

Benchmarking a New Parallel Language Implementation

Malcolm Watt

Supervisor: Phil Trinder

Final Year Dissertation

MSc Computer Services Management

Heriot-Watt University

Declaration

I, Malcolm Watt, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it or the works of other authors in any form (e.g. ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of references employed is included.

Signed:

Date:

Abstract

The advancement and emergence of parallel technologies has enabled new areas and problems to be solved which would have been impossible under sequential computing. The growth in parallel architectures such as multi-core and many-core systems has enabled commercial and scientific industries to exploit the full use of parallelism. However, to truly take advantage of parallelism, a high level of dependency rests on the programming of the parallel technology. There is a large range of parallel languages used to program these parallel systems each with their own strengths and weaknesses. This dissertation provides a performance comparison of three parallel programming implementations on a set of parallel program benchmarks. The three languages used are Glasgow Parallel Haskell, Eden and the newly implemented Glasgow Distributed Haskell. The results help identify the performance of the new GdH- implementation against two similar parallel Haskell implementations and help create further work on the GdH- implementation.

This dissertation was achieved through carrying out a literature survey on parallelism exploring both parallel programming and the three individual parallel functional languages used in this project.

Three benchmark programs were developed for each of the three languages which tested three different parallel paradigms on each language.

These benchmarks were executed on two different multi-core architectures to calculate the median runtime, speedup and variability on each implementation.

Once all programs were executed the data was collated to give a full comparison of the parallel languages.

This comparison helped to provide a conclusive analysis of the GdH- implementation's current true performance.

Acknowledgements

First of all I would like to thank my supervisor Phil Trinder for his support and guidance throughout this project helping me develop my knowledge of parallel languages as well as my presentation and research skills.

I would also like to thank Patrick Maier for his invaluable help with parallel programming languages and presentation of results.

Finally I would like to thank my current employers at Advocates Pharmaceuticals for supporting and encouraging me to do this MSc programme and ultimately getting the opportunity to work on this project.

Contents

Declaration.....	i
Abstract.....	ii
Acknowledgements.....	iii
Contents.....	iv
List of Figures	vi
List of Tables	vii
1 Introduction	10
1.1 Aim and Objectives	10
1.2 Contributions	10
2 Literature Survey.....	13
2.1 Introduction	13
2.2 Parallelism.....	13
2.3 Parallel Architectures.....	14
2.3.1 Single Instruction Single Data stream	14
2.3.2 Single Instruction Multiple Data stream	15
2.3.3 Multiple Instruction Single Data	16
2.3.4 Multiple Instruction Multiple Data stream	16
2.3.5 Shared Memory	17
Distributed Memory.....	18
Hybrid Distributed-Shared Memory	19
2.4 Parallel Trends.....	20
2.4.1 Multi-Core	20
2.4.2 Many-Core	21
2.5 Parallel Programming.....	22
2.6 Parallel Functional Languages.....	23
2.6.1 Glasgow Parallel Haskell	23
2.6.2 Eden	26
2.6.3 GdH-	28

2.7 Benchmarking	29
3 Experiment Design	32
3.1 Hardware Specification	32
3.2 Structure of Measurements.....	33
3.3 Test Programs	35
3.3.1 Fibonacci	35
3.3.2 Sum of Totients.....	36
3.3.3 Queens	37
4. Experiment Results	39
4.1 Fibonacci	39
4.1.1 Runtime.....	39
4.1.2 Speedup	42
4.1.3 Variability	44
4.2 Sum of Totients	47
4.2.1 Runtime.....	47
4.2.2 Speedup	49
4.3.2 Variability	51
4.3 Queens	54
4.1.3 Runtime.....	54
4.2.3 Speedup	56
4.3.3 Variability	58
4.4 Results Summary.....	61
4.5 Discussion of results.....	63
5. Conclusion	65
5.1 Summary	65
5.2 Limitations.....	66
5.3 Future Work	67
References	
Appendix A – Program Listings	
Appendix B – Full Results	

List of Figures

2.1	Single Instruction Single Data stream	15
2.2	Single Instruction Multiple Data stream.....	16
2.3	Multiple Instruction Single Data stream.....	16
2.4	Multiple Instruction Multiple Data stream	17
2.5	UMA Shared Memory.....	18
2.6	NUMA Shared Memory.....	18
2.7	Distributed Memory	19
2.8	Hybrid Distributed-Shared Memory.....	20
2.9	Ring Process Structure	27
4.1	Runtime of up to 8-cores Fibonacci program on lxpara2.....	39
4.2	Runtime of up to 8-cores Fibonacci program on Beowulf Cluster	40
4.3	Runtime of up to 224-cores Fibonacci program on Beowulf Cluster	41
4.4	Absolute Speedup of up to 8-cores Fibonacci program on lxpara2.....	42
4.5	Absolute Speedup of up to 8-cores Fibonacci program on Beowulf Cluster.....	42
4.6	Relative Speedup of up to 224-cores Fibonacci program on Beowulf Cluster	44
4.7	Variance of Fibonacci program on up to 8-cores lxpara2 and Beowulf cluster	45
4.8	Variance of Fibonacci program on up to 224-cores Beowulf cluster	46
4.9	Runtime of up to 8-cores Sum of Totients program on lxpara2.....	47
4.10	Runtime of up to 8-cores Sum of Totients program on Beowulf Cluster	47
4.11	Runtime of up to 224-cores Sum of Totients program on Beowulf Cluster	48
4.12	Absolute Speedup of up to 8-cores Sum of Totients program on lxpara2.....	49
4.13	Absolute Speedup of up to 8-cores Sum of Totients program on Beowulf Cluster	50
4.14	Relative Speedup of up to 224-cores Sum of Totients program on Beowulf Cluster ..	51
4.15	Variance of Sum of Totients program on up to 8-cores lxpara2 and Beowulf cluster ..	52
4.16	Variance of Sum of Totients program on up to 224-cores Beowulf cluster	53

4.17	Runtime of up to 8-cores Queens program on Ixpara2.....	54
4.18	Runtime of up to 8-cores Queens program on Beowulf Cluster.....	54
4.19	Runtime of up to 224-cores Queens program on Beowulf Cluster	55
4.20	Absolute Speedup of up to 8-cores Queens program on Ixpara2	56
4.21	Absolute Speedup of up to 8-cores Queens program on Beowulf Cluster.....	57
4.22	Relative Speedup of up to 224-cores Queens program on Beowulf Cluster	58
4.23	Variance of Queens program on up to 8-cores Ixpara2 and Beowulf cluster	59
4.24	Variance of Queens program on up to 224-cores Beowulf cluster	60

List of Tables

4.1	Runtime of up to 8-cores Fibonacci program on lxpara2.....	40
4.2	Runtime of up to 8-cores Fibonacci program on Beowulf Cluster.....	41
4.3	Runtime of up to 224-cores Fibonacci program on Beowulf Cluster	41
4.4	Absolute Speedup of up to 8-cores Fibonacci program on lxpara2.....	43
4.5	Absolute Speedup of up to 8-cores Fibonacci program on Beowulf Cluster.....	43
4.6	Relative Speedup of up to 224-cores Fibonacci program on Beowulf Cluster	44
4.7	Variance of Fibonacci program on lxpara2	45
4.8	Variance of Fibonacci program on Beowulf Cluster	45
4.9	Variance of Fibonacci program on up to 224-cores Beowulf Cluster	46
4.10	Runtime of up to 8-cores Sum of Totients program on lxpara2	48
4.11	Runtime of up to 8-cores Sum of Totients program on Beowulf Cluster	48
4.12	Runtime of up to 224-cores Sum of Totients program on Beowulf Cluster	49
4.13	Absolute Speedup of up to 8-cores Sum of Totients program on lxpara2.....	50
4.14	Absolute Speedup of up to 8-cores Sum of Totients program on Beowulf Cluster	50
4.15	Relative Speedup of up to 224-cores Sum of Totients program on Beowulf Cluster ..	51
4.16	Variance of Sum of Totients program on up to 8-cores lxpara2	52
4.17	Variance of Sum of Totients program on up to 8-cores Beowulf Cluster	52
4.18	Variance of Sum of Totients program on up to 224-cores Beowulf cluster	53
4.19	Runtime of up to 8-cores Queens program on lxpara2.....	55
4.20	Runtime of up to 8-cores Queens program on Beowulf Cluster.....	55
4.21	Runtime of up to 224-cores Queens program on Beowulf Cluster	56
4.22	Absolute Speedup of up to 8-cores Queens program on lxpara2	57
4.23	Absolute Speedup of up to 8-cores Queens program on Beowulf Cluster.....	57
4.24	Relative Speedup of up to 224-cores Queens program on Beowulf Cluster	58
4.25	Variance of Queens program on up to 8-cores lxpara2	59
4.26	Variance of Queens program on up to 8-cores Beowulf cluster	59

4.27	Variance of Queens program on up to 224-cores Beowulf cluster	60
4.28	Comparison of median runtimes	61
4.29	Comparison of speedups	61
4.30	Comparison of variability	62

1 Introduction

1.1 Aim and Objectives

The aim of this dissertation was to analyse and evaluate the potential of a brand new parallel language implementation. Glasgow Distributed Haskell was recently ported onto the EDI framework for parallel architectures and this project involved benchmarking this implementation. Additionally, this project involved comparing the results of the benchmark tests on GdH- with two other Haskell parallel implementations: Glasgow Parallel Haskell and Eden. The performance comparison of the three languages was used to determine the efficiency and execution of GdH-. To accomplish this aim the following four objectives were identified:

- 1.) *Learn GdH-, GpH and Eden*
- 2.) *Develop three test programs in the parallel languages*
- 3.) *Perform parallel comparative benchmarks of the GdH-, Eden and GpH programs*
- 4.) *Compile the results into a report and evaluate results*

1.2 Contributions

This project was achieved through the following steps:

- A literature survey was undertaken on parallelism which featured parallel architectures and parallel programming including a description of the three parallel functional languages used in this project. The literature survey, which can be seen in Chapter 2, also covered benchmarking including measurement techniques.

- Three parallel benchmark programs were developed in three different parallel language implementations. The three languages were all extensions of the Haskell language and were namely: Glasgow Parallel Haskell, Eden, and the new implementation of Glasgow Distributed Haskell. The three benchmarks were parallel implementations of the Fibonacci sequence, the sum of Euler numbers, and the n-queens problem. All benchmark problems were developed to use the same strategy in solving the problems. A description of these benchmarks and the conditions of these experiments can be found in Chapter 3 while the full program listings can be found in Appendix A and the accompanying CD.
- The nine programs were executed on two multi-core architectures. The two architectures were lxpara2, which is an 8 core 32-bit machine, and the Beowulf cluster, which is a 32 node cluster of 64-bit 8-core machines resulting in a maximum of 256-cores. However, for the purposes of this project, all programs were run on 32 bit processor on the Beowulf cluster. Each program was run and sufficient data was collected to calculate the median runtime, variability and speedup of each program on each implementation. A collection of tables and graphs of these results can be found in Chapter 4.
- A comparison was made of the performances of the three languages on the benchmarked problems in Chapter 4 which found:
 - That GdH- currently lags behind the performances of both GpH and Eden in the benchmarks selected. Although it recorded a maximum absolute speedup of 5.8 over 8-cores on the Fibonacci test program, its other results were not as flattering with the next best absolute speedup being 4.2 over 8-cores. It also showed a fair amount of variance recording the highest percentage of the three languages at 84.4% on lxpara2 for the Fibonacci program.
 - GpH was shown to be the most predictable and reliable of the three languages. It recorded the lowest average variance of the three languages, for example in the Fibonacci tests on lxpara2 the highest amount of variability was 2.7%. GpH also gave fairly good speedup results with its best absolute speedup being 5.3 for the Queens program on lxpara2.

- Eden recorded most of the best runtimes in the testing environment. It recorded the lowest runtime of all the benchmarks by solving the Sum of Totients program in 2.5 seconds over 224-cores. It also recorded the best speedup of 5.9 for the Fibonacci program on lxpara2 as well as the best relative speedup of 85.1 for the Sum of Totients program over 224-cores.

2 Literature Survey

2.1 Introduction

In this section I will give a brief background on parallel computing and the languages to be used in this project based on literature researched. This will involve defining what parallelism is, looking at the architectures used, parallel programming, and a brief analysis of benchmarking parallel program implementations.

2.2 Parallelism

Parallel computing is used to increase the speed of execution of a computational problem. It involves using multiple processors to break down a problem into separate parts which can be solved concurrently resulting in a more efficient execution of a given problem. In contrast to sequential computing where a problem is handled by a single processing unit and instructions are executed in a single sequence, parallel computing enables several problems to be solved in a fraction of the time [4]. It also enables more complex or larger problems to be solved where sequential computing would not cope. Using multiple resources, problems can be broken down further into more manageable instructions and executed, resulting in problems outside the scope of sequential computing being solved.

Parallel computing is an emerging technology and its uses are expanding [1]. Generally, it has been used to help solve difficult scientific problems in the domain of applied physics, electrical engineering, bioscience and mathematics. However, many commercial applications are starting to require faster computers with parallel execution. This can involve search engines, pharmaceutical design, data mining and medical diagnosis applications [12]. This is driving the demand for parallel computing and bringing it from the high end of computing towards the mainstream.

However, parallel computing does impose a harder task for developers and programmers. As the scale of the problems increases, so do the issues that programmers must address. The main problem that developers encounter when creating parallel algorithms is with

communication and synchronisation between different subtasks [4]. Issues such as race, deadlocks and mutual exclusion can cause problems which will reverse the purpose of parallel computing and slow down the execution time rather than speed up. It is imperative that the algorithm and programming behind it must be correct for parallel computing to be truly effective.

This results in parallel computing being seen as a rewarding but frustrating area within the domain of computer science. However, increased research and advances in technology are evolving parallel computing and helping to make it easier to realise its full potential [2].

2.3 Parallel Architectures

In Michael J. Flynn's paper *Parallel Architectures* [7] he identifies four classifications of parallel architectures based on the instruction and data streams:

- 1- Single Instruction Single Data stream (SISD)
- 2- Single Instruction Multiple Data stream (SIMD)
- 3- Multiple Instruction Single Data stream (MISD)
- 4- Multiple Instruction Multiple Data stream (MIMD)

Each of these types of architectures invokes a different type of parallelism.

2.3.1 Single Instruction Single Data stream

This category is the most common with it being present in serial computers with a single processor. A single instruction stream is executed on the CPU and a single data stream is used as input during any one clock cycle. This process is deterministic meaning it behaves predictably resulting in the programmer being able to recreate the process easily at any time. Although concurrency does not seem apparent in this type of architecture, there are possible executions of parallelism. One such way is through pipelining, where an instruction can be processed concurrently at different phases. Although this does not achieve concurrency of execution it does achieve concurrency of process. Another form of parallelism in SISD is through superscalar architecture. It implements instruction level

parallelism (ILP) which analyses dependencies between operations within the instruction stream and schedules appropriate operations to execute concurrently. This allows for faster throughput at any given clock rate. Most modern day computers implement SISD architecture and it is the oldest architecture implemented [6].

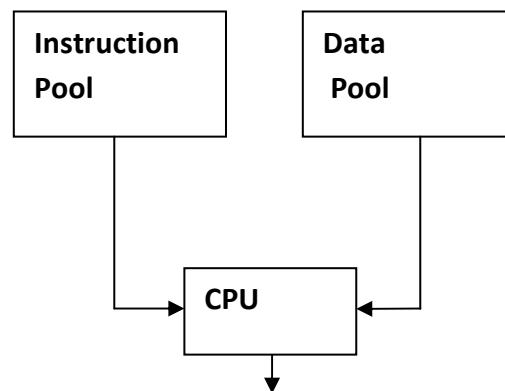


Figure 2.1 Single Instruction Single Data stream

2.3.2 Single Instruction Multiple Data stream

This type of architecture involves processing units executing the same instruction at any different clock cycle but processing units can operate on a different data element. There are two types of processors which have been developed to instigate this architecture are array processors and vector processors. Array processors operate many processor elements on many data elements in parallel whereas a vector processor uses a single processor element operating in sequence on multiple data elements. In the main, much of the development in this area has focussed on vector processors with very little significant work on array architecture, due to lack of market demand and application base [6].

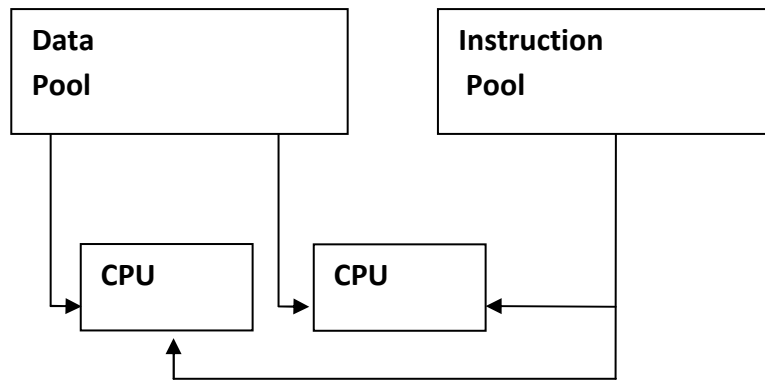


Figure 2.2 Single Instruction Multiple Data stream

2.3.3 Multiple Instruction Single Data

This parallel architecture's basis is that a single data stream is fed into multiple processing units which operate on the data independently. However, SISD and MIMD are seen as more appropriate common data parallel techniques rather than this one. Therefore there have not been many instances of this architecture with it seen as having limited demand but there are still some niche examples of MISD such as the space shuttle flight control computers [6].

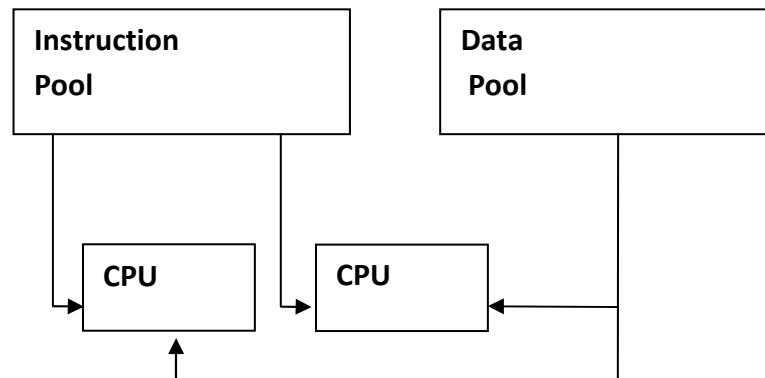


Figure 2.3 Multiple Instruction Single Data stream

2.3.4 Multiple Instruction Multiple Data stream

This architecture involves multiple processors performing different instruction streams on different data streams. Unlike in the SIMD architecture each processor works independently. This results in MIMD architecture being either synchronous or asynchronous. MIMD is currently the most common type of parallel computer with most

modern computers stemming from this architecture. This is due to it having the fastest execution, being extremely cost effective and offering the highest level of parallelism.

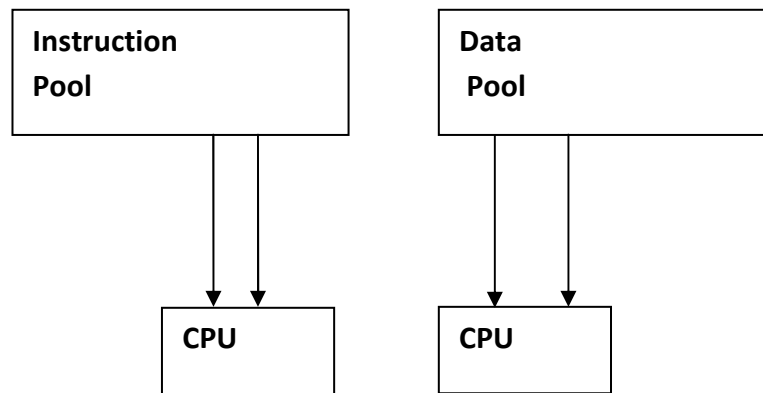


Figure 2.4 Multiple Instruction Multiple Data stream

With MIMD architectures the organisation of memory becomes an issue. There are two types of memory architectures for parallel computers: shared and distributed. The main difference between these memory architectures is the designation of processor memory. Shared memory inclines to connect all processors to the same memory allocation allowing multiple accesses to the same variables. Distributed memory gives each processor its own local memory with its own variables.

2.3.5 Shared Memory

The general idea of shared memory is to have global address space for all processors to access. This means that resources are shared even with multiple processors operating independently. However, any changes to memory by one processor will be visible to all processors. Shared memory can be further divided into two classes [6]: Uniform Memory Access (UMA) and Non-uniform Memory Access (NUMA).

UMA architectures involve identical processors sharing the memory equally. This results in equal access time to memory. Each processor can use a private cache and peripherals are shared. UMA architecture helps speed up the execution of a single large program and is suitable for applications for multiple users.

NUMA architectures involve linking two or more symmetric multiprocessor machines. These machines can access each others memory connected by a bus interconnect.

However, not all processors have equal access times to different memory addresses with access across links being considerably slower.

