

Automatic Parallelization of Data-Driven JStar Programs

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
at the
University of Waikato
by
Min-Hsien Weng



THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

University of Waikato
2013

Abstract

Data-driven problems have common characteristics: a large number of small objects with complex dependencies. This makes the traditional parallel programming approaches more difficult to apply as pipe-lining the task dependencies may require to rewrite or recompile the program into efficient parallel implementations. This thesis focuses on *data-driven* JStar programs that have rules triggered by the tuples from a bulky CSV file or from other sources of complex data, and making those programs run fast in parallel. JStar is a new declarative language for parallel programming that encourages programmers to write their applications with implicit parallelism.

The thesis briefly introduces the JStar language and the implicit default parallelism of the JStar compiler. It describes the root causes of the poor performance of the naive parallel JStar programs and defines a performance tuning process to increase the speed of JStar programs as the number of cores increases and to minimize the memory usage in the Java Heap. Several graphic analysis tools were developed to allow easier analysis of bottlenecks in parallel programs. The JStar compiler and runtime were extended so that it is easy to apply a variety of optimisations to a JStar program without changing the JStar source code. This process was applied to four case studies which were benchmarked on different multi-core machines to measure the performance and scalability of JStar programs.

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Mark Utting for the useful comments, remarks and engagement for the duration of this thesis. Furthermore, I would like to thank the Waikato Symphony administrators and NeSI Pan Cluster support team who assisted me during the final benchmark experiments. I would like to thank my family and my friends, who have supported me throughout the entire process by cheering me up all the time.

Contents

1	Introduction	1
1.1	Goals	1
1.2	PvWatts Example	3
1.3	Definitions	4
1.3.1	Workload	4
1.3.2	Executors	6
1.3.3	Dependency and Granularity	7
1.3.4	Latency and Throughput	9
1.3.5	Locality	10
1.3.6	Speedup	13
1.3.7	Efficiency	14
1.3.8	Scalability	15
1.3.9	Performance Trade-Offs	16
1.3.10	Perfect Parallelism	17
1.4	Contributions	18
2	Introduction to JStar	20
2.1	The JStar Language (Delta Tree and Gamma Database)	21
2.2	How the JStar Compiler Works	22
2.3	The Default Delta Tree and Gamma Database	23
2.3.1	The Delta Tree	24
2.3.2	The Gamma Database	29
3	Related Work	30
3.1	Library-based Shared Memory Programming Languages	31
3.2	Partitioned Global Address Space Languages	32
3.2.1	Titanium and X10	32
3.2.2	Chapel	33
3.3	Intel Concurrent Collections	34
3.4	Domain-Specific Languages	35
3.4.1	DSL Characteristics	36
3.4.2	The Delite Framework	37

4	Generating Parallel Code	40
4.1	Sequential (-seq flag) versus Parallel Code (-par flag)	40
4.1.1	Tuple Lifecycle	41
4.1.2	Optimisations	42
4.1.3	Code Generation	43
4.2	Speed of the Gamma Database	44
4.2.1	The Gamma Table Data Structure Choices	46
4.2.2	The <code>PvWattsHashTable</code> Gamma Table	48
4.2.3	Benchmark Result of the Sequential Tuple Insertion	49
4.2.4	Benchmark Result of the Parallel Tuple Insertion	52
4.3	Speed of the Delta Tree	54
4.3.1	The <code>DeltaNode</code> Data Structure Choices	55
4.3.2	Benchmark Result	56
4.4	Optimisation Strategy	56
4.4.1	<i>noDelta</i> Optimisation	57
4.4.2	<i>noGamma</i> Optimisation	58
4.5	Summary	59
5	Performance Tuning Process	60
5.1	Performance Tuning Procedure	60
5.2	Performance Tuning Tools	62
5.2.1	Task Dependency Graph	62
5.2.2	Debug Trace Output	63
5.3	Summary	66
6	Case Study: PvWatts	68
6.1	JStar PvWatts Program	68
6.2	Benchmark of the Naive JStar PvWatts Program	71
6.3	Performance Tuning Process	74
6.3.1	Analyzing Tuple Dependency	74
6.3.2	Inlining Tuples	75
6.3.3	Data Structures of the PvWatts Gamma Table	78
6.4	Phase Experiment	80
6.5	Disruptor Version	83
6.5.1	RingBuffer	83
6.5.2	Producer	85
6.5.3	Consumer	87
6.5.4	Disruptor PvWatts Program	90
6.5.5	Benchmark Result	93
6.6	Conclusion	94

7 Case Study: Dijkstra's Shortest Path Algorithm	95
7.1 Dijkstra's Shortest Path Algorithm	95
7.2 JStar Dijkstra Program	96
7.3 Benchmark Configuration	98
7.4 Performance Tuning Process	98
7.4.1 In-lining Tuples	99
7.4.2 Improving the Parallelism	101
7.4.3 Optimizing the Delta Tree Data Structures	102
7.4.4 Optimizing the Done Gamma Table	103
7.4.5 Optimizing the Edge Gamma Table	105
7.5 Conclusion	107
8 Case Study: Median-Finding	108
8.1 JStar Median-Finding Program	109
8.2 Benchmark Configuration	110
8.3 Performance Tuning Process	111
8.3.1 In-lining Tuples	111
8.3.2 Optimizing the Data Gamma Table	113
8.3.3 Benchmark Results	114
8.4 Conclusion	115
9 Case Study: Matrix Multiplication	116
9.1 JStar MatrixMult Program	118
9.2 Benchmark Configuration	119
9.3 Performance Tuning Process	120
9.3.1 In-lining Tuples	120
9.3.2 Optimizing the Matrix Gamma Table	121
9.3.3 Benchmark Results	122
9.4 Conclusion	123
10 Conclusions and Future Work	125
Appendices	132
A JStar Tuple Timing Graph Installation Guide	132
B Symphony Benchmark S.O.P	134
C NeSI Benchmark S.O.P	136
D Case Study: PvWatts	139
E Case Study: Dijkstra's Shortest Path Algorithm	143

F Case Study: Median-Finding	149
G Case Study: Matrix Multiplication	153

List of Figures

1.1	The ForkJoinTask diagram	7
1.2	Task dependency graph of a JStar program.	8
1.3	The Low-Latency and High-Latency diagram.	10
1.4	The Intel multi-core processor diagram.	11
1.5	The efficiency chart of a program on a 8-core machine.	14
2.1	The Delta tree diagram.	26
4.1	The lifecycle of a JStar tuple.	42
4.2	The lifecycle of a JStar tuple with three optimisations.	43
	(a) <i>noDelta</i> optimization.	43
	(b) <i>noGamma</i> optimization.	43
	(c) <i>noDelta</i> and <i>noGamma</i> optimizations.	43
4.3	The HashMap insertion behaviour diagram.	47
4.4	Performance of inserting tuples in sequential into the Gamma table.	50
	(a) Total execution time with varying initial capacity.	50
	(b) Speedup with varying initial capacity.	50
4.5	Performance of inserting tuples in parallel into the Gamma table.	53
	(a) Total execution time with varying Fork/Join pool size.	53
	(b) Speedup with varying Fork/Join pool size.	53
5.1	Task dependency graph of the JStar PvWattsGamma Program	63
5.2	The JStar tuple timing graph.	66
6.1	Table schema of the JStar PvWatts Program.	69
6.2	Performance of the naive JStar PvWatts program.	73

