THESIS TITLE

by

FName LName

Submitted in Fulfilment of the Requirements for the Degree of Doctor of Philosophy

University of Glasgow

College of Science and Engineering
School of Computing Science

September 2014

Abstract

Write abstract here..

Contents

List of Tables

List of Figures

Dedication

To my Mother and Father.

Acknowledgements

Write here...

Declaration

I declare that, except where explicit reference is made to the contribution of others, that this dissertation is the result of my own work and has not been submitted for any other degree at the University of Glasgow or any other institution.

Write your name here

Chapter 1

Introduction

1.1 Context

How to use Macros. The Glasgow Haskell Compiler (GHC) is write.

How to cite. According to Akhter and Roberts [1] write. According to Shende et al. [2] write.

How to use glossaries. The number of Processing Elements (PEs) write. The Haskell on a Shared Memory Multiprocessor (GHC-SMP) is write.

1.2 Thesis Statement

1.3 Contributions

1. **Write.**. detail here.

2. **Write.**. detail here.

1.4 Authorship and Publications

1.5 Thesis Structure

The structure of this thesis is as follows:

Chapter 2 gives....

Chapter 2

Background

2.1 Introduction

2.2 Parallel Architectures

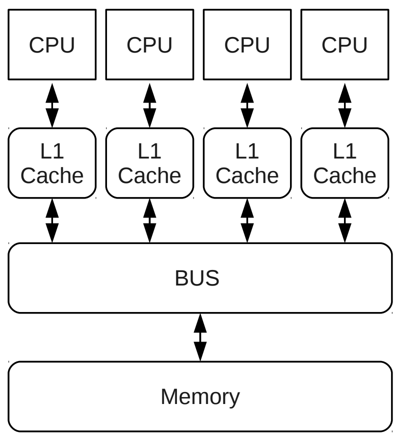

Figure 2.1: Shared Memory SMP Architecture

2.3 Summary

Chapter 3

Conclusion and Future Work

3.1 Summary

3.2 Limitations

3.3 Future Work

Appendix A

Benchmarks

Listing A.1: Fibonacci Benchmark Implementation.

```haskell

-- / sequential Fibonacci

fib :: Int -> Integer
fib n | n <= 1    = 1
      | otherwise = fib (n-1) + fib (n-2)


-- / parallel Fibonacci; shared memory

par_fib :: Int -> Int -> Par Integer
par_fib seqThreshold n
  | n <= k    = force $ fib n
  | otherwise = do v <- new
                   let job = par_fib seqThreshold (n - 1) >>=
                             force >>=
                             put v
                   fork job
                   y <- par_fib seqThreshold (n - 2)
                   x <- get v
                   force $ x + y
  where k = max 1 seqThreshold

-- / parallel Fibonacci; distributed memory

dist_fib :: Int -> Int -> Int -> Par Integer
dist_fib seqThreshold parThreshold n
  | n <= k    = force $ fib n
  | n <= l    = par_fib seqThreshold n
  | otherwise = do
      v <- new
      gv <- glob v
      spark $(mkClosure [| dist_fib_abs (seqThreshold, parThreshold, n, gv) |])
      y <- dist_fib seqThreshold parThreshold (n - 2)
      clo_x <- get v
      force $ unClosure clo_x + y
  where k = max 1 seqThreshold
        l = parThreshold

dist_fib_abs :: (Int, Int, Int, GIVar (Closure Integer)) -> Thunk (Par ())
dist_fib_abs (seqThreshold, parThreshold, n, gv) =
  Thunk $ dist_fib seqThreshold parThreshold (n - 1) >>=
          force >>=
          rput gv . toClosure

-- / parallel Fibonacci; distributed memory; using sparking d-n-c skeleton

spark_skel_fib :: Int -> Int -> Par Integer
spark_skel_fib seqThreshold n = unClosure <$> skel (toClosure n)
  where
    skel = parDivideAndConquer
             $(mkClosure [| dnc_trivial_abs (seqThreshold) |])
```

```
                $(mkClosure [| dnc_decompose |])
                $(mkClosure [| dnc_combine |])
                $(mkClosure [| dnc_f |])

dnc_trivial_abs :: Int -> Thunk (Closure Int -> Bool)
dnc_trivial_abs (seqThreshold) =
  Thunk $ \ clo_n -> unClosure clo_n <= max 1 seqThreshold

dnc_decompose =
  \ clo_n -> let n = unClosure clo_n in [toClosure (n-1), toClosure (n-2)]

dnc_combine =
  \ _ clos -> toClosure $ sum $ map unClosure clos

dnc_f =
  \ clo_n -> toClosure <$> (force $ fib $ unClosure clo_n)

-- / parallel Fibonacci; distributed memory; using pushing d-n-c skeleton

push_skel_fib :: [Node] -> Int -> Int -> Par Integer
push_skel_fib nodes seqThreshold n = unClosure <$> skel (toClosure n)
  where
    skel = pushDivideAndConquer
                nodes
                $(mkClosure [| dnc_trivial_abs (seqThreshold) |])
                $(mkClosure [| dnc_decompose |])
                $(mkClosure [| dnc_combine |])
                $(mkClosure [| dnc_f |])
```