Yet despite this in NUMA architecture, shared memory is still a quicker architecture of data sharing between tasks. The global address space is also a user-friendly perspective to memory, with multiple accesses to one block of memory. Shared memory does have a few disadvantages namely the addition of more processors. More processors would result in an increase in traffic on the CPU-paths and would also become increasingly expensive with new amounts of shared memory having to be re-produced with the increase in processors.

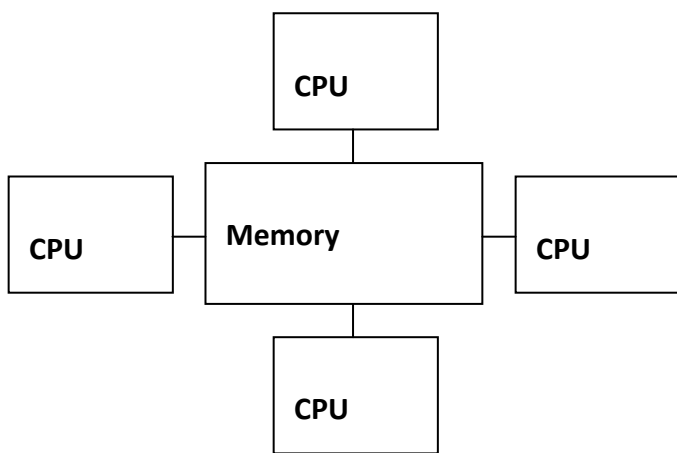


Figure 2.5 UMA Shared Memory

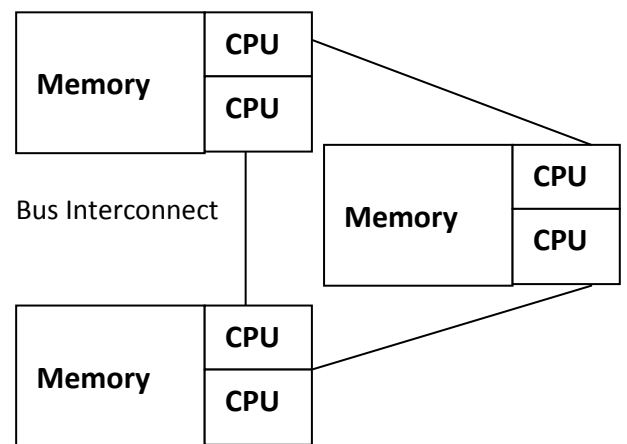


Figure 2.6 NUMA Shared Memory

Distributed Memory

The common characteristic of distributed memory is that the systems require a communication network for the connection of processor memory. Each processor has its own independent memory which does not reference any other processors' memory, so unlike shared memory there is no global address space. However, when a processor needs to access data on another processor's memory they must communicate through some interconnection between multiple processors.

With distributed memory, access time to a processor's own memory is rapid and without interference. This memory architecture also has the advantage of being scalable with the number of processors also increasing the amount of memory.

However, one problem with distributed memory is mapping data structures to this memory organisation.

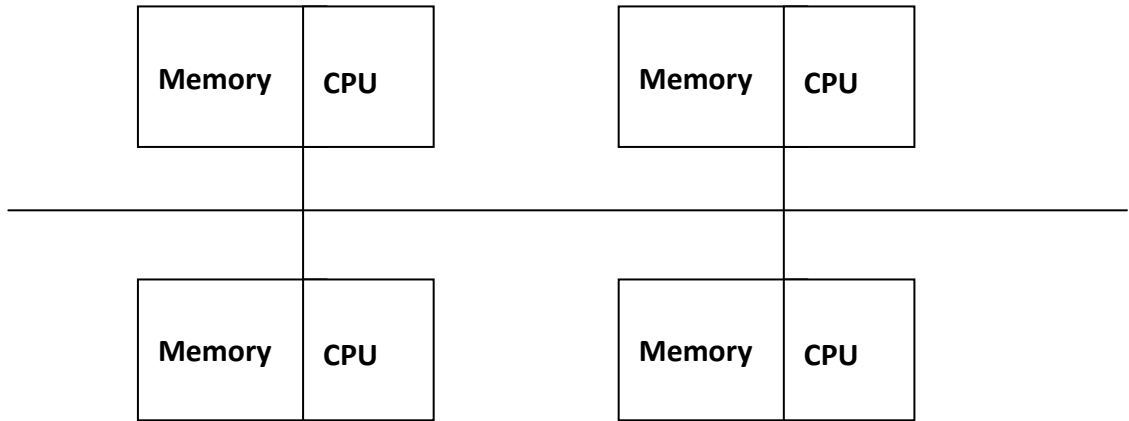


Figure 2.7 Distributed Memory

Hybrid Distributed-Shared Memory

For large scale computers it would be advantageous to have the best of both memory architectures. Many large computers employ both distributed and shared memory architectures with this technique becoming the increasingly common memory structure [6]. It works by the shared component being a symmetric multiprocessor machine with the processors on that machine addressing its memory as global. This machine is connected via an interlink to similar multiprocessor machines with global memory. Each multiprocessor machine knows only about its own memory and must communicate via a network to access or move data to another machine. This results in a usable, scalable and fast memory architecture. However, one main flaw with this as well as both shared and distributed memory is the reliance on the programmer. On the distributed side, the programmer is responsible for the many details regarding communication and on the shared side the programmer is responsible for synchronisation of effective global addresses. This once again leads to the problem that any parallel architecture is heavily dependent on the programmer in order to become successful [6].

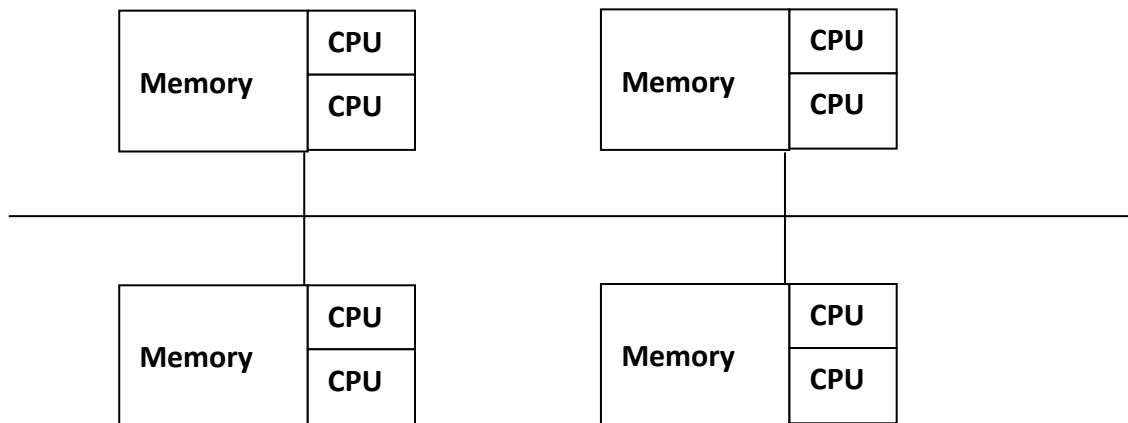


Figure 2.8 Hybrid Distributed-Shared Memory

2.4 Parallel Trends

With the increased trend in high performance computing applications, parallelism is becoming much more of a dependent process [15]. Standard processors are becoming defunct with their inability to solve these intense and highly complex problems. Therefore, the need for parallel architectures has grown with these models being integrated into new and existing systems. There exist special purpose processors, known as ‘accelerators’, which exploit parallel architectures which improve parallel performance. There are several developments of these accelerators such as FPGAs (Field Programmable Gate Arrays) and GPUs (Graphic Processing Units) but for this report I will mainly be looking at two parallel architectural trends: multi-core systems and many-core systems.

2.4.1 Multi-Core

The original concept of having instructions read and executed through one core has been eclipsed by the growing trend of multi-core processing. A multi-core processor is a single unit which contains at least two independent processors and is integrated onto a chip multiprocessor (CMP) or multiple circuit dies [16]. Each processor’s core works in parallel and time-slices multiple threads much like in single processors.

Multi-core architectures exploit the main advantages of parallel computing. If parallelism is properly programmed, execution of instructions is faster and a greater scope of problems can be solved. Also, with multiple processors on a single die, signals between CPUs travel

shorter distances resulting in clock rate improvement and lower degradation of signals. With many organisations such as Valve and Apple developing implementations of multi-core applications [11], this architecture is continuing to expand.

2.4.2 Many-Core

Where multi-core architectures lack is in their growing scale affecting the overall efficiency and resources. In 1965 the co-founder of Intel Gordon Moore predicted that transistor density on integrated circuits will double roughly every two years. This prediction more commonly known as Moore's Law had come to fruition, yet the increase in processors does not necessarily increase efficiency. Fred Pollack, a lead engineer at Intel, added a side note to this law, known as Pollack's rule [16], which states that: *"performance increase is roughly proportional to square root of increase in complexity"*. What this means is that doubling the number of transistors does not automatically double the performance. In fact it is identified that doubling the number of transistors will only succumb to a speedup increase of 40%. With power consumption being linearly proportional to the increase in processors, this implies that the increase in performance is at an expense of a greater decrease in energy efficiency and increase in cost.

As these laws suggests, the ever advancement of multi-core architecture will eventually be deemed as too big an expense in power consumption to consider it a feasible performance enhancer. Yet, with many-core architecture the ratio of power consumption and throughput becomes much more manageable [16]. Many-core architecture involves having many small less complex units which individually deliver a lower performance but, with the greater quantity of cores, the throughput becomes much greater than a multi-core system with the same power consumption [9]. Multi-core systems generally use under 32 cores but many-core can use hundreds of small cores with the potential of expanding core numbers in the thousands. Many-core architectures have solved the problem proposed by multi-core in delivering high performance machines which are also energy efficient. The emergence of the many-core architecture has further built on the continuous development of parallel computing and helps build new technologies which will advance commercial and scientific applications to bigger areas

2.5 Parallel Programming

As stated earlier, although parallel architectures have enabled new areas and programs to be processed, high performance is still dependent on the programming of the parallel machine [18]. Parallel programming is the method of creating programs for computation that exploit parallelism. It can be divided into two characteristics:

Explicit Parallelism

This feature involves the programmer explicitly defining the parallelisation in a program through constructs describing the parallel computation [18]. This makes parallelism a much more flexible task but also makes it a much more complicated task for the programmer who will hold all responsibility for parallel computation. The programmer has to consider factors such as component detection, synchronisation and communication when coding making it a more arduous process.

Implicit Parallelism

This feature in a programming language decides automatically what parts of a program will be run in parallel and what parts would be run in sequence [18]. Essentially the programmer would simply have to write a sequential program and the compiler would automatically apply parallelism, making the programs naturally parallel. Implicit parallel programs become much more productive with the programmer not needing to concern themselves with communication or task division and simply concentrate on the problem that needs to be solved [17]. However, there has been limited success with creating a purely implicit parallel high level language and in some cases the programmer will still have to control parts of the parallelism co-ordination.

Parallel Programming Models

A parallel programming model is an expression of a parallel program which can be compiled and executed. These can be invoked from extendable languages, imported libraries in sequential languages and completely new executions. Parallel programming models can be divided into two areas: process interaction and problem decomposition.

Process interaction is concerned with the way processes communicate with each other. The two most common forms of process interaction are shared memory, as identified in parallel architectures, and message passing.

In a message passing model, data is exchanged via sending messages between processing elements [18]. Message passing can be synchronous or asynchronous but there is no global shared location for processors and specific senders and receivers must be identified.

Problem decomposition is concerned with splitting a problem into identified separate processes. The two most common forms of this are task parallelism and data parallelism.

Task parallelism concentrates on the threads of execution being processed on the same or different data. Generally different threads communicate with one another as they are processed and data is passed from one thread to the next like a sequence of steps. Task parallelism expresses message-passing communication [18].

Data parallelism involves different elements of a data set being acted on by different processors at the same time. As opposed to task parallelism, it concentrates on the nature of the data rather than the thread of execution. Data parallelism uses shared addresses so that information is passed globally.

2.6 Parallel Functional Languages

For this project I will be using three parallel programming implementations. These are Glasgow Parallel Haskell, Glasgow Distributed Glasgow and Eden [28], [30], [29]. All three implementations are extensions of the functional programming language Haskell and will each be briefly described in the following section.

2.6.1 Glasgow Parallel Haskell

The majority of this section is based on the "*Gentle Introduction to GpH*" [20] section from the Glasgow Parallel Haskell home page. Glasgow Parallel Haskell is the dominant programming model for Haskell. It is a high-level parallel programming language with only a few key aspects of parallelism needing to be defined by the programmer.

GpH extends Haskell by adding the primitive **par** which identifies a possible parallel execution of a program expression. For example in the expression `a `par` b` it is implied that `a` could be evaluated by a new parallel thread while the parent thread could continue to evaluate `b`. Additionally, GpH adds a primitive **pseq** to control the sequencing of composition specifying how much evaluation a thread should perform. In the expression `a `pseq` b` the value `a` is evaluated into weak head normal form before returning `b` resulting in all data structures being evaluated up to the top level constructor. An example of GpH's impact on the Haskell language can be seen in a simple Fibonacci program. The sequential Haskell version is shown below:

```

nfib :: Int -> Int
nfib 0 = 1
nfib 1 = 1
nfib n = nf1+nf2+1
        where nf1 = nfib (n-1)
              nf2 = nfib (n-2)

```

However, when GpH is applied for parallelisation, the program is transformed as follows:

```

parfib :: Int -> Int
parfib 0 = 1
parfib 1 = 1
parfib n = nf2 `par` (nf1 `pseq` (nf1+nf2+1))
        where nf1 = parfib (n-1)
              nf2 = parfib (n-2)

```

Note that the addition of the primitives ``par`` and ``pseq`` have ordered the program so that `parfib nf2` is computed on a parallel thread, then the expression `parfib nf1` is evaluated and finally the main expression is evaluated. The main expression is however dependent on `nf2` being computed before it can be evaluated.

However, with increased program structures, the algorithm of the program becomes less obvious due to the behaviour. Evaluation strategies are used to separate the algorithm from the behaviour and raise the level of abstraction in GpH. The general purpose of evaluation strategies is that *"it should be possible to understand the semantics of a function without considering its dynamic behaviour."* [20]

A strategy is a function that will preserve the value of a type when specifying the behaviour of the computation of that type. It does not affect the value being computed by the algorithmic component but is evaluated for pure effect. Strategies are applied in GpH with

the function ``using`` which will apply a strategy to a value e.g. `func a b = expr`using` strategy.`

Simple strategies are concerned with merely the evaluation degree. With this method simple strategies can be created: for example, below the strategy `r0` will perform no reduction and `rseq` will reduce its argument into Weak Head Normal Form.

```
r0 :: Strategy a
r0 _ = ()

rseq :: Strategy a
rseq x = x `seq` ()
```

Along with these simple strategies, you can reduce an expression to Normal Form which will contain no redexes using the following `rnf` strategy:

```
class NFData a where
  rnf :: Strategy a
  rnf x = x `seq` ()
```

`NFData` can be defined for many types e.g. `NFData Int`, but can also be defined for constructors for example lists and pairs:

```
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs

instance (NFData a, NFData b) => NFData (a,b) where
  rnf (x,y) = rnf x `seq` rnf y
```

As well as the evaluation degree a strategy can specify parallelism and sequencing. Data-oriented parallelism describes the behaviour adhering to a data structure: for example the `parList` applies the strategy to every element of a list in parallel:

```
parList :: Strategy a -> Strategy [a]
parList strat [] = ()
parList strat (x:xs) = strat x `par` (parList strat xs)
```

This can then be used in the parallel map function `parMap` which will apply its argument to every element in a list:

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat
```

Alternatively for complex data-structures we can apply data-oriented parallelism through strategic function application. This is defined with the construct `$||` such as in `f $|| s $ x` which applies the strategy `s` to `x` while also applying the function `f`. An alternative to the `parMap` function above, using this technique allows the definition of data-oriented parallelism without changing the definition of `map` itself:

```
g $|| parList rnf $ map f xs
```

The separation of algorithmic and behavioural code used in evaluation strategies aids in the tuning of performance in parallel programs helping the user experiment with techniques. The semantics used here for GpH can be combined and rewritten into a parallel program to use evaluation strategies.

GpH is an expanding language and is proving its worth in increasing the speedup of Haskell programs on multi-core machines. With its evaluation strategies it is also extremely useful in exploring alternative parallelisations and is an excellent tool in parallel prototyping. It is also a useful tool in parallel symbolic applications such as natural language processors. This makes GpH a simple but extremely useful language for parallel architectures.

2.6.2 Eden

Eden like GpH extends the Haskell language however it takes a different approach to parallelism. Where GpH takes an annotation-based parallel approach, Eden incorporates explicit process creation and interconnection used by the programmer to match a given architecture. Eden's target programs are not only transformational programs but concurrent programs as well [23].

The Eden language transforms a function into a process abstraction via the `process` construct which defines schemes for processes in a functional style. For example a process structure mapping inputs to outputs is written as:

```
process(input1,input2,...inputn)->(output1,output2,...output)
where equation1.....equationx
```

A process abstraction in Eden is represented with the syntax `Process a b` where `Process` is the newly introduced type constructor, `a` is the process interface input, and `b` is the process output interface. For example, the ring process structure identified in the paper “*From [sequential] Haskell to [parallel] Eden*” [23] which maps a list of input channels to output channels defining the behaviour of the process via an abstraction is shown below:

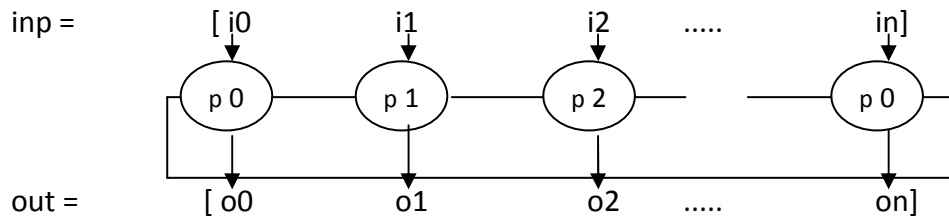


Figure 2.9 Ring Process Structure

This can be defined in Eden as:

```
procRing :: (Int -> Process(i,r) (o,r)) -> Process [<i>] [<o>]
procRing p
= process inp -> out
where (out,ring) = createRing p 0 inp ring

createRing :: (Int -> Process(i,r) (o,r)) -> Int [i]->r-> ([o],r)
createRing p k [] ringIn = ([],ringIn)
createRing p k (i:is) ringIn = (o:os,ringOut)
where (o,ringMid) = p k # (i,ringIn)
      (os,ringOut) = createRing p (k+1) is ringMid
```

In this program, the function `createRing` builds the process ring element by element until no inputs are available. The process instantiation `p k # (i,ringIn)` creates the ring processes together with their communication interface so as to bring the two ends of the ring connection together. Communication between processes is unidirectional in Eden i.e. one writer connects to one reader. When a process is created the communication channels between parent and child for input and output are already created for transmission. If input is not received the evaluation is suspended until that input has been successfully retrieved. Eden’s performances on parallel applications have generally been quite good with its explicit process model yielding good speedup times and performances. In “*GpH and Eden: Comparing Two Parallel Functional Languages on a Beowulf Cluster*” [21] it is noted that

Eden's speedup times surpass the GpH implementation which suggests that the Eden implementation should be a strong and useful language for parallel languages.

2.6.3 GdH-

This section is mainly taken from the paper *"The Design and Implementation of Glasgow Distributed Haskell"* [24]. Glasgow Distributed Haskell (GdH-), is an extension of GpH and Concurrent Haskell. Using the implicit pure threads defined in GpH combined with the explicitly placed I/O threads defined in Concurrent Haskell GdH- builds a high-level distributed programming model.

Users can interact with threads in GdH- through the primitive `PEid` which identifies processor entities. As processor entities are explicit in GdH-, a program can explicitly utilise resources within a processor. Additionally IO thread creation can be made in GdH- using the `rForkIO` primitive which is built on the Concurrent Haskell's `fork` primitive. This will take an IO thread and `PEid` and spawn the specified thread on the specified PE. This function can be blocked until the remote thread finishes using the primitive `revalIO`. A sample piece of code from the *"Design and Implementation of Glasgow Distributed Haskell"* using these operations is shown below:

```
main = do
  pr <- allPEid
  m <- newEmptyMVar
  let work = do
        i <- myPEid
        o <- owningPE m
        return (i,o)
    px <- mapID(\p -> (revalIO work p)) pr
  putStrLn (show px)
```

This program uses `mapM` to apply the operation `revalIO` on all identified PEs. Another simple program which shows the GdH- language is the Ping program taken from the same paper:

```

main = do
  pex <- allPEid
  putStrLn ("PEs = " ++show pex)
  mapM loop pex
  where
    loop ps = do
      putStr("Pinging " ++show ps++".....")
      (name,mx) <- timeit (revalIO remote ps)
      putStrLn ("at " ++name++"time = " ++show mx++"mx")
    remote = getEnv "HOST"

```

This lists the available PEs using `allPEid` and then `mapM` loops across this list. For each PE, we time how long it takes to execute the operation `revalIO` with the operation `timeIT` and then display the results.

Communication in GdH- is done through virtual shared memory in which each thread can use to communicate globally. High level constructs are abstracted over distributed MVars to provide explicit synchronisation.

The explicit threads and mapping used by GdH- along with the identification of PEs make it a unique language stemming from already established languages. This results in GdH- inheriting many mature functional language technology already tried and tested. However, GdH- has still a lot of untested qualities and its full assessment is not yet complete. Further more it has recently been imported onto a new framework with its performance yet to be assessed. This makes the potential for many projects, such as this one, to try and further establish GdH- as a distributed programming model.

2.7 Benchmarking

Benchmarking involves assessing the relative performance of an implementation through running a specified program. In the scope of parallel programming this can be useful to analyse the efficiency and effect of exploiting parallelism on a system.

The first and most obvious performance measurement is the runtime of a program in a parallel architecture. This is simply the time it takes for a program to execute successfully. In parallel computing it is assumed that a higher number of processors would equal a lower execution time of a program. This however, is not always the case and to truly test the ratio of multiple processors to single processor execution the speedup of an algorithm must be calculated.

Speedup is calculated with the formula $S = T1/T2$, where T1 represents the time taken for a program to execute under a single processor and T2 is the time taken for the program to execute over multiple processors [19]. In parallel programs, not all operations can be executed in parallel computation. There are still sequential operations that will exist within these parallel algorithms which increasing the number of processors will not affect. The increase in processors will bring down the execution time of the parallel operations but eventually the entire execution time will be determined by the sequential operations. This is explained in Amdahl's law [26] which is used to determine the maximum speedup that can be expected from parallel algorithms given the amount of parts that must be computed sequentially. It is calculated by $S = 1 / ((P/N) + (1-P))$ where P refers to the operations being performed in parallel, and N refers to the number of processors. The ideal situation for a speedup would be to have a linear relationship with the increase in processors having a consistent increase in speedup.

Additionally, programs can be assessed through the variance of their runtime. Due to interference with other processes or synchronisation issues, runtimes of a program can vary from different runs [31]. The variance of the runtimes can be calculated as a percentage by running a program a certain amount of times obtaining the smallest largest and median runtime. A percentage can be obtained by subtracting the smallest runtime from the largest and dividing the result by the median to give the decimal value which can be multiplied by 100 to give the variance. The variability of a program shows the overall reliability of a system and can expose potential flaws in a systems communication or distribution methods [31].

There are many ways to measure a programming languages performance but the ideal goal is to compare these performances with that of other program languages in order to analyse the best language with the best potential. There currently exist a number of benchmark suites with existing languages' problems and performances such as the *Alioth Shootout* benchmarks [33]. This is a collection of simple benchmarks written in over 24 different programming languages charting which languages produce correct results in the quickest time. This helps to compare what languages give the best performances.

However, for parallel languages there is not yet a comparable system for parallel languages. However, the ongoing SICSA MultiCore Challenge is an ongoing development which hopes to make ground in providing a concrete parallel language comparison. The SICSA MultiCore

Challenge is a project aiming to compare several approaches to parallel computation [34]. Developers are invited to implement selected applications on multi-core systems with a chosen parallel language system. The results are then summarised and presented at a workshop for all developers to discuss the strengths and weaknesses of current parallel implementations. This aims to achieve a better understanding of the parallel systems available and the potential of each system. Currently, two phases have run in this group and the third and final phase has been planned to take place at the Trends in Functional Programming symposium at St.Andrews in 2012 [35]. This final phase will help the parallel computing community in understanding the applicability, stability, limitations and support of current systems as well as exploring new approaches to parallelism.

3 Experiment Design

This section will describe the method and conditions under which the benchmark programs were tested. This will include specifying the hardware and operating system used for this project, the overall structure of the testing, and a description of the test programs.

3.1 Hardware Specification

The benchmarks were conducted on two different platforms running the Linux CentOS 5.5 Operating System. The first platform is lxpara2, which is a single 8-core 32-bit machine running Intel Xeon 2.33GHz processors with a cache size of 6144KB and 8 GB worth of RAM. The other platform was the Beowulf cluster which is a 64-bit machine containing 32 8-core nodes resulting in a possible 256-core machine. Each node is running either a set of Intel Xeon 2.00GHz processors or 2.13GHz processors with cache sizes of 4096KB, clock speeds of 2.13 GHz and 12GB worth of RAM. For the purposes of this project, all executables run on the Beowulf cluster were run under 32-bit due to the Eden compiler installed for these measurements set up as a 32-bit compiler.

The two compilers needed for these benchmark programs were the GHC version 6.12.3 compiler and the Eden version 7.0 compiler. Both Eden and Glasgow Distributed Haskell used the software package Parallel Virtual Machine (PVM) in order to gather multiple processors and exploit parallel architectures. PVM works by collecting a number of different machines and combining them into a single large parallel computer. This was used to collect all the nodes in the Beowulf cluster giving a potential of 256 available processors for the GdH- and Eden benchmarks to exploit. However, Glasgow Parallel Haskell does not use PVM and so was only comparable with preliminary benchmarks on 8 processors on lxpara2 and an individual Beowulf node.

3.2 Structure of Measurements

For each problem a sequential Haskell program was written as well as a GpH, Eden and GdH-implementation. The sequential program was built with the GHC compiler and was measured with the Linux *time* command to give the programs' runtimes. The results would enable the absolute speedup of each test program to be calculated.

Due to GpH being limited to only 8-cores, two sets of test data were used for these benchmarks. The initial input data run on the sequential program was used to benchmark all three parallel languages on 1, 2, 4, 6, 7 and 8 cores. This gave a fair comparison of both GdH- and Eden to GpH and also ensured absolute speedup could be calculated by running the same input data as the sequential version. The number of cores used was determined by doubling the number until the maximum of 8. However, results were also measured for 6 and 7 cores as the maximum core number of 8 potentially producing unreliable results. This is due to *lxpara2* utilising some processor for system tasks and using all 8 results in contention between system and user tasks. The second set of input data was a much larger value in order to get a reasonable set of runtimes on a larger amount of cores. This data was only used for Eden and GdH- on 8, 16, 32, 64, 128, 192 and 224 cores spread over up to 28 nodes. Through PVM cores are distributed over nodes by allocating each node to a core: for example, 8-cores would use 1-core from 8 nodes. Although able to accommodate 256 cores on the Beowulf cluster, using all processors was not viable and resulted in defective runtimes therefore the maximum reliable core number was set at 224. Another change in these benchmarks is that, due to sequential and lower processor number runtimes taking an extensive amount of time, the results start from 8-cores onwards to enable them to be of a suitable range. This meant that the absolute speedup was not recorded but rather the relative speedup compared to the runtime of the 8-core run.

When benchmarking the three parallel languages the results were collected to calculate the median runtime, speedup, and variability of each. In order to effectively calculate each of these properties, each program was run a total of seven times on each determined core number. The runtime was determined by obtaining the median value from the runtimes gathered. As stated, the speedup was calculated differently for the different inputs. The

absolute speedup was calculated by dividing the median runtime of a test program by the runtime of the sequential program so as to determine how much faster the parallel algorithm was compared to the sequential. The relative speedup was recorded by dividing the median runtime of an n-core test program by the median runtime of the Eden's test program on 8-cores. When calculating the relative speedup it was decided to use Eden's runtime on 8-cores to truly compare GdH's performance to Eden's. As was discovered in the experiments, GdH- suffered from extremely high runtimes on lower cores and so the relative speedup calculated using GdH's 8-core run produced highly aberrant speedup graphs. Using Eden's 8-core run gave a better insight into the performance of GdH-. The speedup formulas are shown below:

Absolute Speedup

$$Sa = \frac{T_s}{T_n}$$

Sa = Absolute Speedup Ts = Sequential Runtime
Tn = Runtime on n-cores

Relative Speedup

$$Sr = \frac{T_r}{T_n}$$

Sr = Relative Speedup Tn = Runtime on n-cores
Tr = Runtime on lowest core number

The variability is used to calculate how reliable the test programs are on each platform dependant on the number of cores used. It is calculated by obtaining the highest runtime of the seven results and subtracting the lowest runtime. This result is then divided by the median runtime and multiplied by 100 to give a percentage of the variability. The equation for variability is shown below:

$$V = \frac{\text{Variability} (R_h - R_s)}{R_m} \times 100$$

V= Variability Rh = Highest Runtime Rs= Lowest Runtime Rm= Median Runtime

3.3 Test Programs

The test programs used for this evaluation were reliable problems used to obtain suitable data. Three different test programs were developed for each platform and a full listing of all programs used can be found in Appendix A and the attached CD. A brief description of each program is given below:

3.3.1 Fibonacci

The Fibonacci test program uses a divide and conquer paradigm and takes in an integer input and produces the Fibonacci number. A Fibonacci number is the sum of the previous two numbers in the Fibonacci sequence e.g. 1, 1, 2, 3, 5 etc. The recurrence relation is stated below:

$$F_n = F_{(n-1)} + F_{(n-2)}$$

where $F_0 = 1$ and $F_1 = 1$

The sequential Haskell Fibonacci code is shown below:

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

In this code, `fib` takes in an integer and returns 1 if it is less than 2. If the input value is over 2 it will recursively call the `fib` function calculating the result of `fib n-1` and `fib n-2` and then adding them together to return the result.

When adding parallelism to this program the calculation of the recursive `fib` call is done in parallel. A threshold of 20 was also introduced to solve lower inputs more quickly rather than adding unnecessary parallelism for small tasks. The parallel `parfib` function for GdH is shown below:

```

parfib :: [PEId] -> Int -> IO Int
parfib pes n | n < 2      = return 1
              | n < 20    = return (fib n)
              | otherwise = do t <- rfork pes (parfib pes (n-2))
                               left <- parfib pes (n-1)
                               right <- join t
                               return $! (left + right)

```

In these benchmarks a test input of 43 was used for core sizes 1 to 8 which generates 121392 sparks while the higher core sizes used a test input of 47 generating 635,621 sparks. The test values were chosen to return a high enough runtime to ensure low enough variability but a low enough runtime to ensure results could be gathered in a suitable timeframe.

3.3.2 Sum of Totients

The Sum of Totients test program is a data parallel paradigm and calculates the Summation of Euler Totients between a lower and upper limit. Euler's Totient function takes in an integer n and returns the number of integers between 1 and n which are coprime to n . This program calculates the sum of Euler's Totient function on a range of numbers between a lower and upper limit. The sequential Haskell code is shown below:

```

sumTotient :: Int -> Int -> Int
sumTotient lower upper = sum (map euler (mkList lower upper))

euler :: Int -> Int
euler n = length (filter (relprime n) [1 .. n-1])

relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1

hcf :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

mkList :: Int -> Int -> [Int]
mkList lower upper = (upper, upper-1..lower)

```

In this code the lower and upper limits are arranged into a list from the upper down to the lower. Each element is then applied to the `euler` function which counts the relative primes of a number with the `map` function. Each result from the list applied to the `euler` function is then added up and returned as an integer.

The method used to add parallelism for this program was to introduce the function `splitAtN` which splits the list of integers into several chunks of integers dependent on a value set by the user. For example an input with a lower limit of 1 and an upper limit of 10,000 could take in the value 200 to split the list of integers to 50 lists of integers. Additionally a parallel map was introduced to replace the top level map function. This applies the `euler` function in parallel to the elements in the list. The `splitAtN` function and parallel map adaptation of the `sumTotient` function in Eden is shown below:

```
sumTotient :: Int -> Int -> Int -> Int
sumTotient lower upper sn = sum([(process (\ z -> (sum . map euler)z))#x |
    x <- splitAtN sn (mkList lower upper)]
    `using` seqList r0)

splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
    where (ys,zs) = splitAt n xs
```

For these benchmarks a test input of 1 to 24,000 with and a chunk size of 300 were used for core sizes 1 to 8 which yielded 80 sparks. For the higher core sizes a test input of 1 to 100,000 with a chunk size of 400 was used yielding 250 sparks. This showed a slight flaw in testing with the relatively low amount of sparks for the test data however, the inputs were decided to obtain runtimes which ensured low variability within feasible process time.

3.3.3 Queens

The queens test program is a nested data parallel paradigm and calculates the *n-queens* problem given an integer input. The *n-queens* problem involves calculating the number of different ways that queens can be placed on a chessboard of a size $n \times n$ so that no two queens can attack each other. For example in an 8x8 chessboard there are 92 distinct solutions for placing a number of queens. The sequential Haskell code used to determine the n-queens is shown below:

```

nqueens :: Int -> Int
nqueens nq = length (gen 0 [])
where
  safe :: Int -> Int -> [Int] -> Bool
  safe x d [] = True
  safe x d (q:1) = x /= q && x /= q+d && x /= q-d && safe x (d+1) 1

  gen :: [[Int]] -> [[Int]]
  gen bs = [ (q:b) | b <- bs, q <- [1..nq], safe q 1 b ]

```

In this code the board is generated and queens added via the `gen` function, which creates a partial board, and the `safe` function, which calculates whether a queen can be added safely on a partial board. These functions create a long list of partial boards with all possible positions of queens on them. The `nqueens` function then returns the length of this list.

To add parallelism to this program a parallel map function was introduced to create individual solutions in parallel on a recursive call of the `pargen` function. A threshold was also introduced in order to solve lower numbered problems sequentially. For example the `pargen` code for GpH is shown below:

```

pargen :: Int -> [Int] -> [[Int]]
pargen n b
  | n >= threshold = iterate gen [b] !! (nq - n)
  | otherwise      = concat bs
  where bs = (map (pargen (n+1)) (gen [b]))
            `using` parList rdeepseq