(a)	Total execution time with varying Fork/Join pool size.	73
(b)	Speedup with varying Fork/Join pool size.	73
6.3	Task dependency graph of the JStar PvWatts program.	74
(a)	The naive program	74
(b)	The optimized program	74
6.4	Speedups of the optimisation strategy.	76
6.5	Performance of the optimized parallel JStar PvWatts program.	78
(a)	Total execution time with varying Fork/Join pool size	78
(b)	Speedup varying with varying Fork/Join pool size	78
6.6	Speedups of the PvWatts Gamma table data struture.	79
(a)	Absolute speedup with varying Fork/Join Pool Size.	79
(b)	Relative speedup with varying Fork/Join Pool Size.	79
6.7	The workflow of the two-phase JStar PvWatts program.	81
6.8	Performance of the phase experiment.	82
6.9	The workflow between one producer and <i>RingBuffer</i>	86
6.10	The workflow among <i>RingBuffer</i> and 12 Consumers.	88
6.11	The workflow of the Disruptor PvWatts program.	91
6.12	Performance of the JStar and Disruptor PvWatts Programs.	93
7.1	The shortest path solved by the Dijkstra's algorithm.	95
7.2	Table schema of the JStar Dijkstra program.	97
7.3	Task dependency graph of the JStar Dijkstra programs.	100
(a)	The naive program	100
(b)	The optimized program	100
7.4	Speedups of the optimized JStar Dijkstra program.	102
7.5	Speedups of the <i>DeltaNodeInt</i> data structures.	103
7.6	Speedups of the <i>CHMDoneTable</i> data structures.	105
7.7	Speedups of the <i>Edge</i> data structures.	106
8.1	Table schema of the JStar Median-Finding program.	110
8.2	Task dependency graph of the JStar Median-Finding program.	112

(a)	The naive program	112
(b)	The optimized program	112
8.3	An example of the <code>CHMDataTable</code> data structure.	114
8.4	Speedups of the optimized JStar Median-Finding program. . .	114
9.1	The table schema of the JStar MatrixMulti program.	118
9.2	Task dependency graphs of the JStar MatrixMult programs. .	120
(a)	The naive program	120
(b)	The optimized program	120
9.3	Speedups of the optimized JStar MatrixMult program.	122

List of Tables

4.1	The JStar compiler option list.	59
5.1	The JStar program option list.	67
6.1	Benchmark configuration of the JStar PvWatts program. . . .	71
	(a) Hardware specification	71
	(b) Java Virtual Machine options	71
6.2	The inline tuple list of the JStar PvWatts program.	76
6.3	The configuration of the Disruptor PvWatts program.	92
7.1	Benchmark configuration of the JStar Dijkstra program. . . .	99
	(a) Hardware specification	99
	(b) Java Virtual Machine options	99
7.2	The inline tuple list of JStar Dijkstra program.	100
8.1	An example of the finding-median iterative algorithm.	108
8.2	Benchmark configuration of the JStar Median-Finding program. .	111
	(a) Hardware specification	111
	(b) Java Virtual Machine options	111
8.3	The inline tuple list of JStar Median-Finding program.	112
9.1	Benchmark configuration of the JStar MatrixMult program. . .	119
	(a) Hardware specification	119
	(b) Java Virtual Machine options	119
9.2	The inline tuple list of JStar MatrixMult program.	121

Chapter 1

Introduction

1.1 Goals

JStar is a new declarative language for parallel programming that encourages programmers to write their applications with implicit parallelism.[4] With JStar, programmers do not have to think how to make the program run in parallel, and they can just focus on the semantic of their algorithms and the order of execution. A key goal of JStar is that by applying optimization options, the JStar compiler can translate a JStar program into efficient sequential or parallel Java source code for a given architecture, without changing the JStar source.

Computers are all composed of at least one central processing unit (CPU) and a memory space for reading and executing the program instructions. Modern cores are implemented as silicon chips, which contain computing components and small circuits on it. Due to a large number of required circuits, early cores were large and power-consuming. Since the integrated circuit technology was invented, the transistor size has greatly reduced each decade by the continuous drive of *Moore law*, allowing faster clock speeds and more sophisticated architectures. By making the line width as small as 20 nano-meters, a CPU is able to contain over billions of transistors and electronic components, and thus the computing power can also be improved. However, this method is facing

technological and financial challenges as the transistor size is approaching the miniaturization limit.

Adding more cores on a single CPU is an alternative to increase the computing power. Ideally, the multi-core CPUs can multiply the speed of a program by the number of cores and shorten the communication time by sharing the same cache and bus interface. In fact, the parallel program often fails to scale up to large number of cores and has difficulties to be portable across heterogeneous platforms. Before this thesis started, the JStar compiler could generate sequential Java code, and a prototype parallel runtime based on the Fork/Join library and splittable `Hashsets` has been developed, but this has only been applied to one case study (matrix multiplication), where it showed the poor scalability[1].

This project aims to improve the performance of parallel JStar programs with concurrent data structures and efficient utilization of computing resources. Our goals are described as follows:

Speed is the execution time that a JStar program completes a given problem.

The goal is to maximize the speed and minimize the total execution time of a JStar program.

Scalability is the ability of a JStar program to increase its speed as the number of cores increases. The goal is to make the scalability as linear as possible.

Resource Usage is the total amount of computing resources which a JStar program uses. The goal is to minimize the usage, including the heap size, memory bandwidth, CPU usage, and garbage collection.

Portability is the ability of a JStar program to be run on multi-platforms.

The goal is to directly run a JStar program across different platforms without extra efforts, such as rewriting the source codes or re-compilation.

All of our goals will be explained with a photovoltaic (PV) energy system example in the next section.

1.2 PvWatts Example

A photovoltaic (PV) station is a solar energy production system with arrays of solar panels. It can continuously convert the sunlight into the direct current electricity (DC Power). Compared to other power generation methods, the PV energy generation method produces no pollution when operating and uses sustainable energy sources. Furthermore, the solar panels are easily mounted on the rooftop or on the ground. The solar power has many benefits to the environment and human beings so that many people have started to be interested in the PV station installation. But the energy production of a PV station is mainly determined by the weather. Thus, the location becomes an important issue for a PV system.

NREL(National Renewable Energy Laboratory) provides a tool to assist people to make this decision. The *PvWatts* program¹ is an energy calculator, simulating a PV energy system in a area and estimating the hour-by-hour power production.[24] The PvWatts program uses the historic weather data in a location to determine the intensity of solar radiation on the PV arrays. By using the parameters of solar arrays and the efficiency of power conversion, the PvWatts program estimates the energy generation (in Watts) for each hour of the year. And all the estimated records for a typical meteorological year (TMY) are exported to a data file (a CSV file). This file consists of one year of records, where each row in the table represents one energy record. The hourly record is composed of the time data and the AC power. The time data has 4 fields: year, month, day, and hour. And the AC power is the electricity wattage generated during an hour.