A.1 Sum Euler

Listing A.2: Sum Euler Benchmark Implementation.

```
-- / Euler's totient function (for positive integers)

totient :: Int -> Integer
totient n = toInteger $ length $ filter (\ k -> gcd n k == 1) [1 .. n]

-- / sequential sum of totients

sum_totient :: [Int] -> Integer
sum_totient = sum . map totient

-- / parallel sum of totients; shared memory

par_sum_totient_chunked :: Int -> Int -> Int -> Par Integer
par_sum_totient_chunked lower upper chunksize =
  sum <$> (mapM get =<< (mapM fork_sum_euler $ chunked_list))
    where
      chunked_list = chunk chunksize [upper, upper - 1 .. lower] :: [[Int]]


par_sum_totient_sliced :: Int -> Int -> Int -> Par Integer
par_sum_totient_sliced lower upper slices =
  sum <$> (mapM get =<< (mapM fork_sum_euler $ sliced_list))
    where
      sliced_list = slice slices [upper, upper - 1 .. lower] :: [[Int]]


fork_sum_euler :: [Int] -> Par (IVar Integer)
fork_sum_euler xs = do v <- new
                       fork $ force (sum_totient xs) >>= put v
                       return v

-- / parallel sum of totients; distributed memory

dist_sum_totient_chunked :: Int -> Int -> Int -> Par Integer
dist_sum_totient_chunked lower upper chunksize = do
  sum <$> (mapM get_and_unClosure =<< (mapM spark_sum_euler $ chunked_list))
    where
```

```
      chunked_list = chunk chunksize [upper, upper - 1 .. lower] :: [[Int]]


dist_sum_totient_sliced :: Int -> Int -> Int -> Par Integer
dist_sum_totient_sliced lower upper slices = do
  sum <$> (mapM get_and_unClosure =<< (mapM spark_sum_euler $ sliced_list))
    where
      sliced_list = slice slices [upper, upper - 1 .. lower] :: [[Int]]


spark_sum_euler :: [Int] -> Par (IVar (Closure Integer))
spark_sum_euler xs = do
  v <- new
  gv <- glob v
  spark $(mkClosure [| spark_sum_euler_abs (xs, gv) |])
  return v

spark_sum_euler_abs :: ([Int], GIVar (Closure Integer)) -> Thunk (Par ())
spark_sum_euler_abs (xs, gv) =
  Thunk $ force (sum_totient xs) >>= rput gv . toClosure

get_and_unClosure :: IVar (Closure a) -> Par a
get_and_unClosure = return . unClosure <=< get

-- / parallel sum of totients; distributed memory (using plain task farm)

farm_sum_totient_chunked :: Int -> Int -> Int -> Par Integer
farm_sum_totient_chunked lower upper chunksize =
  sum <$> parMapNF $(mkClosure [| sum_totient |]) chunked_list
    where
      chunked_list = chunk chunksize [upper, upper - 1 .. lower] :: [[Int]]


farm_sum_totient_sliced :: Int -> Int -> Int -> Par Integer
farm_sum_totient_sliced lower upper slices =
  sum <$> parMapNF $(mkClosure [| sum_totient |]) sliced_list
    where
      sliced_list = slice slices [upper, upper - 1 .. lower] :: [[Int]]

-- / parallel sum of totients; distributed memory (chunking/slicing task farms)

chunkfarm_sum_totient :: Int -> Int -> Int -> Par Integer
chunkfarm_sum_totient lower upper chunksize =
  sum <$> parMapChunkedNF chunksize $(mkClosure [| totient |]) list
    where
      list = [upper, upper - 1 .. lower] :: [Int]


slicefarm_sum_totient :: Int -> Int -> Int -> Par Integer
slicefarm_sum_totient lower upper slices =
  sum <$> parMapSlicedNF slices $(mkClosure [| totient |]) list
    where
      list = [upper, upper - 1 .. lower] :: [Int]
```

Glossary

GHC-SMP Haskell on a Shared Memory Multiprocessor.

PE Processing Element.

Bibliography

[1] S. Akhter and J. Roberts. *Multi-Core Programming: Increasing Performance Through Software Multi-threading.* Richard Bowles, 2006.

[2] S. Shende, A. D. Malony, J. Cuny, P. Beckman, S. Karmesin, and K. Lindlan. Portable Profiling and Tracing for Parallel, Scientific Applications Using C++. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '98, pages 134–145, New York, NY, USA, 1998. ACM.