Sematic of Parallel Primitive Haskell Programming Language

Dr. Mustafa Kh. Aswad
Department of Computer Engineering & IT
Subratha University, Subratha, Libya
Email: mustafaasawd@gmail.com

Abstract

Nowadays, heterogeneous multi-core has become the mean-stream computer architecture. It emerging hundreds of cores and combining accelerators like GPUs on traditional chip. Programming software models need to exploit all the resources at the hand of a programmer with minimum efforts. This paper describes the Semitics of parallel primitives of Haskell Programming Language. Primitives are exploiting the underlying architecture resources. New primitive added to a programming language we need to prove its expected behavior. This paper specified the expected behaviors of the constructs by Haskell functions, achieving an executable specification. We have formulated several properties as Haskell predicates and used ***Quickcheck*** to check them on random input. The three basic properties represent sanity checks of the semantics. Two proposed implementation relevant properties did not hold, and counter examples extracted from ***Quickcheck*** identified diffusion of sparks to be the problem. In the implementation, we avoided this problem by resetting the boundaries after one fishing stage. The final property, checked with ***Quickcheck***, shows that with this modification, the desired property holds.

1. Introduction

The introduction of multi-core processors has renewed interest in parallel functional programming and there are now number of parallel programing models that explore the advantages of a functional language for writing explicitly parallel code or implicitly paralellizing code written in a pure functional language [2]. There are more developments in the areas of softwares such as transactional memory and nested data parallelism [6] [11].

Parallel programs are written to gain performance. This goal is achieved by exploit the potential of a real parallel computing resource like a multi-core processor [8]. Concurrency is a programming techniques that allows us to model computations as hypothetical independent activities. Those computations can communicate and synchronize [10]. This paper describes in Section [2.1] number of parallel programming model include explicit, semi-explicit and implicit parallel programs. It will know that fully implicit parallelism leads to very small tasks which con not hide the over hide communications. Also the explicit parallel program is too difficult for non-experience programmer. This paper propose the semi-explicit parallel programing as good approach for writing parallel programming, and easy to control the granularity of parallel tasks on modern operating systems and processors. Haskell function language is a semi-explicit parallel programming model constrains performance and simplicity. It is particularly fox on formal semantics for the new constructs.

2. Background

This section presents an introduction to number of parallel programming models. It in mainly constrain on the techniques for writing concurrent parallel programs. The parallel program is written gain performance. In other-words it written to reduce the execution time. Therefore, we need to exploit the potential of a real parallel architecture platform like multi-core cores.

2.1. Parallel Applications

The development of parallel programs increases the number of challenges that already exist in developing sequential programs. A parallel programs aims to achieve performance cross different parallel platform without changes and hide the latency between cores and I/O operations to disks and network devices. A parallel program should remain easy to constructs.

As mentioned the new mainstream microprocessors are move towards hundreds of cores or more in one platform. To achieve performance from each individual core, it is necessary to split a given work into tasks and distribute these tasks across multiple processing cores. Splitting a work to number of tasks requires a program language capable to automatically parallelize the sequential code or semi-explicit or explicitly parallel program which is then scheduled onto multiple cores by the operating systems. Haskell function language is semi-explicit parallel programming.

2.2. Haskell Functional Language

Haskell Functional Language represents code in mathematical function sense. One of its features is laziness which means functions don't evaluate their arguments. This feature helps to write parallel code very easy in Haskell functional language.

Exploiting Parallelism in Haskell, in general every expression in most functional language can be evaluated in parallel. This can be automatically exploited. But exploiting all parallelism in a program has side effects. It creates too many small tasks. Which cannot be efficiently scheduled and parallelism is limited by fundamental data dependencies in the source program. Haskell provides a mechanism to allow the user to control the granularity of parallelism by indicating what computations may be usefully carried out in parallel. This kind of parallelism calls Semi-Explicit Parallelism. It provides two primitives par and pseq [12].

```
par :: a -> b -> b
pseq :: a -> b -> b
```

Fig 1: par and pseq Function

The function par indicates to the Glassgow Haskell Compiler (GHC) run-time system that it may be beneficial to evaluate the first argument (a) in parallel with the second argument (b). It returns the second argument as result. The notion of a lazy future provided by Haskell language allow the runtime system to create spark for Frost arguments (a) which has the potential to be executed on a different thread from the parent thread. But not necessarily create a thread to compute the value of the expression (a).

