List of Table 3
List of figure 4
1. Introduction 5
1.1 Overview 5
1.2 Contributions 5
1.3 Dissertation Outline 6
2 Backgrounds 8
2.1 Parallel Programming 8
2.1.1 Parallel Computer 9
2.1.3 Parallelism 10
2.1.4 Design process of parallel programming 11
2.2 Parallel Programming Paradigm 14
2.2.1 Data Parallelism 14
2.2.2 Control Parallelism 15
2.3 Skeleton Parallelism 16
2.4 Programming Languages 17
2.4.1 Parallel Programming Languages 17
2.4.2 Functional Programming 18
2.4.2 Haskell 21
2.5 Parallel and Distributed Functional Programming (GdH and GpH) 21
2.5.1 Glasgow Parallel Haskell (GpH) 21
2.5.2 Glasgow distributed Haskell (GdH) 22
2.5.3 Relationship among GpH, GdH and GHC 23
3 Designing and Implementing Skeletons 25
3.1 ParMap 25
3.2 Divide and Conquer 31
4. Evaluating Skeletons 36
4.1 ParMap 36
4.1.1 Regular Data 37
4.1.2 Irregular Data 40
4.1.3 Discussion 42
4.2 Divide and Conquer 46
5 Conclusion 50
5.1 Summary 50
5.2 Future work 51
Reference 52
Appendix 55
A Map Skeletons 55
A.1: GdHParMap (Regular Data) 55
A.2: GdHFarmMap (Regular Data) 59
A.3: GpHParMap (Regular Data) 62
A.4: GpHChunkMap (Regular Data) 63
A.5: GpHShuMap (Regular Data) 64
A.6: GdHWorkPool (Regular Data) 66

A.7: GdHParMap (Irregular Data) 71
A.8: GdHFarmMap (Irregular Data) 75
A.9: GpHParMap (Irregular Data) 79
A.10: GpHChunkMap (Irregular Data) 80
A.11: GpHShuMap (Irregular Data) 81
A.12: GdHWorkPool (Irregular Data) 84
B Divide and Conquer Skeletons 88
B.1 GpHDC 88
B.2 GdHDC 90

List of Table

Table 4-1-1 Regular Data Measurement……………………………………37

Table 4-1-2 Regular Data Speedup…………………………………………39

Table 4-1-3 Irregular Data Measurement……………………………………40

Table 4-1-4 Irregular Data Speedup…………………………………….……41

Table 4-1-5 GdHParMap Speedup……………………………………………44

Table 4-2-1 Divide and Conquer Measurement…………………...…………..46

Table 4-2-2 Divide and Conquer Speedup……………………………………47

List of figure

Figure 2-2-1 Multicomputer……………………………………………………………9

Figure 2-2-1 ParMap………………………………………………………………15

Figure 2-3-1 ParMap Code……………………………………………………..16

Figure 2-3-2 Pipeline Code…………………………………………………17

Figure 2-5-1 Relationship of GdH, GpH and GHC…………………………24

Figure 3-1-1 GdHParMap Behaviour…………………………………………....27

Figure 3-1-2 GdHFarmMap Behaviour………………………………………30

Figure 3-2-1 DC Divide……………………………………………………………32

Figure 3-2-2 DC Conquer……………………………………………………….32

Figure 3-2-3 GdHDC Divide…………………………………………………33

Figure 4-1-1 Regular Data Measurement……………………………………38

Figure 4-1-2 Regular Data measurement……………………………………39

Figure 4-1-3 Irregular Data Measurement…………………………………40

Figure 4-1-4 Irregular Data Speedup…………………………………………..41

Figure 4-1-5 GpHShuMap……………………………………………………43

Figure 4-2-1 Divide and Conquer Measurement………………………………..47

Figure 4-2-2 Divide and Conquer Speedup……………………………………..48

1. Introduction

1.1 Overview

Parallel computing is an important aspect in computing, as it can reduce the time to solve large problems. Skeletons are a common and easy way to do parallel computing. In this project we develop and measure two skeletons: “ParMap” and “divide and conquer” in parallel and distributed version of Haskell [12]. Haskell is de facto standard of functional programming language. GdH and GpH are extension of Haskell. Both GdH and GpH provide the ability of parallel computing. GpH uses implicit parallelism and GdH uses explicit parallelism.

1.2 Contributions

The main contribution of this thesis is to design and implement “ParMap” and “divide and conquer” skeletons in GdH to compare their efficiency with other skeletons. In “ParMap” skeleton we discuss how and why we optimize the simple “ParMap” skeleton. Base on this experience we develop an efficiency “divide and conquer” skeleton which avoids the pitfall of the simple “ParMap” skeleton.

We measure both GdH and GpH skeletons and compare them. There are two version of “Map” skeletons we develop. They are “GdHParMap” and “GdHFarmMap”. To measure these “Map” skeletons we use both regular data and irregular data. “GdHFarmMap” get speedup up to 14.34 (irregular data) and 13.06 (regular data). It is much better than corresponding GpH skeleton which get speedup at about 7.47 (irregular data) and 8.8 (regular data). GdH “divide and conquer” skeleton also get excellent speedup which is 16.54 in this case. Again GdH performance is better than GpH whose “divide and conquer” skeleton get speedup at about 8.84.

1.3 Dissertation Outline

Chapter 2 is a section of background which contains parallel programming, parallel computer, parallel Programming Paradigm, skeleton parallelism, parallel and distributed programming languages especially GdH and GpH.

Chapter 3 discusses how we implement “GdHParMap”, “GdHFarmMap” and “Divide and Conquer” skeleton.

Chapter 4 contains the results of measurement. The result includes regular data results and irregular data results. Then we compare and analysis the results.

In Chapter 5 we summary what we have done and what we can do in the future.

2 Backgrounds

In this chapter we discuss some background acknowledgement. First we introduce parallel programming and parallel computer in which we define what parallelism is. And the following section we introduce two type of parallelism: data parallelism and control parallelism. Then we define skeleton parallelism. At the end we discuss some programming which includes Haskell, GpH, GdH ,etc

2.1 Parallel Programming

The processor become faster and faster. No matter how powerful the processor is, one-processor computer can not meet the requirement we need. A good idea is to use many processors concurrently. The computer, using a set of processors to solve computational problem, is named parallel computer. To take advantage of parallel computers we need to use parallel programming. Parallel programming entails partitioning the problem into smaller problems, and scheduling the execution of these smaller sub-programs (processes) onto multiple processors in parallel to solve the problem. This section will introduce some parallel programming languages, parallelism and design process of parallel programming.

2.1.1 Parallel Computer

A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem.[9] The model of parallel computer includes multicomputer; a distributed-memory MIMD (multiple instruction multiple data) computer with a mesh interconnect; a shared-memory multiprocessor, and a local area network etc.[9] Here we will discuss “Beowulf” system which is a multicomputer.

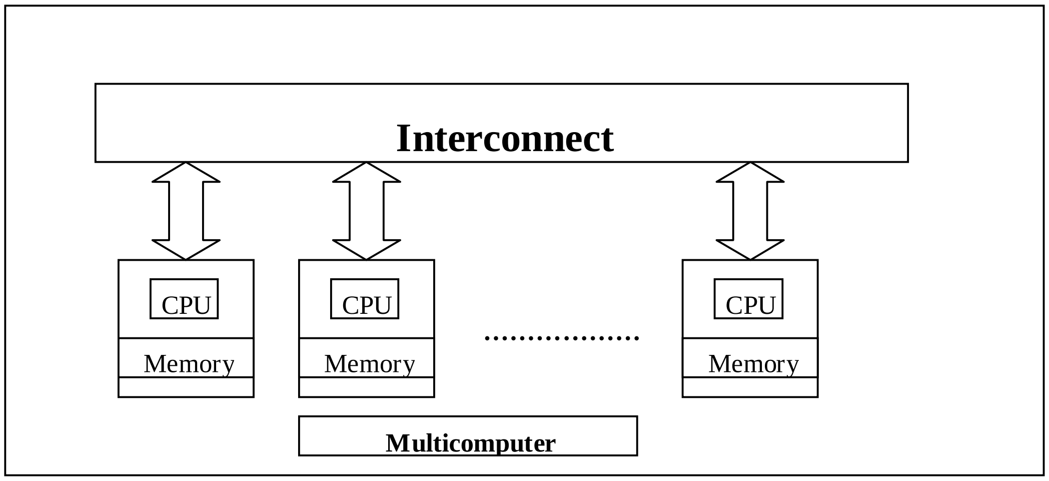

Figure 2-2-1 Multicomputer

A multicomputer consists of a number of computers, or nodes, linked by an interconnection network. Each computer executes its own program. This program may access local memory and may send and receive messages over the network. Messages are used to communicate with other computers or, equivalently, to read and write remote memories. In the idealized network, the cost of sending a message between two nodes is independent of both node location and other network traffic, but does depend on message length.[9]

A Beowulf cluster is a collection of personal computer (PCs) interconnected by widely available networking technology running any one of several open-source Unix-like operating systems.[28] In this project we have a “Beowulf” system which contains 32 computers. Each computer has 533MHz Celeron processor, 128kB cache, 128MB of DRAM and 5.7GB of IDE disk. They are connected through a 100Mb/s fast Ethernet switch with a latency of 142 us. The operation system is Red Hat 8.0. But we can not guarantee all of these computers can always work robustly at all time. So we measure our program on up to 16 processors.

2.1.3 Parallelism

What is “Parallelism”? Here we list some definition below:

1. “An approach to performing large, complex and/or lengthy tasks that involves concurrent operation on multiple processors” [6].
2. “Decomposition of a task into smaller tasks to be performed simultaneously, i.e. in parallel” [7].
3. “Parallelism is the use of similar patterns of words (or grammatical forms) to express similar or related ideas or ideas of equal importance” [8].

From the definitions above, we found:

1. Parallelism is not an object. It is an abstracted methodology or pattern.
2. A large, complex and/or lengthy task should be decomposed to some smaller related tasks.
3. These tasks could be performed simultaneously.
4. Parallelism is just the abstracted methodology or pattern to perform the point 2 and 3 listed above.

We defined parallelism as “Parallel is the abstracted methodology or pattern which decompose large, complex tasks to related smaller tasks and these smaller tasks can be run simultaneously”.

There are two levels of parallelism: implicit parallelism and explicit parallelism [19]. Implicit Parallelism means a parallel processing system decides automatically which parts to run in parallel. In contrast explicit parallelism requires programmers annotate his program to indicate which parts should be executed as independent parallel tasks.

2.1.4 Design process of parallel programming

The design process of parallel programming includes four distinct stages: Partitioning, Communication, Agglomeration and Mapping. There are three attribute we should think about: Concurrency, Scalability and Locality. Concurrency refers to the ability to perform many actions simultaneously; this is essential if a program is to execute on many processors. Scalability indicates resilience to increasing processor counts and is equally important, as processor counts appear likely to grow in most environments. Locality means a high ratio of local memory accesses to remote memory accesses (communication); this is the key to high performance on multicomputer architectures. [29]

Partitioning:

“The partitioning stage of a design is intended to expose opportunities for parallel execution” [9]. This stage focus on “a fine-grained decomposition of a problem” [9]. First, programmer find out the data associated with problem. Second, programmer determines the right partition for the data. Finally, programmer decides how to associate computation with data.

Communication:

As we have talked in parallelism, the tasks generated by partitioning stage are related. This means data must be transferred between tasks. “This information flow is specified in the communication phase of a design” [9]. There are two phases in this stage: 1.We defines channels structure that links the sender and recipient. 2. We specify the message to be sent and received. In [9] communication has been classify into four categorize: local/global, structured/unstructured, static/dynamic, and synchronous/asynchronous. Local communication means each task communicates with a small set of other tasks (its “neighbours"); in contrast, global communication requires each task to communicate with many tasks. Structured communication refers a task and its neighbours form a regular structure, such as a tree or grid; in contrast, unstructured communication networks may be arbitrary graphs. In static communication, the identity of communication partners does not change over time; in contrast, the identity of communication partners in dynamic communication structures may be determined by data computed at runtime and may be highly variable. Synchronous communication means producers and consumers execute in a coordinated fashion, with producer/consumer pairs cooperating in data transfer operations; in contrast, asynchronous communication may require that a consumer obtain data without the cooperation of the producer. [29]

Agglomeration

“In the third stage, agglomeration, we move from the abstract toward the concrete. We revisit decisions made in the partitioning and communication phase with a view to obtaining an algorithm that will execute efficiently on some class of parallel computer” [9]. The number of tasks will be reduced and the single task may become greater than before. The goal of this stage is: reducing communication cost, retaining flexibility, and reducing software engineering costs.