threshold = 10

```

In these benchmarks a test input of 13 was used for core sizes 1 to 8 which produced 10,372 sparks. The larger test input for core sizes 8 to 224 was 15, producing 15,480 sparks. Once again these test values were selected to give a feasible runtime with low variability.

4. Experiment Results

This section presents and discusses the results obtained from the experimental design outlined in section 3. This chapter is split into three sections discussing the runtime, speedup and variability of each test program on each platform. Each section will present each test program's results of the up to 8-core runs on both lxpara2 and the Beowulf cluster, as well as the up to 224-core results on the Beowulf cluster. A discussion on the results is presented in section 4.5.

A full list of results is available in Appendix B.

4.1 Fibonacci

4.1.1 Runtime

The sequential runtime for Fibonacci program was 89.2 seconds on the Beowulf cluster and 73.7 seconds on lxpara2. Figures 4.1 and 4.2 show the runtimes of the Fibonacci program on both lxpara2 and the Beowulf cluster respectively, with a given input of 43 producing 121,392 sparks:

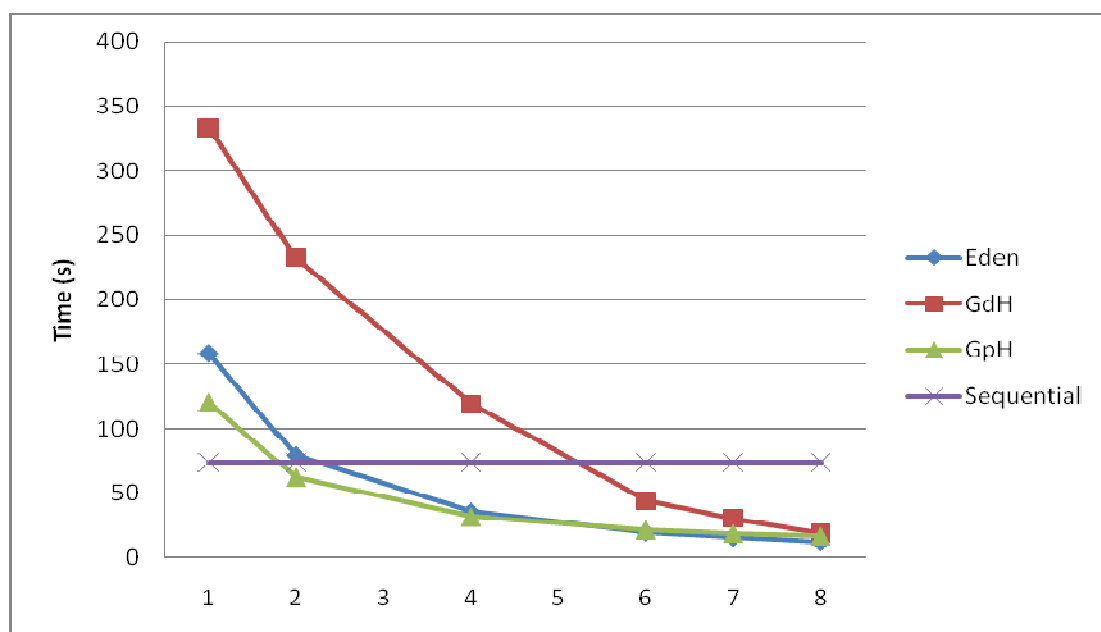


Figure 4.1 Runtime of up to 8-cores Fibonacci program on lxpara2

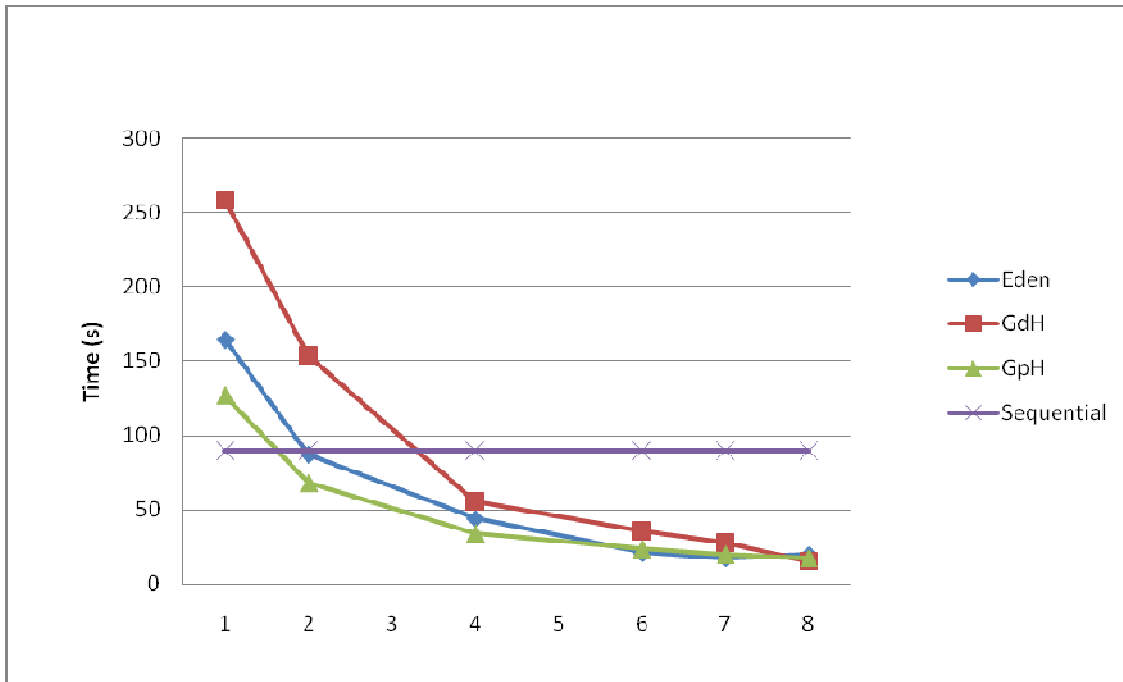


Figure 4.2 Runtime of up to 8-cores Fibonacci program on Beowulf Cluster

It can be observed that GpH gives the best overall runtime up to 6-cores where Eden records the lowest runtimes. Another noticeable point is the runtime of GdH- on 1 and 2 cores. It can be seen that GdH- has a much greater runtime than both Eden and GpH on the first two cores with a considerable gap in runtime. However, beyond 2-cores this gap is reduced significantly and GdH- shows much more competitive results on 4 to 8-cores. The median runtimes are recorded in the following two tables:

Table 4.1 Runtime of up to 8-cores Fibonacci program on Ixpara2

	1	2	4	6	7	8
Eden	158.4	79.6	35.7	20.1	16.0	12.3
GdH-	333.4	232.7	119.0	44.0	30.5	18.8
GpH	120.5	62.3	31.8	21.8	18.9	16.9

Table 4.2 Runtime of up to 8-cores Fibonacci program on Beowulf Cluster

	1	2	4	6	7	8
Eden	164.6	87.4	44.3	21.3	17.8	19.9
GdH-	258.1	153.9	55.5	35.8	28.1	15.5
GpH	126.9	67.9	34.4	23.8	20.1	17.6

When the test input was increased to 47 producing 635,621 sparks on the Beowulf cluster to test runtimes between 8 and 224-cores, the following graph is produced:

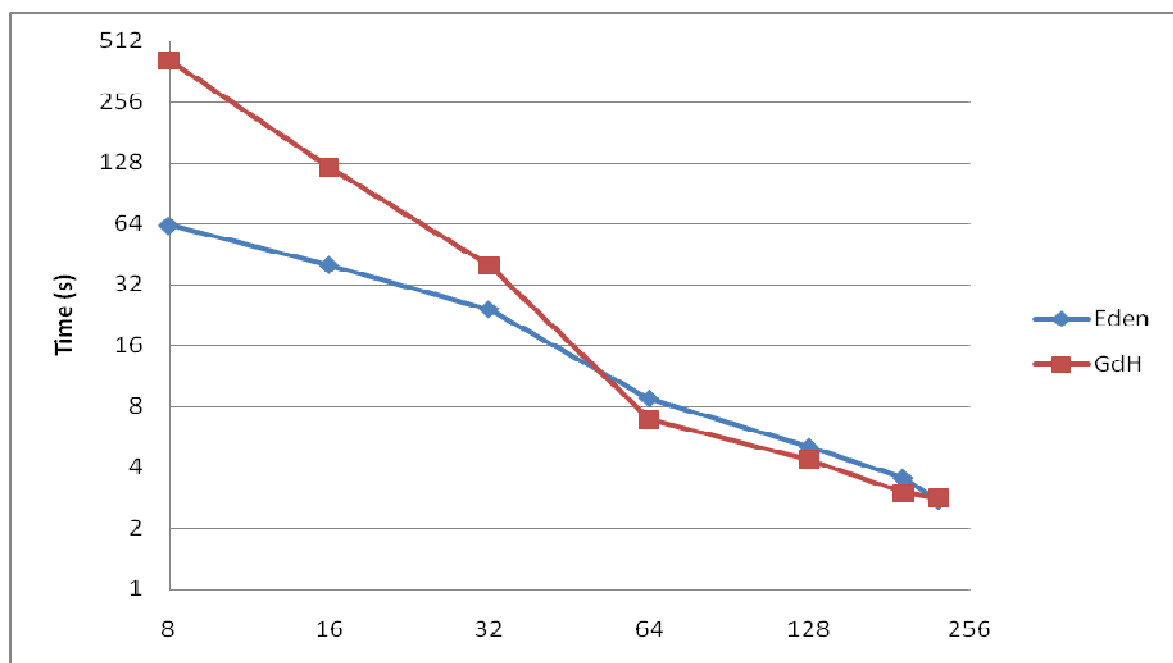


Figure 4.3 Runtime of up to 224-cores Fibonacci program on Beowulf Cluster

Once again GdH- has a substantial gap in runtime compared to Eden on the initial cores i.e. 8 and 16. However, on the larger amount of cores, GdH- not only competes well with Eden but surpasses it in runtime in some cases. The results are shown below:

Table 4.3 Runtime of up to 224-cores Fibonacci program on Beowulf Cluster

	8	16	32	64	128	192	224
Eden	62.9	40.1	24.3	8.8	5.0	3.5	2.7
GdH-	410.4	121.1	39.7	6.9	4.4	3.0	2.8

4.1.2 Speedup

The graphs below show the speedup of the three programming implementations for the Fibonacci program with a test input of 43 on an 8-core architecture:

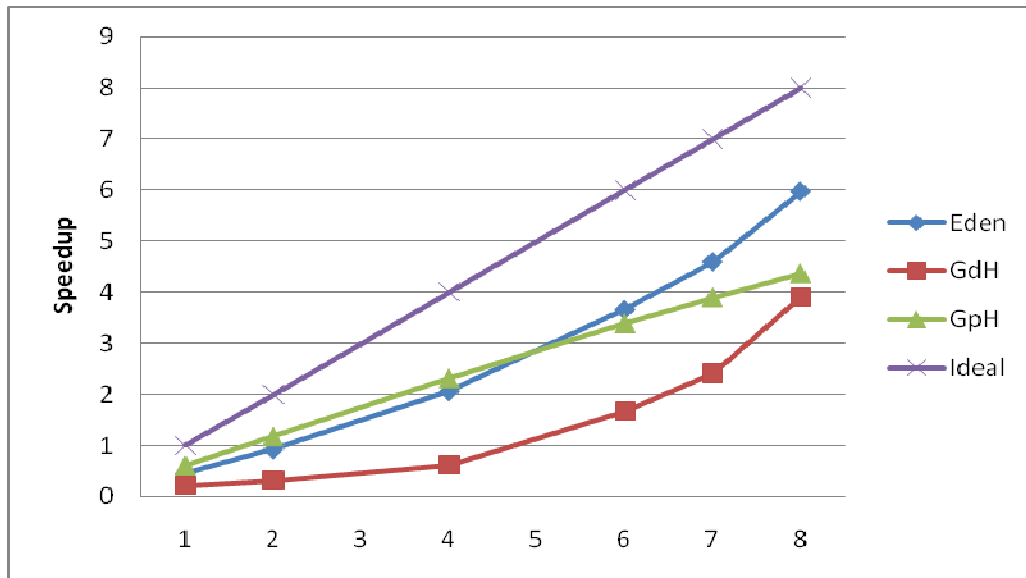


Figure 4.4 Absolute Speedup of up to 8-cores Fibonacci program on lxpara2

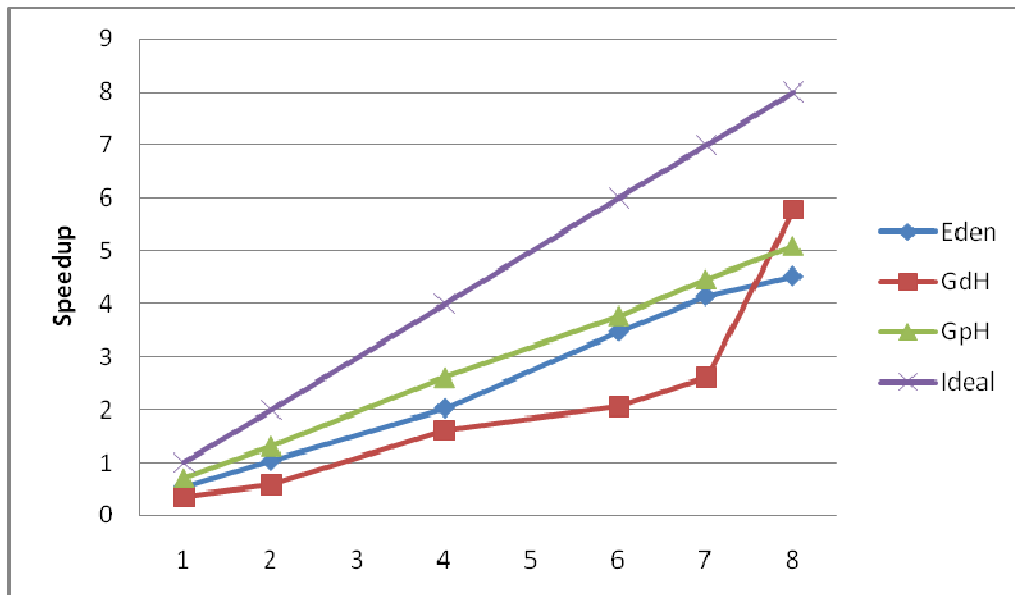


Figure 4.5 Absolute Speedup of up to 8-cores Fibonacci program on Beowulf Cluster

Due to the unreliability of the runtimes over all 8-cores, the 7-core results are taken as a more accurate observation of each system's speedup performance. Eden showed the best

result on 7-cores on lxpara2 with a speedup of 4.6 surpassing GpH and GdH-'s speedups of 3.9 and 2.4 respectively. However, on the Beowulf Cluster GpH records the best speedups up to 7-cores. On 8-cores, on the other hand, it is seen that GdH- shows a dramatic increase in absolute speedup, exceeding both Eden and GpH but, as stated earlier, the 8-core runs cannot be fully reliable due to system tasks taking processor utilisation. The following tables show the speedups of the Fibonacci program:

Table 4.4 Absolute Speedup of up to 8-cores Fibonacci program on lxpara2

	1	2	4	6	7	8
Eden	0.5	0.9	2.1	3.7	4.6	5.9
GdH-	0.2	0.3	0.6	1.7	2.4	3.9
GpH	0.6	1.2	2.3	3.4	3.9	4.3

Table 4.5 Absolute Speedup of up to 8-cores Fibonacci program on Beowulf Cluster

	1	2	4	6	7	8
Eden	0.5	1.0	2.0	3.5	4.1	4.5
GdH-	0.4	0.6	1.6	2.1	2.6	5.8
GpH	0.7	1.3	2.6	3.8	4.5	5.1

The relative speedup graph for the Fibonacci program for the up to 224-cores run is shown below:

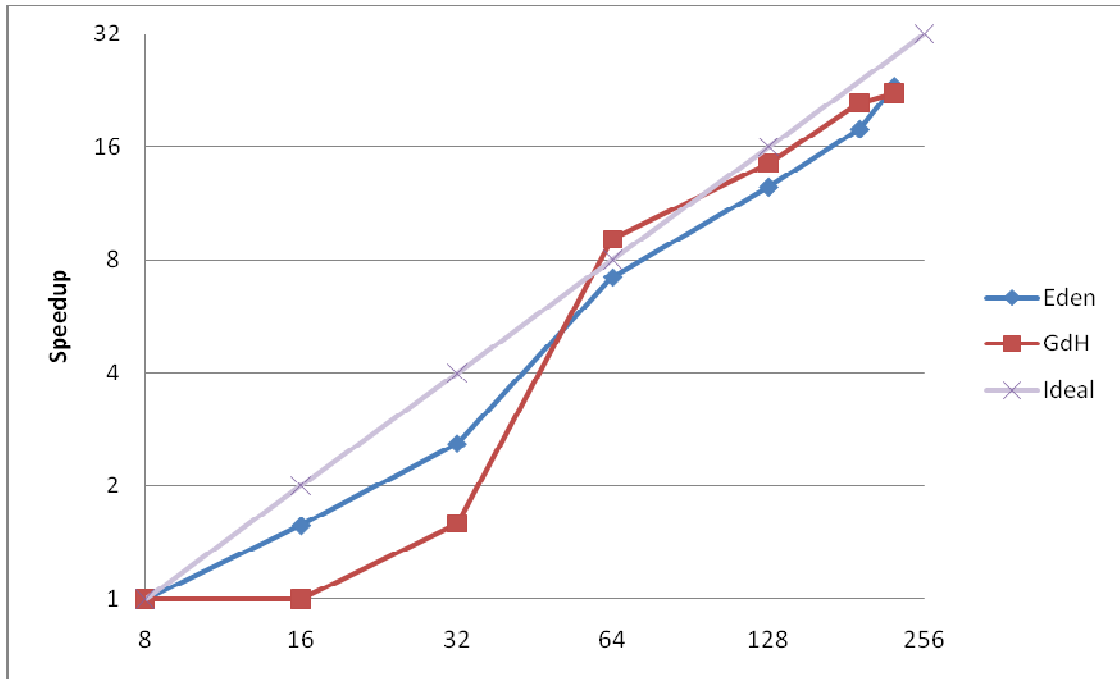


Figure 4.6 Relative Speedup of up to 224-cores Fibonacci program on Beowulf Cluster

From this graph it is observed that GdH- shows a distinctively lower speedup than Eden up to 32 cores. However, from 64 to 192-cores it records significantly better speedups than Eden. Considering that this is being measured using Eden’s initial runtime GdH- shows an extremely impressive performance for the Fibonacci program on larger amounts of cores. However, it is let down by its performance on lower amounts of cores. The speedup times are shown below:

Table 4.6 Relative Speedup of up to 224-cores Fibonacci program on Beowulf Cluster

	8	16	32	64	128	192	224
Eden	1.0	1.6	2.6	7.2	12.5	17.8	23.1
GdH-	0.2	0.5	1.6	9.1	14.4	20.9	22.3

4.1.3 Variability

The charts and tables below show the variance of runtime percentages of the Fibonacci programs on the up to 8-cores tests over seven runs:

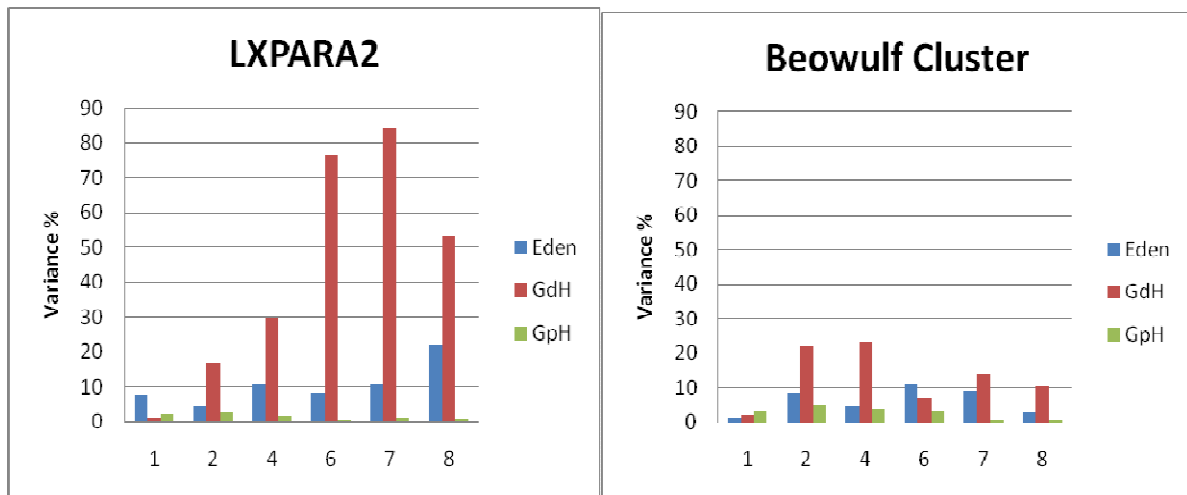


Figure 4.7 Variance of Fibonacci program on up to 8-cores lxpara2 and Beowulf cluster

Table 4.7 Variance of up to 8-cores Fibonacci program on lxpara2

	1	2	4	6	7	8
Eden	7.9	4.1	10.9	8.3	10.8	22.0
GdH-	0.9	17.0	29.8	76.3	84.4	53.2
GpH	2.2	2.7	1.2	0.5	1.0	0.8

Table 4.8 Variance of up to 8-cores Fibonacci program on Beowulf Cluster

	1	2	4	6	7	8
Eden	1.5	8.7	4.7	10.8	9.3	2.9
GdH-	1.9	21.9	23.3	6.8	13.7	10.4
GpH	3.3	4.9	3.9	3.2	0.6	0.6

The main noticeable aspect of these results is GdH’s extremely high variability on lxpara2. On both systems GdH- gave the highest variance percentages but on lxpara2 it reached up to 84.4% which is significantly higher than any of Eden or GpH’s top variance percentages. GdH- also demonstrated higher variance in the 224-core tests shown in the chart and table below:

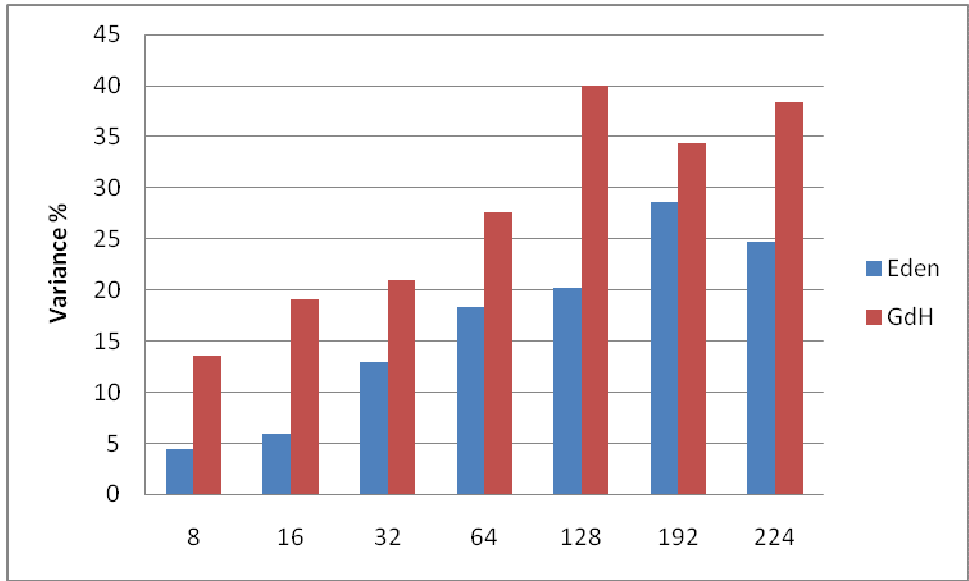


Figure 4.8 Variance of Fibonacci program on up to 224-cores Beowulf cluster

Table 4.9 Variance of up to 224-cores Fibonacci program on Beowulf Cluster

	8	16	32	64	128	192	224
Eden	4.4	5.9	12.9	18.3	20.2	28.6	24.8
GdH-	13.6	19.1	20.9	27.6	39.9	34.3	38.3

Once again GdH- shows higher variance but not quite as excessive as its results on Ixpara2. Also both languages show a common trend increasing variance on more cores.

4.2 Sum of Totients

4.2.1 Runtime

The sequential runtime of the Sum of Totients program was 44.8 seconds on lxpara2 and 55.3 seconds on the Beowulf cluster. The following graphs show the up to 8-cores runtime on both systems with a test input of 24,000 and a chunk size of 300 producing 80 sparks:

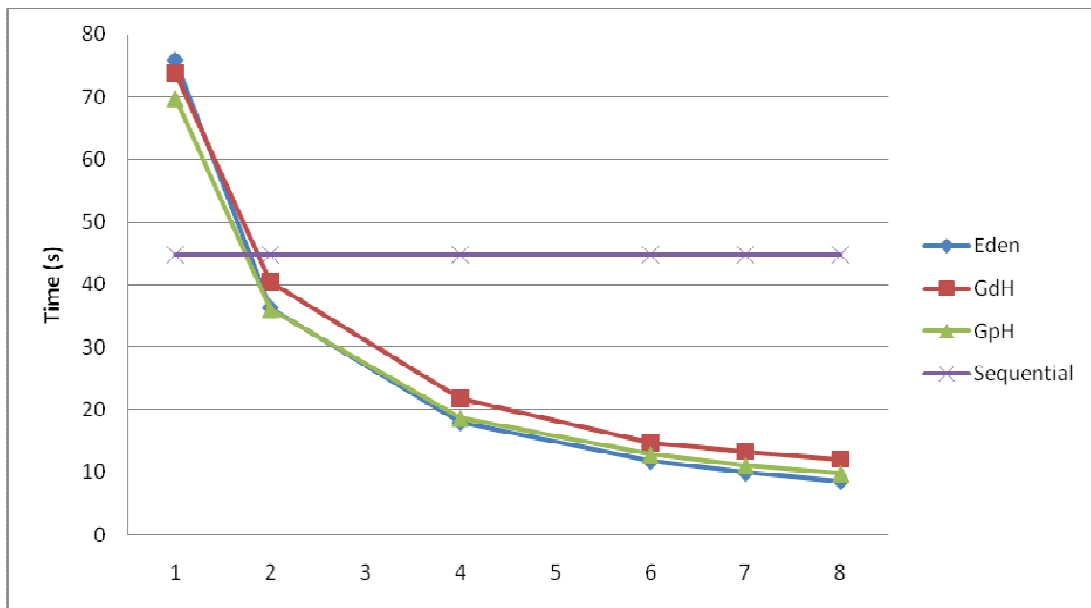


Figure 4.9 Runtime of up to 8-cores Sum of Totients program on lxpara2

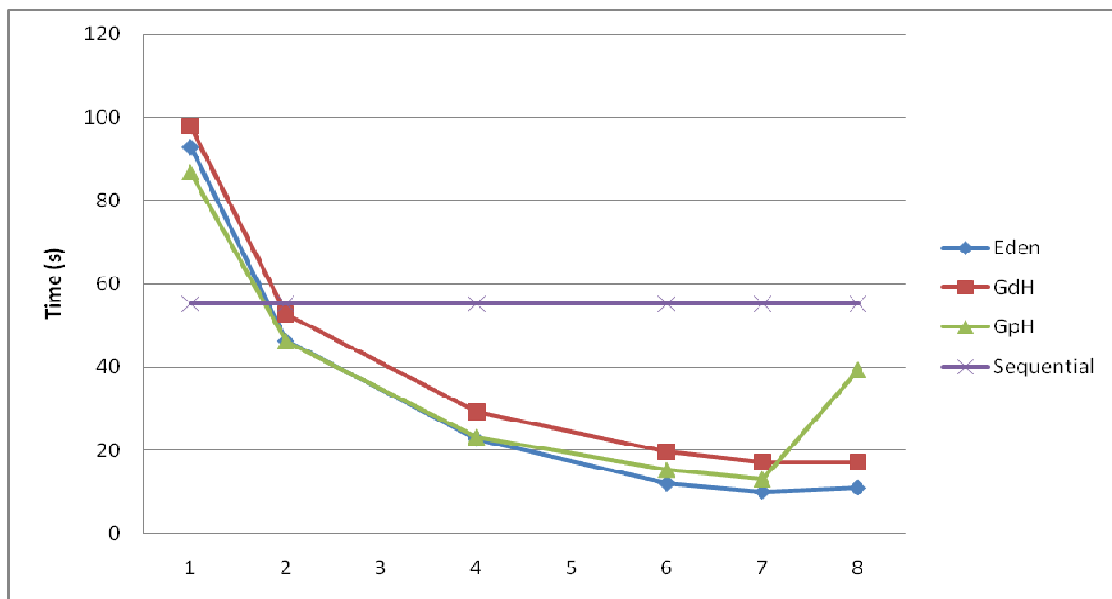


Figure 4.10 Runtime of up to 8-cores Sum of Totients program on Beowulf cluster

As can be seen, Eden was the quicker performer on average with GdH- performing the slowest for the majority of instances. The performance of GpH using 8 processors on the Beowulf cluster shows the runtime to significantly increase as predicted due to OS and system tasks taking up processor performance. The results are shown below:

Table 4.10 Runtime of up to 8-cores Sum of Totients program on lpara2

	1	2	4	6	7	8
Eden	75.9	36.3	18.0	11.8	9.9	8.6
GdH-	73.8	40.4	21.8	14.7	13.4	12.1
GpH	69.8	36.0	18.7	12.9	11.1	9.9

Table 4.11 Runtime of up to 8-cores Sum of Totients program on Beowulf Cluster

	1	2	4	6	7	8
Eden	92.9	46.2	22.8	12.1	9.9	10.9
GdH-	97.9	52.7	29.2	19.6	17.1	17.1
GpH	86.8	46.2	23.3	15.4	13.1	39.4

For the 8-cores to 224-cores run the input was increased to 100,000 with a chunk size of 400, giving 250 sparks which produced the following graph:

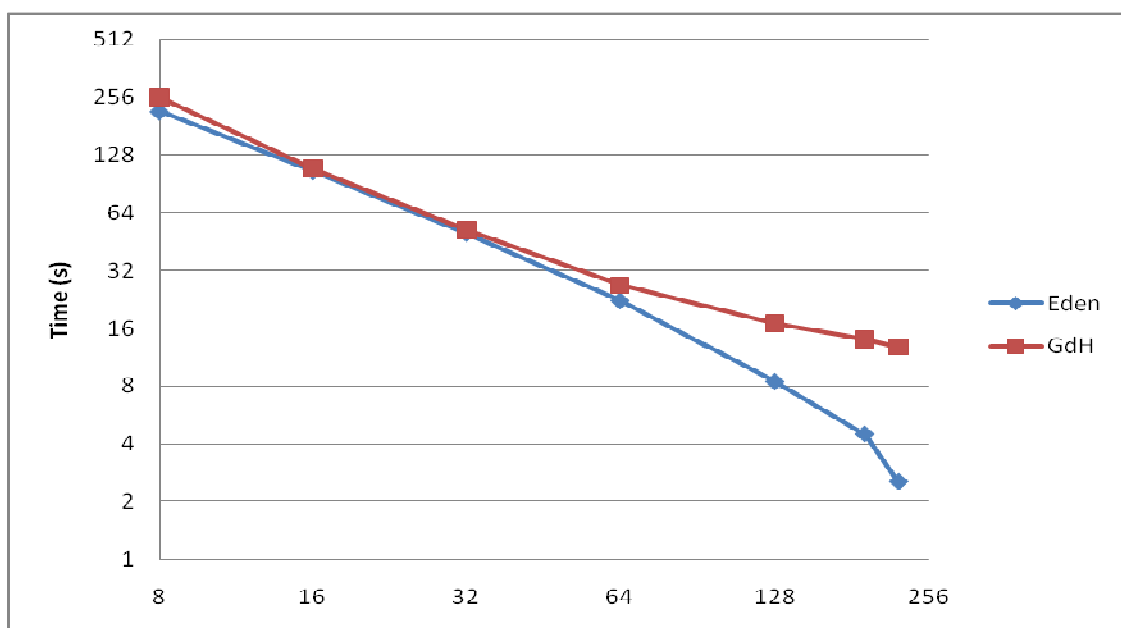


Figure 4.11 Runtime of up to 224-cores Sum of Totients program on Beowulf cluster

Once again Eden outperforms GdH- over the majority of cores, with GdH- showing little improvement in runtime by increasing cores. It is surprising to see Eden produce a relatively linear graph from the runtimes obtained, producing an overly superior performance. Beyond 128-cores GdH-'s performance appears to decline which is probably a result of how GdH- distributes its tasks. The results are shown below:

Table 4.12 Runtime of up to 224-cores Sum of Totients program on Beowulf Cluster

	8	16	32	64	128	192	224
Eden	215.9	105.2	50.1	22.3	8.4	4.5	2.5
GdH-	256.6	108.7	51.7	26.8	16.9	14.0	12.8

4.2.2 Speedup

The absolute speedup for the Sum of Totients program on the up to 8-cores run is shown below:

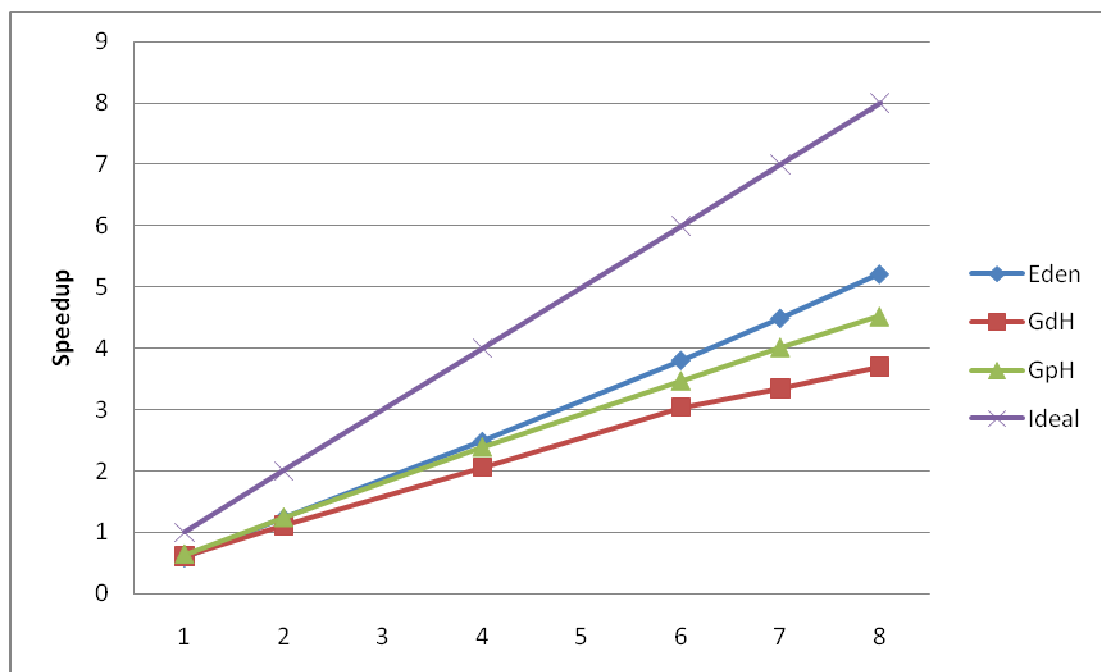


Figure 4.12 Absolute Speedup of up to 8-cores Sum of Totients program on lxpara2

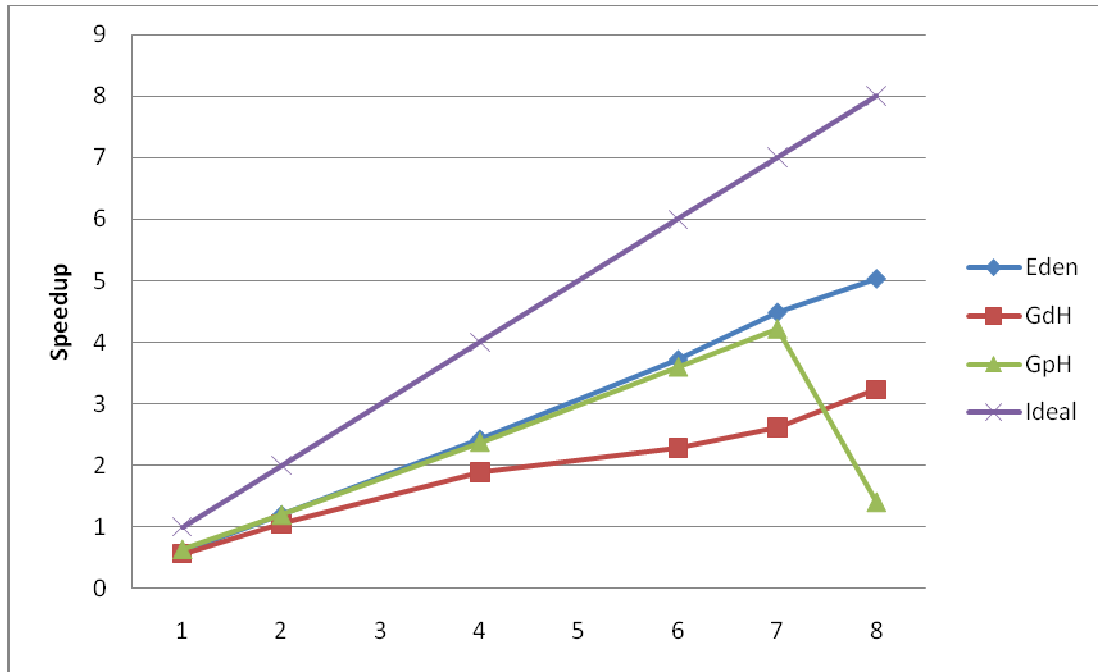


Figure 4.13 Absolute Speedup of up to 8-cores Sum of Totients program on Beowulf cluster

Once again Eden shows the best speedup over 7-cores on Ixpara2 with 4.5 but also shows the best on the Beowulf cluster with 4.4. GpH’s speedup was relatively steady as well only showing a drop on 8-cores on the Beowulf cluster due to the processor allocation over 8-cores. GdH- had the lowest speedups for the majority of instances but showed a steady improvement through the addition of cores on Ixpara2. However, on the Beowulf cluster it showed a slower trend and ultimately fewer speedups. The tables below show the speedups of the Sum of Totients program:

Table 4.13 Absolute Speedup of up to 8-cores Sum of Totients program on Ixpara2

	1	2	4	6	7	8
Eden	0.6	1.2	2.5	3.8	4.5	5.2
GdH-	0.6	1.1	2.1	3.0	3.4	3.7
GpH	0.6	1.2	2.4	3.5	4.0	4.5

Table 4.14 Absolute Speedup of up to 8-cores Sum of Totients program on Beowulf cluster

	1	2	4	6	7	8
Eden	0.6	1.2	2.4	3.7	4.4	5.0
GdH-	0.6	1.1	1.9	2.3	2.6	3.2
GpH	0.6	1.2	2.4	3.6	4.2	1.4

The relative speedup results of the Sum of Totients program on the Beowulf Cluster are shown on the following graph:

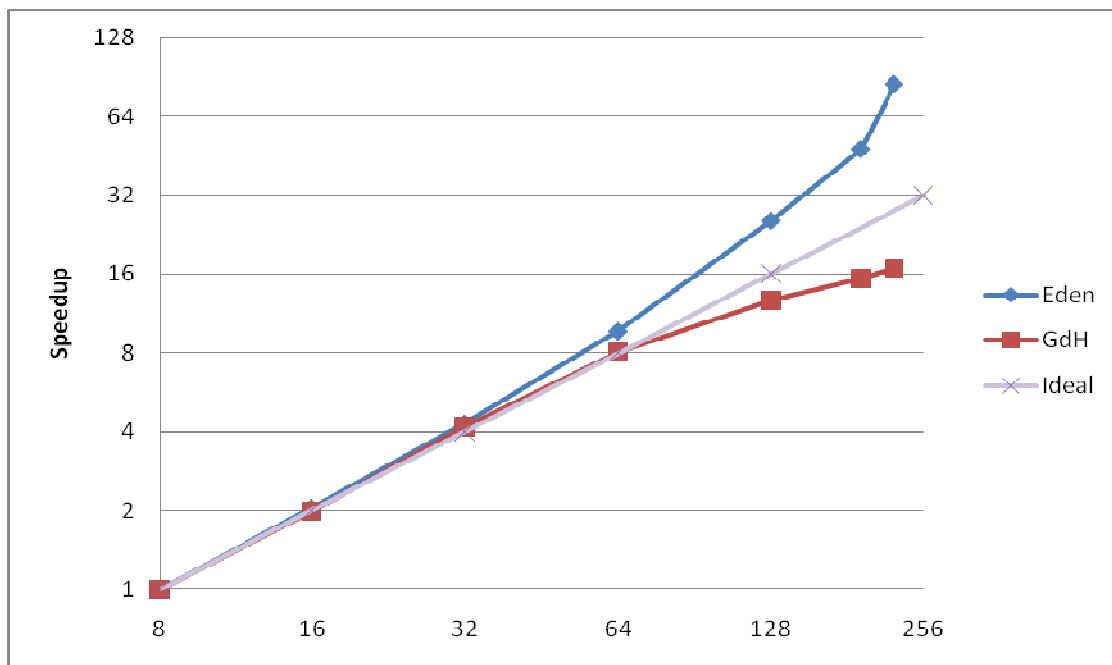


Figure 4.14 Relative Speedup of up to 224-cores Sum of Totients program on Beowulf Cluster

Although both Eden and GdH- showed similar relative speedups up to 32-cores, Eden clearly produced the higher speedup result from 64-cores onwards. As mentioned in the runtime graph, Eden appears to excel beyond expectation from 64-cores onwards whereas GdH- shows a decline. This is due to how each language distributes the relatively small 250 tasks produced. The relative speedups are shown below:

Table 4.15 Relative Speedup of up to 224-cores Sum of Totients program on Beowulf cluster

	8	16	32	64	128	192	224
Eden	1.0	2.0	4.3	9.7	25.6	48.1	85.2
GdH-	0.8	1.9	4.2	8.1	12.7	15.4	16.8

4.3.2 Variability

The charts and tables below show the variance percentages of the Sum of Totients programs on the up to 8-cores tests:

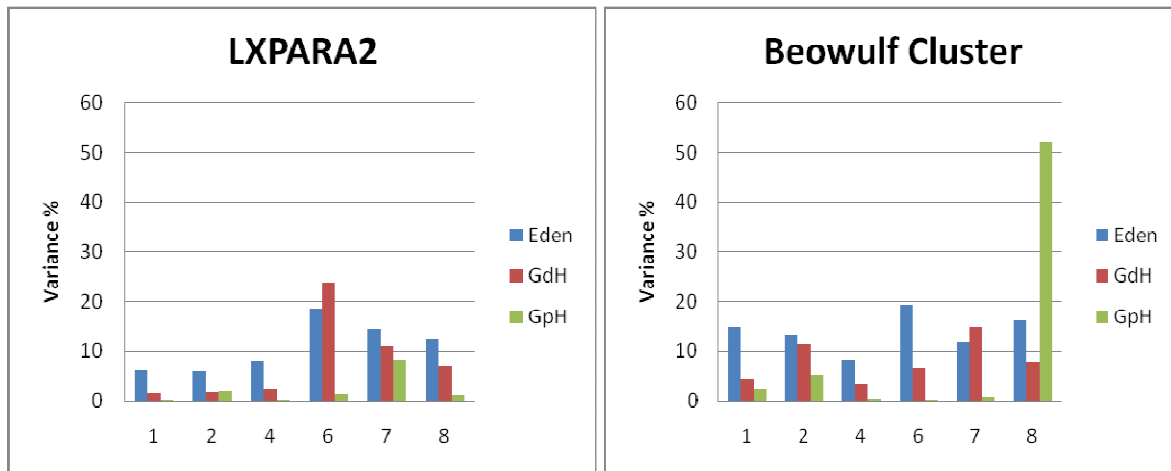


