Parallel execution of functional programs on loosely coupled multiprocessor systems

Tetsurou Tanaka * Masato Takeichi *

Abstract

It has been suggested that functional programs are suitable for programming parallel computers owing to their inherent parallelism. We propose a parallel evaluation model of functional programs based on the STG(Spineless Tagless G-machine) model proposed for sequential evaluation, and describe our parallel implementation of a functional language Gofer on the AP1000 parallel computer.

1 Introduction

It has been suggested that functional programs are suitable for programming parallel computers. One of such conviction comes from the fact that functional programs have a great degree of parallelism owing to freeness in the evaluation order of their component expressions. However, excessive parallelism may cause useless computation and execution overhead, and some mechanism for controlling parallelism is required for efficient parallel evaluation.

In eager functional languages, every argument of a function should be evaluated in any case, and hence it may be evaluated in parallel with no worrying about useless computation. This is not the case, however, for lazy functional languages. Since arguments should not be evaluated until their values are required, we have to use strictness information or to specify the arguments being executed speculatively in order to avoid useless computation.

For this purpose, we introduce a primitive function `spec` which is to be used as

```
spec f x = f x
```

for expressing parallel evaluation of `f x` and `x`. Such an extension to the language does not change the syntax nor the compiler itself. Nevertheless it is so flexible to express many kinds of parallel algorithms in a concise way[4, 5].

*University of Tokyo {tanaka,takeichi}@ipl.t.u-tokyo.ac.jp

We have introduced the function `spec` into Gofer[2], and proposed a new parallel execution model of functional programs based on the STG(Spineless Tagless G-machine) model. We will explain the basic idea of our parallel STG model, and give some experimental results obtained so far with our implementation of Gofer on the AP1000 parallel computer.

2 STG

The STG model is a sequential evaluation model of lazy functional languages [3]. The STG code is designed so that it is easily translated into sequential procedural languages such as C, and the translated code runs fast on stock computers.

2.1 Structure of closures

To archive lazy evaluation, `f x y` in

```
g1 x y = h (f x y)
g2 f x y = h z z
  where z = f x y
```

need to be represented in a structure with which the value can be computed when needed. Although most execution models make distinction between values in a form of WHNF(Weak Head Normal Form) and thunks for representing unreduced expressions, STG does not distinguish them. The STG model unifies these two into a common structure called closure(Fig. 1).

When a closure is evaluated, the address of the closure is assigned to a global variable `Node`, and then the control is transferred indirectly to the 0-th entry of the Info-table(Information table).

Since STG represents function applications as a flat structure(Spineless), function definitions are easily accessed. And since STG does not distinguish values and thunks, tags for representing object types are no more necessary(Tagless).

2.2 Stacks

STG uses two stacks: A-stack for passing arguments and returning results, and B-stack for controlling and updating.

In functional languages, function application is often reduced to another function application. If it is translated into a tail transfer C function, the translated program will use many C stack words. In the proposed implementation of the STG model, function application is compiled into C code which transfers control to another C function instead of making a call the C function. This technique reduces the depth of the C stack considerably.

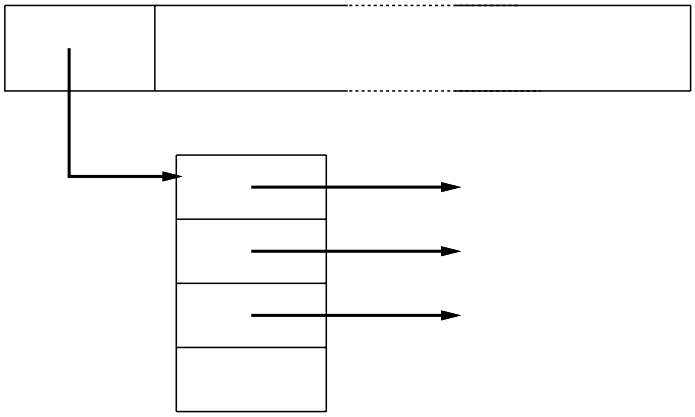

Figure 1: Structure of closures

Function call in more general contexts than tail recursion can also be compiled into a jump which transfers control to either of the continuation addresses on the stack. The modification saves stack usage, too. In the STG model, this idea is extended to putting vectors of continuation addresses which correspond to the constructors of the data type of function results.