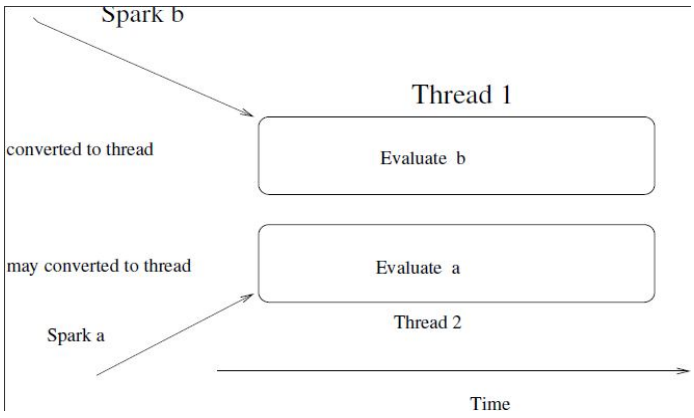

Fig 2: Semi-explicit Parallel Haskell.

Load balance in Haskell is a mechanism used to distribute the work among the participated cores. Haskell uses a work stealing algorithm to distribute a work. Within a multicore it will search for a spark by directly accessing spark pool [4]. A spark is described in the Section [2.2] in a network it sends a fish message searching for work. It is a dynamic mechanism for automatically distributing work and data on a cluster as shown in Figure 3.

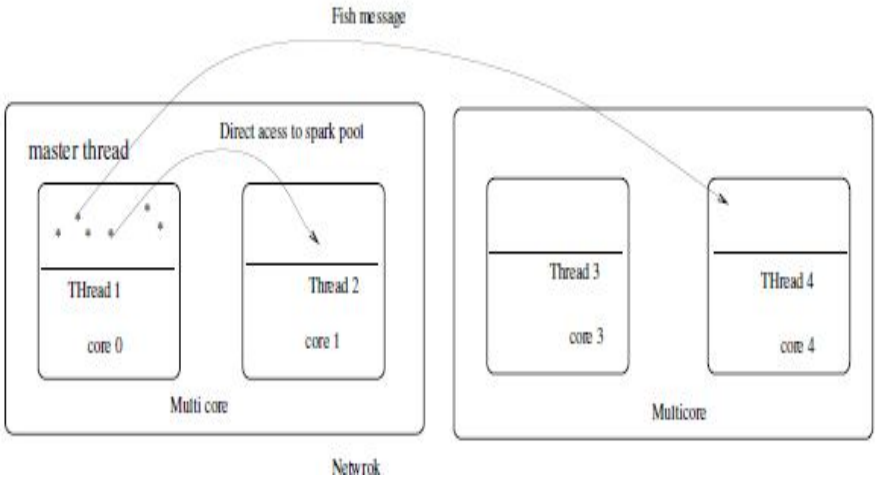

Fig 3: Workstealing Algorithm in Parallel Haskell.

2.3. OpenMP

OpenMP uses an imperative parallel programming style [4]. It is a portable programming interface for shared memory multithreaded programming using C/C++ and FORTRAN as host languages. OpenMP consists of a set of compiler directives, library routines, and environment variables that affect run-time behavior. OpenMP uses a fork-join threading model; a master thread forks a task into a number of worker threads that share the work and then wait until they finish to join before continuing. OpenMP is a scalable model that gives programmers a simple and flexible interface for developing parallel applications for a range of parallel architectures. The model is identified as easy to use and portable. The programmer does not need to put significant effort into parallelizing the existing sequential program. However, this is not always the case, as the multicore resources are not fully utilized if the programmer is not expert in parallel programming.