The parallel JStar PvWatts program could make use of the multi-core computing power to shorten the execution time and increase the speed. As this program calculates the total power generation for each month of the year, it reads each hourly record in the file and averages the monthly energy production. The program starts by parsing command line arguments, and ends

¹See <http://www.nrel.gov/rredc/PvWatts/>

with printing out monthly power production for a PV station. Because the PvWatts program uses readers to read each record in the table, I/O communication between the data file and system memory may limit its execution time. Through the JStar parallelism, the speed of a JStar program could increase as the number of employed cores.

The parallel JStar PvWatts program could reduce the memory usage during the execution. As the input file contains a large number of hourly records, the program could use most of the memory space to create the data objects, which are used at one time. These short-lived objects may cause load on the Java garbage collector, or worse, may lead to the out-of-memory error. To avoid downgrading the performance from the busy garbage collector, the JStar parallelism should efficiently utilize the memory space on the multi-core machine and also ensure the program to function correctly.

The parallel JStar PvWatts program should be run on the multi-core machine without changing the original JStar source program but simply by setting the number of cores (threads). As each parallel hardware has different characteristics, porting a parallel program often requires the programmer to rewrite and recompile the source code. But rewriting the program sometimes may have a good performance but introduce unexpected errors during the execution. The JStar parallelism should implicitly hide the parallel hardware and provide a simple mechanism to make use of the multiple cores.

1.3 Definitions

1.3.1 Workload

The Java application is a sequence of actions expressed in Java language. A Java program needs to be compiled into the platform-independent byte-code, so that the Java Virtual Machine (JVM) can execute it.[5] JVM is not real hardware but a computer program. It provides a run-time environment for the Java byte-code and can be run on heterogeneous computer systems. JVM

makes the Java applications to be portable across a variety of platforms. Apart from dynamic memory allocation, the JVM can use a fixed-size memory space (heap) to store objects during the execution of a program. The JVM has three definition for computation workload[5]:

Task is a set of the program instructions. It is the smallest unit to measure the application's workload. In JStar language, a task is regarded as a *Rule* which use the input tuples to do some computation and output the tuples. For example, one rule in the JStar PvWatts program is triggered after a request from command line argument is received. It reads lines from the input file, parses the fields and creates **PvWatts** tuples with the the date and time and the hourly power production.

Thread is used to execute one or more tasks.

Process is an execution environment which can manage its own memory space and execute many threads in parallel. A single process can execute the threads in the thread pools to run tasks.

The process and thread both have many common characteristics.[18] They both provide a execution environment. But a process has better control over its memory space. The JVM is an example of a single process. As the threads are created and executed in one pool inside the JVM, threads inside a process can use the process's memory space to communicate with each other. A thread is a light-weight process, taking up fewer resources than a process and quickly being created and destroyed. A thread can execute short-lived tasks efficiently while a process can execute the threads concurrently.

Multi-process applications are more complicated. Because inter-process communication requires additional implementations, such as the message passing interface, or the remote procedure calls. Thus exchanging the messages among the processes takes longer than in an multi-threading application and may cause latency problems.

1.3.2 Executors

The *Executors* class is one of these high-level concurrent objects provided by the Java platform.[5] The *Executors* class provides a separate way of launching and managing the threads in a Java concurrent application.[17] Using the executor can separate the application from the thread management. The executor creates all the threads automatically in a pool. When a thread finishes its task, the executor would either reuse it to do other tasks or destroy it permanently. As the executor manages the threads from creation to termination, Java programmers do not need to write extra code to deal with the thread life-cycle issues, which would sometimes cause dead lock. Two executor implementations are:

Thread Pool is introduced in JDK 1.5 and widely used in concurrent applications.[5]

In the pool, the executor creates a number of *worker threads*, which are able to perform tasks on behalf of the application. A worker thread can normally execute multiple tasks, so the number of threads is often less than the number of tasks. After a thread completes its task and returns its result, it would continue taking up a new task until all the tasks have been processed.

Fork/Join Framework is implemented on JDK 7 and tries to use all computing power of the multi-cores machine.[5] Like other Java threading frameworks, the Fork/Join framework uses a `ForkJoinPool` to host all worker threads and distribute tasks to them. The `ForkJoinTask` is a task that runs within the `ForkJoinPool`. At the beginning, an initial `ForkJoinTask` is submitted to the pool and executed by a worker thread automatically. This main `ForkJoinTask` is split into more `ForkJoinTasks` by recursively calling the `Fork` method and these subtasks are asynchronously executed by other available threads. When a worker thread receives a task, the thread will either split the task or process it immediately. If the task work is too heavy, then the thread will split this task

into two sub-tasks, submit them to the pool for future execution (waiting for their result). Otherwise, the thread will execute the task and return its result immediately. The *split-merge* procedure will not stop until all of the sub-tasks have been completed and the main task gets its results.

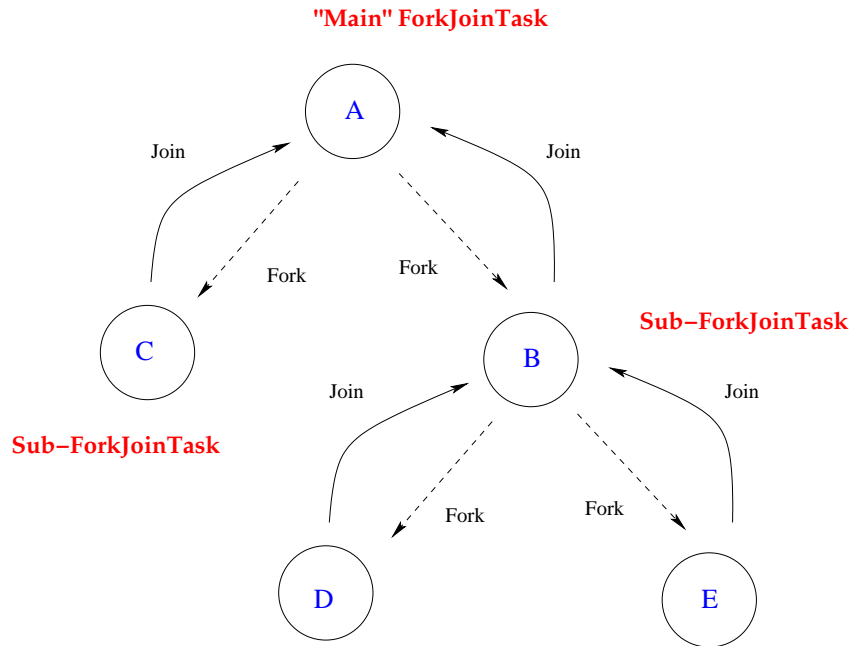


Figure 1.1: The ForkJoinTask diagram

Figure 1.1 illustrates the procedure of the Fork/Join framework. When the *main* ForkJoinTask is submitted to the ForkJoinPool, this main task will be split into two sub-ForkJoinTasks (B and C) by one thread. When a thread starts computing the left subtask, the right sub-task will be split into two sub-tasks (D and E). When two individual threads start to run Task D and Task E, Task B is being held and waiting for the completion of Task D and Task E. After summing up the results of Task D and Task E, Task B rejoins the Task A. Once Task C finishes, Task A aggregates the results of Task B and Task C, and outputs the final result.