Mapping

“In the fourth and final stage of the parallel algorithm design process, we specify where each task is to execute” [9]. To get better performance tasks should be able to execute in different processors and communicate frequently on the same processor.

2.2 Parallel Programming Paradigm

There are many parallel programming paradigms. Data parallelism and control parallelism will be introduced. And we will discuss skeleton parallelism later.

2.2.1 Data Parallelism

Data parallelism involves performing similar computation on many data objects concurrently. It means some code segment runs concurrently on different data elements. For example, an operation named “Double” doubles each element in an array. One processor deals with the first element in that array. And other processor deal with other elements in the array at the same time. The most popular data parallel languages are based on FORTRAN and C [17].

For instance, the GdHParMap (Chapter 3.1) is a typical example of data parallelism. It splits input data list and send them to difference processor to evaluate.

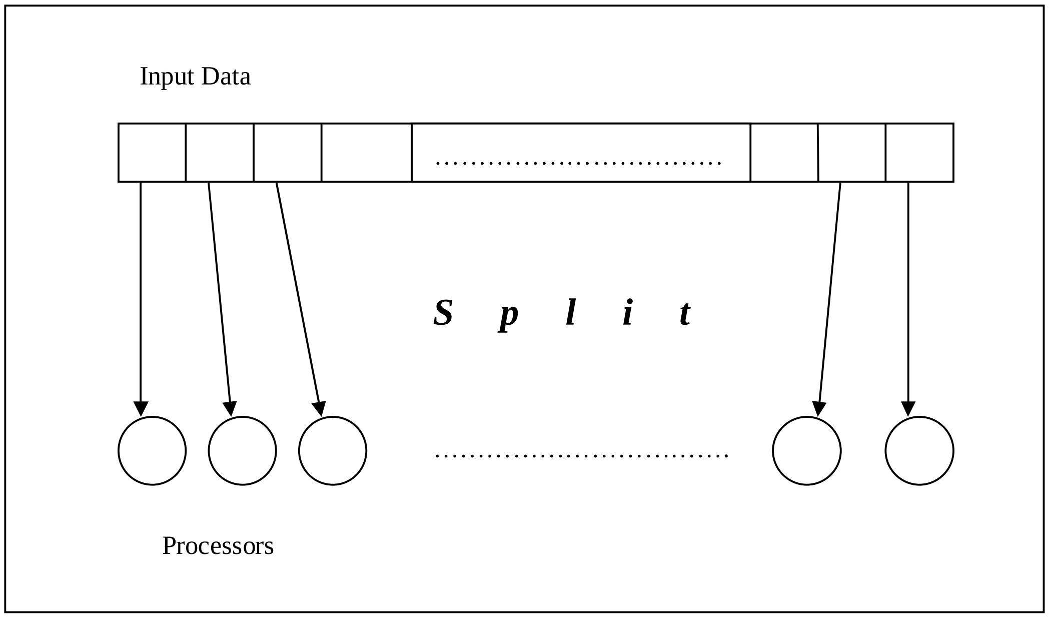

Figure 2-2-1 ParMap

2.2.2 Control Parallelism

Control parallelism means different operations can be performed simultaneously on different processors. It involves: functional decomposition; tasks with independent control flow; communication with each other.

Two well-known types of control parallelism are pipelining, in which different processors, or groups of processors, operate simultaneously on consecutive stages of a program, and functional parallelism, in which different functions are handled simultaneously by different parts of the computer. One part of the system, for example, may execute an I/O instruction while another does computation, or separate addition and multiplication units may operate concurrently. [30]

2.3 Skeleton Parallelism

“Within the existing body of parallel algorithms a number of patterns recur frequently. These patterns are composed of computations and the interactions between them and can be conceptually abstracted from the detail of the activities they control. Such abstractions have come to be known as algorithmic skeletons or simply as skeletons” [10]. ----The definition of “Skeleton”.

“Skeleton Parallelism” is to use skeleton algorithms as a technique to parallelism functional language and program parallel machines. It can be either data or control parallelism. For example, “parMap” (Figure 2-3-1) is a simple skeleton which uses data parallelism. And “Pipeline” [31] is a example of control parallelism. (Figure 3-2-2)

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]

parMap strat f xs = map f xs `using` parList strat
```

Figure 2-3-1 ParMap Code

```
pipe_naive :: transmissible a => [[a] -> [a] -> [a]

pipe_naive fs xs = (ppipe fs) # xs


ppipe :: transmissible a => [[a] -> [a]] -> process [a] [a]

ppipe [f] = process xs -> f xs

ppipe (f:fs) = process xs -> (ppipe fs) # (f xs)
```

Figure 2-3-2 Pipeline Code

2.4 Programming Languages

2.4.1 Parallel Programming Languages

There are many parallel programming languages. This is an introduction of some typical parallel programming languages.

1. Eden

Eden is a declarative language for parallel and concurrent programming which is defined as an extension of Haskell [1].

2. High Performance FORTRAN (HPF)

“High Performance Fortran is an informal standard for extensions to Fortran to assist its implementation on parallel architectures, particularly for data-parallel computation “[2]. The first version of HPF is HPF 1.0 which developed between March 1992 and May 1993 [3]. The latest version is HPF 2.0.

3. ZPL

“ZPL is an array programming language designed from first principles for fast execution on both sequential and parallel computers” [5]. Foremost ZPL is an array programming language. But it is better than traditional array programming language for some distinguish characteristics. For example, ZPL has a performance model that allows users to know roughly how well their programs will run on parallel machines. This property has been pioneered by ZPL in the parallel context. [5]

2.4.2 Functional Programming

“Functional Programming is based on the simplest of models, namely that of finding the value of an expression” [11]. One of its distinguishing characteristic is that it focus on the on the high-level "what" rather than the low-level "how". A functional programming is consisted of defining functions and other values.

Given a function named “swap” which swaps two values. We can define it by Haskell as below:

```
-----------------------------------------------------
swap :: ( Int, Int ) -> ( Int, Int )
swap :: ( a, b ) = ( b, a )
-----------------------------------------------------
```

We found we don't need to care how to swap the two values. We just tell system what we want. The following code is the same function defined by C/C++. It focuses on how we swap the two values.

```
//swap function by C/C++
void swap( int &a, int &b ){
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}
//end swap function
```

There are lots of functional programming languages. For example, Haskell, Lisp, SML etc are functional programming language. We will discuss Haskell later. Here we introduced Lisp and SML.

Lisp:

“Lisp is a programmable programming language.” –John Foderaro [14]. Lisp is the second oldest (high level programming) computer language. (FORTRAN is the oldest one.) Lisp was designed by John McCarthy in the late 1950’s. It is a strict, dynamically type language. Now lisp has evolved into a family of languages. Today Common Lisp and Scheme are most widely used. “In 1994, Common Lisp became the first ANSI standard to incorporate object oriented programming” [14].

SML:

“Standard ML is a safe, modular, strict, functional, polymorphic programming language with compile-time type checking and type inference, garbage collection, exception handling, immutable data types and updatable references, abstract data types, and parametric modules ”[13].

The first version of SML was defined in *Definition of Standard ML (Milner, Tofte, Harper, MIT Press, 1990).* The latest version is SML ’97 which was defined in The Definition of *Standard ML (Revised) (Milner, Tofte, Harper, MacQueen, MIT Press, 1997)*

2.4.2 Haskell

Haskell is a pure functional computer programming language. “In particular, it is a *polymorphic typed, non-strict, purely functional* language, quite different from most other programming languages ”[12]. The features of Haskell are: brevity, ease of understanding, no core dumps, and code re-uses strong glue, powerful abstractions, and built-in memory management. The latest version is Haskell 98.

2.5 Parallel and Distributed Functional Programming (GdH and GpH)

Without any doubt there are many parallel or distributed functional programming languages which are being used. E.g. Curry[32], Goffin[33]. In this chapter we focus on Glasgow Parallel Haskell (GpH) and Glasgow distributed Haskell (GdH).

2.5.1 Glasgow Parallel Haskell (GpH)

“Glasgow Parallel Haskell (GpH) is a small extension to sequential Haskell that executes on multiple processor elements (PEs)”[15]. GpH provides pure threads which are non-side-effecting and so perform no I/O.

There are two composition operators in GpH. One is “par” which takes two parameters that are to be evaluated in parallel [21]. The expression a par b has the same value as b. The other is “seq” which is a sequential composition operator. “par” operator describes which operations should be evaluated in parallel. It is not strict in its first argument. The first argument will be sparked. At the same time the second will be evaluated by another parallel thread. This operator can produce too many small tasks. Sometimes this will not give you performance. Another problem is we can not control the order of evaluation. To solve these problem “seq” was produced. “seq” defines the order of evaluation. Given expression “a ‘seq’ b”, if “a” is not bottom, this expression has the value of “b”. Otherwise it has the value of “a”. The corresponding dynamic behavior is to evaluate “a” to weak head normal form (WHNF) before returning “b” [21].

2.5.2 Glasgow distributed Haskell (GdH)

“Glasgow distributed Haskell (GdH) is intended to provide a high-level distributed programming model consisting of a hierarchy of threads – the explicitly placed I/O threads of Concurrent Haskell and the Implicit pure threads of GpH ”[15].

To provide a high-level distributed programming model it consist of two classes of threads: pure threads and side-effecting I/O threads [22]. The implicit pure threads are achieved by using shared variables. It also is introduced and synchronised using parallel and sequential composition. “Explicit communication and synchronisation is provided using polymorphic semaphores (MVars) within the I/O monad “[22]. Without doubt the MVars has been extended for the distributed context to provide explicit synchronisation and communication between remote I/O threads.

Four primitive mechanisms were implemented in GdH. They are listed as follows: 1. a list of processor element (PE) identifiers. It provides the calls to read the existing PE tables held in the runtime system (RTS). 2. A remote co-routine operation (revalIO). The revalIO operation encapsulates both remote thread creation and the communication of result [22]. 3. A mechanism for retrieving the owning PE of an immobile object. 4. Distributed exception handling.

2.5.3 Relationship among GpH, GdH and GHC

“Haskell is the de facto standard non-strict functional language and the Glasgow Haskell Compiler (GHC) is arguably the best non-strict Haskell implementation” [15]. Glasgow Haskell Compiler (GHC) not only implements Haskell 98 but also supports concurrency and exceptions for Concurrent Haskell. So the version, which base on GUM RTS, supports parallelism and a shared heap across multiple PEs for GpH. Meanwhile GdH is a minimal superset of the GpH and Concurrent Haskell languages.

The relationship between GpH, Concurrent Haskell, and GdH is shown as follow:

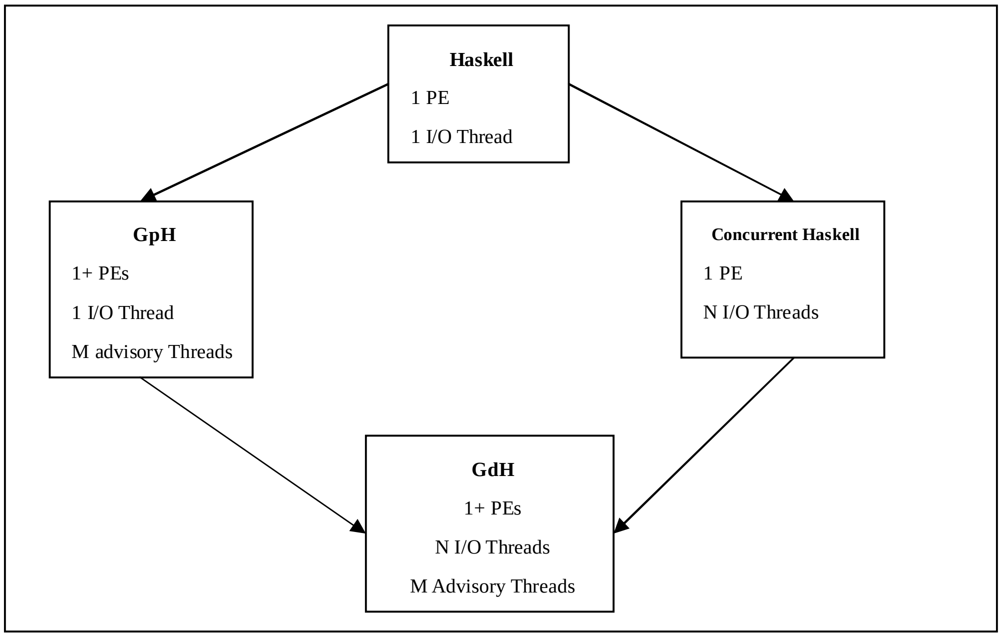

Figure 2-5-1 Relationship of GdH, GpH and GHC

3 Designing and Implementing Skeletons

To investigate the use of explicit coordination in Glasgow distributed Haskell (GdH) for constructing parallel skeletons, we implement two skeletons: “ParMap” and “Divide and Conquer” using GdH. In theory, GdH should get better performance than GpH which use implicit parallelism. We choose corresponding GpH skeletons to measure and compare with GdH skeletons. The GpH skeletons we chose are parMap, chunkMap and divConq which you can find them in [34]. We also implement a GpH skeleton which tries to use explicit parallelism. From this program we can know what performance we can get if we use explicit parallelism over implicit parallelism. In the following section we just explain GpH program briefly. For detail you can find in [34].

3.1 ParMap

The first skeleton we developed is “map”. In a map, each data item appearing onto the input stream is split into its basic components, each one of the components is processed using a function “f”, which performs computing over the basic component, and finally the computed components are re-assembled into a data structure to the original one.

The whole process of map skeleton is explained as follow:

Given the “map f xs” has the type: a vector stream -> b vector stream

Assumes the function “f” has type: a -> b

Provided that the input stream is xn: …: x1: x0

GpH

GpH ParMap simply use “par” to evaluate first data on one PE and then evaluate the rest data concurrently. The core code is listed below:

```
parMap f (x:xs) = fx `par` fxs `seq` (fx:fxs)
            where
              fx = f x
              fxs = parMap f xs
```

The simple GpH ParMap above produces too many communications when number of processors is equal or more than 2. If we can force one processor to evaluate more data we can reduce a lot of communication. “chunkMap” is just the function that we can specify how many data to be evaluate once on a processor. The usage is listed as follow:

```
chunkMap 6 rnf fib (replicate 100 30)
```

GdH

First the input stream is split into basic components: xn… x1, x0. Then each of components is sent to a worker to be evaluated. At the end the computed components are re-assembled into original data structure. The figure 3-1-1 shows the behavior of map.

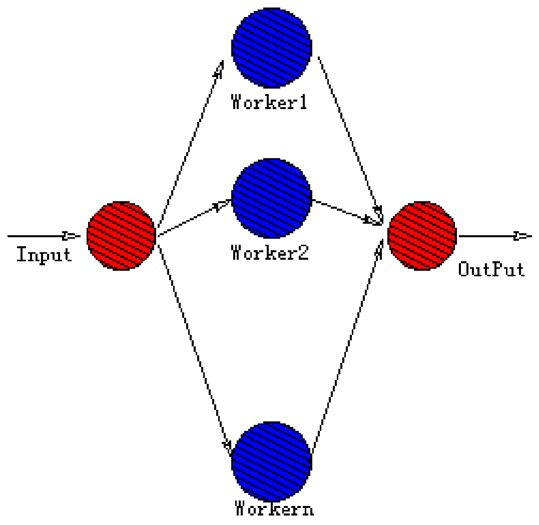

Figure 3-1-1 GdHParMap Behavior

The core GdH source code is explained as follow:

First we get all processors' ID. If there is only one processor we will evaluate input stream by sequence map function and return results. Otherwise we will evaluate it in parallel as follow:

--we define how the data will be evaluated on remote processor and save result into an MVar value:

```
let remote te mVar = do
        put mVar mVar te `demanding` rnf te
        return ()
```

--we build a list of PEId whose length is as same as the length of input stream.

Let ips = repeatPE (length ls) ps

--we also build a list of MVar whose length is as same as the length of input stream. And the list of MVar will be stored in an MVar value. They store each result. So we can fetch them from the MVar value later.

```
createMVarList (length ls) mList

ms <- takeMVar mList

--combine all elements and create remote tasks on remote processors.

Let zs = zip3 ips ls ms

Let work (pe,ts,mVar) = do

                        Let te = f ts

                        rforkIO (remote te mVar) pe

mapMV work zs

--At the end we fetch each result and return.

Rs <- getResult ms

Return rs
```

In order to reduce the number of communication we develop another map skeleton named “FarmMap”. As what we said, if there are n basic components this skeleton needs n workers to evaluate input stream. In this project we use Beowulf Systems to evaluate our skeletons. A Beowulf is a collection of personal computers (PCs) interconnected by widely available networking technology running any one of several open-source Unix-like operating systems[28]. So the communication between two processors is network communication. This type of communication is very expensive. It would be better if we could reduce this communication. For this reason we developed a

Farm Map skeleton. The whole process is explained as follow:

Given the “map f xs” has the type: a vector stream -> b vector stream

Assumes the function has type: a -> b

Provided that the input stream is xn: …: x1: x0

Assumes the number of processor(s) is m

First we divide input data into m chunks. If the number of components is less than m there will be (n-m) empty chunk(s). When (mod n m) equals 0 each chunk contains n/m components. Otherwise the last chunk contains n-n/(m-1) components and other chunks contain n/(m-1) components. Each chunk is treated as a task and is picked by a worker. Because the number of processor(s) equals tasks’ number the communication’s scale in proportion to the amount of processors. Normally there are m*2 communications (send and receive) in Farm Map skeletons. If n>>m we can reduce a large number of communications.

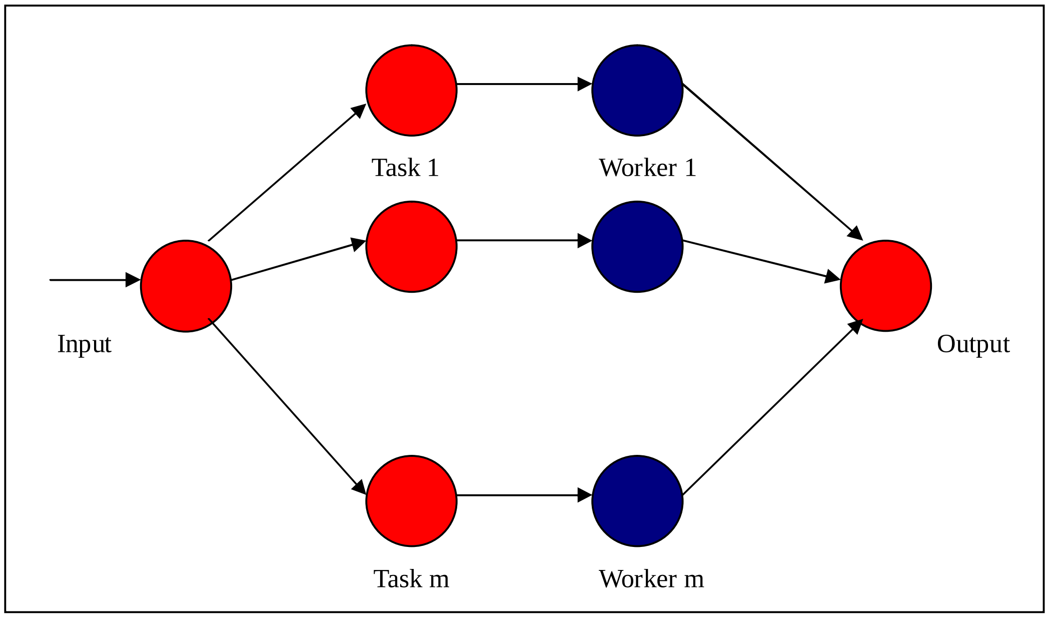

Figure 3-1-2 GdHFarmMap Behavior

The core GdH source code is explained as follow:

--we get the number of available processors first. We assume there are n processors. The input stream will be split into n chunks.

```
ps <- allPEId

let n = length ps

let ts = unshuffle n ls

--defines the remote te and prepare MVar values list

let remote te mVar = do

        putMVar mVar te `demanding` rnf te

        return ()

mList <- newEmptyMVar

createMVarList n mList

ms <- takeMVar mList
```

```
--combine elements and evaluate data in parallel
let zs = zip3 ps ts ms
let work (pe,ts,mVar) = do
            let te = map f ts
            rforkIO (remote te mVar) pe
mapMV work zs
--get result. Now the result is not the original data structure
rs <- getResult ms
--reassemble the result to original data structure and return it
return (shuffle rs)
```

3.2 Divide and Conquer

The second skeleton we developed is "Divide and Conquer" skeleton. This skeleton is a data-parallel and nested skeleton. The simple divide and conquer skeleton is to divide problem into two (or more than two) sub-problem if it can be divided. And the sub-problem will be divided into other two (or more than two) sub-problem if it can be divided again. This action will be processed until it can not be divided. Then all the lowest level sub-problem will be evaluated in parallel. The results will be transmitted to upper level and be combined by "Conquer" which is defined by user. This action will be repeated until the all problems have been computed. Figure 3-2-1 shows how the problem is to be divided. Figure 3-2-2 shows how the results are to be return and be combined.

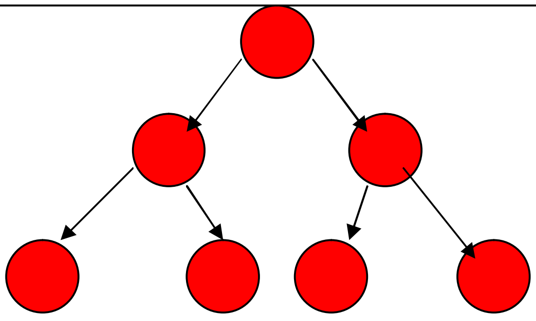
Figure 3-2-1 DC Divide

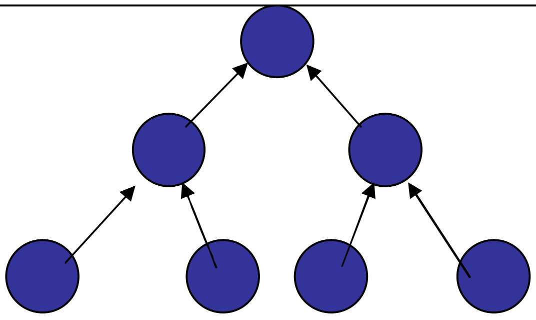
Figure 3-2-2 DC Conquer

In our project we use this skeleton to evaluate "Fib n". As a result each data will be spitted into two parts ("n-1", "n-2") if it can be spitted. In GdH one part will be evaluated on original processor and another part will be evaluated on other processor. Figure 3-2-3 shows one possible distribution. In GpH we can not specify which processor we should use. The only thing we could do is to tell system that we want these tasks to be evaluated in parallel. In order to avoid extra network communication we add a condition over whether the data can be further divided or not. If there is one or more than one free processors it means the data can be divided. Otherwise the data can not be divided.

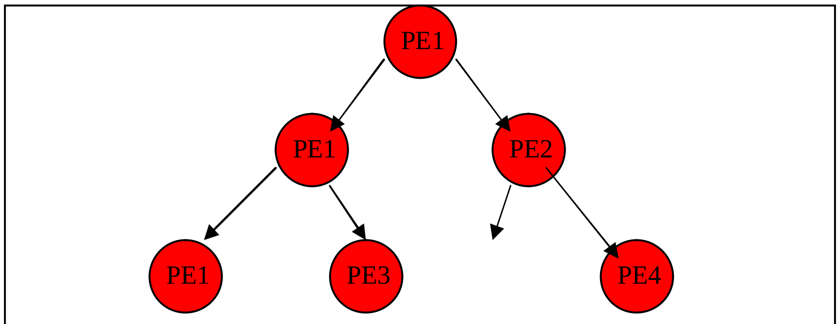

Figure 3-2-3 GdHDC Divide

The core GdH source code is explained as follow:

--There are two important function in our divide and conquer skeleton. One is “divConq”, the other is “divConqToIP”. “divConq” is the main function which should be used by users. “divConqToIP” is the function to implement parallelism. “divConq” creates some necessary values and sends them to “divConqToIP” when it invokes “divConqToIP”. "divConqToIP” does nested actions to compute data in parallel. It decides which processor can be used to evaluate sub-problem from a PEId list. It will delete the PEId from that list if “divConqToIP” decides to use this processor. Or it will evaluate data in sequence if there is no more free processor or the data can not be divided.

--we define pFib which uses divide and conquer skeleton to evaluate “fib n” function.

pFib :: Int -> IO Int

