Parallel Distributed Programming with Haskell+PVM

Noel Winstanley and John O'Donnell

University of Glasgow

Abstract. This paper presents a methodology to transform a pure functional specification into a distributed message-passing program via equational reasoning. The methodology uses a formal model of a message passing system. This abstract system can be implemented using PVM or MPI, and thus executable programs produced.

1 Introduction

This paper presents a methodology to transform a specification in Haskell, a purely functional language, into a distributed message-passing program via equational reasoning. This methodology has been applied to small examples and the resulting parallel programs proved correct and executed successfully on a network of workstations.

The methodology has two main stages. In a lazy functional language the evaluation order is implicit, defined by data dependencies between the values calculated. This is unsuitable for a message passing implementation, so we first create an explicit order of execution. The second stage of the transformation produces parallelism by introducing message-passing actions into the intermediate program.

An often-cited advantage of functional programming languages is their support for formal equational reasoning. Communications libraries such as PVM [3] and MPI [2] consist of large quantities of C code, thus it is impossible to reason formally about them. We use an abstract message passing model, characterised by a set of laws. Program equivalence proofs are then secure in the context of this model. The message passing model is made concrete by implementing using a communication library; we assume this implementation has the same semantics as the model.

Typically, an implementation of the communication model will have constraints in its use. For instance, in our implementation using PVM only basic types can be used to construct messages. This means that functions cannot be communicated between processes, which is inconvenient for a language with higher-order functions. We can work around this by transmitting symbolic representations of functions: these are encoded and decoded by the communicating tasks.

Due to limited space, in this paper we concentrate on some of the more interesting features of the methodology. The rest of the paper is structured as follows: Section 2 presents the methodology in more detail, execution order is examined

in Section 3, and the message passing model is described in Section 4. We conclude in Section 5. We assume some familiarity with functional programming, i.e. as described by Bird and Wadler [1], and with the concrete syntax and use of Haskell [4].

2 Methodology

The methodology transforms a functional specification into a parallel program which can be implemented using a message-passing library. The main stages of the methodology are as follows –

Calculate Dependencies Parallelism may be introduced where computations can be executed independently of one another. This can be found by analysing the dependencies of the specification (Section 3). Some algorithms have more capacity for parallelism than others; if little potential is found one should consider expressing the algorithm differently.

Order the Specification The specification should be transformed into a program with explicit sequencing of calculations,(Theorem. 5). An explicit execution order allows communication actions to be introduced; optimising re-arrangements of the order can also be made.

Introduce Tasks and Communication Decisions must be made as to which expressions to evaluate as new tasks. We believe that Haskell+PVM is best suited to large-grain parallelism, due to the computational cost of creating tasks and the latency of messages in a distributed memory system. In general the best expressions to parallelise are those at the highest level of the program. Tasks are introduced using equivalences between actions which produce local and distributed computations (Equations 12,13).

Optimise Reordering the execution of the program may increase parallelism. For example, operations to create new tasks should be executed as early as possible. We assume that this is an expensive operation; early creation allows more time for the tasks to be initialised before they are required. Rearrangements of code are made using dependency preserving transformations (Equations 1–6) and the laws of the message passing model (Section 4). This ensures that the resulting program has the same semantics.

Implement using a Message Passing Library The comunication primitives of the model must be translated into equivalent operations in the implementation of the model. Further optimisation is possible by replacing some model operations with different system calls. For instance, if the same message is sent to a set of tasks, a sequence of send operations can be replaced by a more efficient multicast.

When examining a sizeable program, there is a tradeoff between the complexity produced and the useful parallelism exposed. A limit may have to be chosen below which there is assumed to be no code worth parallelising, due to small granularity.

3 Dependency and Evaluation Order

The outcome of a computation may rely upon computations performed previously. This constrains the order of execution, and thus limits the ways an algorithm may be parallelised. These constraints may be found using *dependency analysis.* Assuming `f` is strict, in the following example `x` and `y` are independent, while `f x y` depends on both `x` and `y`.

```
let x = e
    y = e'
    in f x y
```

We define the following relations on the dependencies between two computations.

Definition. $a :\gg: b$ *('b depends on a').* The outcome of b relies upon the evaluation of a. $:\gg:$ is transitive, but not reflexive. $a :\|: b$ *('a is independent of b').* The evaluation of a and b are totally independent of one another. $:\|:$ is reflexive, but not transitive. $:\gg:$ and $:\|:$ are related by $\neg(a :\gg: b) \wedge \neg(b :\gg: a) \Leftrightarrow a :\|: b$.