1.3.3 Dependency and Granularity

Dependencies define the execution sequence of tasks. Dependencies can ensure that the behaviour of a program is run according to what users expected.

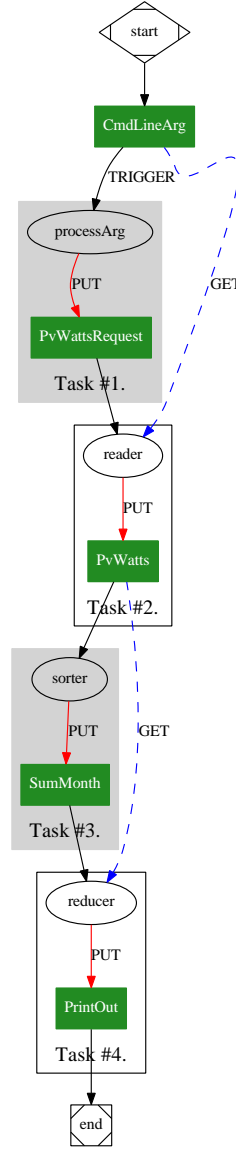


Figure 1.2: Task dependency graph of a JStar program.

We will demonstrate the task dependencies with the JStar PvWatts example. Each JStar task is composed of one rule and its output tuples, and optional the query tuple. The *Rule* is the task and sometimes needs tuples from other tables (query tuples). After finishing the computation, each rule/task outputs tuples which is used to trigger other tasks. The order of tuple execution are formed in a connected graph, as shown in the Figure 1.2. Even though the task dependencies may limit the parallelism, they provide a way of reasoning the correctness of a JStar program. For example, sorters are used to categorize the **PvWatts** tuples by the month value. The sorters in this program are used to force the reducers to wait until the readers finish their work. Even though

the sorters prevents the concurrent execution of the reducers and the readers, they ensure that the reducers obtain a complete and full set of the **PvWatts** tuples and output the correct final result.

The *granularity* can relax the constraint caused by the task dependencies. Based on the frequency across the threads, the granularity is classified as *coarse* or *fine*. The coarse-grain computation refer to the loose dependencies among threads while the fine-grain one refer to the close dependencies. For example, if the JStar PvWatts program uses two readers to read the input file in parallel, then the file will be divided into two segments and two readers are created to read each one of them concurrently. Since each reader just needs to read one half of the whole records, the reading time can be shorten. However, if the file is chopped too fine, then the program will create too many readers and cause the increase of the overhead costs and reduce the benefits of granularity. For example, if we use millions of parallel readers, then streaming each file segment will take up a lot of time and slow down the performance. Besides, compared with a single reader, the parallel readers need to spend the extra time synchronizing records. So the level of granularity should be set up to meet the hardware specification and the users' needs.

1.3.4 Latency and Throughput

The *latency* refers to the amount of time to complete a unit of work and the *throughput* are the amount of work that can be finished per unit time. They are both used to measure the performance of a parallel program but have different ways of achieving the parallelism. Figure 1.3 illustrates the difference between the low and high latency. The low-latency runs each task one after one and the high-latency run each task with an overlap of starting time. The low-latency method has the shorter task time but a longer completion time than the high-latency as the task needs to waiting for the last task. The low-latency method can have a shorter task time but the high-latency have a better throughputs. Note that the latency can be hidden by using the multi-core machine. When

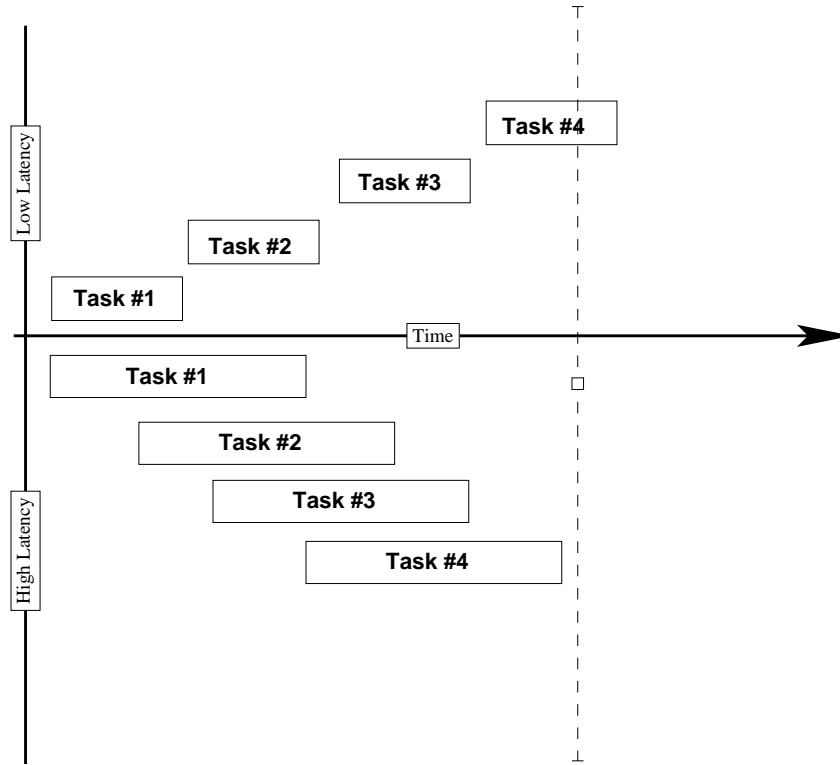


Figure 1.3: The Low-Latency and High-Latency diagram.

one thread is executing a high-latency task, the multi-core machine may create two or more threads to do other tasks on the same core. As these threads share the data in the cache of the core, the latency from the memory can be hidden and thus the execution time will be shortened. But this method does not really reduce the latency but the number of requests to the main memory.

1.3.5 Locality

The *cache* is the memory space between the cores and the main memory space and has been implemented widely in the modern CPUs for its low-latency performance. By pre-fetching the recent data off the main memory, a core can directly read and write the data in the caches without the communication between the memory buses and the controllers. As the cache increase accessing speed, the overall performance can be improved. Figure 1.4 illustrates the memory reference in a multiple cores environment.[13] According to the cache block size, the caches are classified into three hierarchies: Level 1 (L1), Level 2 (L2) and Level 3 (L3).