2.4. Message Passing Interface (MPI)

A standard defines an interface for sending and receiving messages [13]. Specifically the interface includes point-to-point communication functions, ***send*** operations performing a data transfer between two concurrently executing tasks, and ***receive*** operations to accept data from another processor into program memory space. It also has other operations, such as broadcast barriers and reduction that explicitly involve a group of processors. It is heavily used in high performance computing and, with considerable tuning, delivers an acceptable performance across a wide range of architectures. MPI is a MIMD style model. However a shared-memory style can be simulated using send and receive messages of MPI. MPI does not provide the dynamic creation or deletion of processes during a program runtime (the total number of processes is fixed [9].

3. Constructs Semantics

The new constructs are not prescriptive: rather than specifying a single PE for a task, they identify sets of PEs within the communication hierarchy of the architecture. We present a simple Haskell specification of the sets of PEs that each construct identifies when executed on any PE of participating PEs. We first need some mechanism specifying paths and distances in the tree hierarchy.

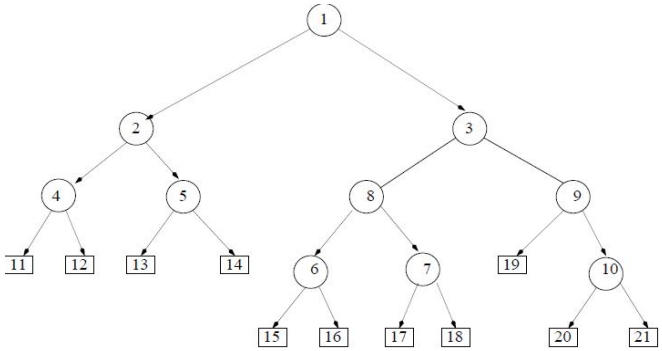

Fig 4: An Example of Hierarchical Architecture.

3.1. Distance Function

In order to illustrate how the ***distance*** function is working, we define a binary tree ***(Tree t)*** structure representing an underlying parallel platform (e.g. the one shown in Figure(4). The definition of the tree data structure is as follows:

```
data Tree a = Node a (Tree a) (Tree a)
            | Leaf {pId ::Int} deriving (Eq, Show)
```

A tree is a leaf with a PE Id as value, or a node represents network possibly parametrized with information such as latency, leaves represents PEs. A node has a value and two branches, each of which is a subtree.

The function ***distance t p1 p2*** calculates the distance, defined as the number of steps to the nearest common node in the hierarchy between two leaves in the architecture hierarchy. The function takes a ***tree t*** representing the architecture and two leaves ***p1*** and ***p2*** as input and returns the distance between the two leaves as an integer.

```
distance ::Tree Int ->Int ->Int ->Int
distance  t  p1  p2  =  d1+d2
       where
          pathTo  p1 = path t  p1
          pathTo  p2 = path t  p2
          comNodes = length (prefixOf   pathTo p1 pathTo p2)
          d1 = length pathTop1 - comNodes
          d2 = length pathTop2 - comNodes
path ::  Tree Int ->Int -> [Int]
path (Leaf p ) s
     | p==s = [s]
     | otherwise = []
path  (Node v t u) s
     | v == s =  [v]
     | left== [] && right  == [] = []
     | left /= [] =   [v]  ++ left
     | right  /= [] = [v] ++ right
     where
       left = path t s
            right = path u s
```

Fig 5: Distance Function

The definition of the distance function uses additional auxiliary functions, path shown in Figure 5. The call path ***(Node v t u) p*** calculates the path to leaf p from the root of the tree represented as a list. It takes a tree ***(Node v t u)*** and leaf p and returns list of nodes that lead to the leaf p. The complete Haskell program defining all auxiliary functions, e.g. for simplicity, we use a tree of integers shown in Figure 4 to demonstrate the constructs semantics. Squares in the tree represent PEs in the hierarchy (Leaf). Circles in the tree represent the networks connecting Pes or sub-networks (Node).

As an example, if we need to compute a distance between Leaf 15 and Leaf 20, we proceed by the following steps.