Computations may depend upon one another in two distinct ways – by a value passed explicitly, or by a change in shared state. We can subdivide $:\gg:$ into $:\gg:_V$ for a value dependency and $:\gg:_S$ for a state dependency; likewise, $:\|:$ separates into $:\|:_V$ and $:\|:_S$. The following now holds

$$mA :\|: mB \Leftrightarrow mA :\|:_S mB \wedge mA :\|:_V mB$$
$$mA :\gg: mB \Leftrightarrow mA :\gg:_S mB \vee mA :\gg:_V mB$$

3.1 Ordering Combinators

All computations that a computation c is dependent upon must be completed before c can be executed. An explicit execution order which captures this can be specified using the following combinators.

Definition. SEQ takes two computations and executes them sequentially. i.e. $a\,\mathrm{SEQ}\,b$ evaluates a to completion, then b. IND takes two computations and executes them in an undefined order. This may be in sequence, in parallel or concurrently.

Theorem 1 : Execution Order. *An execution order for a sequence of computations is valid if, for all computations a,b,x in the sequence, the following is true.*

$$a :\gg: b \Rightarrow a\,\mathrm{SEQ}\,b \quad \text{or} \quad a\,\mathrm{SEQ}\,x\,\mathrm{SEQ}\,b \qquad \textit{where } \neg\, b :\gg: x \wedge \neg\, x :\gg: a$$
$$a :\|: b \Rightarrow a\,\mathrm{IND}\,b$$

A sequence of computations may be reordered and still give a valid result. This is useful, as alternative orderings may produce more parallelism. The following theorem gives transformations that can be used to reorder a sequence.

Theorem 2 : Transformation of SEQ **and** IND. *Let* $\mathcal{M}[\![a]\!]$ *denote the semantics of computation a. We define a transformation relation* $\stackrel{t}{\Longrightarrow}$ *such that for computations a,b,* $a \stackrel{t}{\Longrightarrow} b \Rightarrow \mathcal{M}[\![a]\!] = \mathcal{M}[\![b]\!]$ *The following are valid transformations —*

$$a \,\mathrm{IND}\, b \stackrel{t}{\Longleftrightarrow} b \,\mathrm{IND}\, a \ . \tag{1}$$

$$a :\|: b \Rightarrow a \,\mathrm{SEQ}\, b \stackrel{t}{\Longleftrightarrow} b \,\mathrm{SEQ}\, a \ . \tag{2}$$

$$a :\|: b \Rightarrow a \,\mathrm{SEQ}\, b \stackrel{t}{\Longleftrightarrow} a \,\mathrm{IND}\, b \ . \tag{3}$$

$$(a \,\mathrm{SEQ}\, b) \,\mathrm{IND}\, c \stackrel{t}{\Longrightarrow} a \,\mathrm{SEQ}\, (b \,\mathrm{IND}\, c) \ . \tag{4}$$

$$(a \,\mathrm{SEQ}\, b) \,\mathrm{IND}\, c \stackrel{t}{\Longrightarrow} (a \,\mathrm{IND}\, c) \,\mathrm{SEQ}\, b \ . \tag{5}$$

$$(a \,\mathrm{SEQ}\, b) \,\mathrm{IND}\, (c \,\mathrm{SEQ}\, d) \stackrel{t}{\Longrightarrow} (a \,\mathrm{IND}\, c) \,\mathrm{SEQ}\, (b \,\mathrm{IND}\, d) \ . \tag{6}$$

Note that the first three transformations are reversible, while the others are one-way, due to a loss of information. Transformation (6) extends for larger combinations of SEQ *and* IND.

3.2 Monads

Arbitrarily executing computations which produce side-effects breaks referential transparency. This can be prevented by defining an explicit order of execution for these computations. In Haskell, this is done using *monads* [5]. Haskell provides a convenient way to sequence monadic computations using 'do notation'. This is a language construct which orders a sequence of computations, binding results to variables and returning the the last computation's result.

Theorem 3 : Equivalence of Do and SEQ. *A do statement is equivalent to a sequence of computations combined with the* SEQ *combinator. The conversion between the two is purely syntactic. Do statements have nothing akin to the* IND *combinator; these must be transformed to* SEQ *combinators before converting to do statement syntax.*

Thus computations in a do statement may be reordered using the $\stackrel{t}{\Longrightarrow}$ relation: provided the same value is returned by the do statement, the semantics remain the same. Associated with monads are two operations —

```
return :: a -> IO a [1]        run :: IO a -> a
```

`return` takes a value and returns a computation, which just produces that value with no side effects. `run` executes a monadic computation, returning the value the computation produces. This operation should be used with care, as if misused can undermine referential transparency. However, we consider it safe within the context it is used in the methodology. The following laws relate `return`, `run` and `do` statements and can be used to remove from code the `run` operations introduced by the methodology.

[1] `IO a` is the type of a computation which will return a value of type `a` when executed.