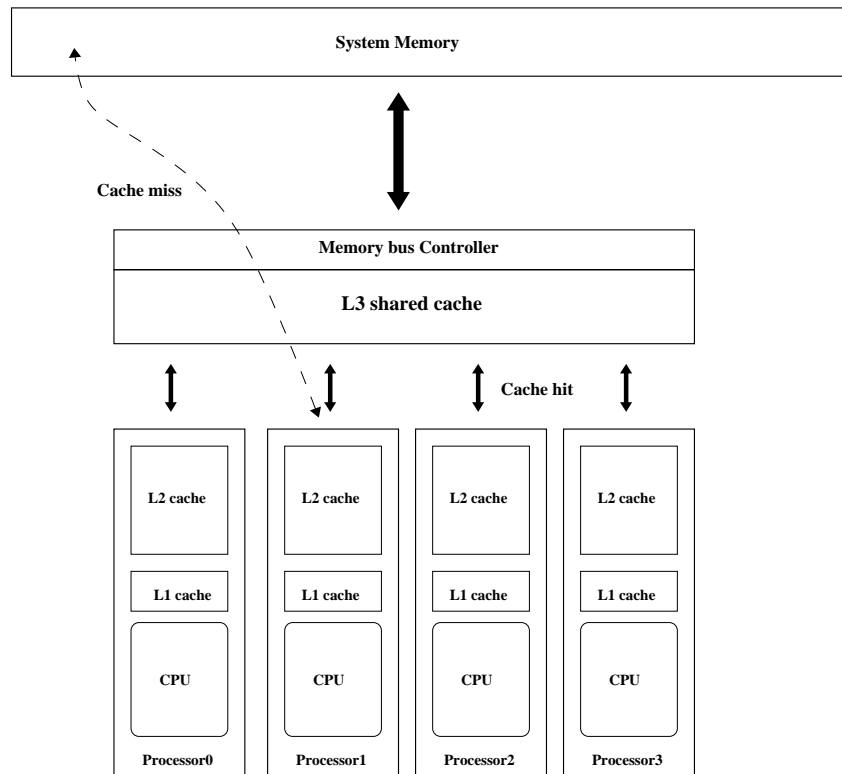


Figure 1.4: The Intel multi-core processor diagram.

- *L1 cache* is the smallest and fastest cache. It stores the copies of data which are the most frequently used in main memory.
- *L2 cache* is the next larger but slower cache, compared with L1. It can hold a chunk of data items near or next to the recently used data.
- *L3 cache* is the largest but slowest cache. It can hold a bigger chunk of data items than L1 and L2.

L1 cache and L2 cache are usually located inside an core whereas the L3 cache is shared among multiple cores. After receiving a request for making data reference, the core first checks L1 cache, followed by L2 cache and L3 cache. If the data reside in any local or shared cache, the core will make a direct and fast reference (*cache Hit*). Otherwise, the core has to access the data from the external main memory (*cache Miss*).[22]

The cache miss causes a higher latency than the cache hit for the requests to access the main memory through the system bus. Besides, when the multiple threads update the same data in the memory, they may compete each other for

its ownership, which results in *race conditions*. [22] The effect of race conditions may slow down the performance as threads are racing each other to access the data. But using the `mutex` may avoid this problem. Assume that the PvWatts program uses a mutex to ensure the mutual exclusion: only one thread can access the shared data file. When a thread is reading the file, the file's mutex is set to be *locked* and other threads have to wait until its state becomes *unlocked*. Even though the atomicity of this file is guaranteed by the mutex, the parallel PvWatts program will run slower than the sequential one as each thread has to spend the extra time waiting until the mutex becomes unlocked.

A program frequently reuses the data in the same location or within the nearby memory space. Based on the time duration and data reference, the locality of reference is divided into two categories: [22]

- Temporal locality: the program might reuse the specific data which have been referenced recently.
- Spatial locality: the program might reuse the data items which are relatively close to the recently accessed item.

The locality rule is to maximize the number of the local references and minimize the number of non-local reference. [22] This rule tries to reduce the dependency between the threads and tasks, so that the parallel program can run faster because most of the communication latency and memory contention are avoided. To illustrate the locality rule, consider the multiple PvWatts reducers which are used to sum up the power for each month. Each reducer queries the tuples from the same table in the database. The tuples of a month are stored in the neighbouring area of the memory space, because they have been sorted by their date and time field values. Thus, the program would fetch a chunk of tuples to the CPU cache and the reducer might get the tuples directly from the cache without the access of the main memory. The performance can be improved as the cache provides the faster speed than the memory.

1.3.6 Speedup

The *speedup* is the main performance index for a parallel program. The *execution time* is from the time when the first core begin executing the program to the time which the last core completes execution. But execution time does not indicate how the parallelism scale up the program. The speedup is defined as the execution time of a sequential program divided by the execution time of a parallelized program that has the same result. But speedup has many issues and may lead to wrong interpretation. The follows are some factors that may affect the speedup and lead to performance loss.[22]

1. The different parallel machines may affect the speedup although they use the same architecture. This is because some of the components they use are slightly different or may have been upgraded with the new generation of technology.
2. The JVM options would affect the speedup. For example, turning off the compiler optimization may increase the execution times of a parallel program. This change affects the speedup and leads to the incorrect interpretation. To avoid this error, the compiler options should be reported along with the program execution.
3. The relative speedup is the speedup relative to the execution time that the program runs with a single core (thread). It is necessary when the problem size is too large to be fit into a sequential program.
4. The JVM warm-up affects the final execution time. The cache behaviour in the first few runs is not well-formed and have more chances of cache misses than the later runs. So the total execution time in the first run is always the longest one. Running the program several times or more can warm up the caches and reduce the chances of cache misses. So averaging the execution time in the later runs can get a stable and real speedup.

5. Some off-core activities (e.g. writing to/reading from disk) could dominate the execution time and completely destroy the parallelism.

1.3.7 Efficiency

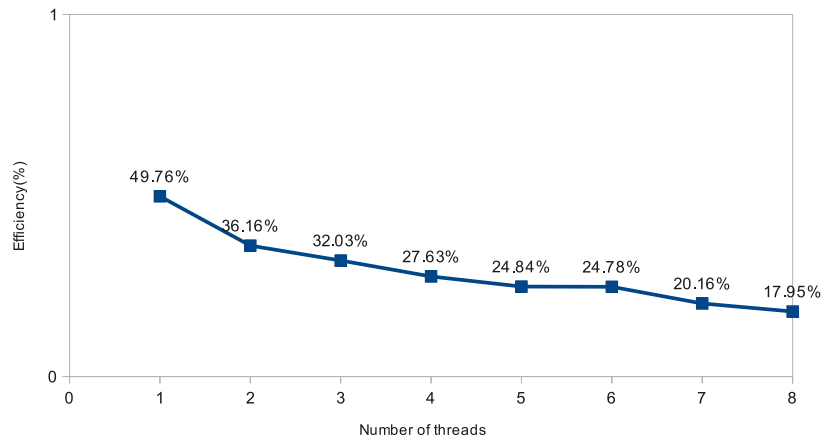


Figure 1.5: The efficiency chart of a program on a 8-core machine.

The efficiency of a program is the normalized speedup (the speedup divided by the number of cores).[23] It shows how much time faster each CPU is used on the parallel program than on the sequential one. The efficiency of 1.0 means that the speed increases linearly to the number of cores. The efficiency is always less than 1 and decreases as the number of cores increase. As shown in Figure 1.5, the parallel program using a single thread takes up most of CPU time whereas the same program executed with 8 threads has the least amount of CPU time.