1) Path to Leaf 15 ➔ pathTop1 = [1,3,8,6,15]

2) Path to Leaf 20 ➔ pathTop2 =[1,3,9,10,20]

3) Length of the longest common prefix of pathTop1 and pathTop2 ➔ comNodes = length [1,3] =2

   Node 3 is the nearest common node between Leaf 15 and Node 20

4) Length of path to Leaf 15 from nearest common Node ➔ d1 = length (pathTop1) - length (comNodes) = 3

5) Length of path to Leaf 20 from nearest common Node =) d2 = length(pathTop2) - length(comNodes) = 3

6) The distance between Leaf 15 and Leaf 20 = d1 + d2 = 6, is the sum of steps moving from one leaf to the nearest common parent and down to other leaf.

```
setparDist :: Tree Int ->Int ->Int ->Int -> [Int]
setparDist t m u p
      | ((m<0) || (u<0)) = []
      | ((m==0) && (u==0))= [p]
      | (u > (length (pp)-1)&& (m==0)) =(rLeaf ( t))
      | ((m==u)||(u > (length (pp)-1)))
                  = exact ( subexact) p
      | m==0 = [p]++setPes
      | otherwise = setPes
     where
      pp = path  t p
commonnu = last (take (length(pp) - u) pp)
commonnm =last ( take (length(pp) - m) pp)
subu=subTree t commonnu
```

```
subexact= subTree t commonnm
complementtree = (complementTreesubu
commonnm)
setPes= filter (/= commonnm) (rLeaf
                    (complementtree))

subTree ::Tree Int ->Int -> ( Tree Int)
subTree (Leaf  p) s =EmptyTree
subTree (Node v t u) s
   | v == s  = (Node v t u)
   | left== EmptyTree&& right == EmptyTree
                        = EmptyTree
   | left /= EmptyTree = left
   | right  /=EmptyTree =right
   where
    left =    subTree t s
    right =   subTree u s

complementTree ::  Tree Int ->Int -> Tree Int
complementTree (Leaf p1) s = (Leaf p1)
complementTree  (Node v l EmptyTree ) s
    | v==s = EmptyTree
    | otherwise =(Node v (complementTree l s)
EmptyTree)
complementTree (Node v t u) s
      | v==s = EmptyTree
      | otherwise  = (Node v
                (complementTree t s)
                              (complementTree u s))
```

Fig 6: setparDist Locations Function

3.2. setparDist Function

The most basic primitive we propose is the ***parDist*** primitive. We therefore start by defining its semantics in terms of the possible locations defined by it. A ***setparDist t m u p*** specifies the set of PEs on which a ***parDist m u*** task may be executed from $\mathbf{PE_p}$ in an architecture ***t***. It takes an architecture tree ***t***, a minimum bound ***m***, maximum bound ***u***, and a leaf ***p***, the current location, as input and returns a list of all possible PEs (Figur [6]). For example, if we need to generate sparks intended to be executed between levels 1 and 3 from Leaf 20 of the tree shown in Figure (4), we perform the following:

1) Calculate path to Leaf 20 =) pathTop = [1,3,9,10,20].
2) Calculate the common node distant by u levels from Leaf 20 ➔common u = last (take (5 - 3) [1,3,9,10,20]) ➔ 3.
3) Calculate the common node that is m levels from Leaf 20 ➔common m= last (take (5 - 1) [1,3,9,10,20]) ➔ 10.
4) Calculate the subtree of the common u leaf 3 ➔ subtree u = (Node 3 (Node 8 (Node 6 (Leaf pId = 15)(Leaf pId = 16)) (Node 7 (Leaf pId = 17) (Leaf pId = 18))) (Node 9 (Leaf pId = 19) (Node 10 (Leaf pId = 20)(Leaf pId = 21))))
5) Calculate the complementary tree of the common node 10. The complementary tree is the original tree excluding the subtree of a given node. In our case, we calculate the complement for ***subtree u*** of node 10. ➔complementtree = Node 3 (Node 8 (Node 6 (Leaf pId = 15) (Leaf pId = 16)) (Node 7 (Leaf pId = 17) (Leaf pId = 18))) (Node 9 (Leaf pId = 19))
6) Finally calculate the leaves of the complementtree subtree ➔setPes = [15,16,17,18,19].

```
setparBound ::Tree a ->Int ->  a -> [a]
setparBound t n p = setparDist t 0 n p
```