For example, since the type of `e1` in `if e1 then e2 else e3` is Boolean consisting of two constructors, `False` and `True`, a vector consisting of two addresses for corresponding continuations `e3` and `e2` is pushed onto the B-stack before `e1` is evaluated. Note that such a vector is can be made at compile time. After evaluating `e1`, control comes to the code for `False` or `True`, which pops up the vector from the B-stack and causes a jump to the corresponding continuation address.

With this technique, `if` expression and `case` expression can be efficiently implemented. The result values of `Int` and `Float` are returned through global variables `RetInt` and `RetFloat` respectively, and values of other types are returned through the A-stack.

The B-stack is used for updating closures as well. Closures can be referred by more than one closures, but should not be evaluated more than once. Every closure is evaluated after its address are pushed onto the B-Stack. When evaluated form drops to a WHNF, the closure pushed on the B-stack is updated to become as indirection node which points to that WHNF. The STG model also takes advantage of the vectored return scheme for updating.

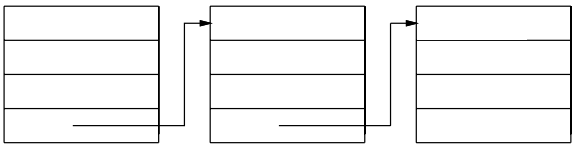

Figure 2: The structure of Goals

3 Extension for parallel execution

Since parallelism introduced by the primitive function `spec` is dynamic, it is impossible to distribute closures to processors at compile time. In addition to this, since a program may include parallelism requiring more processors than equipped, each processor should deal with as many contexts as required.

Every context is represented by a stack region. Since allocation of stack regions for as many contexts as necessary will cause memory exhaustion, we represent a context as a closure called *goal*. A goal holds the contents of the stacks and is linked to the active goal queue when created.

3.1 Goals

When `spec f x` is evaluated, a goal for evaluating `x` is created and exported to another processor. And each imported goal is linked to the active goal queue of that processor. The goal has the form represented as Fig. 2.

Each goal is evaluated without preemption until it suspends or completes. Upon entering a closure, each processor checks the message buffer whether there exist messages and processes them, if any.

When evaluation of a goal completes, the closure pointed by the goal should be updated. In principle, the caller of the closure pushes entries for update corresponding to the type. But the type of the closure called by a goal cannot be known by the processor evaluating that closure. Hence updating at the toplevel should be done in such a way that the closure is updated by a closure consisting of `RetInt`, `RetFloat` and the contents of the A-stack, which will be used when this closure is referred by the caller.

When the value of such a closure is requested, it is recorded in the update frame on the B-stack, and `RetInt`, `RetFloat` and the contents of the A-stack are restored before returning with the corresponding index of the vector as usual.

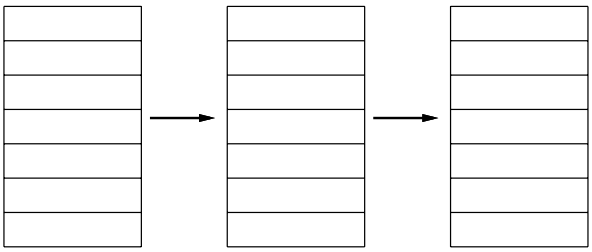

Figure 3: State transition of import table

3.2 Migration

On loosely coupled multiprocessors, goals and values need to be transferred via message passing. When goals and values are moved, closures are used as units. Export and Import tables are used in order to evaluate a closure only once by either one of the processors.

- WHNF
  When closures representing WHNF are exported to other processors, their copies are made. Closures pointed by such a closure are copied recursively to some depth.

- others
  Other closures are registered in the export table, and their addresses of the export table are passed with the processor number. The receiver allocates an import table closure for each closure passed from other processors and records the address of the export table of the sender with the processor number of the sender.

When a goal is exported, the closure itself needs to be exported for evaluation. The sender of the goal allocates a closure of the import table, and sends its address and the contents of the closure. Along with this, the original closure is overwritten by an indirection reference to the closure. The receiver of the goal allocates a closure of the export table and creates a goal for evaluating the closure and returning the result to the closure of the import table of the sender.

Each element of the import table has one of three states, which are distinguished by the info(Fig. 3)

1. ReadVar
   Each closure in ReadVar state contains a processor number of the partner and the address of the export table, which holds the closure having been imported. When entering it, `read_var` message is sent to the partner, and

the current goal is hooked to the suspend list. Then the info of the closure is rewritten to WaitVar.