Figure 4.15 Variance of Sum of Totients program on up to 8-cores lxpara2 and Beowulf cluster

Table 4.16 Variance of up to 8-cores Sum of Totients program on lxpara2

	1	2	4	6	7	8
Eden	6.2	5.9	7.9	18.6	14.4	12.2
GdH-	1.5	1.7	2.2	23.8	10.9	6.9
GpH	0.1	1.9	0.1	1.4	7.9	1.1

Table 4.17 Variance of up to 8-cores Sum of Totients program on Beowulf Cluster

	1	2	4	6	7	8
Eden	14.9	13.4	8.0	19.2	11.9	16.2
GdH-	4.4	11.4	3.2	6.6	14.9	7.7
GpH	2.1	5.2	0.4	0.3	0.7	52.1

The first noticeable point of these results is the large percentage of variance - 52.1% - of GpH on 8-cores on the Beowulf cluster. The GpH program produced relatively low amounts of variance this result while GdH- and Eden had a similar trend in variance. The 8-cores to 224-cores variance results are shown in the following chart and graph:

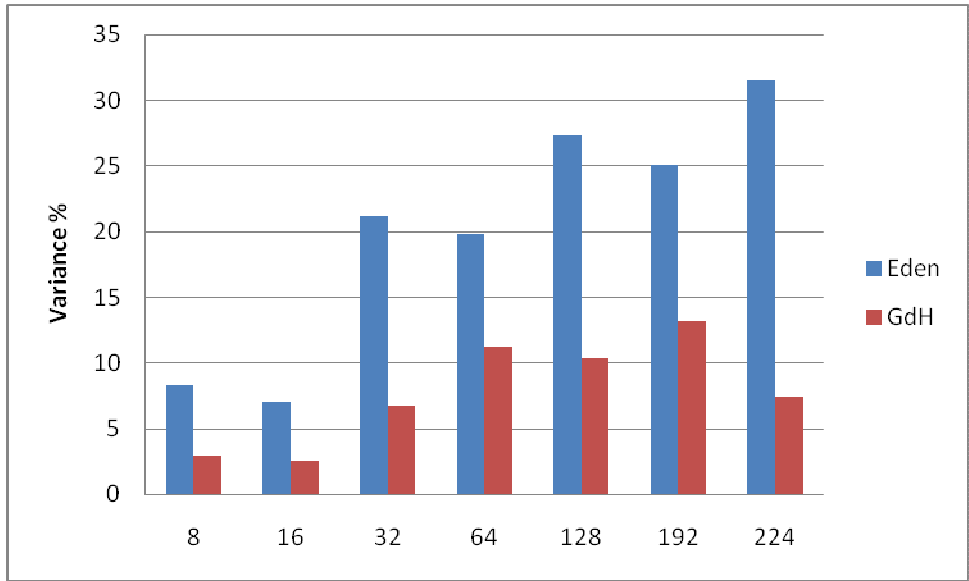


Figure 4.16 Variance of Sum of Totients program on up to 224-cores Beowulf cluster

Table 4.18 Variance of up to 224-cores Sum of Totients program on Beowulf Cluster

	8	16	32	64	128	192	224
Eden	8.3	7.1	21.2	19.8	27.3	25.1	31.5
GdH-	2.9	2.6	6.7	11.2	10.4	13.1	7.4

What can be observed here is GdH’s lower amount of variance compared to Eden. Both have a similar trend but Eden shows a greater amount of variance, averaging treble the amount of GdH-. However, as stated, this may be due to surprisingly accelerated speedup in these runs.

4.3 Queens

4.1.3 Runtime

The sequential runtime of the Queens program was 40.7 seconds on lxpara2 and 45.7 seconds on the Beowulf Cluster with an input of 13 producing 10,372 sparks. This resulted in the following graphs being produced from the 8-core runs:

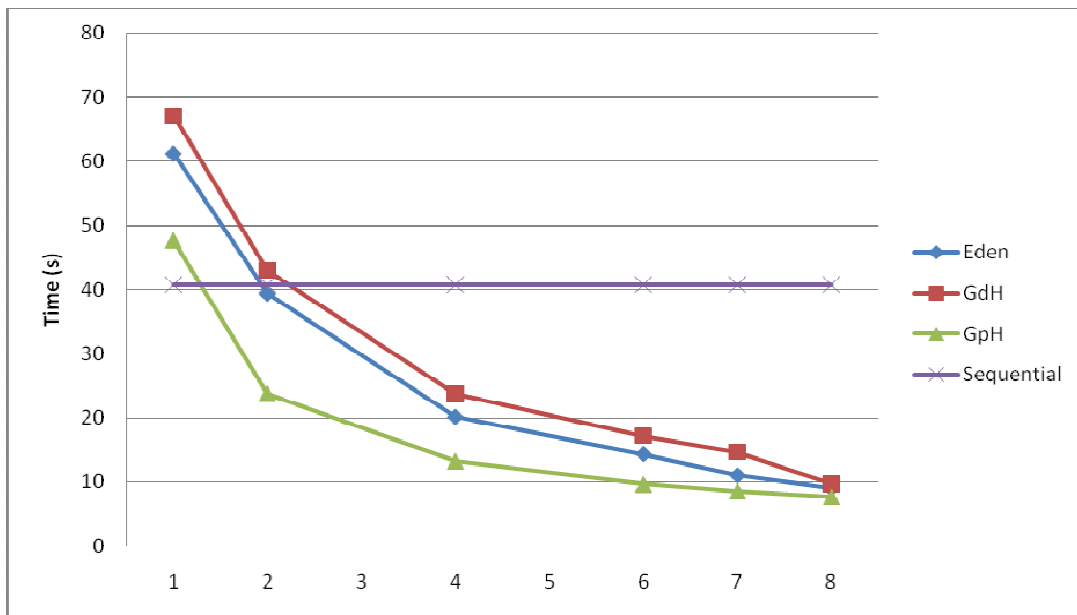


Figure 4.17 Runtime of up to 8-cores Queens program on lxpara2

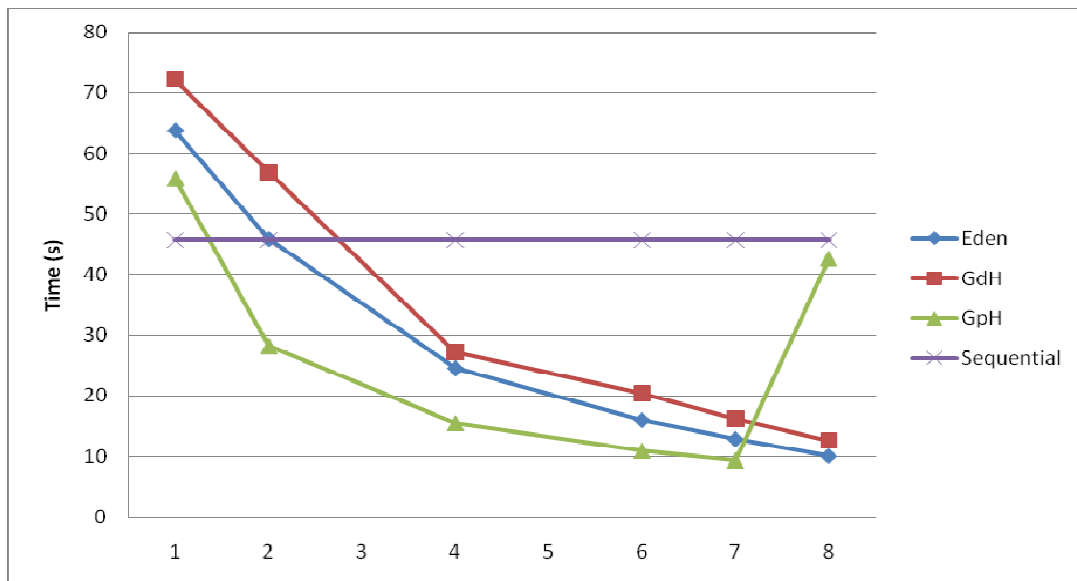


Figure 4.8 Runtime of up to 8-cores Queens program on Beowulf Cluster

GpH clearly showed the better runtimes up to 7-cores with the 8-cores runs once again producing unreliable results. GdH- gave the slowest runtimes again with a significant gap between it and Eden for the majority of instances. The tables below show the runtimes for the queens program on both systems:

Table 4.19 Runtime of up to 8-cores Queens program on lxxpara2

	1	2	4	6	7	8
Eden	61.2	39.3	20.2	14.3	11.0	8.9
GdH-	67.2	43.0	23.7	17.2	14.7	9.7
GpH	47.7	23.8	13.3	9.7	8.6	7.7

Table 4.20 Runtime of up to 8-cores Queens program on Beowulf Cluster

	1	2	4	6	7	8
Eden	63.8	45.9	24.6	16.1	12.9	10.1
GdH-	72.2	56.9	27.2	20.4	16.2	12.6
GpH	55.9	28.3	15.5	10.9	9.4	42.7

For the 8-cores to 224-cores run the test input was increased to 15, creating 15,480 sparks to produce the following runtime graph:

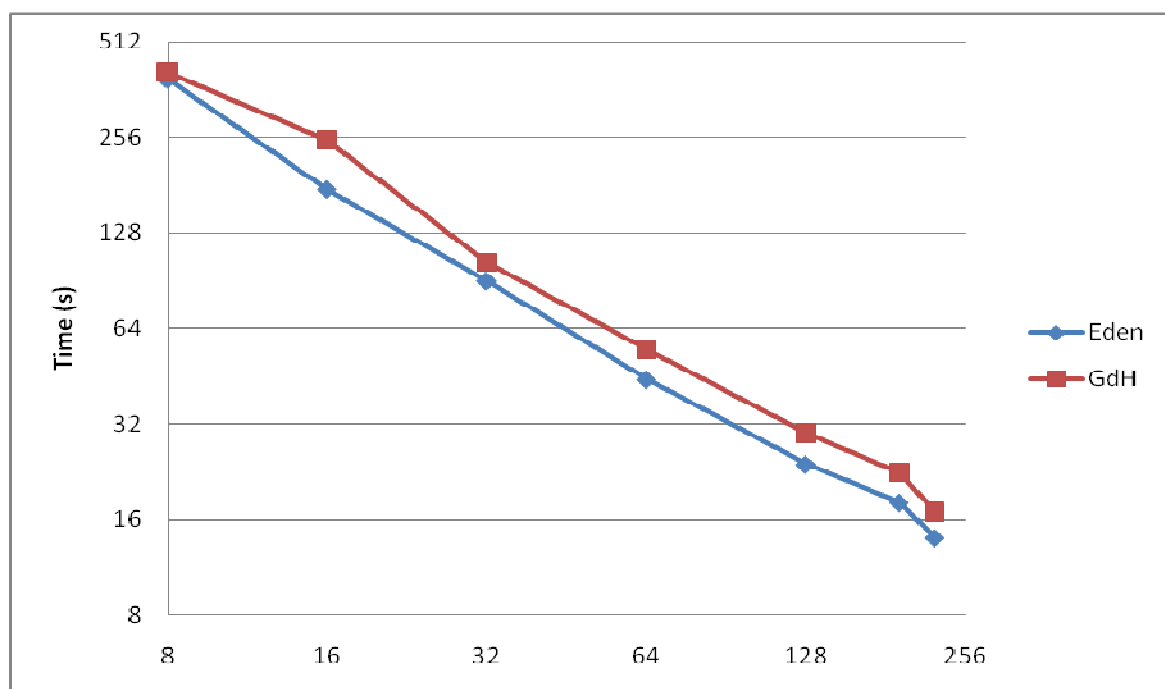


Figure 4.19 Runtime of up to 224-cores Queens program on Beowulf Cluster

Throughout the cores, Eden clearly outperforms GdH- steadily although both show a distinctly linear trend in runtimes. This is shown in the table below:

Table 4.21 Runtime of up to 224-cores Queens program on Beowulf Cluster

	8	16	32	64	128	192	224
Eden	390.8	175.9	90.4	44.3	23.9	18.1	14.0
GdH-	410.2	250.2	103.2	55.0	29.9	22.5	16.9

4.2.3 Speedup

The absolute speedup of the Queens program on both lxpara2 and the Beowulf cluster are displayed in the following graphs:

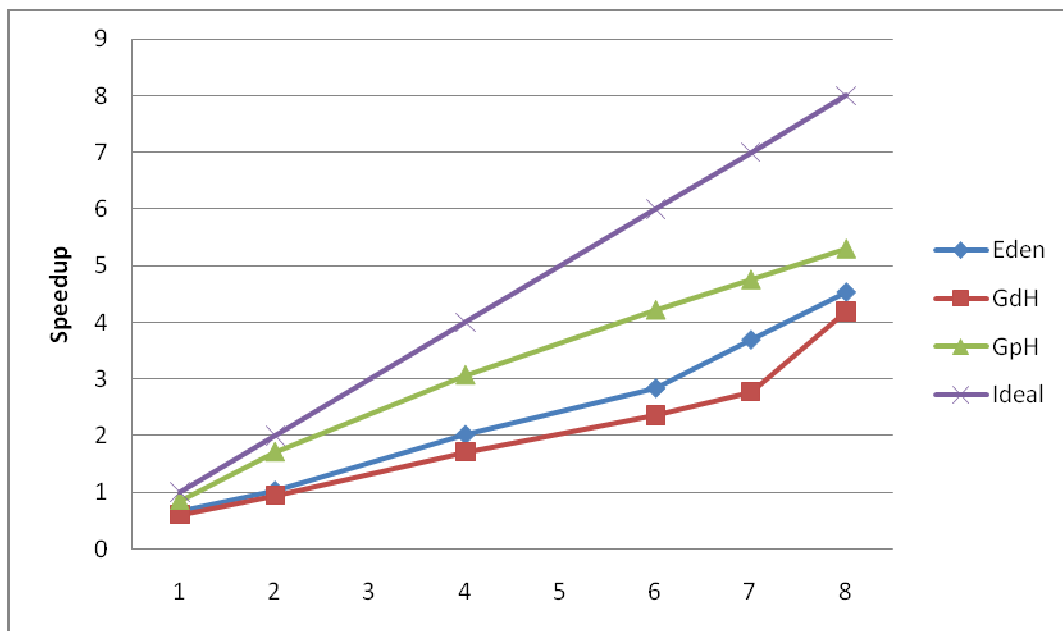


Figure 4.20 Absolute Speedup of up to 8-cores Queens program on lxpara2

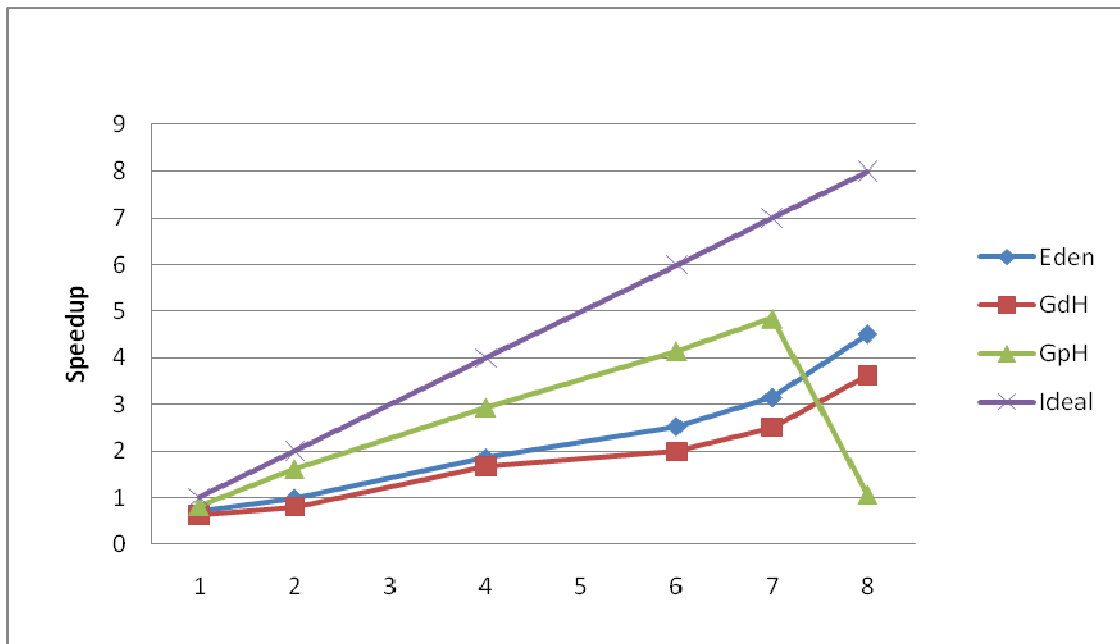


Figure 4.21 Absolute Speedup of up to 8-cores Queens program on Beowulf Cluster

For both lxpara2 and the Beowulf Cluster it is seen that GpH produces the best absolute speedup over these tests. Both Eden and GdH- showed steady improvements through the addition of more cores with GdH- showing the lower speedup results. The complete results for the 8-core runs are shown below:

Table 4.22 Absolute Speedup of up to 8-cores Queens program on lxpara2

	1	2	4	6	7	8
Eden	0.7	1.0	2.0	2.8	3.7	4.5
GdH-	0.6	0.9	1.7	2.4	2.8	4.2
GpH	0.9	1.7	3.1	4.2	4.8	5.3

Table 4.23 Absolute Speedup of up to 8-cores Queens program on Beowulf cluster

	1	2	4	6	7	8
Eden	0.7	0.9	1.9	2.5	3.2	4.5
GdH-	0.6	0.8	1.7	1.9	2.5	3.6
GpH	0.8	1.6	2.9	4.2	4.9	1.1

The relative speedup results on the 8-cores to 224-cores tests are shown on the following graph:

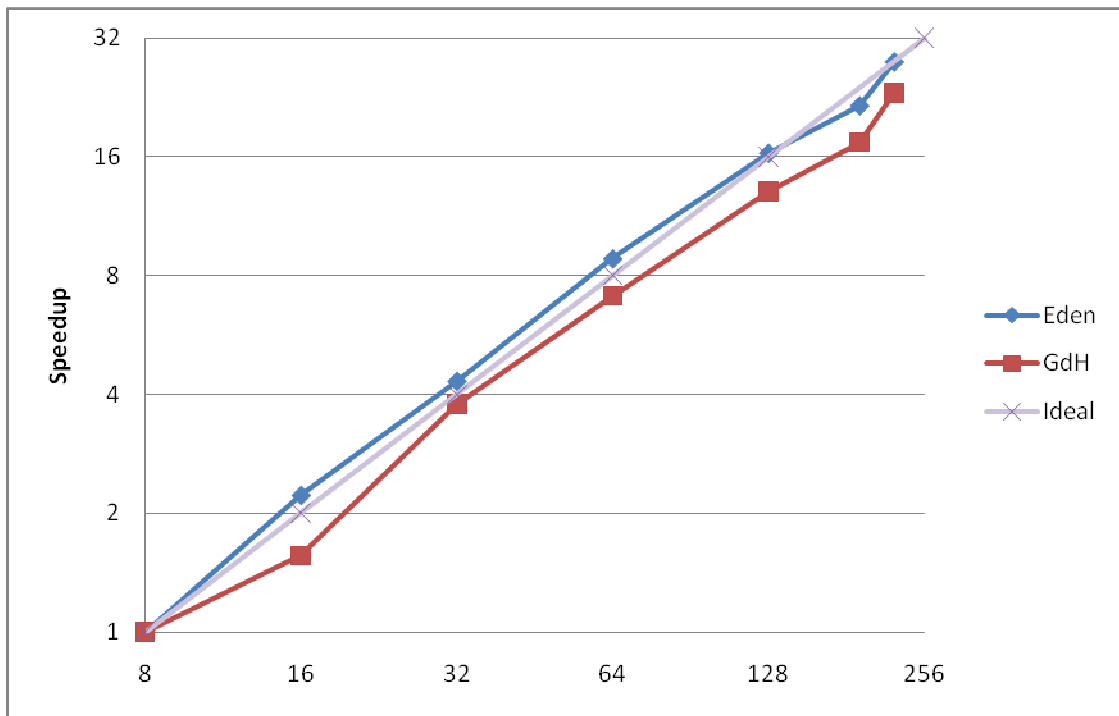


Figure 4.22 Relative Speedup of up to 224-cores Queens program on Beowulf Cluster

Much like the absolute speedups on the 8-core tests both Eden and GdH- show a similar trend for the relative speedups up to 224-cores. Once again Eden produces a larger speedup than GdH-, yet both showed a fairly linear trend for the relative speedups. The following table shows the speedups of the 8-cores to 224-core runs:

Table 4.24 Relative Speedup of up to 224-cores Queens program on Beowulf cluster

	8	16	32	64	128	192	224
Eden	1.0	2.2	4.3	8.8	16.4	21.6	27.9
GdH-	0.9	1.6	3.8	7.1	13.1	17.4	23.2

4.3.3 Variability

The bar charts and tables below show the variance percentages of the Queens programs on the up to 8-cores tests:

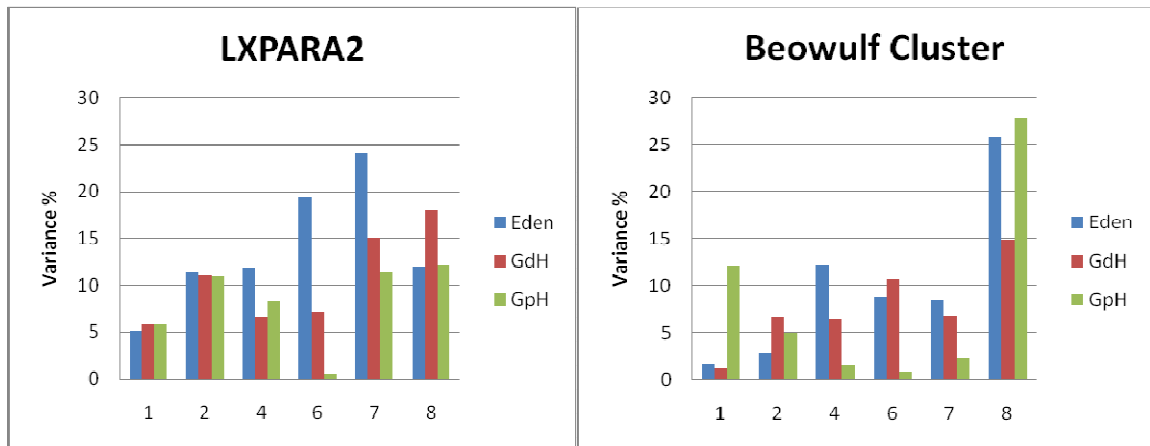


Figure 4.23 Variance of Queens program on up to 8-cores lxpara2 and Beowulf cluster

Table 4.25 Variance of up to 8-cores Queens program on lxpara2

	1	2	4	6	7	8
Eden	5.1	11.3	11.8	19.4	24.1	12.1
GdH-	5.8	11.0	6.5	7.2	15.1	18.0
GpH	5.8	10.9	8.4	0.6	11.3	12.1

Table 4.26 Variance of up to 8-cores Queens program on Beowulf Cluster

	1	2	4	6	7	8
Eden	1.7	2.8	12.2	8.8	8.5	24.8
GdH-	1.1	6.6	6.4	10.8	6.8	14.9
GpH	12.2	5.0	1.6	0.9	2.3	27.8

The most noticeable result is the higher variance of Eden compared to GdH- and GpH for the majority of test instances on the Queens program. All three languages also gave high percentages on 8-cores on the Beowulf cluster due to the unreliability of using all 8-cores. The variance for the 8-cores to 224-cores tests are shown in the following chart and table:

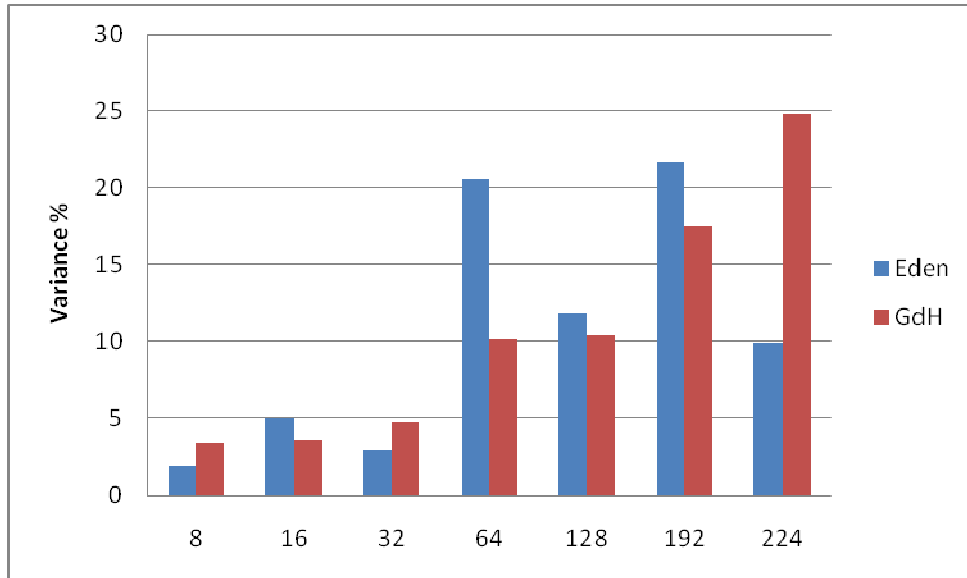


Figure 4.24 Variance of Queens program on 8-core lxpara2 and Beowulf cluster

Table 4.27 Variance of up to 224-cores Queens program on Beowulf Cluster

	8	16	32	64	128	192	224
Eden	1.9	4.9	2.7	20.6	11.8	21.7	9.9
GdH-	3.5	3.6	4.7	10.1	10.4	17.5	24.8

In these results GdH- shows a steady increase in variance as more cores are added, although it shows a similar or lesser percentage than Eden until 224-cores. It is at that point that GdH- gives its highest variance result of the Queens program of 24.8%, which is over double the percentage of Eden. However, it can be observed that both Eden and GpH's maximum variance on 192 and 224-cores respectively are relatively comparable.

4.4 Results Summary

This section summarises each language’s performance on these benchmarks by presenting the best runtime, speedup and variability in comparative tables. The runtime comparison table shown below presents each languages’ best (i.e. lowest) median runtime and the core number that it was achieved on each benchmark run. The best result of the three languages in each run is highlighted in bold.

Table 4.28 Comparison of median runtimes

	8-cores lxpara2			8-cores Beowulf			224-cores Beowulf	
	GpH	Eden	GdH-	GpH	Eden	GdH-	Eden	GdH-
Fibonacci	16.9 <i>8-cores</i>	12.3 8-cores	18.8 <i>8-cores</i>	17.6 <i>8-cores</i>	19.9 <i>8-cores</i>	15.5 <i>8-cores</i>	2.7 224-cores	2.8 <i>224-cores</i>
Sum of Totients	9.9 <i>8-cores</i>	8.6 8-cores	12.1 <i>8-cores</i>	13.1 <i>7-cores</i>	9.9 7-cores	17.1 <i>8-cores</i>	2.5 224-cores	12.8 <i>224-cores</i>
Queens	7.7 8-cores	8.9 <i>8-cores</i>	9.7 <i>8-cores</i>	9.4 7-cores	10.1 <i>8-cores</i>	12.6 <i>8-cores</i>	14.0 224-cores	16.9 <i>224-cores</i>

It is seen that Eden gives the best runtimes on the up to 224-core runs as well as the up to 8-core run Sum of Totients. These results will be further discussed in section 4.5.

The speedup comparison table is shown below. Once again this takes the best (i.e. highest) speedup achieved on each run with the best result of the three languages displayed in bold:

Table 4.29 Comparison of speedups

	8-cores lxpara2			8-cores Beowulf			224-cores Beowulf	
	GpH	Eden	GdH-	GpH	Eden	GdH-	Eden	GdH-
Fibonacci	4.4	5.9	3.9	5.1	4.5	5.8	23.1	22.3
Sum of Totients	4.5	5.2	3.7	4.2	5.0	3.2	85.1	16.8
Queens	5.3	4.5	4.2	4.9	4.5	3.6	27.9	23.2

As expected, this table mirrors the runtime comparison’s presentation of the languages which produce the best results.

The variability comparison table takes the highest variability percentage achieved from each run. The language which produced the lowest maximum percentage is determined to have achieved the best variance and is shown in bold. The table is presented below:

Table 4.30 Comparison of variability

	8-cores lxpara2			8-cores Beowulf			224-cores Beowulf	
	<i>GpH</i>	<i>Eden</i>	<i>GdH-</i>	<i>GpH</i>	<i>Eden</i>	<i>GdH-</i>	<i>Eden</i>	<i>GdH-</i>
Fibonacci	2.7	22.0	84.4	4.9	10.8	23.3	28.6	39.9
Sum of Totients	7.9	18.6	23.8	52.1	19.2	14.9	31.5	11.2
Queens	11.3	24.1	18.0	27.8	24.8	14.9	21.7	24.8

What can be seen here is that GpH consistently produces the lowest maximum variance on lxpara2. GdH- also produces low maximum variance on the up to 8-core Beowulf runs; however, a full discussion of each language’s performance will be given in the next section.

4.5 Discussion of results

From these results it can be seen how each language performs and how GdH- compares to both Eden and GpH.

GpH performed generally well on Ixpara2 and on the Beowulf node up to 7-cores but suffered when running on all 8-cores. For both the Queens and Sum of Totients programs its runtimes were larger than expected on 8-cores and brought down its speedup. This was due to the Beowulf node utilising one of its cores for system operations so that when GpH would use all 8-cores on the node this would cause the system to run more slowly than usual. This is a known issue in parallel programming and has been discussed by S.Marlow in his paper *Runtime Support for Multicore Haskell* [32] for the International Conference on Functional Programming 2009. Yet GpH still gave strong results up to 7-cores with it outperforming the other languages on both the Queens runs and the Fibonacci Beowulf Cluster run. It is also worth noting that although GpH did not always give the best speedup, its absolute speedup curves are the closest to linear and therefore it is the most predictable language. In addition to this GpH showed quite low variance for most test runs.

Eden, too, gave strong results with high absolute speedup results which would usually be the best speedup compared to GpH and GdH-. It is only on both of the Fibonacci runs on the Beowulf cluster and the Queens 8-core runs that Eden does not produce the best speedup result. Another point to note on Eden is its runtime curve for the Sum of Totients up to 224-cores. It is observable that, from 64-cores onwards, the runtime curve is unexpectedly linear which suggests that Eden distributes the relatively small 250 tasks produced quite efficiently or the 8-core run results were defectively high as a result of slow node. However, Eden does produce a high amount of variance in these test runs but its faster runtimes and its good results patterns make it the superior language in these tests.