Fig 7: setparBound Locations Function

3.3. setparBound Function

As mentioned in the previous section, ***parDist*** is the most basic primitive. We can use it to define the other constructs. A ***setparBound t n p*** (Figure 7) specifies the set of PEs that tasks generated by a ***parBound n*** may be executed on, from P Ep in architecture t. For example if we need to generate sparks bounded by two levels from leaf 11 of the tree shown in Figure 4, we just call ***setparDist*** with the following parameters ***t 0 2 11***, where t is the tree. The result is [11, 12, 13, 14], a list of leaves with a distance of at most 2 in the architecture tree (t)

3.4.setparAtLeast Function

A ***parAtLeast*** is similar to ***parBound***, as it takes an additional integer parameter specifying the minimum distance in the communication hierarchy that the computation may be communicated. Therefore, it can be defined in a similar way, as shown in Figure 8.

```
setparAtLeast::(Ord a, Show a)=>Tree a->Int ->  a -> [a]
setparAtLeast t  n  p = setdFun  t  n maxLevel  p
        where
            maxLevel = 3
```

Fig 8: setparAtLeast Locations Function

So, if we need to generate sparks intended to be executed at least two levels from leaf 11 of the tree shown in Figure 4, we just call ***setparDist*** t 2 ***maxLevel*** 1 1, where t is the tree and ***maxLevel*** is the

maximum distance that sparks can be sent within the architecture hierarchy. In this example, the result is [15, 16, 17, 18, 19, 20, 21].

3.5. Construct Properties Test

This section presents implementation-relevant properties that the architecture-aware semantics should satisfy. These properties are expressed as Boolean functions in Haskell and validated using ***QuickCheck*** [5] that is the properties are written as Haskell functions and can be automatically checked on either random input or with custom test data generators. Two types of properties are tested: the basic properties and specialized properties.

1) Basic Properties: Let P be the set of PEs which are the leaves of the tree representing a given hierarchical architecture. The domain H is the domain of all possible tree hierarchies. The domain P is the domain of all possible processor elements.
   a. Basic property one. For any processing element $p \in P$; the only possible placement of a bounded spark withupper and lower bounds of 0 and 0 is the p itself. Formally, this is written as:

$$\forall h \in H; \forall p \in P \text{setparDist h 0 0 p} = \{\text{p}\}$$

   b. Basic property two. Let path p be a function that returns the longest path to the PE from the root of the tree hierarchy. Let rLeaf h be a function that returns all processor elements (PEs) in the tree hierarchy. The path and rLeaf functions are described in Section [3.1]. For $p \in P$ and$h \in H$, the set of PEs returned by calling the setparDist h 0 (length (path h p)) p function is equal to the set of all PEs in the hierarchy, as returned rLeaf h.

$$\forall h \in H ; \forall p \in P \text{setparDist h 0 (length (path h p)) p = rLeaf h}$$

c. Basic property three. If the upper bound u is less than 0 or lower bound m is greater than the longest path to the PE from root of the tree hierarchy then the set of PEs returned by calling ***setparDist h m u p*** function is an empty set of PEs.

$$\text{setparDist h m u p} = \{\} \qquad \forall \text{ h Z}$$

The above three basic properties can be considered sanity checks of the semantics. All basic properties have passed Quickcheck testing using one hundred randomly generated inputs, each with a randomly generated tree hierarchy.

2) Specialised Properties: The proposed architecture-aware model exposes the tree hierarchy to the programmer through the ***parDist*** primitive. The ***parDist*** primitive provides a mechanism to spark tasks that can be executed in certain levels of the architecture hierarchy. We believe, for the implementation, it is important that these sparks do not leave their neighborhood, where neighborhood is the set of PEs specified by ***parDist*** primitive. Otherwise the bounded spark may diffuse to arbitrary locations after several steps of fishing (work stealing, as outlined in Section [2.2]. We define the following property to formally specify and check this property.

   a. **Specialised proposed property one**. This property reflects the initial intention of the bounded ***parDist***. For $p \in P$ and $h \in H$, the set of PEs returned by ***setparDist h 0 u p*** is equal to the set of PEs returned by ***setparDist h 0 u p'***, where p' is a possible location after one step of fishing. The aim is to guarantee that if the spark is fished again from p' it will be executed in the same neighborhood specified by the original p.