```
pFib n = divConq fib n threshold conq divide

        where

        --If n<=30 we evaluate it in sequence.

        threshold n = n <= 30

        --The way we divide the data.

        divide n = (n-1, n-2)

        --The definition of conquer

        conq l r = l + r + 1

--some source code in “divConqToIP” function

--Create two Mvar values to store the divided sub-problem’s result

mL <- newEmptyMVar

mR <- newEmptyMVar

--Here we run the two divided sub-problems

right mR (head nextPEl)

left mL

--Waits for the result

mrsR <- takeMVar mR

mrsL <- takeMVar mL

--Combines the two results

let te = conquer mrsL mrsR

putMVar mrs te `demanding` rnf te

return ()
```

```
where
    (lt,rt) = divide arg
    left ml = do
        --evaluate one part on original processor
        divConqToIp f lt threshold conquer divide mPEs mAllPE ml
        --evaluate another part on other processor
        right mr nextPE = do
        rforkIO (divConqToIp f rt threshold conquer divide mPEs mAllPE mr)
nextPE
```

4. Evaluating Skeletons

In this chapter we discuss how we evaluate our skeleton: we evaluate “Map” skeleton with regular data and irregular data. And we use “fib 42” to evaluate “divide and conquer” skeleton. We list all result and build speedup graph. At the end we compare and analysis these results.

4.1 ParMap

To evaluate these skeletons we chose regular data and irregular data and the function is the Fibonacci (fib) function. The reasons we chose this data and function are below: The “fib” function is a well-known and very simple function. The key reason we chose this function is it can consume a long CPU time in a simple way. To evaluate parallelism, regular data can give us reliable result. On the other hand irregular data always occurs in real world. So we chose both regular data and irregular data to evaluate skeletons. As we said the communication between two processors is very expensive. This communication can not be avoided in parallel computing. So the smaller ratio which the communications occupy in the whole process the better result we can get. The work could not be too small. As a result we chose “fib 30” for regular data and chose “fib x (25<x<32) “for irregular data. The scale of tasks can not be too small. For this reason we chose 100 tasks.

For each skeleton we evaluate it many times and record all results. We have two steps to choose final result: First we abandon all extreme results. For example, if we got 5 results: 50s, 10s, 48s, 120s and 51s, we will abandon two bad results (10s and 120s). They occur by accident and are not common results. Secondly we choose the median. In this case “50s” will be our final result.

4.1.1 Regular Data

As we said we evaluate the skeletons many times. And record all result. At the end the final result is chosen after two steps. The “GdHWorkPool” designed by Dr Robert. The “GpHShuMap” which is given in Appendix A5 is poorly designed GpH skeleton. We also evaluate them as contrast.

The final results are below:

PEs / Skeletons	1	2	4	8	16
GpHParMap	176.76	227.63	128.73	36.30	20.69
GpHChunkMap	170.36	69.75	42.45	33.31	19.35
GdHParMap	140.75	140.57	65.67	32.61	16.08
GdHFarmMap	143.11	72.90	36.80	19.76	10.96
GdHWorkPool	107.92	57.69	28.63	14.53	8.64
GpHShuMap	168.45	113.86	56.32	33.18	40.2

Table 4-1-1 Regular Data Measurement

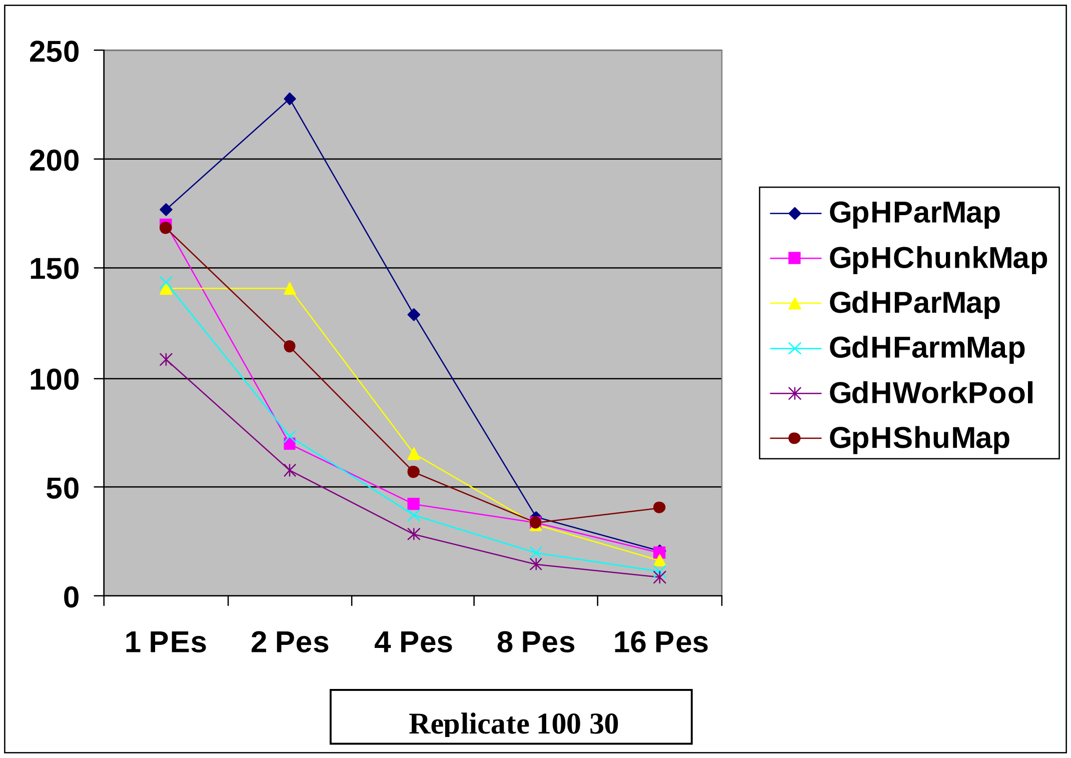

Figure 4-1-1 Regular Data Measurement

The following table and figure are speedup result. It bases on the runtime result. If we assume the runtime results are R1, R2 … Rn. Then speedup result will be Xn=R1/Rn.

The speedup:

Skeletons \ PEs	1	2	4	8	16
Ideal	1	2	4	8	16
GpHParMap	1	0.78	1.37	4.87	8.54
GpHChunkMap	1	2.44	4.01	5.11	8.8
GdHParMap	1	1.00	2.14	4.32	8.75
GdHFarmMap	1	1.96	3.89	7.24	13.06
GdHWorkPool	1	1.87	3.77	7.43	12.50
GpHShuMap	1	1.48	3.00	5.08	4.19

Table 4-1-2 Regular Data Speedup

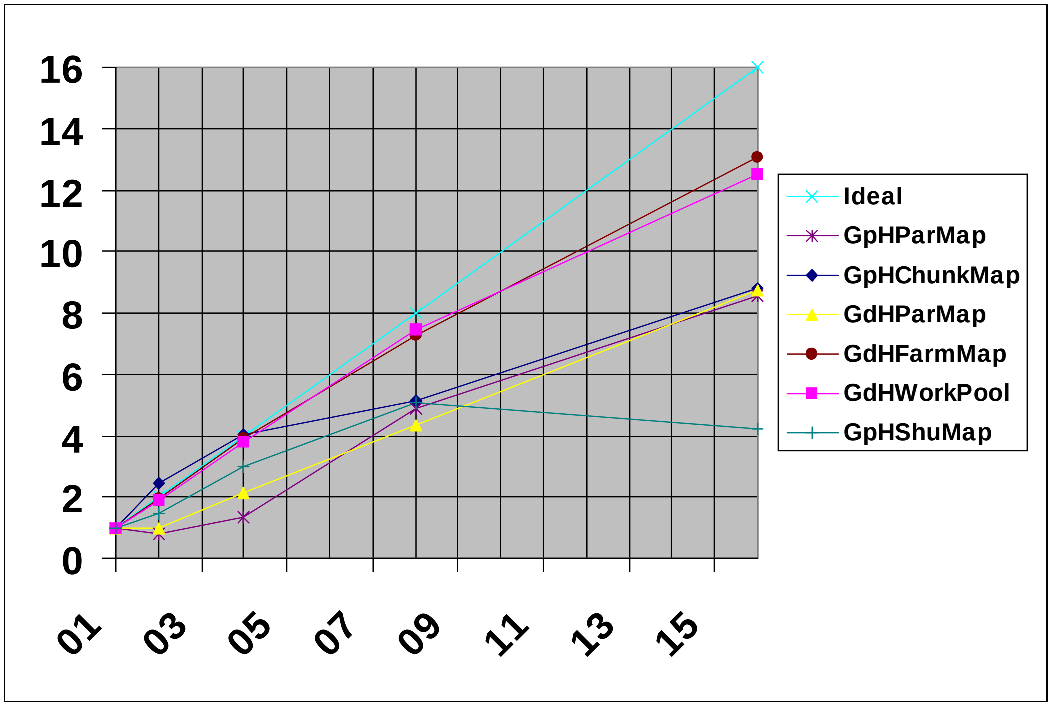

Figure 4-1-2 Regular Data measurement

4.1.2 Irregular Data

The runtime result:

Skeletons \ PEs	1	2	4	8	16
GpHParMap	132.31	165.88	97.41	27.65	22.92
GpHChunkMap	165.48	73.16	38.47	62.30	21.37
GdHParMap	86.05	76.47	42.96	18.97	8.89
GdHFarmMap	87.46	42.76	25.00	11.71	6.13
GdHWorkPool	108.01	57.63	28.63	14.53	8.64

Table 4-1-3 Irregular Data Measurement

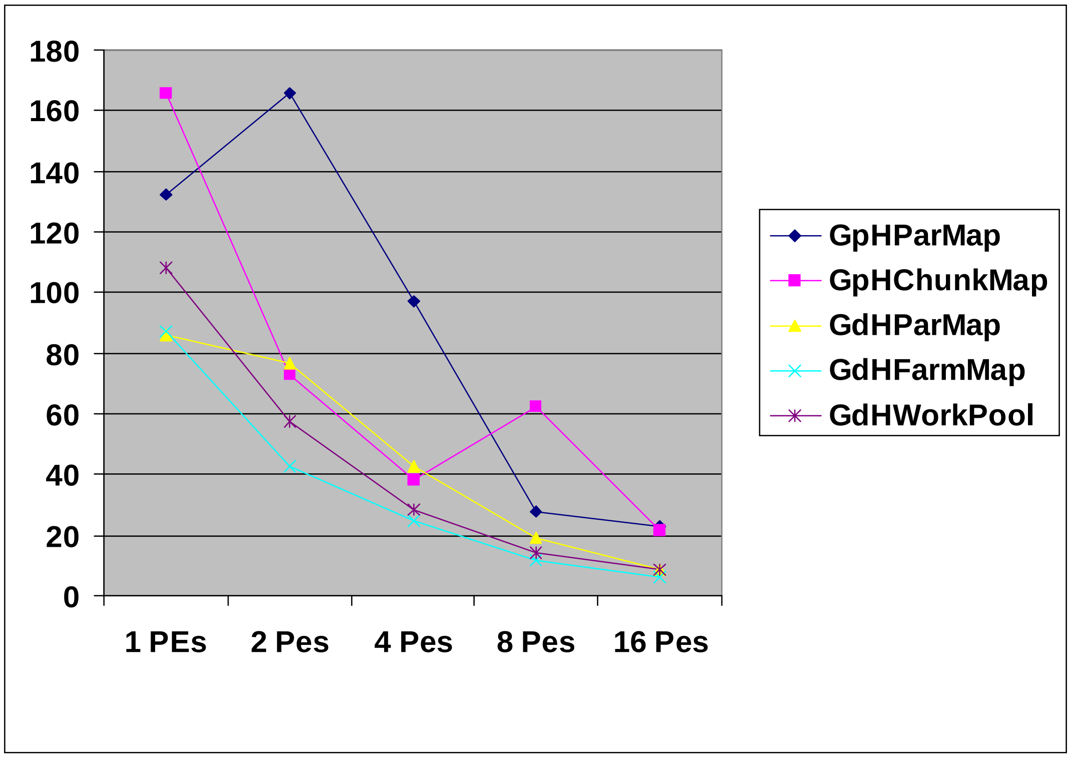

Figure 4-1-3 Irregular Data Measurement

The speedup results:

Skeletons \ PEs	1	2	4	8	16
Ideal	1	2	4	8	16
GpHParMap	1	0.80	1.36	4.79	5.77
GpHChunkMap	1	2.26	4.30	2.67	7.74
GdHParMap	1	1.13	2.00	4.54	9.68
GdHFarmMap	1	2.05	3.50	7.47	14.34
GdhWorkPool	1	1.88	3.78	7.44	12.51

Table 4-1-4 Irregular Data Speedup

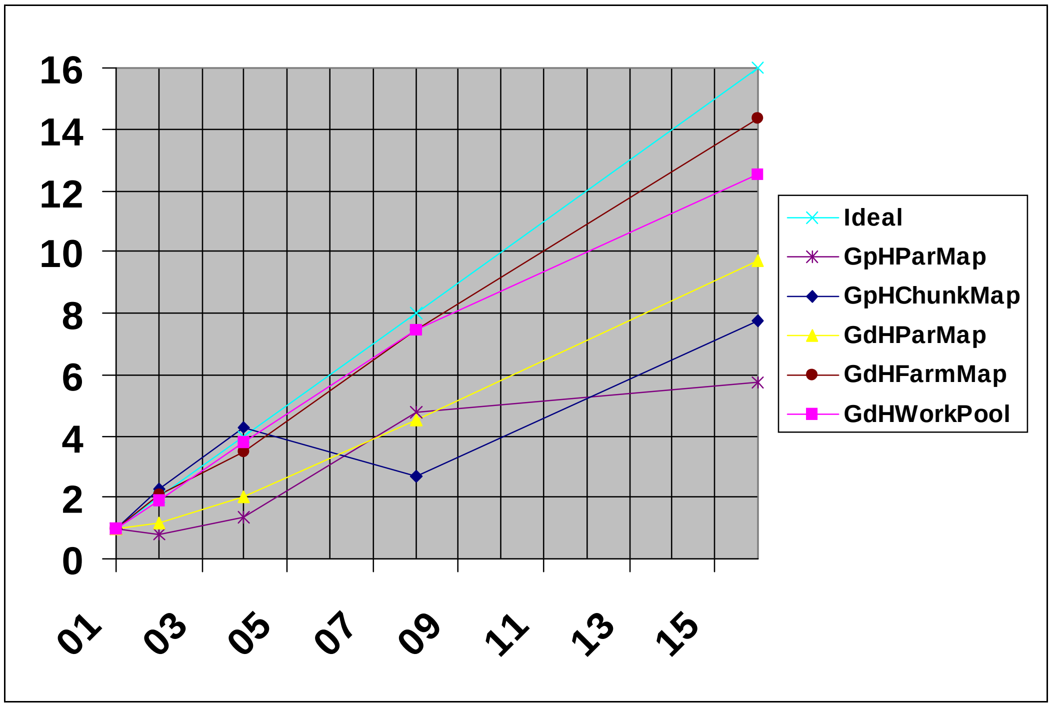

Figure 4-1-4 Irregular Data Speedup

4.1.3 Discussion

Gdh VS GpH

From figure 4-1-1 we can find that some GdH skeletons run slower than GpH on 1 PE. But they always run faster than GpH at 16 PEs. GdH scales much better than GpH and the figure 4-1-2 confirms this. As GpH use implicit parallelism, it needs extra communications to indicate which processor can be used. In contrast GdH uses explicit it does not need that communication. In theory GdH should get better performance than GpH. So this result is what we expected. In GdH we can specify the processor to evaluate problem. On the other hand, we don’t need to worry about which processor we should choose to evaluate our problem in GpH. GpH evaluates the problem by implicit parallelism. As a result we must do more jobs to select processor for the tasks in GdH. At the same time GpH need some time to inquire which processor can be used. But on 1 processor this communication is lightweight. When the number of processor grows to 16, the communication changes to cross processor communication. This kind of communication is very expensive. That is why GpH runs faster on 1 processor and slower on 16 processors.

In the irregular data runtime (Figure 4-1-3) and speedup graphs (Figure 4-2-4) we fund that the above conclusion is not always the valid. The reason is the different parallelism and the strategy we used. The source data we evaluated is chosen from 25 to 33 randomly. To evaluate fib 33 takes much more time than to evaluate fib 25. In GdH, except gdhWorkPool, we only consider the number of tasks, not matter how long each task need we evaluate them on the some way. The worst situation is: Some heavy tasks are sent to the same processor and all other processors have to wait this processor. In GpH system inquires which processor can be used. The worst situation above can not happen if each task is small enough. The exception is gdhWorkPool. It sends task to processor when the processor is free. So the worst situation is avoided.

GpHShuMap (Appendix A5 or A11):

This function is a poorly designed GpH function. As we said GpH uses implicit parallelism. “GpHShuMap” try to use explicit parallelism in GpH. It divides the list into some chunks and tries to dump each to one processor. But GpH will relocate the task to processors. This behaviour force GpH to do some extra communication. This is the reason why GpHShuMap has longer runtime on 16 processors than on 8 processors.

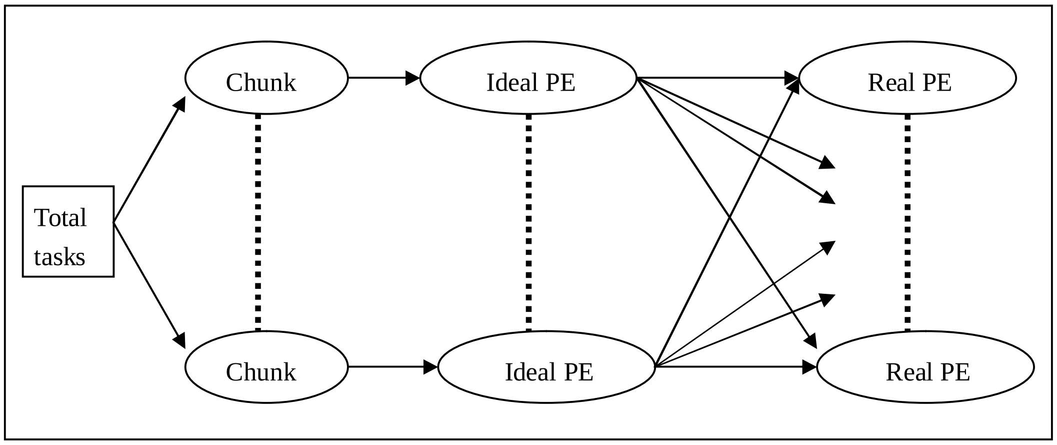

Figure 4-1-5 GpHShuMap

GdHParMap (Appendix A1 or A7)

We fund that the time ParMap run on 2 processors is so close to the time on 1 processor. As we said GdH need some expensive communication when processors are more than 1. We define Te as the time to evaluate. Np is number of processor. C is cross-processors communication. Ta is total time. The time on 1 processor is: Ta = Te. The time on more than 1 processor is: Ta = Te/Np + C. If C is close to Te, there is not speedup from 1 processor to 2 processors. But it has speedup from 2 to N (N>2) processors. As a result GdHParMap didn't get good speedup from 1 to 2 processors. But it runs faster and faster when processor is increased. From table 4-1-5 we can see the excellent speedup from 2 to 16 processors. (Results base on regular data.)

GdHParMap	2PEs	4PEs	8PEs	16PEs
Runtime	140.57	65.67	32.61	16.08
Ideal	1	2	4	8
Speedup	1	2.14	4.31	8.74

Table 4-1-5 GdHParMap Speedup

GpHParMap (Appendix A3 or A9)

GpHParMap is a simple GpH skeleton given in Appendix A3 or A9. When the processors increased from 1 to 2 it can have no speedup because of the expensive communication. The reason is similar to the same situation of GdHParMap. From table 4-1-2 we can see the GpHParMap and GdHParMap get very similar speedup at the end. It does not mean they have similar performance. GdHParMap is optimized for 1 processor. GdHParMap does not do extra job on 1 processor. It just evaluates the task in sequence. That is why it runs fastest on 1 processor. Compared with GpHParMap, GdHParMap is always better on 1 processor all the way to 16 processors.