Some programs can solve the problem faster by simply adding more number of threads. But the speedup can not always be improved in such a way and has a theoretical limitation, depending on how much a program is parallelized. A parallel program contains a sequential part and a parallel part. Using a multi-core machine can speed up the parallel part but not the sequential one. So the speedup of a parallel program is limited by the sequential part. The more of a sequential part in a program relative to the parallel part, the poorer speedup the program gets.

Amdahl's law defines the maximal speedup a parallel program can achieve with additional computing resources. Assume that we have a machine with N cores and F is the sequential fraction of the program. The maximal/theoretical speedup is[23]:

$$Speedup \leq \frac{1}{F + \frac{(1-F)}{N}}$$

As the N increases to an infinite, the speedup is close to $1/F$. This means that the speed of a parallel program would converge to a constant (the inversion of its serial fraction) and could not have any improvement by adding more number of cores to the machine. For example, the program with a quarter of sequential parts can at best have a speedup of four regardless of the number of cores.

1.3.8 Scalability

Deciding a problem size is difficult. If the problem size of a program is small enough to fit or be handled on the single-core machine, then it is unreasonable to run this program on the multi-core machine. In order to get a fairly good performance on the multicore machine, the problem of a parallel program should be scaled up as the number of cores increases. The scalability of the problem implies[23]:

1. The design of the many-core or multi-core machine does not requires a high-frequency CPU as the multicore machine with low-frequency CPU can achieve a similar or close performance as the high-speed single core.
2. The software batching technique will be widely used in the parallel program as it decreases the overhead costs of communication among threads and improves the efficiency by performing the latency-hiding technique.
3. The problem size of a parallel program should be scalable to maintain the efficiency when more cores are added to run the program.

1.3.9 Performance Trade-Offs

The performance of a parallel program can be affected by a variety of factors, e.g. the communication costs, task dependencies and the CPU idle time. And the factors are dynamic as they may vary from one problem to another. And each factor may have a trade-off relationship with the others. That is, lowering one factor may result in an increase of others. The common performance trade-offs in a parallel program are described as follows:[23]

1. Communication costs are reduced by the independent parallel tasks.

Each independent computation can be run by one thread without any communication cost or waiting time among tasks. But the independent task may create some redundant computations to remove the task dependencies. For example, an input file needs to be split into several parts so that each reader can take one part to process in parallel. Even though the redundant computations increase the costs, the communication costs can be reduced.

2. The parallelism often requires a large amount of memory.

If the data are too big to fit into the cache, then they will be moved into the main memory space which has the slower speed than the cache. To make use of the cache as much as possible, the *privatization* and *padding* methods can be used to reduce the memory usage in a program. The privatization is to replace the shared/global variables with the private/local ones. The benefit is that the thread does not have to interact with shared memories all the time even though the privatization requires the additional memory costs. The padding is to make each thread to do the same portion of a task so as to keep the variables on the same cache line. Padding can make those dependent variables to be independent, that is, to remove the false sharing and improve the performance.[9]

3. Overhead costs prevent the parallelism.

Consider the following three trade-offs between the overhead cost and the

parallelism. *a)* The overhead cost occurs when one reducer accumulates the results from all of the threads. This reducer can be the bottleneck for the performance in this case. But if the intermediate combiners are used to categorize the data before the reducer, then the summation workload can be parallelized by using the multiple reducers. *b)* The fine-grained task can improve the load balance than the coarse-grained one. The coarse tasks have different workload and different completion time. This unbalanced workload may cause some threads to be busy all the time while some are idle and waiting for other threads. But over-decomposing a task may lead to an increase in the communication costs. *c)* Batching technique can improve the parallelism by performing a group of tasks rather than one task at one time. But batching processing may cause the contention or the race conditions and reduce the efficiency.

1.3.10 Perfect Parallelism

Perfect parallelism is when the execution time of a program is sped up in proportion to the number of cores. The following reasons explain why it is hard to achieve perfect parallelism.[23]

1. Parallelism has expensive overhead costs.

The overhead costs of parallelism are communication costs, synchronization costs and memory usage. Some communication costs can be avoided but some can not (e.g. the shared memory communication). Synchronization costs are hard to detect as the messages are passed between threads. And the memory size sometimes can constrain the performance of a parallel program.

2. Some computations are non-parallelized.

Non-parallelizable parts in a program are those which must be executed in sequential. Amdahl law shows that the maximum performance of a parallel implementation is determined by its sequential fraction (the

ratio of its sequential computations over all its computations). That is, the sequential computation limits the maximal speedup of a program.

3. The idle time is never avoided.

The CPU idle time results from the unbalanced and memory-bound computation. The unbalance load means that each core has different amount of workloads. Running a sequential program on the multi-core machine would incur this problem. The memory bandwidth constraint is still a problem to the parallel programming when the CPUs write/read the data from the memory.

4. Contention causes the slowdown of whole system.

Contention decreases the speed of a program because it increases the workloads on the memory and shared memory bus.

1.4 Contributions

This section lists the main improvements to the JStar compiler, methods, results, and tools that have been achieved. During this thesis, they include:

1. Added the *-seq* and *-par* options to the JStar compiler, so that users can easily generate code for either sequential or parallel machines. Before our thesis, a Fork/Join prototype was developed for JStar. It was based on splitable hashsets of tuples, so this prototype had poor speedups and significant overheads for tuples with small amount of computation.[1]
2. Defined two optimisations (*noDelta* and *noGamma*) to reduce the number of tuples in the JStar data storages and the latency of triggering the tuple rules.
3. Extended the JStar runtime to allow users to specify the debugging verbosity to report different kinds of runtime information about the execution of JStar programs.

4. Added *tuple grouping* feature to the JStar runtime so that Delta nodes with many tuples can group multiple tuples into a single fork/join task, to reduce the overhead of having many small tasks.
5. Extended the JStar runtime to allow users to choose between alternative Gamma table and Delta tree data structures at runtime. This makes it easy to measure the performance of alternative data structures.
6. Evaluated the performance of different Gamma data structures.
7. Developed a *Task Dependency Graph Tool* for visualising the dependencies between rules and tables in an execution of a JStar program.
8. Added logging features to the JStar runtime, and wrote the *JStar Tuple Timing Graph* that visualises the log output as a timeline of tuple executions. This is helpful for identifying bottlenecks and performance problems within some JStar programs.
9. Defined the standard operating procedure (S.O.P) to benchmark the JStar programs on the Symphony cluster or the NeSI cluster.
10. Defined a performance tuning process for JStar programmers to follow.
11. Applied that tuning process to several case studies (with different styles of parallelism) and demonstrated that it produces efficient programs, usually with quite good speedup.
12. Used the Disruptor data structures to speed up one JStar program without changing the JStar source code.