$\forall h \in H;\ \forall p;\ p' \in P;\ u\ 2\ Z\ ;\ p' \in$ (setparDist h 0 u p))

setparDist h 0 u p' = setparDist h 0 u p

On closer examination, this property fails under the ***quickcheck*** test. In the case of an unbalanced tree hierarchy, the result of ***setparDist h 0 u p*** may return a subset of the set returned by ***setparDist h 0 u p***.

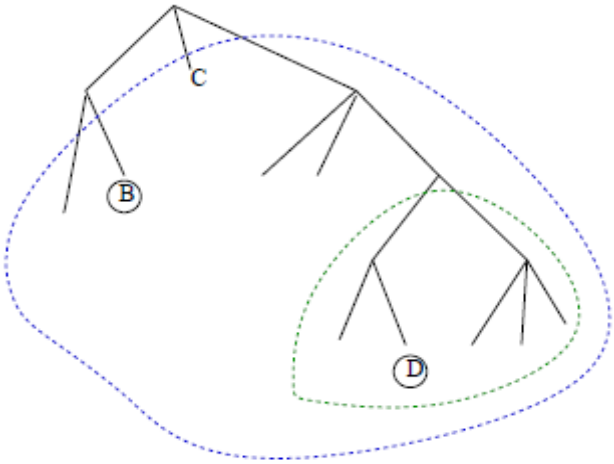

Fig 9: Tree Example of Specialized Proposed Property One

For example, in the tree hierarchy shown in Figure 9, if PE (B) launches a spark with boundaries 0 and 2, then any PE in the outer circle can fish the spark. In particular, it can be fished by PE (D). In second step the spark can be fished only from PEs in the inner circle. That is why the property fails. However, this is not always true, illustrated in the next proposed property.

b. Specialized proposed property two. For $p \in P$ and$h \in H$, the set of PEs returned by setparDist h 0 u p' is a subset of the set of PEs returned by setparDist h 0 u p.

$$\forall\, h \in H \;, \forall\, p\,, p' \in P\;, u \in Z \in p' \in$$
$$(\, setparDist\; h\; 0\; u\; p)\;) \;\; =>$$
$$setparDist\; h\; 0\; u\; p' \;\leq\; setparDist\; h\; 0\; u\; p$$

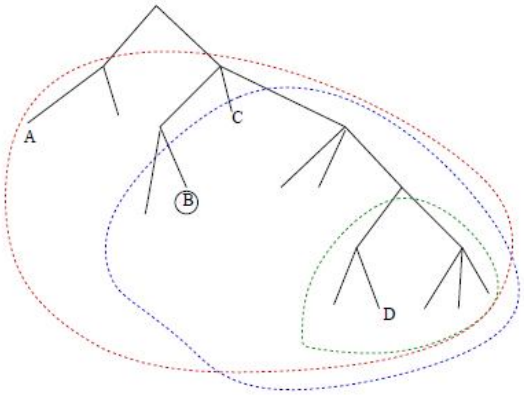

Fig 10: Tree Example of Specialized Proposed Property Two.

The property also fails the ***quickCheck*** test, because of unbalanced tree hierarchies. In the second step, the spark may be fished by a PE which is not one of the elements of the original PE neighborhood. For example, in the tree hierarchy showed in Figure 10, if PE (B) launches a spark with boundaries 0 and 2, then any PE in the blue circle can fish the spark. In particular, it can be fished by PE(C). In a second step, the spark can be fished from PE (C) by any PE in the red circle. In particular, it can be fished by PE (A), which is outside the original neighborhood. We call this behavior of the fishing mechanism diffusion of sparks. In the implementation, we must prevent this scenario from happening. We achieve this by resetting the boundaries of the spark to be 0 and 0, after the first fishing stage. This forces evaluation of the spark on the initial target PE, and thus within the neighborhood specified by the spark.

c. Specialized proposed property two. For p 2 P and h 2 H, after fishing a spark from p to p', which is within the bound u and after resetting the bound u for the spark to 1, this spark can only be fished by a p inside the original neighborhood of p. The set of PEs returned by ***setparDist h 0 1 p'*** is a subset of the set of PEs returned by ***setparDist h 0 u p.***

$$\forall h \in H; \forall p;\ p' \in P;\ u \in Z;\ p' \in$$

$$(\ setparDist\ h\ 0\ u\ p)\ )$$
$$setparDist\ h\ 0\ 1\ p' setparDist\ h\ 0\ u\ p$$

This property passes the ***quickCheck*** and guarantees that there is no diffusion of sparks, i.e. sparks always remain in the neighborhood specified by the original **parDist**.