Law 4 : Monad and Do Laws.

$$\texttt{run . return} = \texttt{id} \tag{7}$$

$$\texttt{(return . run) mA} = \texttt{mA} \tag{8}$$

$$\texttt{mA} = \texttt{do \{mA\}} \tag{9}$$

$$\texttt{do \{mA (run mB)\}} = \texttt{do \{v <- mB; mA v\}} \tag{10}$$

$$\texttt{do \{v <- do \{mA; mB\}; mC v\}} = \texttt{do \{mA; v <- mB; mC v\}} \tag{11}$$

3.3 Let Expressions and Do Statements

Let expressions are a fundamental feature of Haskell. The following theorem shows how they can be transformed to do statements. Thus we give an explicit order of execution to a functional program. This allows side-effecting computations (such as message passing) to be added to the program without breaking referential transparency.

Theorem 5 : Let to Do. *A let expression may be transformed into an do statement with the same semantics, provided the let expression contains no mutually recursive values.*[2] *During this transformation variables may have to be renamed whenever name spaces are combined.*

Proof of Theorem 5, by induction.
Base Case : A let expression of form `let v = exp in f v`

```
  = let v = (run . return) exp in f v                          (7)
  = let v = (run . return) exp in (run . return) (f v)         (7)
  = (run . return)(f ((run . return) exp))
  = run (do {return (f ((run . return) exp))})                 (9)
  = run (do {v <- return exp; return (f v)})                   (10)
```

Inductive Step : A let expression of form[3]

```
    let v = exp in let v' = exp' in ℰ v
  = let v = (run . return) exp in let v' = exp' in ℰ v         (7)
  = let v = (run . return) exp in
     run . return)(let v' = exp' in ℰ v)                       (7)
  = (run . return)(let v' = exp' in ℰ ((run . return) exp))
  = run (do {return (let v' = exp' in ℰ ((run . return) exp))}) (9)
  = run (do {v <- return exp; return (let v' = exp' in ℰ v)})  (10)
  = run (do {v <- return exp;
     return (run (do {v' <- return exp'; return (ℰ v)}))})     (ind. hyp.)
  = run (do {v <- return exp; v' <- return exp'; return (ℰ v)}) (8,11)
```

□

[2] A mutually recursive definition is indicated by two values such that $a :\gg: b \wedge b :\gg: a$

[3] Any non-mutually recursive let expression can be expressed in this form

Transforming Do to Let A similar transformation can be made from a do statement to a let expression. However, a let expression's execution can only be ordered by data dependencies: thus a complete transformation is only possible when every state dependency in the do statement is shadowed by a similar value dependency.

4 A Formal Model of a Message Passing System

In a distributed memory implementation of a Haskell specification, values must be communicated explicitly between computations running as separate tasks. This is because there is no common heap underlying the entire program. This section presents a simple model of a message passing system.

A *task* is a distinct unit of work within the system. A task may create, destroy and communicate with other tasks. Let $TASK$ be the set of tasks in the system, and $T_1[\![c]\!]$, where $T_1 \in TASK$ denote that T_1 performs the computation c at some time during its existence. Similarly, $T[\![c]\!]^i$ denotes the ith time that computation c has been performed by T. The system call `myTid` returns an identifier, of type TID, to the calling task. The following law states that we require that this identifier is unique for each task within the system.

Law 6 : Task Equality.

$$\forall T_1, T_2 \in TASK \;.\; T_1[\![\texttt{myTid}]\!] = T_2[\![\texttt{myTid}]\!] \Leftrightarrow T_1 = T_2 \;.$$

It follows from Law 6 that a value of type TID can be used to select a task from $TASK$. Let $TASK[tid]$ select an element from $TASK$ such that —

$$\forall t_1 \in TIDS \;.\; TASK[t_1][\![t_2 \texttt{ <- myTid}]\!] \Leftrightarrow t_1 = t_2 \;.$$

4.1 Task Creation

The operation `task :: a -> IO TID` takes an expression, and creates a task to evaluate it, in parallel to the main program. `task` returns the identifier of the newly created task. If the expression to evaluate is a function, the new task will expect to receive messages, from the parent, containing arguments. After evaluating the expression, the task sends a result back to the parent. There are different ways to define the behaviour of `task`. Does the task compute once and then die, or does it persist and live to compute another day? For simplicity, we choose a 'one shot' task, which will evaluate once, and then terminate. The following lemma states that no computation can be performed by a task until it has been created.

Lemma 7 : Task Existence.

$$\forall T \in TASK, \texttt{x :: a} \;.\; T[\![\texttt{t <- task c}]\!] :\gg:_s TASK[\texttt{t}][\![\texttt{x}]\!] \;.$$

The following law gives the behaviour of the system call `parent`. When called by a child task, it returns the identifier of the task which created it.