GdH- did not perform as well as expected for the most part of these tests. It usually gave the highest runtimes resulting in it gaining relatively low speedups on small number of cores. One of the main issues, however, was its seemingly longer runtimes on lower

numbers of cores. Especially apparent in the Fibonacci results, GdH- struggled on 1 and 2 cores producing extremely high runtimes yet would drastically reduce its runtime from 4-cores onwards. This pattern also emerged in the Fibonacci results on 8-224 cores where GdH- gave high runtimes on 8 and 16 cores but radically dropped in runtime after more cores were added. This resulted in GdH- recording exceedingly high relative speedups. Although GdH- did have some instances where it outperformed GpH and Eden in runtime, there was never a distinct pattern emerging which showed GdH- to be consistently better. In terms of variance, GdH- generally had lower percentages than Eden although, in the Fibonacci results, GdH- showed extremely high variance.

Overall it is apparent that GdH- lags behind Eden and GpH in terms of performance. In most results there is not a vast gulf of difference between GdH- and the other two but with Eden and GpH regularly achieving lower run times than GdH-, it is apparent that Eden and GpH are currently better performing languages.

5. Conclusion

This section will summarise the achievements and results gained from this project as well as identifying the limitations. This section will also outline possible future work in this area.

5.1 Summary

This project has explored parallel computing, by benchmarking a new implementation of the parallel language Glasgow Distributed Haskell and comparing it to two other parallel languages: Eden and Glasgow Parallel Haskell.

A literature survey was undertaken exploring the emergence of parallel computing and the growing area of parallel programming.

A set of benchmark programs were developed for all three parallel languages which would allow a fair comparison of each language's capabilities. These programs covered three different types of parallel problems: data parallel in the Sum of Totients program; nested data parallel in the Queens program; and a divide and conquer problem in the Fibonacci program.

The benchmarks were tested on two different systems: an 8-core 32-bit machine and a Beowulf Cluster containing 32 8-core nodes. Both systems were used to test each benchmark up to 8-cores and the Beowulf cluster was used to test Eden and GdH- on larger test input up to 224-cores. These tests were used to measure the runtime, speedup, and variability of each parallel language. This gave a fair comparison to analyse the performance of each language and the effectiveness of GdH- in contrast to Eden and GpH.

The results were collated into a set of performance graphs and tables showing how each language compared to each other. A summary table was also constructed showing each languages best result for each program's runtime, speedup and variability.

The results showed that GdH- did not perform quite as well as both Eden and GpH, regularly recording longer runtimes than the other two languages. GdH- generally produced the lowest runtimes and gave the highest instance of variability with 84.4%. However the only instance were GdH- produced was with the Fibonacci test program over the higher number of cores on the Beowulf Cluster. Other than a lone absolute speedup of 5.8 on the Beowulf 8-core Fibonacci run, GdH- failed to get any absolute speedup results over 4.2. GdH-'s inferior results could be caused by the way in which the language distributes tasks.

GpH gave decent speedups and had relatively low variance as well as being the most predictable language of the three. GpH's highest absolute speedup was 5.3 on the lxpara2 Queens run on 8-cores.

Eden produced some of the best runtimes and over larger cores generally outperformed GdH-. It produced the highest absolute and relative speedup of 5.9 and 85.1 on the Fibonacci 8-core run on lxpara2 and the Sum of Totients 224-core run on the Beowulf Cluster respectively.

It was concluded from this benchmark testing that GdH- has not reached the same level as both Eden and GpH yet.

5.2 Limitations

The benchmarks used for this project test only a limited area of each parallel language. A more comprehensive evaluation of these parallel languages could have been achieved if more expansive benchmark implementations were added to these tests. However, this was not viable due to time constraints.

The maximum amount of cores tested for Eden and GdH- was 224 from a possible 256. This was due to the unreliability of using the maximum amount of processors available on the Beowulf Cluster with runs on 256-cores causing the program to stall or crash. Therefore, it was decided that 224-cores would be the maximum value on which to test these benchmarks.

When calculating the speedup on the Eden and GdH- benchmarks from 8-cores to 224-cores it was decided to calculate the relative speedup by dividing a result by Eden's runtime on 8-cores. Although this gives a good comparison of GdH-'s performance compared to Eden, determining the absolute speedup would render more conclusive results. However, the extremely large runtime on the sequential program for these test inputs made it not viable to calculate in the given timeframe.

5.3 Future Work

5.3.1 Extended Benchmark Suite

As stated earlier, these benchmarks only tested a limited area of each language's full potential. Increasing the range of test programs in either complexity or problem domain could produce a different set of results and possibly give an alternative evaluation of the GdH- implementation. Alternatively a much larger set of test inputs could be measured to evaluate GdH-'s performance under more intense processes. This could show the GdH- implementation to perform closer to Eden in these circumstances.

5.3.2 Investigation into contention on small number of cores with GdH-

From the results collected, one of the noticeable trends was GdH-'s high runtimes on 1 or 2-core runs. This high runtime, especially in the Fibonacci test program, was significantly larger than both Eden and GpH, resulting in GdH- initially being viewed as an inferior language. However, the addition of more cores greatly reduced the runtime and showed GdH- to be closer to Eden and GpH. There is a possibility that GdH- shows some contention when run on a small amount of cores and there is a chance to investigate why GdH- prefers performing on larger amount of cores.

References

References

[1] **Parallel Computer Architecture: A Hardware/Software Approach**

D Culler

Kaufmann publishing 1998

ISBN: 9781558603431

[2] **Introduction to Parallel Computing (2nd Edition)**

A Grama, A Gupta, G Karypis, V Kumar

Wesley Addison publishing 2003

ISBN: 9780201648652

[3] **Parallel Computing**

Available from:

<http://www.azalisaudi.com/para/Para-Week1-Intro.pdf>

Accessed on 3 March 2011

[4] **Overview of Parallel Computing**

Available from:

<http://www.psc.edu/training/petascale/lectures/ParallelComputingOverview.pdf>

Accessed on 5 March 2011

[5] **Data Parallel Architectures**

Available from:

<http://www-leland.stanford.edu/class/ee392c/notes/lec02/notes02.pdf>

Accessed on 8 March 2011

[6] **Introduction to Parallel Computing**

Available from:

https://computing.llnl.gov/tutorials/parallel_comp/

Accessed on 5 March 2011

[7] **Parallel Architectures**

M J Flynn

Stanford University article, March 1996

[8] **Parallel Programming: for Multicore and Cluster Systems**

T Rauber, G Runger

Springer publishing 2010

ISBN: 9783642048173

[9] **Many-Core Processor**

Available from:

<http://software.intel.com/en-us/articles/many-core-processor/>

Accessed 15 March 2011

[10] **The Art of Multiprocessor Programming**

M Herlihy, N Shavit

Kauffmann publishing 2008

ISBN: 0123705916

[11] **Valve goes multi-core**

J Reimer

Ars article 2006

Available from:

<http://arstechnica.com/gaming/news/2006/11/valve-multicore.ars>

Accessed 15 March 2011

[12] **Algorithms and Architectures for parallel processing**

S Yeo, J H Park, L T Yang, C Hsu

Springer publishing 2010

ISBN: 3642131352

[13] **Implicit and Explicit Parallel Programming in Haskell**

Available from:

<http://web.cecs.pdx.edu/~mpi/pubs/par.html>

Accessed on 14 March 2011

[14] **Moore's Law: Made real by Intel innovations**

Available from:

<http://www.intel.com/technology/mooreslaw/>

Accessed on 8 March 2011

[15] **Parallel Computing: Numerics, Applications, and Trends**

R Trobec, M Vajteric, P Zinterhof

Springer publishing 2009

ISBN: 1848824084

[16] **Pollack's Rule**

Available from:

http://en.wikipedia.org/wiki/Pollack's_Rule

Accessed on 8 March 2011

[17] **Implicit Parallelism**

Available from:

<http://www.cs.nmsu.edu/~epontell/adventure/node6.html>

Accessed 19 March 2011

[18] **Parallel Programming Model**

Available from:

http://en.wikipedia.org/wiki/Parallel_programming_model

Accessed 20 March 2011

[19] **The Science of Computer Benchmarking**

R W Hockney

Society for Industrial & Applied Mathematics

ISBN: 0898713633

[20] **A Gentle introduction to GpH**

Available from:

<http://www.macs.hw.ac.uk/~dsg/gph/docs/Gentle-GPH/sec-gph.html>

Accessed on 14 March 2011

[21] **GpH and Eden: Comparing two parallel functional languages on a Beowulf cluster**

H-W Loidl, U Klusik, K Hammond, R Loogen, P W Trinder

2000

[22] **Parallel Functional Programming in Eden**

R Loogen, Y Ortega-Mallen, R Pena

Journal of Functional Programming 15(3), 2005, pages 431-475

[23] **From (sequential) Haskell to (parallel) Eden**

S Breitinger, U Klusik, R Loogen

1998

[24] **The Design and Implementation of GdH-**

R F Pointon, P W Trinder, and H-W Loidl

Scottish Functional Programming Workshop, St Andrews, Scotland, July 2000

[25] **Review of GdH- August 1999-February 2000**

R F Pointon

Internal report, February 2000

[26] **Parallel Benchmarks and Comparison-Based Computing**

C P Breshars

1995

[27] **Principles of Parallel Programming**

Y C Lin, L Snyder

Wesley Addison publishing 2005

ISBN: 0321487907

[28] **Glasgow Parallel Haskell**

Available from:

<http://www.macs.hw.ac.uk/~dsg/gph/>

Accessed on 14 March 2011

[29] **Eden Parallel Functional Language**

Available from:

<http://www.mathematik.uni-marburg.de/~eden/>

Accessed 15 March 2011

[30] **Glasgow Distributed Haskell**

Available from:

<http://www.macs.hw.ac.uk/~dsg/GdH-/>

Accessed on 20 March 2011

[31] **The relationship of variance to interaction contrast in parallel systems factorial technology**

J T Townsend, A.Diedrich

British journal of Mathematical and Statistical Psychology, 1996

[32] **Runtime Support for Multicore Haskell**

S. Marlow, S.P. Jones, S. Singh

International Conference on Functional Programming, March 2009

[33] **Computer Language Benchmark Game**

Available from:

<http://shootout.alioth.debian.org/>

Accessed on 10 August 2011

[34] **MultiCoreChallenge SICSASE wiki**

Available from:

<http://www.macs.hw.ac.uk/sicsawiki/index.php/MultiCoreChallenge>

Accessed on 10 August 2011

[35] **Symposium Trends in Functional Programming**

Available from:

<http://www-fp.cs.st-andrews.ac.uk/tifp/>

Accessed on 10 August 2011

Appendix A – Program Listings

Eden

Fibonacci program: parFibEden