Chapter 2

Introduction to JStar

JStar is a new declarative language that aims to encourage implicit parallel programming[28]. The semantics of JStar language is a subset of Datalog with negation, and explicitly defines a causality ordering which ensures the correct sequence of execution flow.[4] The JStar compiler has already been implemented to translate a JStar program into a Java parallel implementation, which can be executed on single-core or multi-core CPUs. Since the compiler generates parallel codes by default, JStar programmers can focus on the design of their program without making any parallelism strategy. The separation of program and parallel implementation facilitates people, who have little or no parallel programming knowledge, to aggressively make use of multi-core computing power. The current version of JStar (V2.0) includes the following features:[28]:

1. The *Tuple Order Visualizer* is a graphic user interface that displays the execution sequence of tuples in a JStar program with a tree structure.
2. The *Satisfiability Modulo Theories (SMT) Connector* translates the theorems from JStar syntax to Standard SMT-LIB format, so that the conforming SMT theorem solvers can check their satisfiability and determine their validity. The theorems in a JStar program include the execution order of rules and tuple invariants. If the rule sequence conflicts with the user-defined causality ordering declaration, or the tuple invariants

are not preserved during the execution, then the SMT connector would return the results and show any available counterexamples.

3. A *Logging system* that can record each table usage and provide a tool to visualize the task dependencies.

2.1 The JStar Language (Delta Tree and Gamma Database)

JStar stores all data in main memory rather than on disk as in-memory database provides faster accessing speed. Since the performance is an important goal for JStar, an in-memory database is the preferable form of data storage.

The JStar language supports a relational programming paradigm: data are organized in a relational database. JStar shares most of the terminology of SQL relational models. A *tuple* is used to describe a basic data object. An *attribute* defines the property of a tuple, including values, data type and name. A tuple is represented as an ordered list of attribute values. A set of tuples, which have the same attributes, are grouped in one *table*.

Rules control the flow of a JStar program. Rules can add tuples to or query tuples from these tables. But they cannot update or delete any existing tuple in the tables. Since tuple values cannot be changed during the execution, and the use of negation and aggregate operators is restricted to avoid data races, the output of JStar program depends purely on the input values. Thus, running a JStar program with the same input values would always produce the same results, though possibly in a different order. This feature makes JStar similar to a pure functional language, and the behaviours of a JStar program are deterministic. The use of immutable tuples does not allow side effects and makes JStar thread-safe. As tuples can not be changed or modified after being created, there is no chance that a single tuple is updated by multiple

threads. With the properties of the functional paradigm, determinism, and thread safety, JStar programs can be aggressively parallelized and optimized by the JStar compiler.

Tuples are inserted into the Delta Tree immediately after they are created and initialized by rules. Then these tuples wait in order in the Delta tree for being processed by the JStar runtime environment. According to the execution order of tuples which has been declared in the causality expression, the JStar runtime takes out a group of minimal number of tuples from the Delta Tree. JStar processes the tuples in the same group with one of the three strategies, which will be detailed in Section 2.3:

- Splitting the group into subgroups
- Iterating sequentially through the tuples in the group
- Creating a list of `Fork/Join` tasks and then processing them concurrently.

2.2 How the JStar Compiler Works

The JStar compiler translates a JStar program to source code written in Java. By using the Java compiler and the Java virtual machine(JVM), these generated Java files are converted into Java byte codes and executed on the different types of operation systems. With appropriate compiler options, the JStar compiler generates parallel Java code. Thus, JStar programs can be executed on single-core machine or multi-core machine without changing the source code.

The JStar compiler is implemented with `Xtext`[10], which provides an open-source framework for programming and domain-specific language development. The JStar compiler is developed with `Eclipse SDK` and uses the `Java Runtime Environment (JRE)` for compilation and execution. The JStar compiler transforms each table declaration in the program into two classes for the tuple and three classes for the table, and places all the Java source code into the `src-gen` folder.

Consider the compilation of the `PvWatts` table in the JStar `PvWatts` program. The `PvWatts` table has 5 fields, ordered as year, month, day, time and watts. Besides, it also adds invariants to each field as described in the following:

```

1 table PvWatts(int year, int month, int day, String time, int watts)
2   orderby (PvWatts)
3   inv 1000<year && 1<=month && month<=12 && 1<=day && day<=31;

```

When the JStar compiler compiles this table declaration, it generates a `PvWatts` class inherited from `Tuple` class and initializes its member variables with the `PvWattsBuilder` class. The member variables of these two classes are inferred from the expression of field declaration. The *orderby* clause imposes a natural ordering on the `PvWatts` class, and generates the `compareTo` method to compare all the fields. For the invariant, the JStar compiler creates and overrides the `invariant` method of `PvWatts` class. This method checks whether the member variables of each `PvWatts` tuple satisfy the conditions in the *inv* declaration, so that each `PvWatts` tuple has valid date and time values.

2.3 The Default Delta Tree and Gamma Database

JStar uses a pool/queue pattern to process new tuples during the execution. When a new tuple arrives, the JStar runtime does not process its task immediately but adds this tuple into a temporary data set. This shared database is called the **Delta tree** and acts like a queue. All new tuples are lined up in order and waiting for processing. Thus, the JStar runtime can create and process tuples at the same time without any waiting. JStar runtime take one or more tuples from Delta tree and execute them concurrently. After being taken out from the Delta tree, each tuple is moved to one table in the Gamma database and then JStar runtime starts to execute the associated rules whose input is this tuple kind. These rules can query tuples from Gamma database and put more tuples to Delta tree. Since JStar runtime frequently interacts with

the Delta tree and the Gamma database, efficient implementations of these two databases are essential to improve the JStar performance. The following subsections describe the default implementations of Delta tree and Gamma set.

2.3.1 The Delta Tree

Listing 2.1: Source Code of the DeltaNode Class.

```

1  package nz.ac.waikato.jstar.runtime.delta;
2  import ...
3
4  public abstract class DeltaNode extends RecursiveAction {
5      final JStarProgram prog;
6      public DeltaNode(JStarProgram prog) {
7          this.prog = prog;
8      }
9      /**
10     * @return the program that this delta node is part of.
11     */
12     public final JStarProgram getProgram() {
13         return prog;
14     }
15     protected void compute() {
16         process(prog);
17     }
18     /**
19     * Insert the given tuple into this subtree, and remove duplicates.
20     * This subtree is the subtree for level 'depth' of the orderby list.
21     *
22     * @param toInsert the tuple to insert
23     * @param depth
24     */
25     public abstract void insert(Tuple toInsert, int depth);
26     /**
27     * Execute all the tuples within this subtree.
28     * They are processed in minimum-first order.
29     * Each tuple is removed after it is processed.
30     * @param prog TODO
31     * @return when the whole subtree is empty.
32     */
33     public abstract void process(JStarProgram prog);
34 }

```

The Delta tree is organized as a multi-level priority queue and made up of `DeltaNode` objects. The `DeltaNode` class is shown in Listing 2.1. As being extended from the `RecursiveAction` class, the `DeltaNode` object is an typical Fork/Join task which does not return any result. The `DeltaNode` class specifies the common behaviours which are performed by the JStar runtime. For

example, the `insertion` method inserts a tuple into the Delta tree. For each kind of tuple, the JStar runtime creates one single `DeltaNode` subtree of the fixed tree-depth predefined in the *order* declaration, and then puts the tuple of its kind to the leaf node. The branch-like data structure can increase the speed to search for the set of minimal tuples. The reasons are described in the follows: *a)* each leaf node in the Delta tree contains only one kind of tuples; *b)* the Delta tree are indexed hierarchically; *c)* duplicate tuples are removed from the Delta tree. **Note that** many different kinds of tuples are inserted into the Delta tree, so it contains a heterogeneous set of tuples. The ordering of subtree is determined by the *order* declarations of each kind of tuple with the Nth level.