GpHChunkMap (Appendix A4 or A10)

For a GpH program it achives good speedup from 1 processor to 2 processors. It is benefited from its strategy of evaluation. It divides all tasks into chunks. If we define “Nc” as the number of chunks, the total time of evaluation is: Ta = Te/Np + C/Nc. So it got good speedup from 1 processor to 2 processors even if cross-processors communication is so expensive. At the same time each chunk contains some tasks. If the tasks in one chunk are very heavy the other processors should wait the processor which evaluates the heavy chunk. This is the reason why it can not get good performance when measure irregular data.

GdHFarmMap (Appendix A2 or A8)

The parallelism of this skeleton is very similar to GpHChunkMap. The difference is implicit and explicit controls of work definition. In GdHFarmMap it can know how many processors it got. So it can divide all tasks into chunks whose number equals the number of processors. It reduces the communication as little as notably. As a result it gets the best speedup in regular data.

Fortunately it gets the best speedup in irregular data as well. The reason is there is no too heavy tasks to be evaluated in one chunk. If one chunk contains some heavy tasks GdHFarmMap could not get good speedup.

4.2 Divide and Conquer

To evaluate this skeleton we also choose “fib” function. After some testing we choose “fib 42” to evaluate. The reason is it has a runtime of about 400 seconds. The result is listed below:

Runtime result:

	1PE	2PEs	4PEs	8PEs	16PEs
GpHDC	361.56	797.70	400*	191.93	40.91
GdHDC	454.21	282.73	108.61	68.78	27.47

Table 4-2-1 Divide and Conquer Measurement

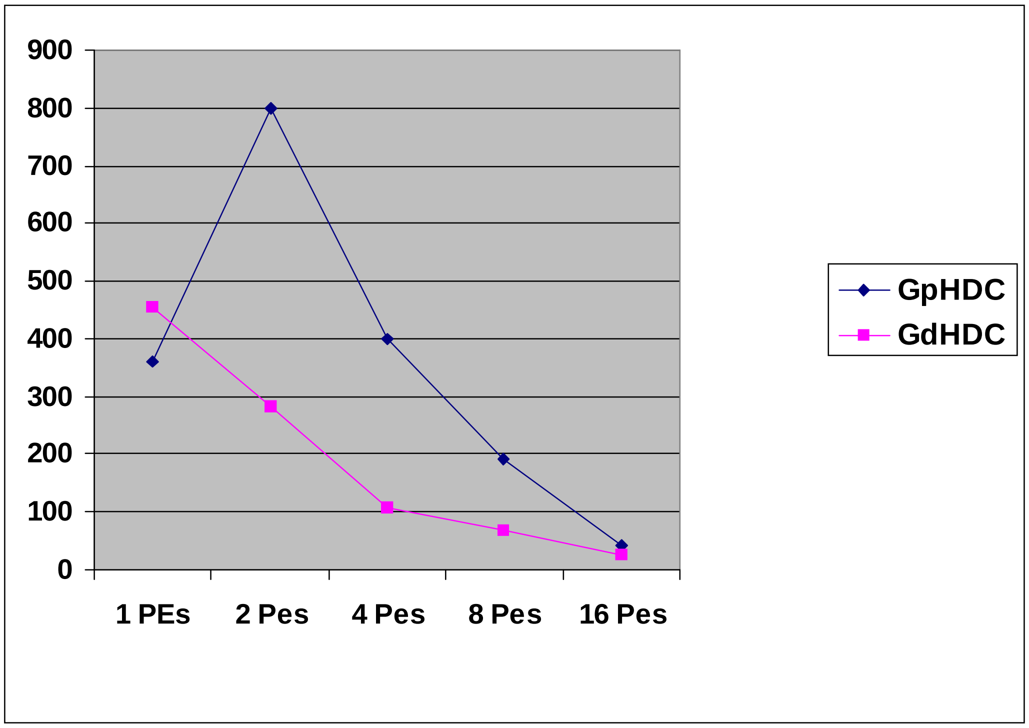

Figure 4-2-1 Divide and Conquer Measurement

The speedup results:

	1PE	2PEs	4PEs	8PEs	16PEs
GpHDC	1	0.45	0.9	1.88	8.84
GdHDC	1	1.61	4.18	6.60	16.54

Table 4-2-2 Divide and Conquer Speedup

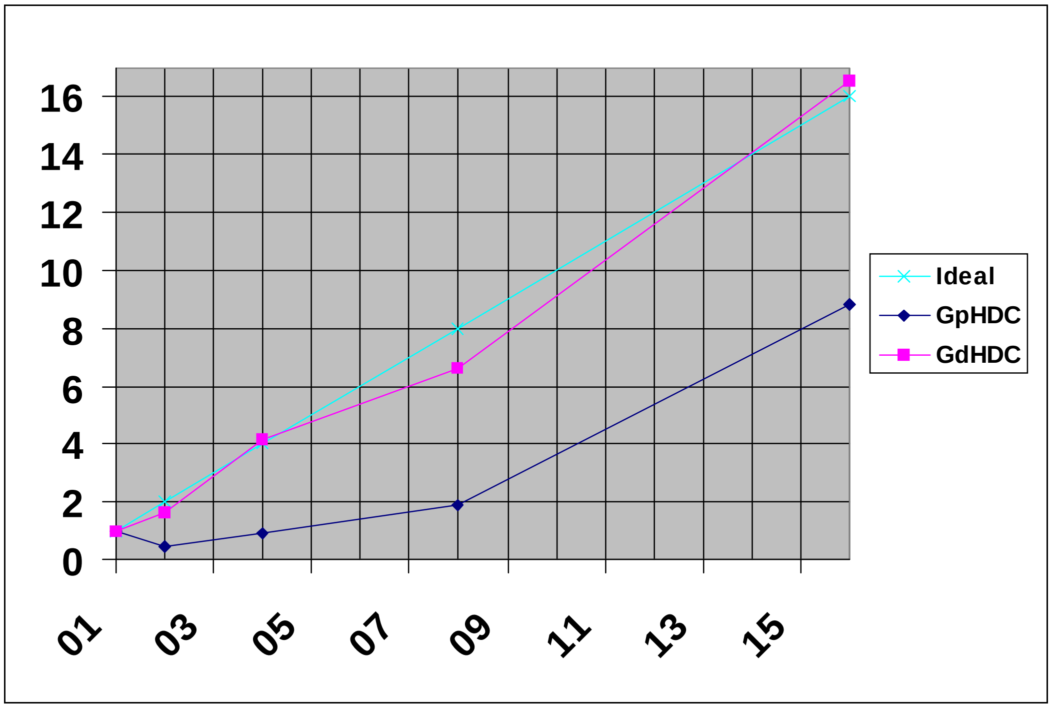

Figure 4-2-2 Divide and Conquer Speedup

GpHDC

In our project we evaluate "fib 42". And we define it divide until "fib 30". There will be at most $2^{(42-30)}$ sub-tasks. If there are two processors there would be at most $2^{(42-30)}/2$ communications. That is why it can not have good speedup on two and four processors. From two processors the communications time has been reduced as the processors increased. In this case it gets speedup from 8 processors.

GdHDC

GdH uses explicit parallelism. So it stops dividing when it meets the requirement or there are no more processors. As a result there are very few communications for each processor. We can see the speedup is so close to ideal speedup.

5 Conclusion

In this chapter we summary what we have done and draw conclusion. And then we discuss how we can extend our project in the future.

5.1 Summary

In chapter 2 we introduce parallel computing, parallelism, skeletons, parallel programming languages, Haskell, GpH, and GdH etc. In this chapter we defined what parallelism is. We also listed other similar languages. We pointed out the relationship of the GdH, GpH and Haskell. Chapter 3 introduced how we design and implement the skeletons. In this section we explain how some core code works. We listed some core code and add comment with the code. Chapter 4 contains the results of measurement. The result includes regular data results and irregular data results. Then we compare and analysis the results. The skeletons which is implemented in GdH is better than the skeletons implemented in GpH.

In Chapter 3 we develop two GdH skeletons which are “map” and “divide and conquer”, evaluate them and compare them with the corresponding GpH skeletons. As GdH have more concept than GpH, we find GdH is a little more difficult than GpH to implement. But it gives us more flexible controls. In GdH we can specify which tasks should be evaluated on which processor. To be a good parallel algorithm one of the keys is to reduce communications. For example, the “GdHParMap” and the “GpHParMap” produces too many communications. As a result, both of them can not get good speedup when processors increase from 1 to 2. (See Table 4-1-2 or Table 4-1-4)

5.2 Future work

To deal with real world problem these two skeletons are not enough. More GdH skeletons should be developed. For example, the “Fold” and the “pipeline” are well-know and useful.

In “GdHFarmMap” there should be a mechanism to predicate the runtime of task. It is dangerous if some heavy tasks are divided into one chunk. In “divide and Conquer” it should have a load-balancing algorithm. Otherwise one process may be heavily loaded while other are idle.

We can compare them with other parallel functional language skeletons. For example Eden.

Reference

[1] Eden, "http://www.mathematik.uni-marburg.de/inf/eden/", retrieved by June 2003.

[2] High Performance FORTRAN, "http://www.math.fu-berlin.de/user/kranz/pp.html", retrieved by June 2003.

[3] J Merlin and B Chapman, "High Performance FORTRAN 2.0", Online tutorial, 1997

[4] High Performance Fortran Forum. "High Performance Fortran Language Specification version 1.0". Sci.Prog, special issue, 1993. Available at http://www.crpc.rice.edu/HPFF or http://www.vcpc.univie.ac.at/information/mirror/HPFF/hpf1/

[5] ZPL, "http://www.cs.washington.edu/research/zpl", University of Washington, retrieved by June 2003.

[6] Parallelism, "http://www.ncsc.org/training/materials/mpicourse/mpi_mppmintro/tsld008.htm", North Carolina Supercomputing Centre, retrieved by June 2003.

[7] "What is Parallelism", "http://mrccs.man.ac.uk/hpctec/courses/HPC/hpc_11.html", A Grant, S Ramsden, retrieved by June 2003.

[8] Parallelism, "http://www.unlv.edu/Writing_Center/Parallelism.htm", Scott Nesbitt, retrieved by June 2003.

[9] http://www-unix.mcs.anl.gov/dbapp/text/book.html, "Designing and Building Parallel Programs, L Foster, retrieved by June 2003.

[10] K Hammond and G Michaelson, "Research Directions in Parallel Functional Programming", Springer, 1999

[11] S Thompson, "Haskell The Craft of Functional Programming", Addison-wesley, 1996

[12] The Haskell Home Page, "http://www.haskell.org", retrieved by May 2003

[13] "Standard ML of New Jersey", "http://www. smlnj.org", retrieved by June 2003

[14] Association of Lisp Users, "http://www.lisp.org", retrieved by June 2003

[15] PTL, The Design and Implementation of Glasgow distributed Haskell

[16] L Foster, "7.1 Data Parallelism", http://www-unix.mcs.anl.gov/dbpp/text/node83.html, retrieved by June 2003

[17] K Hammond and G Michaelson, "Research Directions in Parallel Functional Programming", Page 191, Springer, 1999

[18] K Hammond and G Michaelson, "Research Directions in Parallel Functional Programming", Page 76, Springer, 1999

[19] K Hammond and G Michaelson, "Research Directions in Parallel Functional Programming", Chapter 3, Springer, 1999

[20] Control Parallelism, "http://www.cs.umass.edu/~weems/CmpSci635/Lecture13/L13.10.html", Chip Weems, retrieved by June 2003.

[21] P W Trinder, “GpH - A Parallel Functional Language”, http://www.macs.hw.ac.uk/~dsg/gph/docs/Gentle-GPH/sec-gph.html

[22] R.F. Pointon, P.W. Trinder, and H-W. Loidl, “The Design and Implementation of Glasgow distributed Haskell”, 2000

[28] T L. Sterling, J Salmon, D J. Becher and D F. Savarese, “How to Build a Beowulf, A Guide to the Implementation and Application of PC Clusters”, The MIT Press, Second Printing, 1999, Page 9

[29] Ian Foster, “Designing and Building Parallel Programs”, “http://www-unix.mcs.anl.gov/dbpp/text/node11.html”, retrieved by June 2003.

[30] Thinking Machines Corporation, “Super Computing and Parallelism”, “www.cs.berkeley.edu/~demmel/cs267-1995/cm5docs/tech-summary/1-Parallel.ps.gz”, retrieved by September 2003.

[31] R Pena and F Rubio, “Parallel Functional Programming at Two Levels of Abstraction”, Paper

[32] Michael Hanus, “A Truly Integrated Functional Logic Language”, “http://www.informatik.uni-kiel.de/~mh/curry/”, retrieved by September 2003.

[33] MANUEL M. T. CHAKRAVARTY, “Distributed Haskell aka Goffin”, “http://www.cse.unsw.edu.au/~chak/goffin/”, retrieved by September 2003.

[34] P.W Trinder, K Hammond, H.W Loidl and S.L Peyton Jones, “Algorithm + Strategy = Parallelism”, 1993

Appendix

A Map Skeletons

A.1: GdHParMap (Regular Data)