```
module Main where
import System
import Control.Parallel.Eden

main = do args <- getArgs
        let
            n = read (args!!0) :: Int
            putStrLn("parfib " ++ (show n) ++ " = " ++ (show (parfib n)))

-- Fibonacci numbers sequentially
fib :: Int -> Int
fib n | n < 2 = 1
      | otherwise = fib (n-1) + fib (n-2)

parfib n | n<2 = 1
         | n<20 = fib n
         | otherwise = nf1+nf2
           where
             nf1 = (process parfib) # (n-1)
             nf2 = (process parfib) #(n-2)
```

Sum of Totients program: parEulerEden

```
module Main where
import Control.Parallel.Eden
import System
import Control.DeepSeq
import IO

sumEuler :: Int -> Int -> Int -> Int
sumEuler lower upper sn = sum([ (process (\ z -> (sum . map euler) z)) # x
| x <- splitAtN sn (mkList lower upper) ]
    `using` seqList r0)

splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
    where (ys,zs) = splitAt n xs

euler :: Int -> Int
euler n = length (filter (relprime n) [1 .. n-1])

relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1

hcf :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

mkList :: Int -> Int -> [Int]
mkList lower upper = [upper, upper-1 .. lower]

main = do args <- getArgs
    let
        lower = read (args!!0) :: Int -- lower limit of the interval
        upper = read (args!!1) :: Int -- upper limit of the interval
        sn = read (args!!2) :: Int -- split
        hPutStrLn stderr ("Sum of Totients between [" ++
            (show lower) ++ ".." ++ (show upper) ++ "] is " ++
            show (sumEuler lower upper sn))
```

Queens program: parQueensEden

```
--Originally adapted from program provided by Mustafa Aswad

import Eden
--import ParPrim
import System.IO.Unsafe(unsafePerformIO) -- for Eden-6 skeletons below
import List(transpose)
import Control.Monad
import System.Environment
import Debug.Trace
-- version of N-queens originally from nofib/imaginary/queens, parallelised
-- by Simon Marlow 03/2010.

main = do
  [n] <- fmap (fmap read) getArgs
  print (nqueens n)

nqueens :: Int -> Int
nqueens nq = length (pargen 0 [])
  where
    safe :: Int -> Int -> [Int] -> Bool
    safe x d [] = True
    safe x d (q:l) = x /= q && x /= q+d && x /= q-d && safe x (d+1) l

    gen :: [[Int]] -> [[Int]]
    gen bs = [ (q:b) | b <- bs, q <- [1..nq], safe q 1 b ]

    pargen :: Int -> [Int] -> [[Int]]
    pargen n b
      | n <= threshold = iterate gen [b] !! (nq - n)
      | otherwise      = concat bs
      where
        bs = (parallelMap (pargen (n+1)) (gen [b]))
              where
                np = noPe
                parallelMap = mw np 2
    threshold = 10

-----
--- Eden stuff
-----

-- Eden parmap:
edenParMap :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
edenParMap f xs = unsafePerformIO ( mapM (instantiate (process f)) xs )

-- version with process parameter:
edenParMap' :: (Trans a, Trans b) => (Process a b) -> [a] -> [b]
edenParMap' proc xs = unsafePerformIO (mapM (instantiate proc) xs)

-- Eden process farm (np processes working on whole lists)
farm :: (Trans a, Trans b) =>
  Int -> (Int -> [a] -> [[a]]) -- n, distribute
```

```

        -> ([[b]] -> [b])           -- combine
        -> Process [a] [b]         -- worker process
        -> [a] -> [b]             -- what to do
farm np distr combine p inputs =
    combine (edenParMap' p (distr np inputs))

-- workpool (alias "master-worker") taken from TFP paper:
spawn :: (Trans a, Trans b) => [Process a b] -> [a] -> [b]
spawn ps is = unsafePerformIO (zipWithM (instantiateAt 0) ps is)

mw :: (Trans t, Trans r) => Int -> Int -> (t -> r) -> [t] -> [r]
mw n prefetch wf tasks = res
  where
    (reqs, res) = (unzip . merge) (spawn workers inputs)
    -- workers   :: [Process [t] [(Int,r)]]
    workers     = [process (zip [i,i..] . map wf) | i <- [0..n-1]]
    inputs      = distribute n tasks (initReqs ++ reqs)
    initReqs    = concat (replicate prefetch [0..n-1])

-- task distribution according to worker requests
distribute :: Int -> [t] -> [Int] -> [[t]]
distribute np tasks reqs = [taskList reqs tasks n | n <- [0..np-1]]
  where taskList (r:rs) (t:ts) pe | pe == r = t:(taskList rs ts pe)
    | otherwise = taskList rs ts pe
    taskList _ _ _ = []

-- functions to distribute and combine a list
-----
-- Helper: take each n-th element
takeEach :: Int -> [a] -> [a]
takeEach n [] = []
takeEach n (x:xs) = x : (takeEach n (drop (n-1) xs))

-- unshuffleN splits a list into n lists (round-robin)
unshuffleN :: Int -> [a] -> [[a]]
-- simple: unshuffleN n xs = [takeEach n (drop i xs) | i <- [0..(n-1)]]
-- optimised by UK (2001):
unshuffleN n xs = unshuffle xs
  where unshuffle xs = map (f xs) [0..n-1]
        where f xs i = g (drop i xs)
              g [] = []
              g xs = head xs : (g (drop n xs))

-- inverse:
-- shuffle . unshuffle == id
shuffle :: [[a]] -> [a]
shuffle = concat . transpose
-----

-- JB sez:
-- helpers:

chunk :: Int -> [a] -> [[a]]
chunk n [] = []
chunk n xs = front:chunk n r
  where (front,r) = splitAt n xs

```

GdH-

Fibonacci program: parFibGdH-

```
--Originally adapted from program provided by Patrick Maier
import Prelude hiding (catch)
import qualified Control.Concurrent
    as Conc (forkIO)
import qualified Control.Concurrent.MVar
    as MV (MVar, newEmptyMVar, putMVar, takeMVar)
import Control.Monad (unless)
import System (getArgs)
import System.IO (stdout, stderr, hSetBuffering, BufferMode(..))
import System.Random (mkStdGen, setStdGen, randomRIO)

import GdHminus (PEId, myPEId, allPEIds, revalIO)