Law 8 : Parent and Child.

$$\forall T \in TASK, \texttt{x :: a} \;.\; T[\![\texttt{t <- task x}]\!] \Rightarrow T[\![\texttt{myTid}]\!] = TASK[\texttt{t}][\![\texttt{parent}]\!] \;.$$

4.2 Communication

Message passing is performed using the operations `receive :: TID -> IO a` and `send :: TID -> a -> IO ()`. `send` transfers a message from one task to the message buffer of another. `receive` searches the message buffer for the first message from the specified task. The ordering of messages in a task's buffer is defined in the following law.

Law 9 : Message Ordering. If m' is the ith message sent by T_1 to T_2 and m'' is the ith message received by T_2 from T_1 then $m' = m''$.

If no messages from $TASK[t_1]$ are present in the message buffer, `receive` t_1 blocks until a message becomes available. Thus there is a state dependency between `receive` and `send`, as given in the next lemma.

Lemma 10 : Receive Dependency.

$$\forall t_1, t_2 \in TIDS \;.\; TASK[t_1]\llbracket \texttt{send}\ t_2\ \mathtt{m} \rrbracket^i :\gg:_s TASK[t_2]\llbracket \texttt{receive}\ t_1 \rrbracket^i \;.$$

4.3 Task Introduction

We now present transformations for the introduction of message passing operations into a Haskell program.

$$exp \Longleftrightarrow \texttt{run (do \{t <- task } exp\texttt{; receive t\})} \tag{12}$$

$$f(e_1 \ldots e_n) \Longleftrightarrow \texttt{run (do\{t <- task } f\texttt{; send t (}e_1 \ldots e_n\texttt{); receive t\})} \tag{13}$$

These laws state that an expression or function can be transformed into a task which evaluates separately. A function's arguments must be sent to the new task before it can return a result. The transformation given is for an uncurried function; a curried function can be parallelised in a similar way, or by using a send operation for each argument.

4.4 Dialogues

A *dialogue* is a two-way communication between two tasks, and is comprised of two coupled streams of requests to the system. A process may simultaneously hold dialogues with many different processes. The interleaving of dialogues is constrained only by data dependencies between dialogues.

The order in which one dialogue may be executed is constrained by the data dependencies of the messages passed between the two tasks. Nonetheless, reordering a dialogue may enable further parallelisation. Provided these dependencies are satisfied, the dialogue can be re-ordered in any way.

Theorem 11 : Dialogue Reordering. *Two tasks, T_1, T_2 in a dialogue produce streams S_1,S_2. Let c_1, c_2 be the computations executed by T_1 and T_2 to produce these streams of requests. c_1 and c_2 can be transformed, using $\overset{t}{\Longrightarrow}$, thus reordering the system requests in the dialogue, producing new streams S_1',S_2' The dialogue has the same semantics if, given c_1 is reordered so that the ith request in S_1 is now executed as the jth request in S_1', c_2 is reordered likewise.*

For optimisation it is useful to know the dependencies between message passing operations. The following law only reasons about *state* dependencies; data dependencies must also be considered.

Law 12 : Dependencies between Operations.

$$\texttt{x1,x2:: a . task x1} \;:\|:_S\; \texttt{task x2} \;. \tag{14}$$

$$\forall t_1, t_2 \in TIDS \;.\; t_1 \neq t_2 \Rightarrow$$
$$t_1 \;\texttt{<-task exp}\; :\|:_S\; \{\texttt{receive}\; t_2 \mid \texttt{send}\; t_2\; \texttt{m}\}^4 \;. \tag{15}$$

$$\forall t_1, t_2 \in TIDS \;.\; t_1 \neq t_2 \Rightarrow$$
$$\{\texttt{receive}\; t_1 \mid \texttt{send}\; t_1\; \texttt{m}\} \;:\|:_S\; \{\texttt{receive}\; t_2 \mid \texttt{send}\; t_2\; \texttt{m}\} \;. \tag{16}$$

5 Conclusion

We have presented a methodology for formally deriving a message passing program from a pure Haskell specification. 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 PVM; with this the derived programs have been executed on a group of workstations. A direction for future work is to extend this methodology to other communication models.

References

1. R. Bird and P. Wadler. *Introduction to Functional Programming.* International Series in Computer Science. Prentice-Hall, 1987.
2. Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. An introduction to the MPI standard. Technical Report UT-CS-95-274, Department of Computer Science, University of Tennessee, January 1995. Tue, 1 Apr 97 18:13:17 GMT.
3. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.
4. J. Peterson[editor], K. Hammond[editor], L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, S. Peyton Jones, A. Reid, and P. Wadler. Haskell 1.3, A non-strict, purely functional language. Report YALEU / DCS / RR-1106, Department of Computer Science, Yale University, May 1996.
5. P. Wadler. The essence of functional programming (invited talk). In *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992*, pages 1–14, New York, NY, USA, 1992. ACM Press.

[4] The notation $\{\cdots \mid \cdots\}$ indicates that the law is true for any alternative between the braces.