```
module Main(main) where


import Distributed

import Strategies


mapMV :: (a->IO b) -> [a] -> IO ()

mapMV f ls = do;mapM f ls;return () -- a mapM which throws away its result


-------------------------------------------------------------


createMVarList :: Int -> MVar [MVar a] -> IO ()

createMVarList 0 ms = return ()

createMVarList n ms = do

              b <- isEmptyMVar ms

              if (b)

                then

                     do
```

```
                  mVar <- newEmptyMVar
                  putMVar ms ([mVar])
                  createMVarList (n-1) ms
              else
                  do
                  mVar <- newEmptyMVar
                  mVars <- takeMVar ms
                  putMVar ms (mVar:mVars)
                  createMVarList (n-1) ms
-----------------------------------------------------------------
getResult :: [MVar a] -> IO [a]
getResult [] = return []
getResult (x:xs) = do
            rs <- takeMVar x
            rss <- getResult xs
            return (rs:rss)
-----------------------------------------------------------------


repeatPE :: Int -> [a] -> [a]
repeatPE 0 _ = []
repeatPE _ [] = []
repeatPE n pes
```

```
    | n < 0 = pes
    | (length pes) >= n =pes
    | otherwise = pes ++ (repeatPE (n - (length pes)) pes)
------------------------------------------------------------
--parMap


gdhParMap :: NFData b => (a->b) -> [a] -> IO [b]
gdhParMap f ls =do
        ps <- allPEId           -- get all the PEs
        let n = length ps
        if n < 2        -- only 1 pe
         then
            do
            let rs = map f ls
            return rs `demanding` rnf rs
         else
            do
            let ips = repeatPE (length ls) ps
            let remote te mVar = do
                putMVar mVar te `demanding` rnf te
                return ()
            mList <- newEmptyMVar
        createMVarList (length ls) mList
        ms <- takeMVar mList
        let zs = zip3 ips ls ms
        let work (pe,ts,mVar) = do
                let te = f ts
                rforkIO (remote te mVar) pe
        mapMV work zs
        rs <- getResult ms
        return rs

-- ------------------------------------------------------------------
fib :: Int -> Int
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2) + 1

---------------------------------------------------------------------

main = do
   rs <- gdhParMap fib (replicate 100 30)
   print rs
```

A.2: GdHFarmMap (Regular Data)