-- local fork and join primitives
type LocalThread a = MV.MVar a
fork :: IO a -> IO (LocalThread a)
fork job =
    do t <- MV.newEmptyMVar
       Conc.forkIO (job >>= MV.putMVar t)
       return t

join :: LocalThread a -> IO a
join t = MV.takeMVar t

-- randomly scheduled remote fork (to go with local join)
rfork :: [PEId] -> IO a -> IO (LocalThread a)
rfork pes job =
    do myPE <- myPEId
       i <- randomRIO (0, length pes - 1)
       let pe = pes!!i
           -- Too much output apparently crashes PVM ...
           -- putStrLn $ " " ++ show myPE ++ " rfork to " ++ show pe
       fork (revalIO job pe)

parfib :: [PEId] -> Int -> IO Int
parfib pes n | n < 2      = return 1
              | n < 20    = return (fib n)
              | otherwise = do t <- rfork pes (parfib pes (n-2))
                               left <- parfib pes (n-1)
                               right <- join t
                               return $! (left + right)

-- Fibonacci numbers sequentially
fib :: Int -> Int
fib n | n < 2 = 1
      | otherwise = fib (n-1) + fib (n-2)
```

```
main :: IO ()
main =
  do hSetBuffering stdout LineBuffering
     hSetBuffering stderr LineBuffering

  -- read arguments
  args <- getArgs
  let
    x = read (args!!0) :: Int
  -- seed random number gen
  --unless (seed < 0) $
  --setStdGen (mkStdGen seed)

  -- compute on all PEs
  pes <- allPEIds
  putStrLn $ "parfib " ++ show pes ++ " " ++ show x ++ " = ..."
  y <- parfib pes x
  putStr $ "parfib " ++ show pes ++ " " ++ show x ++ " = " ++ show y
```

Sum of Totients program: sumEulerGdH

```
import Prelude hiding (catch)
import qualified Control.Concurrent
  as Conc (forkIO)
import qualified Control.Concurrent.MVar
  as MV (MVar, newEmptyMVar, putMVar, takeMVar)
import Control.Monad (unless)
import System (getArgs)
import System.IO (stdout, stderr, hSetBuffering, BufferMode(..))
import System.Random (mkStdGen, setStdGen, randomRIO)
import GdHminus (PEId, myPEId, allPEIds, revalIO)

main = do args <- getArgs
  let
    lower = read (args!!0) :: Int -- lower limit of the interval
    upper = read (args!!1) :: Int -- upper limit of the interval
    sn = read (args!!2) :: Int -- upper limit of the interval
    y <- computeSumEuler lower upper sn
    putStrLn ("Sum of Totients between [" ++
      (show lower) ++ ".." ++ (show upper) ++ "] is " ++
      show (y))

euler :: Int -> IO Int
euler n = return (length (filter (relprime n) [1 .. n-1]))

relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1

hcf :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

mkList :: Int -> Int -> [Int]
mkList lower upper = [upper, upper-1 .. lower]

splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
  where (ys,zs) = splitAt n xs

job :: IO Int -> PEId -> IO (MV.MVar Int)
job f pe = do mv <- MV.newEmptyMVar
  Conc.forkIO (revalIO f pe >>= MV.putMVar mv)
  return mv
```

```
computeSumEuler :: Int -> Int -> Int -> IO Int
computeSumEuler lower upper sn = do pes <- allPEIds
  let
    infpes = cycle pes
    pesData = zip infpes (mkList lower upper)
    splitpes = splitAtN sn pesData
  result <- mapM(mapM (\(pe,x) -> job (euler x) pe)) (splitpes)
  eulers <- mapM(mapM MV.takeMVar) (result)
  let
    eulef = (concat eulers)
  return (sum eulef)
```

Queens program: queensGdH

```
import Prelude hiding (catch)
import qualified Control.Concurrent
  as Conc (forkIO)
import qualified Control.Concurrent.MVar
  as MV (MVar, newEmptyMVar, putMVar, takeMVar)
import Control.Monad (unless)
import System (getArgs)
import System.IO (stdout, stderr, hSetBuffering, BufferMode(..))
import System.Random (mkStdGen, setStdGen, randomRIO)
import GdHminus (PEId, myPEId, allPEIds, revalIO)
import System.Environment

main = do
  [n] <- fmap (fmap read) getArgs
  res <- (nqueens n)
  print res

nqueens :: Int -> IO Int
nqueens nq = do len <- (pargen 0 [])
  return (length len)

where
  safe :: Int -> Int -> [Int] -> Bool
  safe x d [] = True
  safe x d (q:l) = x /= q && x /= q+d && x /= q-d && safe x (d+1) l

  gen :: [[Int]] -> [[Int]]
  gen bs = [ (q:b) | b <- bs, q <- [1..nq], safe q 1 b ]

  pargen :: Int -> [Int] -> IO [[Int]]
  pargen n b
    | n >= threshold = return (iterate gen [b] !! (nq - n))
    | otherwise       = do mp <- (parMap (pargen (n+1)) (gen [b]))
      return (concat mp)

  threshold = 10

parMap :: (a -> IO b) -> [a] -> IO [b]
parMap f xs = do pes <- allPEIds
  let
    infpes = cycle pes
    pesData = zip infpes xs
    mvars <- sequence (map (\(pe,x) -> job (f x) pe) pesData)
    result <- mapM MV.takeMVar (mvars)
  return result

job :: IO a -> PEId -> IO (MV.MVar a)
job f pe = do mv <- MV.newEmptyMVar
  Conc.forkIO (revalIO f pe >>= MV.putMVar mv)
  return mv
```

GpH

Fibonacci program: parFibGpH

```
module Main where
import System
# if defined(EVAL_STRATEGIES)
import Control.Parallel
import Control.Parallel.Strategies
import Control.DeepSeq
#endif

main = do args <- getArgs
        let
            n = read (args!!0) :: Int
            putStrLn("parfib2 " ++ (show n) ++ " = " ++ (show (parfib n)))

parfib :: Int -> Int
parfib n | n <= 1    = 1
         | n <= 20   = fib n
         | otherwise = x `par` (y `pseq` x + y)
                    where x = parfib (n-1)
                          y = parfib (n-2)

-- Fibonacci numbers sequentially
fib :: Int -> Int
fib n | n < 2 = 1
      | otherwise = fib (n-1) + fib (n-2)
```

Sum of Totients program: parEulerGpH

```
module Main(main) where
import System
# if defined(EVAL_STRATEGIES)
import Control.Parallel
import Control.Parallel.Strategies
import Control.DeepSeq
import Data.List
#endif
import IO

sumTotient :: Int -> Int -> Int ->Int
sumTotient lower upper sn = sum([sum(map euler n) | n <- splitAtN sn
(mkList lower upper)]
`using` parList rdeepseq)

splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
    where (ys,zs) = splitAt n xs

euler :: Int -> Int
euler n = length (filter (relprime n) [1 .. n-1])

relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1

hcf :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

mkList :: Int -> Int -> [Int]
mkList lower upper = [upper, upper-1 .. lower]

main = do args <- getArgs
    let
        lower = read (args!!0) :: Int -- lower limit of the interval
        upper = read (args!!1) :: Int -- upper limit of the interval
        sn = read (args!!2) :: Int --the number to split
        hPutStrLn stderr ("Sum of Totients between [" ++
            (show lower) ++ ".." ++ (show upper) ++ "] is " ++
            show (sumTotient lower upper sn))
```

Queens program: queensGpH

```
import Control.Parallel
import Control.DeepSeq
import Control.Parallel.Strategies
import System.Environment

main = do
  [n] <- fmap (fmap read) getArgs
  print (nqueens n)

nqueens :: Int -> Int
nqueens nq = length (pargen 0 [])
  where
    safe :: Int -> Int -> [Int] -> Bool
    safe x d [] = True
    safe x d (q:l) = x /= q && x /= q+d && x /= q-d && safe x (d+1) l

    gen :: [[Int]] -> [[Int]]
    gen bs = [ (q:b) | b <- bs, q <- [1..nq], safe q 1 b ]

    pargen :: Int -> [Int] -> [[Int]]
    pargen n b
      | n >= threshold = iterate gen [b] !! (nq - n)
      | otherwise      = concat bs
      where bs = (map (pargen (n+1)) (gen [b]))
                `using` parList rdeepseq
    threshold = 10
```

Appendix B – Full Results

Note: All results are sorted in descending order with the median highlighted

Eden

Fibonacci lpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	160.250	159.214	159.036	158.430	156.221	154.037	147.660
2-cores	80.998	80.037	79.725	79.580	78.638	78.024	77.714
4-cores	37.552	36.201	35.914	35.680	35.227	34.824	33.668
6-cores	20.925	20.503	20.436	20.113	20.031	19.880	19.250
7-cores	17.140	17.032	16.241	16.027	15.928	15.903	15.412
8-cores	14.585	13.040	12.627	12.331	12.071	11.947	11.870

Fibonacci Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	166.183	165.002	164.814	164.586	164.303	164.032	163.758
2-cores	92.750	90.314	89.202	87.384	87.048	86.321	85.167
4-cores	44.407	44.392	44.314	44.302	44.103	43.268	42.307
6-cores	22.667	22.052	21.632	21.250	20.936	20.843	20.365
7-cores	18.990	18.482	17.954	17.814	17.517	17.401	17.336
8-cores	20.307	20.108	19.932	19.851	19.761	19.785	19.722

Fibonacci Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
8-cores	63.120	63.077	63.038	62.994	61.834	61.032	60.331
16-cores	42.355	41.210	40.884	40.117	40.108	40.041	39.990
32-cores	25.850	25.039	24.715	24.268	23.741	23.238	22.715
64-cores	9.520	9.030	8.889	8.767	8.623	8.041	7.915
128-cores	5.640	5.581	5.237	5.046	5.003	4.952	4.620
192-cores	3.880	3.742	3.663	3.536	3.411	3.294	2.870
224-cores	3.115	3.041	2.863	2.725	2.700	2.623	2.440

Sum of Totients lpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	78.943	77.602	76.203	75.889	75.402	74.965	74.221
2-cores	37.021	36.926	36.837	36.343	35.916	35.325	34.884
4-cores	18.877	18.632	18.234	18.001	17.935	17.621	17.445
6-cores	13.070	12.532	12.068	11.749	11.571	11.253	10.885
7-cores	11.214	10.924	10.235	9.955	9.821	9.804	9.782
8-cores	8.965	8.803	8.774	8.583	8.310	8.029	7.921

Sum of Totients Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	108.242	102.842	96.315	102.599	100.342	98.664	92.929
2-cores	52.371	47.325	46.854	46.282	46.203	46.191	46.172
4-cores	24.536	23.421	23.100	22.790	22.781	22.736	22.704
6-cores	14.025	12.934	12.357	12.045	11.993	11.982	11.712
7-cores	10.850	10.712	10.068	9.987	9.731	9.689	9.666
8-cores	12.743	11.366	11.201	10.988	10.982	10.975	10.965

Sum of Totients Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
8-cores	221.885	219.360	216.447	215.947	209.784	207.603	204.020
16-cores	111.660	110.241	107.680	105.150	104.925	104.338	104.204
32-cores	57.850	53.777	52.650	50.132	49.836	49.221	47.220
64-cores	24.554	23.934	23.410	22.262	21.924	21.437	20.150
128-cores	9.250	8.919	8.846	8.423	7.347	7.224	6.950
192-cores	4.987	4.812	4.725	4.493	4.035	3.987	3.861
224-cores	3.050	2.803	2.714	2.536	2.502	2.346	2.250

Queens Ixpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	63.021	62.038	61.778	61.225	61.077	60.681	59.870
2-cores	42.664	40.873	39.831	39.328	39.071	38.764	38.210
4-cores	22.254	22.036	20.884	20.148	20.009	19.925	19.880
6-cores	15.990	15.191	14.858	14.338	14.003	13.854	13.214
7-cores	12.065	12.048	11.267	11.022	10.884	10.438	9.412
8-cores	9.525	9.274	9.005	8.984	8.903	8.675	8.440

Queens Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	64.200	64.192	63.899	63.852	63.611	63.301	63.110
2-cores	46.251	46.037	45.936	45.887	45.713	45.067	44.953
4-cores	25.851	25.003	25.001	24.585	23.741	23.632	22.850
6-cores	16.985	16.821	16.773	16.047	16.030	15.964	15.580
7-cores	13.311	13.084	12.926	12.890	12.714	12.523	12.221
8-cores	12.434	10.774	10.452	10.113	10.036	9.991	9.827

Queens Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
8-cores	392.400	392.010	391.884	390.770	387.569	386.735	384.712
16-cores	177.010	176.552	175.998	175.890	172.030	171.248	168.333
32-cores	91.886	91.035	90.671	90.443	89.905	89.392	89.201
64-cores	52.118	52.000	48.318	44.262	44.010	43.852	43.012
128-cores	24.669	24.558	24.097	23.880	23.197	22.364	21.850
192-cores	20.025	19.621	19.038	18.113	17.977	17.580	16.100
224-cores	15.044	14.622	14.581	14.028	13.927	13.821	13.654

GdH-

Fibonacci lpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	333.870	333.671	333.469	333.373	332.047	331.825	330.664
2-cores	246.816	239.500	237.484	232.651	231.905	230.964	207.258
4-cores	123.264	120.887	119.854	119.025	100.221	94.647	87.745
6-cores	60.981	47.651	46.069	44.024	42.130	38.444	27.381
7-cores	47.771	40.637	31.447	30.512	28.374	24.046	22.024
8-cores	28.313	19.224	19.003	18.848	18.705	18.425	18.288

Fibonacci Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	262.263	260.664	259.784	258.122	257.904	257.687	257.383
2-cores	182.245	161.228	158.713	153.855	150.620	149.774	148.579
4-cores	59.263	57.201	55.886	55.468	55.044	49.310	46.353
6-cores	37.001	36.376	36.074	35.770	35.662	34.994	34.587
7-cores	29.881	29.741	29.034	28.113	27.664	27.271	26.024
8-cores	17.023	16.173	15.667	15.510	15.458	15.481	15.407

Fibonacci Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
8-cores	454.432	432.880	419.334	410.406	403.667	400.308	398.725
16-cores	138.740	131.667	125.220	121.152	118.036	117.354	115.580
32-cores	45.114	43.089	42.113	39.665	39.210	37.821	36.795
64-cores	7.819	7.621	7.038	6.929	6.554	6.133	5.909
128-cores	5.909	5.403	5.002	4.364	4.261	4.207	4.164
192-cores	3.950	3.327	3.087	3.002	2.966	2.924	2.919
224-cores	3.198	3.055	2.902	2.826	2.771	2.458	2.115

Sum of Totients lpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	74.242	74.061	73.921	73.837	73.796	73.661	73.140
2-cores	40.462	40.458	40.430	40.414	40.300	39.828	39.793
4-cores	22.116	22.028	21.852	21.745	21.703	21.691	21.648
6-cores	17.704	15.368	14.952	14.730	14.521	14.288	14.194
7-cores	14.252	14.068	13.881	13.361	13.025	12.967	12.800
8-cores	12.144	12.130	12.102	12.084	11.705	11.302	11.169

Sum of Totients Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	99.876	99.407	97.936	97.899	97.321	97.006	95.612
2-cores	58.644	53.682	52.981	52.688	52.687	52.685	52.645
4-cores	29.480	29.382	29.228	29.225	29.021	28.933	28.534
6-cores	20.225	19.991	19.703	19.640	19.401	19.058	18.936
7-cores	19.107	18.335	17.652	17.110	17.036	16.928	16.550
8-cores	17.305	17.224	17.107	17.095	17.005	16.712	15.992

Sum of Totients Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
8-cores	257.810	257.358	256.927	256.599	255.658	255.302	250.240
16-cores	110.210	109.027	108.848	108.671	108.301	107.921	107.442
32-cores	54.458	52.066	52.017	51.744	51.683	51.205	50.985
64-cores	27.026	27.003	26.924	26.821	25.109	24.993	24.024
128-cores	17.800	17.031	17.004	16.990	16.822	16.085	16.042
192-cores	15.342	14.527	14.221	14.020	13.936	13.821	13.500
224-cores	12.850	12.844	12.837	12.821	12.358	12.109	11.900

Queens Ixpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	69.855	69.058	68.221	67.152	66.001	65.999	65.991
2-cores	44.965	44.307	44.024	43.004	42.225	42.010	40.220
4-cores	24.550	24.003	23.858	23.740	23.668	23.661	23.001
6-cores	17.822	17.706	17.355	17.214	17.025	16.788	16.587
7-cores	16.211	15.581	15.027	14.668	14.352	14.321	14.001
8-cores	10.778	10.030	9.822	9.714	9.699	9.652	9.025

Queens Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	72.668	72.428	72.301	72.225	72.003	71.958	71.854
2-cores	58.250	58.001	56.982	56.880	55.978	55.645	54.490
4-cores	28.002	27.868	27.586	27.223	26.881	26.678	26.260
6-cores	21.740	21.668	21.302	20.440	20.105	19.821	19.541
7-cores	17.110	17.092	16.884	16.225	16.201	16.107	16.000
8-cores	13.880	12.934	12.887	12.624	12.532	12.358	12.001

Queens Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
8-cores	423.440	421.581	413.846	410.225	410.221	409.994	409.240
16-cores	252.110	250.938	250.667	250.174	249.066	248.236	243.123
32-cores	104.850	104.620	104.239	103.220	101.871	101.365	100.003
64-cores	57.666	55.991	55.378	55.002	53.958	53.006	52.114
128-cores	31.664	30.282	30.001	29.870	29.268	28.994	28.550
192-cores	24.042	23.728	23.098	22.448	21.924	21.617	20.117
224-cores	19.067	18.031	17.824	16.850	16.002	15.936	14.885

GpH

Fibonacci Ixpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	121.472	121.025	120.956	120.500	119.880	119.743	118.802
2-cores	62.868	62.820	62.771	62.345	62.003	61.982	61.209
4-cores	32.020	31.999	31.851	31.787	31.720	31.695	31.652
6-cores	21.817	21.789	21.788	21.753	21.749	21.740	21.713
7-cores	18.934	18.920	18.913	18.907	18.882	18.801	18.743
8-cores	16.923	16.901	16.899	16.883	16.855	16.806	16.784

Fibonacci Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	130.094	129.553	127.416	126.939	126.027	125.997	125.875
2-cores	69.066	68.685	68.410	67.995	66.838	66.021	65.735
4-cores	34.489	34.455	34.402	34.374	34.307	33.952	33.166
6-cores	23.906	23.800	23.798	23.788	23.347	23.258	23.146
7-cores	20.133	20.102	20.094	20.090	20.082	20.044	20.016
8-cores	17.691	17.611	17.599	17.591	17.589	17.587	17.584

Sum of Totients lpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	69.850	69.841	69.831	69.826	69.801	69.785	69.762
2-cores	36.206	36.101	36.074	36.038	35.952	35.887	35.511
4-cores	18.714	18.712	18.706	18.704	18.701	18.698	18.693
6-cores	12.950	12.900	12.892	12.880	12.805	12.799	12.767
7-cores	11.901	11.625	11.604	11.127	11.104	11.015	11.012
8-cores	9.992	9.974	9.901	9.889	9.886	9.885	9.885

Sum of Totients Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	88.389	87.003	86.889	86.780	86.693	86.620	86.566
2-cores	46.347	46.308	46.292	46.229	44.851	44.358	43.934
4-cores	23.247	23.235	23.228	23.211	23.210	23.210	23.203
6-cores	15.371	15.371	15.369	15.367	15.361	15.358	15.324
7-cores	13.184	13.145	13.128	13.113	13.104	13.093	13.091
8-cores	47.554	42.579	40.887	39.375	32.471	32.066	27.021

Queens Ixpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	48.225	47.955	47.720	47.691	47.063	45.986	45.459
2-cores	24.628	24.214	24.014	23.792	23.701	23.647	22.014
4-cores	13.994	13.588	13.547	13.263	13.011	12.998	12.885
6-cores	9.665	9.662	9.649	9.647	9.625	9.619	9.612
7-cores	8.589	8.580	8.574	8.559	8.257	7.885	7.620
8-cores	7.950	7.902	7.774	7.678	7.485	7.221	7.020

Queens Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	60.751	59.048	58.722	55.946	55.082	54.777	53.953
2-cores	28.795	28.748	28.334	28.275	28.004	27.668	27.380
4-cores	15.548	15.544	15.543	15.534	15.397	15.388	15.294
6-cores	11.074	11.050	11.001	10.993	10.990	10.985	10.979
7-cores	9.59	9.587	9.480	9.416	9.403	9.385	9.373
8-cores	50.655	49.775	45.271	42.731	42.025	40.887	38.784

Sequential

Fibonacci Ixpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	74.294	74.002	73.884	73.656	73.482	73.423	73.410

Fibonacci Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	90.024	89.711	89.706	89.623	89.601	89.588	89.580

Sum of Totients Ixpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	45.032	44.925	44.892	44.758	44.709	44.659	44.651

Sum of Totients Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	55.887	55.594	55.447	55.250	55.184	55.003	54.878

Queens Ixpara2

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	41.234	40.884	40.828	40.692	40.604	40.603	40.558

Queens Beowulf

	Result 1	Result 2	Result 3	Result 4	Result 5	Result 6	Result 7
1-core	46.025	45.850	45.692	45.688	45.588	45.542	45.301
