

    11_Request_array ra(iCount);

    if( nRank == 0 ){

        for(int i=1; i<nSize; i++)

            for(int j=(i-1)*iCount; j<oro_33409388→args→size(); j++){

                oMsg = ((Function*)oro_33409388→args→at(j));

                11_COMM_WORLD[i].Send( oMsg );

            }

            oro_19184575 = ((Function*)new pfl_par_map())→init();

            for(i=1; i<nSize; i++)

                for(int j=(i-1)*iCount; j<oro_33409388→args→size(); j++){

                    11_COMM_WORLD[i].Recv( oMsg );

                    ((Function*)oro_19184575)->apply( oMsg );

                }

    }

    else{

        Object* oMsg_19184575;

        for(int i=0; i<iCount;i++){

            11_COMM_WORLD[0].Recv( oMsg );

            oMsg_19184575 = ((Function*) (( new pfl_plusInteger() ) → init() ) → apply( (new Long( 3 )) ) )) → apply( oMsg )) → expression();
```

```
            11_COMM_WORLD[0].Send( oMsg_19184575 );
        }
        ra.Waitall();
        return oro_19184575;
    }
};
```

Conclusions

In this paper a methodology for implementation a message–passing paradigm from a PFL specification was shown. Sections of this methodology are mechanical in nature, and could be supported by transformation tools. Using this methodology, we have derived small parallel programs from specifications. The message–passing model has been implemented using MPI; with this the derived programs have been executed on a group of workstations. A direction for future work is to improve effectivity of message–passing implemetation. For now, we use simple synchronisation mechanism (*ra.Waitall()*) for waiting to finish message–passing communications. Detailed description about particular part of implementation (object hierarchy, garbage collector, etc.) will be presented in further papers.

References

[1] Finne, S. and Jones, S., P. J.: Concurrent Haskell, In Principles Of Programming Languages, St. Petersburg Beach, Florida, 1996, pp. 295–308

[2] Hammond, K.: Parallel Functional Programming: An Introduction, International Symposium on Parallel Symbolic Computation, Hagenberg/Linz, 1994, 19 pp.

[3] Jones, S. P., Gordon, A., Finne, S.: Concurrent Haskell, Conference Record of POPL '96: The 23rd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, Glasgow, 1996, 11 pp.

[4] Kollár, J.: Control–driven Data Flow, Journal of Electrical Engineering, Vol. 51. No. 3–4, 2000, pp. 67–74

[5] Kollár, J., Porubän, J., Václavík, P., Vidiščak, M.: Lazy State Evaluation of Process Functional Program, 5th International Conference ISM 2002, Rožnov pod Radhoštim, Czech Republic, April 22–24, 2002, ISBN 8085988704

[6] Kollár, J.: PFL Expressions for Imperative Control Structures. Computer Engineering and Informatics conference with International participation, Oct. 14–15, 1999, Herľany, Slovakia, pp. 23-28 ISBN 80-88922-05-4

[7] Kollár, J.: Process Functional Programming, 33rd Spring International Conference MOSIS'99–ISM'99, Rožnov pod Radhoštem, Czech Republic, April 27–29, 1999, ACTA MOSIS No. 74, pp. 41–48

[8] MPI–2: Extensions for the Message–Passing Interface, Message Passing Interface Forum July 18, 1997, Tennessee, 323 pp.

[9] Porubän, J.: Profiling process functional programs, PhD Thesis, DCI FEI TU Košice, 2004, 90 pp. (in Slovak)

[10] Porubän, J.: Profiling process functional programs, Research report, DCI FEI TU Košice, 2002, 51.pp.

[11] Squyres, J. M., Willcock, J., McCandless, B. C., Rijks, P. W., Lumsdaine, A.: Object Oriented MPI (OOMPI): A C++ Class Library for MPI, Open System Laboratory, Pervasive Technologies Labs, Indiana University, September 3, 2003, 71 pp.

[12] Trinder, P. W., Barry, E., Davis, M. K., Hammond, K., Junaidu, S. B., Loidl, H., Loogen, R., Jones, S., P.: GpH: An Architecture–independent Functional Language, IEEE Transactions on Software Engineering, 1998, 23 pp.

[13] Trident, P., W., Hammond, K., Mattson, J., S., Partridge, A., S., Jones, S., P.: GUM: A Portable Parallel Implementation of Haskell, PLDI'96– Programming Languages Design and Implementation, Philadelphia, 1996, pp. 78–88

[14] Václavík, P.: Abstract types and their implementation in a process functional programming language, PhD Thesis, DCI FEI TU Košice, 2004, 111 pp. (in Slovak)

[15] Václavík, P.: Abstract types and their implementation in a process functional programming language, Research report DCI FEI TU Košice, 2002, 48 pp. (in Slovak)