```
module Main(main) where


import Distributed

import Strategies


mapMV :: (a->IO b) -> [a] -> IO ()

mapMV f ls = do;mapM f ls;return () -- a mapM which throws away its result


--------unshuffle function---------------------

unshuffle :: Int -> [a] -> [[a]]

unshuffle n [] = replicate n []

unshuffle n xs

        | len < n = map (:[]) xs ++ replicate (n-len) []

        | otherwise = zipWith (:) (take n xs) xss

                where xss = unshuffle n (drop n xs)

                    len = length xs


--------shuffle function-----------------------

shuffle :: [[b]] -> [b]

shuffle [] = []

shuffle xss = (map head nonEmptyxss) ++ shuffle (map tail nonEmptyxss)
```

```
      where nonEmptyxss = filter (not.null) xss

--------------------------------------------------------------

createMVarList :: Int -> MVar [MVar a] -> IO ()
createMVarList 0 ms = return ()
createMVarList n ms = do
        b <- isEmptyMVar ms
        if (b)
         then
            do
            mVar <- newEmptyMVar
            putMVar ms ([mVar])
            createMVarList (n-1) ms
         else
            do
            mVar <- newEmptyMVar
            mVars <- takeMVar ms
            putMVar ms (mVar:mVars)
            createMVarList (n-1) ms
--------------------------------------------------------------
getResult :: [MVar a] -> IO [a]
getResult [] = return []

getResult (x:xs) = do

        rs <- takeMVar x

        rss <- getResult xs

        return (rs:rss)

--------------------------------------------------------------

--farm


gdhFarmMap :: NFData b => (a->b) -> [a] -> IO [b]

gdhFarmMap f ls =do

        ps <- allPEId            -- get all the PEs

        let n = length ps

        let ts = unshuffle n ls   -- build up a list of task lists

        let remote te mVar = do

                putMVar mVar te `demanding` rnf te

                return ()

        mList <- newEmptyMVar

        createMVarList n mList

        ms <- takeMVar mList

        let zs = zip3 ps ts ms

        let work (pe,ts,mVar) = do

                let te = map f ts
```

```
                    rforkIO (remote te mVar) pe
          mapMV work zs
          rs <- getResult ms
          return (shuffle rs)

-- ------------------------------------------------------------------
fib :: Int -> Int
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2) + 1

---------------------------------------------------------------------

main = do
     rs <- gdhFarmMap fib (replicate 100 30)
     print rs
```

A.3: GpHParMap (Regular Data)

```
module Main(main) where

import Parallel
import Strategies


---------------------------------------------------

--fib

fib :: Int->Int

fib 0 = 1

fib 1 = 1

fib n = f (n-1) + f (n-2) + 1


---------------------------------------------------

--main

main = print ( parMap rnf f (replicate 100 30) )
```

A.4: GpHChunkMap (Regular Data)

```
module Main(main) where


import Strategies


---------------------------------------------------------------

--chunkMap

chunkMap :: Int -> Strategy b -> (a->b) -> [a] -> [b]

chunkMap n strat f xs = map f xs `using` parListChunk n strat
```

```
-----------------------------------------------------------------

--fib

fib :: Int -> Int

fib 0 = 1

fib 1 = 1

fib n = fib(n-1) + fib(n-2) + 1


main =  print (chunkMap 6 rnf fib (replicate 100 30))
```

A.5: GpHShuMap (Regular Data)

```
module Main(main) where



import Parallel

import System

import Strategies



----------------------------

--unshuffle function

unshuffle :: Int -> [a] -> [[a]]

unshuffle n [] = replicate n []

unshuffle n xs
```

```
          | len < n = map (:[]) xs ++ replicate (n-len) []
          | otherwise = zipWith (:) (take n xs) xss
                      where xss = unshuffle n (drop n xs)
                            len = length xs


-------------------------------
--shuffle function
shuffle :: [[b]] -> [b]
shuffle [] = []
shuffle xss = (map head nonEmptyxss) ++ shuffle (map tail nonEmptyxss)
            where nonEmptyxss = filter (not.null) xss


-------- ---------------------------
--fibonacci  function
f :: Int -> Int
f 0 = 1
f 1 = 1
f n = f (n-1) + f (n-2)+ 1


----------------------------------
--map function
shuMap :: (Strategy b) -> Int -> (a->b) -> [a] -> [b]
shuMap s _ f [] = []

shuMap s 0 f xs = map f xs

shuMap s 1 f xs = map f xs

shuMap s n f xs = shuffle (parMap (seqList s) (map f) xn)

            where

             xn = unshuffle n xs



----------------------------------

--main function

main = print (shuMap rnf 6 f (replicate 100 30))
```

A.6: GdHWorkPool (Regular Data)

```
module Main(main) where



import Distributed

import MutSig

import Bounded

import PrelIOBase (unsafePerformIO)

import Strategies



import IO

debug :: String -> IO ()
debug s = return ()

-- -----------------------------------------------------------------------
--wrapper

data NFData a => Task a =  Task (MVar a) a

parBody :: NFData b => ([Task b] -> IO ()) -> (a->b) -> [a] -> [b]
parBody way f ls = unsafePerformIO (
    do
    debug "parBody"
    let ms = map f ls          -- the map
    ts <- mapM newTask ms      -- allocate task for each list item
    forkIO (way ts)            -- start evaluating all tasks
    let rs = map waitTask ts  -- method to access each tasks result
    return rs )

newTask :: NFData a => a -> IO (Task a)
newTask l = do
    m <- newEmptyMVar
    return (Task m l)
```

```
doTask :: NFData a => Task a -> IO ()
doTask (Task m v) = do
    debug "doTask"
    putMVar m v `demanding` rnf v

waitTask :: NFData a => Task a -> a
waitTask (Task m _) = unsafePerformIO (
    do
    r <- takeMVar m
    putMVar m r       -- in case we fetch it again
    return r )

mapMV :: (a->IO b) -> [a] -> IO ()
mapMV f ls = do;mapM f ls;return () -- a mapM which throws away its result

-- ---------------------------------------------------------------------
--workpool

workPool :: NFData b => (a->b) -> [a] -> [b]
workPool f ls = parBody way f ls
    where way ts = do
        ps <- allPEId           -- get all the PEs
```

```
c <- newMutex ts          -- protect the list of tasks

let getTaskRmt = lock c safe
    where
      safe []     = return ([],Nothing)
      safe (t:ts) = return (ts,(Just t))

let pre = 2

let worker = do
    b <- newBound pre

    let driver = do
        jt <- getTaskRmt
        writeBound b jt
        case jt of
          (Just t)  -> driver
          (Nothing) -> return ()

    let slave = do
        jt <- readBound b
        case jt of
```

```
                 (Just t) -> do
                      doTask t
                      slave
                 (Nothing) -> return ()

            forkIO driver
            slave

       mapMV (\p -> rforkIO worker p) ps  -- create the workers

-- ---------------------------------------------------------------
fib :: Int -> Int
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)+ 1

-----------------------------------------------------------------

main = print (workPool fib (replicate 100 30))
```

A.7: GdHParMap (Irregular Data)

```
module Main(main) where



import Distributed

import Strategies



mapMV :: (a->IO b) -> [a] -> IO ()

mapMV f ls = do;mapM f ls;return () -- a mapM which throws away its result



-----------------------------------------------------------------



createMVarList :: Int -> MVar [MVar a] -> IO ()

createMVarList 0 ms = return ()

createMVarList n ms = do

              b <- isEmptyMVar ms

              if (b)

                then

                     do

                     mVar <- newEmptyMVar

                     putMVar ms ([mVar])

                     createMVarList (n-1) ms

                else
                do
                mVar <- newEmptyMVar
                mVars <- takeMVar ms
                putMVar ms (mVar:mVars)
                createMVarList (n-1) ms
-------------------------------------------------------------
getResult :: [MVar a] -> IO [a]
getResult [] = return []
getResult (x:xs) = do
          rs <- takeMVar x
          rss <- getResult xs
          return (rs:rss)
-------------------------------------------------------------


repeatPE :: Int -> [a] -> [a]
repeatPE 0 _ = []
repeatPE _ [] = []
repeatPE n pes
     | n < 0 = pes
     | (length pes) >= n =pes
     | otherwise = pes ++ (repeatPE (n - (length pes)) pes)
------------------------------------------------------------
```

```
--farm

gdhParMap :: NFData b => (a->b) -> [a] -> IO [b]
gdhParMap f ls =do
        ps <- allPEId          -- get all the PEs
        let n = length ps
        if n < 2       -- only 1 pe
         then
            do
            let rs = map f ls
            return rs `demanding` rnf rs
         else
            do
            let ips = repeatPE (length ls) ps
            let remote te mVar = do
                   putMVar mVar te `demanding` rnf te
                   return ()
            mList <- newEmptyMVar
            createMVarList (length ls) mList
            ms <- takeMVar mList
            let zs = zip3 ips ls ms
            let work (pe,ts,mVar) = do
```

```
                  let te = f ts
                  rforkIO (remote te mVar) pe
           mapMV work zs
           rs <- getResult ms
           return rs

-- ----------------------------------------------------------------------
fib :: Int -> Int
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2) + 1

------------------------------------------------------------------------

main = do
    rs <- gdhParMap fib dataL
    print rs
    where
      dataL = [28, 27, 30, 32, 28, 29, 30, 25, 28, 31, 26, 27, 25, 29, 25, 27, 29, 26, 29,
31,
          25, 28, 29, 27, 30, 32, 28, 29, 30, 25, 28, 31, 26, 26, 27, 25, 29, 25, 27, 29,
          29, 30, 25, 28, 31, 26, 26, 27, 29, 30, 25, 28, 31, 26, 27, 26, 27, 29, 30, 25,
        27, 29, 30, 25, 28, 31, 26, 27, 26, 27, 29, 27, 30, 32, 28, 29, 30, 25, 32, 21,
        26, 27, 26, 27, 29, 27, 30, 32, 29, 30, 25, 28, 31,
        26, 26, 27, 31, 29, 25, 28]
```

A.8: GdHFarmMap (Irregular Data)

```
module Main(main) where


import Distributed
import Strategies


mapMV :: (a->IO b) -> [a] -> IO ()
mapMV f ls = do;mapM f ls;return () -- a mapM which throws away its result


--------unshuffle function--------------------
unshuffle :: Int -> [a] -> [[a]]
unshuffle n [] = replicate n []
unshuffle n xs
    | len < n = map (:[]) xs ++ replicate (n-len) []
    | otherwise = zipWith (:) (take n xs) xss
        where xss = unshuffle n (drop n xs)
          len = length xs
```

```
--------shuffle function-----------------------

shuffle :: [[b]] -> [b]

shuffle [] = []

shuffle xss = (map head nonEmptyxss) ++ shuffle (map tail nonEmptyxss)

        where nonEmptyxss = filter (not.null) xss


-------------------------------------------------------------


createMVarList :: Int -> MVar [MVar a] -> IO ()

createMVarList 0 ms = return ()

createMVarList n ms = do

         b <- isEmptyMVar ms

         if (b)

          then

             do

             mVar <- newEmptyMVar

             putMVar ms ([mVar])

             createMVarList (n-1) ms

          else

             do

             mVar <- newEmptyMVar

             mVars <- takeMVar ms
```

```
               putMVar ms (mVar:mVars)

               createMVarList (n-1) ms

------------------------------------------------------------

getResult :: [MVar a] -> IO [a]

getResult [] = return []

getResult (x:xs) = do

          rs <- takeMVar x

          rss <- getResult xs

          return (rs:rss)

-------------------------------------------------------------

--farm


gdhFarmMap :: NFData b => (a->b) -> [a] -> IO [b]

gdhFarmMap f ls =do

          ps <- allPEId              -- get all the PEs

          let n = length ps

          let ts = unshuffle n ls   -- build up a list of task lists

          let remote te mVar = do

                    putMVar mVar te `demanding` rnf te

                    return ()

          mList <- newEmptyMVar

          createMVarList n mList
```

```
        ms <- takeMVar mList

        let zs = zip3 ps ts ms

        let work (pe,ts,mVar) = do

                let te = map f ts

                rforkIO (remote te mVar) pe

        mapMV work zs

        rs <- getResult ms

        return (shuffle rs)