2. WaitVar
   This state corresponds to either of two cases: the ReadVar closure of the import table has been entered, or the closure entry of the import table is waiting the result to be returned from the goal having been exported. When entering an closure in this state, the current goal becomes suspended.

3. GroundVar
   When the result of evaluation is sent to a WaitVar closure of the import table, the closure entry of the import table is changed to this state. A node in this state works same as a indirect node, and it is reclaimed at garbage collection time.

Different types of closure require different actions of importing and exporting. These actions are achieved by adding corresponding entry in the Info-tables.

3.3 Suspension and Activation

When the current goal enters an import table entry which cannot be evaluated immediately, such as ReadVar and WaitVar, it makes a goal that holds the current context and hooks it to the suspend list of the import table, and suspends by itself.

When execution of a goal is completed, goals and export tables which are hooked on the goal are activated. The goals are linked to the active goal queue, and export tables are rewritten to GroundExport and messages. Messages are sent to the closures of the import tables waiting that value. When a message is sent to an import table in the WaitVar state, goals which have been hooked on the table are activated.

4 Evaluation

We modified the source code of the Gofer system by Mark P. Jones[2] to make the internal representation of Gofer programs into our extended STG code.

At first the processor 0 evaluates a function named `main` which is of the Dialogue type, and goals are forked with `spec` migrate to others processors and all goals are executed in parallel. The host program only receives and processes the requests of I/O from cells.

To examine the performance of the system, we used the naive Fibonacci program and the parallel version of it.

```
fib_spec 0 = 1
```

Table 1: Results(fib 29, 64 cells)

program	execution time (seconds)	goals sum/max/min	suspensions
fib_spec	17.7	1330318/37651/36639	510396
fib_hy(2)	7.6	431445/9040/8169	120067
fib_hy(4)	6.0	162753/3554/3005	44720
fib_hy(6)	4.0	60813/1307/1037	16130
fib_hy(8)	3.5	22980/599/303	5974
fib_hy(10)	3.7	8805/244/127	2306
fib_hy(12)	4.1	3315/120/31	844
fib	75.3	1/1/0	0
fib(SS2)	47.6		

```
fib_spec 1 = 1
fib_spec (n+2) = spec ((+) (fib_spec n))
                      (fib_spec (n+1))
fib_hy 0 = 1
fib_hy 1 = 1
fib_hy (n+2) = if (n>10) then
        spec ((+) (fib_hy (n+1)))
             (fib_hy n)
      else
fib_hy (n+1) + fib_hy n
```

We examined a program that is ececuted in sequential fashion for small `n` to make the granularity variable. In the program `fib_hy`, the threshold is 10. We made experiments for various values of `n`. The result AP1000(64 cells) is shown in Fig. 1.

Another example program is one for solving the 8-queens problem. The result is shown in Fig 2

5 Conclusion

We have augmented the STG with several primitives for parallel evaluation, and have implemented it on a loosely coupled multi-processor system. Current implementation is in a preliminary stage for developing efficient systems for functional languages, and there are many problems to be done. It is convincing, however, that parallel implementation of functional languages based on our proposal is a well-engineered technique as our implementation illustrates. The performance could be considered to be sufficient for real applications.

Table 2: Results(queens 8, 64 cells)

program	execution time (seconds)	goals sum/max/min	suspensions
foldl_spec	4.0	333508/8828/5995	134803
foldl	36.8	1/1/0	0
(SS2)	14.0		

References

[1] Bird, R. and Wadler,P.: Introduction to Functional Programming, Prentice-Hall, 1988.

[2] Jones, M. P.: An Introduction to Gofer, Universityof Oxford, 1991.

[3] Jones, SL, P.: Implementinglazy functional languages on stock hardware: the Spineless TaglessG-machine, Journal of Functional Programming 1992.

[4] Tanaka, K., Iwasaki, H. and Takeichi, M.: Abstract description of algorithms and parallel execution of programs, Japan Society for Software Science and Technology 9th Conference Proceedings Japan, pp. 1-4, 1992.

[5] Tanaka, T., Iwasaki, H. and Takeichi, M.: Parallel execution of functional programs by committed choice, Japan Society for Software Science and Technology 9th Conference Proceedings Japan, pp. 85-88, 1992.