4. Related Work

Many of the semantics primitives prove described in this section are similar to the semantics prove presented in this paper. Berry and el.[3]defined a transitional semantics of a simple language to preserve the expected behavior of sequential programs. Mosses and el. [7] has investigated the sequential behavior of ML action semantics and extended the language with concurrency primitives.

5. Conclusion

We have presented the semantics for the architecture aware constructs, specifying the set of possible locations when providing boundaries to the sparks. We have specified the expected behavior of the constructs by Haskell functions, achieving an executable specification. We have formulated several properties as Haskell predicates and used ***Quickcheck*** to check them on random input. The three basic properties represent sanity checks of the semantics. Two proposed implementation relevant properties did not hold, and counterexamples

extracted from ***Quickcheck*** identified ***diffusion of sparks*** to be the problem. In the implementation, we avoided this problem by resetting the boundaries after one fishing stage. The final property, checked with ***Quickcheck***, shows that with this modification, the desired property holds.

References

1. ARCHIBALD, B., MAIER, P., STEWART, R., TRINDER, P., ANDDE BEULE, J. Towards generic scalable parallel combinatorial search .In Proceedings of the International Workshop on Parallel Symbolic Computation (New York, NY, USA, 2017), PASCO 2017, ACM, pp. 6:1–6:10.
2. BARROSO, L. A., GHARACHORLOO, K., MCNAMARA, R., NOWATZYK, A., QADEER, S., SANO, B., SMITH, S., STETS, R., AND VERGHESE, B. Piranha: A scalable architecture based on single-chip multiprocessing. SIGARCH Comput. Archit. News 28, 2(May 2000), 282–293.
3. BERRY, D., MILNER, R., AND TURNER, D. A Semantics for ML Concurrency Primitives. 2 1992, pp. 119–129.
4. CHAPMAN, B., JOST, G., AND RUUD, V. Using OpenMP. Portable Shared Memory Parallel Programming. No. ISBN-13: 978-0-262-53302-7. The MIT Press Cambridge, Massachusetts, London, England,2008.
5. CLAESSEN, K., AND HUGHES, J. Quick Check: A Lightweight Tool for Random Testing of Haskell Programs. In Acmsigplan notices (2000), vol. 35, ACM, pp. 268–279.
6. DAMRON, P., FEDOROVA, A., LEV, Y., LUCHANGCO, V., MOIR, M., AND NUSSBAUM, D. Hybrid transactional memory. SIGPLAN Not. 41,11 (Oct. 2006), 336–346.
7. MOSSES, P., AND MUSICANTE, M. An action semantics for ml concurrency primitives. BRICS Report Series 1, 20 (1994).

8. SKILLICORN, D. B., AND TALIA, D. Models and languages for parallel computation. ACM Comput. Surv. 30, 2 (June 1998), 123–169.
9. SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D., AND DONGARR,J. MPI: The Complete Reference , vol. 1. The MIT Press, 1998.MIT, Cambridge.
10. SUTTER, H., AND LARUS, J. Software and the concurrency revolution. Queue 3, 7 (Sept. 2005), 54–62.
11. TARDITI, D., PURI, S., AND OGLESBY, J. Accelerator: Using data parallelism to program gpus for general-purpose uses. SIGOPS Oper. Syst. Rev. 40, 5 (Oct. 2006), 325–335.
12. TRINDER, P. W., HAMMOND, K., MATTSON, JR., J. S., PARTRIDGE,A. S., AND PEYTON JONES, S. L. Gum: A portable parallel implementation of haskell. SIGPLAN Not. 31, 5 (May 1996), 79–88.
13. WALKER, D., AND DONGARRA, J. MPI: a Standard Message Passing Interface. Supercomputer 12 (1996), pp. 56–68. ASFRA BV.