-- ------------------------------------------------------------------

fib :: Int -> Int

fib 1 = 1

fib 2 = 1

fib n = fib (n-1) + fib (n-2) + 1


--------------------------------------------------------------------



main = do

    rs <- gdhFarmMap fib dataL

    print rs

    where

      dataL = [28, 27, 30, 32, 28, 29, 30, 25, 28, 31, 26, 27, 25, 29, 25, 27, 29, 26, 29,
31,
        25, 28, 29, 27, 30, 32, 28, 29, 30, 25, 28, 31, 26, 26, 27, 25, 29, 25, 27, 29,
        29, 30, 25, 28, 31, 26, 26, 27, 29, 30, 25, 28, 31, 26, 27, 26, 27, 29, 30, 25,
        27, 29, 30, 25, 28, 31, 26, 27, 26, 27, 29, 27, 30, 32, 28, 29, 30, 25, 32, 21,
        26, 27, 26, 27, 29, 27, 30, 32, 29, 30, 25, 28, 31,
        26, 26, 27, 31, 29, 25, 28]
```

A.9: GpHParMap (Irregular Data)

```
module Main(main) where


import Parallel



------------------------------------------
--fib
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib(n-1) +fib(n-2) +1


----------------------------------------
-- map function
```

```
parMap :: (a->b) -> [a] -> [b]

parMap f [] = []

parMap f (x:xs) = fx `par` fxs `seq` (fx:fxs)

              where

                fx = f x

                fxs = parMap f xs



---------------------------------------

-- main function

main = print ( parMap f dataL )

       where

        dataL = [28, 27, 30, 32, 28, 29, 30, 25, 28, 31, 26, 27, 25, 29, 25, 27, 29, 26, 29, 31,

               25, 28, 29, 27, 30, 32, 28, 29, 30, 25, 28, 31, 26, 26, 27, 25, 29, 25, 27, 29,

               29, 30, 25, 28, 31, 26, 26, 27, 29, 30, 25, 28, 31, 26, 27, 26, 27, 29, 30, 25,

               27, 29, 30, 25, 28, 31, 26, 27, 26, 27, 29, 27, 30, 32, 28, 29, 30, 25, 32, 21,

               26, 27, 26, 27, 29, 27, 30, 32, 29, 30, 25, 28, 31, 26, 26, 27, 31, 29, 25, 28]
```

A.10: GpHChunkMap (Irregular Data)

```
module Main(main) where



import Strategies
```

```
chunkMap :: Int -> Strategy b -> (a->b) -> [a] -> [b]

chunkMap n strat f xs = map f xs `using` parListChunk n strat



fib :: Int -> Int

fib 0 = 1

fib 1 = 1

fib n = fib(n-1) + fib(n-2) + 1



main =  do

    let tmp =  chunkMap 6 rnf fib dataL

    print tmp

    where

     dataL = [28, 27, 30, 32, 28, 29, 30, 25, 28, 31, 26, 27, 25, 29, 25, 27, 29, 26, 29, 31,

         25, 28, 29, 27, 30, 32, 28, 29, 30, 25, 28, 31, 26, 26, 27, 25, 29, 25, 27, 29,

         29, 30, 25, 28, 31, 26, 26, 27, 29, 30, 25, 28, 31, 26, 27, 26, 27, 29, 30, 25,

         27, 29, 30, 25, 28, 31, 26, 27, 26, 27, 29, 27, 30, 32, 28, 29, 30, 25, 32, 21,

         26, 27, 26, 27, 29, 27, 30, 32, 29, 30, 25, 28, 31, 26, 26, 27, 31, 29, 25, 28]
```

A.11: GpHShuMap (Irregular Data)

```
module Main(main) where



import Parallel
import System

import Strategies


----------------------------

--unshuffle function

unshuffle :: Int -> [a] -> [[a]]

unshuffle n [] = replicate n []

unshuffle n xs

        | len < n = map (:[]) xs ++ replicate (n-len) []

        | otherwise = zipWith (:) (take n xs) xss

                  where xss = unshuffle n (drop n xs)

                       len = length xs


------------------------------

--shuffle function

shuffle :: [[b]] -> [b]

shuffle [] = []

shuffle xss = (map head nonEmptyxss) ++ shuffle (map tail nonEmptyxss)

          where nonEmptyxss = filter (not.null) xss


-----------------------------------

--fibonacci  function
```

```
fib :: Int -> Int

fib 0 = 1

fib 1 = 1

fib n = f (n-1) + f (n-2)


----------------------------------

--map function

shuMap :: (Strategy b) -> Int -> (a->b) -> [a] -> [b]

shuMap s _ f [] = []

shuMap s 0 f xs = map f xs

shuMap s 1 f xs = map f xs

shuMap s n f xs = shuffle (parMap (seqList s) (map f) xn)

             where

              xn = unshuffle n xs


---------------------------------

--main function

main =  print shuMap rnf 6 f dataL

       where

         dataL = [28, 27, 30, 32, 28, 29, 30, 25, 28, 31, 26, 27, 25, 29, 25, 27, 29, 26, 29, 31,

                25, 28, 29, 27, 30, 32, 28, 29, 30, 25, 28, 31, 26, 26, 27, 25, 29, 25, 27, 29,

                29, 30, 25, 28, 31, 26, 26, 27, 29, 30, 25, 28, 31, 26, 27, 26, 27, 29, 30, 25,
27, 29, 30, 25, 28, 31, 26, 27, 26, 27, 29, 27, 30, 32, 28, 29, 30, 25, 32, 21,

26, 27, 26, 27, 29, 27, 30, 32, 29, 30, 25, 28, 31, 26, 26, 27, 31, 29, 25, 28]
```

A.12: GdHWorkPool (Irregular Data)

```
module Main(main) where


import Distributed

import MutSig

import Bounded

import PrelIOBase (unsafePerformIO)

import Strategies


import IO

debug :: String -> IO ()

debug s = return ()


-- ---------------------------------------------------------------------

--wrapper


data NFData a => Task a =  Task (MVar a) a


parBody :: NFData b => ([Task b] -> IO ()) -> (a->b) -> [a] -> [b]
parBody way f ls = unsafePerformIO (
    do
    debug "parBody"
    let ms = map f ls         -- the map
    ts <- mapM newTask ms     -- allocate task for each list item
    forkIO (way ts)           -- start evaluating all tasks
    let rs = map waitTask ts  -- method to access each tasks result
    return rs )

newTask :: NFData a => a -> IO (Task a)
newTask l = do
    m <- newEmptyMVar
    return (Task m l)

doTask :: NFData a => Task a -> IO ()
doTask (Task m v) = do
    debug "doTask"
    putMVar m v `demanding` rnf v

waitTask :: NFData a => Task a -> a
waitTask (Task m _) = unsafePerformIO (
    do
```

```
   r <- takeMVar m
   putMVar m r    -- in case we fetch it again
   return r )

mapMV :: (a->IO b) -> [a] -> IO ()
mapMV f ls = do;mapM f ls;return () -- a mapM which throws away its result

-- ------------------------------------------------------------------
--workpool

workPool :: NFData b => (a->b) -> [a] -> [b]
workPool f ls = parBody way f ls
   where way ts = do
      ps <- allPEId            -- get all the PEs
      c <- newMutex ts           -- protect the list of tasks

      let getTaskRmt = lock c safe
         where
          safe []     = return ([],Nothing)
          safe (t:ts) = return (ts,(Just t))

      let pre = 2
let worker = do
    b <- newBound pre

    let driver = do
        jt <- getTaskRmt
        writeBound b jt
        case jt of
          (Just t)  -> driver
          (Nothing) -> return ()

    let slave = do
        jt <- readBound b
        case jt of
          (Just t) -> do
              doTask t
              slave
          (Nothing) -> return ()

    forkIO driver
    slave
```

```
        mapMV (\p -> rforkIO worker p) ps  -- create the workers


-- -------------------------------------------------------------------

fib :: a -> b

fib 1 = 1

fib 2 = 1

fib n = fib (n-1) + fib (n-2)


---------------------------------------------------------------------


main = print (workPool fib (replicate 100 30))
```

B Divide and Conquer Skeletons

B.1 GpHDC

```
module Main(main) where


import Strategies


----------------------------------------------------------

--GpHDC
```

```
divConq :: (a->b) -> a -> (a->Bool) -> (b->b->b) -> (a->Bool) -> (a->(a,a)) ->b

divConq f arg threshold conquer divisible divide

  | not (divisible arg) = f arg

  | otherwise      = conquer left right `using` strategy

      where

          (lt,rt) = divide arg

          left    = divConq f lt threshold conquer divisible divide

          right   = divConq f rt threshold conquer divisible divide

          strategy= \_ -> if threshold arg

                         then (seqPair rwhnf rwhnf) $ (left,right)

                         else (parPair rwhnf rwhnf) $ (left,right)


----------------------------------------------------------------------------

--fib

fib :: Int -> Int

fib 1 = 1

fib 2 = 1

fib n = fib (n-1) + fib (n-2)


---------------------------------------------------------------------------

--pfib

pfib :: Int -> Int
pfib n = divConq fib n threshold conq divisible divide
      where
        threshold n = n <= 30
        divide n    = (n-1, n-2)
        divisible n = n > 30
        conq m n = m + n + 1


main = print (pfib 42)
```

B.2 GdHDC

```
module Main(main) where


import Distributed
import Strategies
--------------------------------------------------------------------------------
--fib
fib :: Int -> Int
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2) + 1
--------------------------------------------------------------------------------
--getNextPE
```

```
getNextPE :: MVar [PEId]-> IO [PEId]
getNextPE mv = do
    ms <- takeMVar mv
    if length ms < 1
     then
        do
        putMVar mv []
        return []
     else
        do
         let headPE = head ms
         let ms1 =  drop 1 ms
         putMVar mv ms1
         return (headPE:[])
---------------------------------------------------------------------------------
--divConqToIp
divConqToIp ::  NFData b => (a->b) -> a -> (a->Bool) -> (b->b->b) -> (a->(a,a))
-> MVar [PEId] -> MVar [PEId] -> MVar b ->IO ()
divConqToIp f arg threshold conquer divide mPEs mAllPE mrs
        | threshold arg =do
                let rs = f arg
```

```
              putMVar mrs rs `demanding` rnf rs
              return ()
| otherwise = do
      nextPEl <- getNextPE mPEs
      if length nextPEl < 1
        then
            do
            let rs = f arg
            putMVar mrs rs `demanding` rnf rs
            return ()
        else
            do
            mL <- newEmptyMVar
            mR <- newEmptyMVar
            right mR (head nextPEl)
            left mL
              mrsR <- takeMVar mR
            mrsL <- takeMVar mL
            let te = conquer mrsL mrsR
            putMVar mrs te `demanding` rnf te
            return ()
            where
            (lt,rt) = divide arg
            left ml = do
               sfPE <- myPEId
               divConqToIp f lt threshold conquer divide mPEs mAllPE ml
            right mr nextPE = do
               rforkIO (divConqToIp f rt threshold conquer divide mPEs
mAllPE mr) nextPE
--------------------------------------------------------------------------------
--deleOwn


delOwn :: Eq a => a -> [a] -> [a]
delOwn own items
   | length items < 1 = []
   | otherwise =
      if((head items) == own)
       then
          drop 1 items
       else
          delOwn own ((tail items) ++  (take 1 items))


--------------------------------------------------------------------------------
--divConq
```

```
divConq :: NFData b => (a->b) -> a -> (a->Bool) -> (b-> b-> b) -> (a->(a,a)) ->IO
b
divConq f arg threshold conquer divide
            | threshold arg = do
                return (f arg)
            | otherwise = do
                ps <- allPEId
                myPE <- myPEId
                let psb = delOwn myPE ps
                mPEs <- newEmptyMVar
                mAllPE <- newEmptyMVar
                mrs <- newEmptyMVar
                putMVar mPEs psb
                putMVar mAllPE ps
                rforkIO (divConqToIp f arg threshold conquer divide mPEs mAllPE
mrs) myPE
                rs <- takeMVar mrs
                return rs
--------------------------------------------------------------------------------------------------------
--pFib
pFib :: Int -> IO Int
pFib n = divConq fib n threshold conq divide
        where
        threshold n = n <= 30
        divide n = (n-1, n-2)
        conq l r = l + r + 1
-----------------------------------------------------------------------------------------------

main = do
     rs <- pFib 42
     print rs
```