Listing 2.2: Table Declaration of the Delta Tree Example.

```

1 table CmdLineArg(int index, String value)
2   orderby (CmdLineArgs)
3   inv 0<=index;
4   /**
5    * A request to generate tuples: from (inclusive) .. to (exclusive)
6    */
7 table GenTuples(int index, int from, int to)
8   orderby (Int, seq index)
9   inv 0 <= index;
10
11 table PvWatts(int year, int month, int day, String time, int watts)
12   orderby (Int, seq year, Int, seq month, PvWatts)
13   inv 1000<year && 1<=month && month<=12 && 1<=day && day<=31;
14
15 order CmdLineArg < GenTuples < PvWatts;
```

The Delta set is implemented as a fixed-depth tree and composed of four kinds of `DeltaNode` objects: `DeltaNodeNamed`, `DeltaNodeObject`, `DeltaNodeInt` and `DeltaNodeSet`. Consider an example of three tuple kinds (`CmdLineArg`, `GenTuples` and `PvWatts`), and its table declaration and ordering of tuples are listed in the Listing 2.2. The `CmdLineArg` tuple passes one program argument to the JStar program; the `GenTuples` tuple requests the program to generate a fixed number of `PvWatts` tuples; the `PvWatts` tuple represents a random hourly `PvWatts` record. The `CmdLineArg` is the first prioritized tuple as it is ordered by the default `CmdLineArgs` table, which has been already implemented in JStar runtime. The `GenTuples` tuple is the second, followed by the

PvWatts tuple.

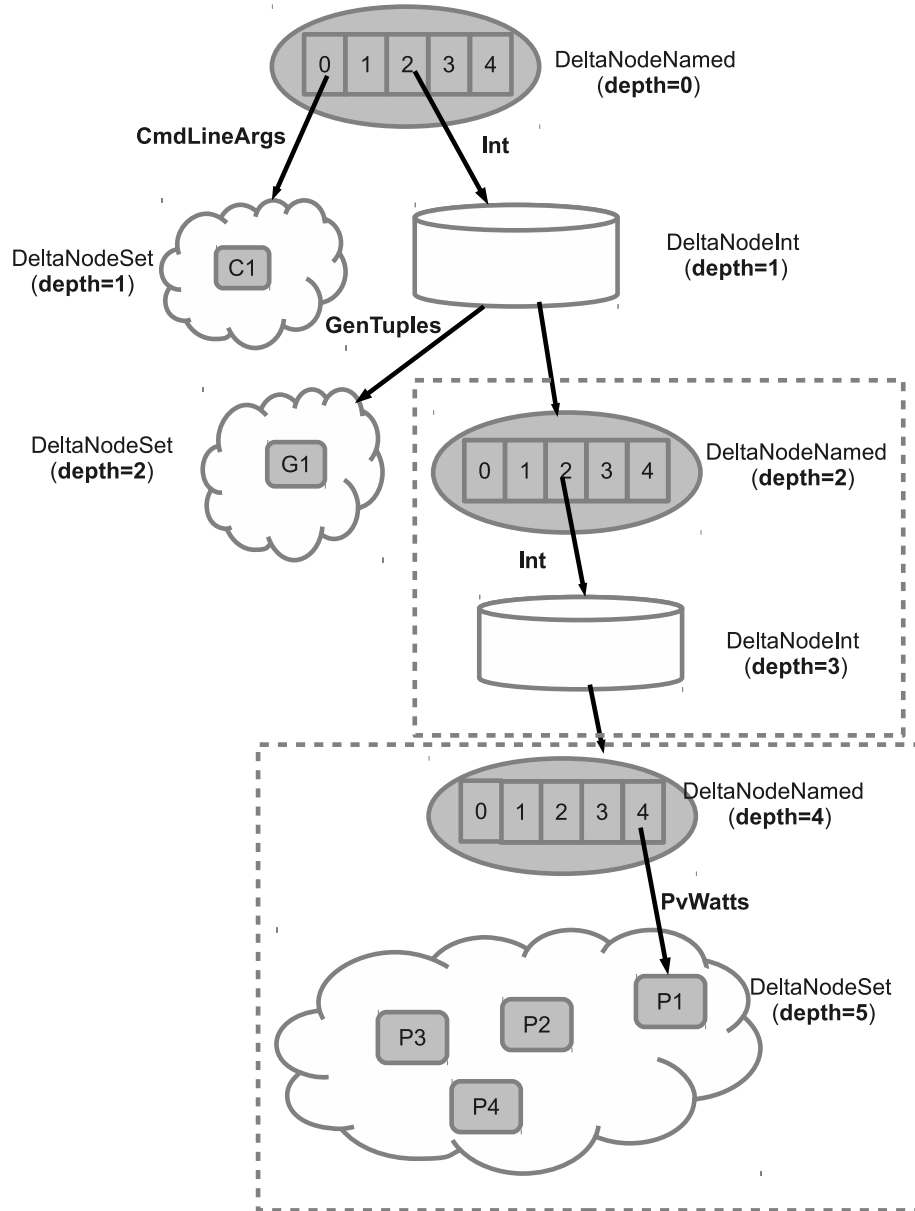


Figure 2.1: The Delta tree diagram with 3 tuple kinds of 5 tree-depths.

Figure 2.1 is the Delta tree of the above example. We will use this graph to describe four DeltaNode objects:

DeltaNodeNamed is used as the root node of the Delta tree and the tuple ordering indicator. According to the `orderby` declaration, it inserts the same kind of tuples into one single subtree. As the root node of a Delta tree, the DeltaNodeNamed object contains a fixed-size array of DeltaNode nodes whose length is determined by the total number of tuple kinds in the program. For example, the above JStar program uses 5

kinds of tuples, including three user-defined tuple kinds and two default-implemented ones (`CmdLineArgs` and `Int`). Thus, the array size inside the `DeltaNodeNamed` object is 5 and the array position indicates the **order** declaration, as listed in the follows: **0)** `CmdLineArgs`; **1)** `CmdLineArg`; **2)** `Int`; **3)** `GenTuples`; **4)** `PvWatts`.

When a new tuple is inserted into the `DeltaNodeNamed`, its **orderby** list determines the position in the array. For example, the `CmdLineArg` tuples are ordered by the `CmdLineArgs`, as shown in the **orderby** declaration of `CmdLineArg` table. The JStar runtime puts each new `CmdLineArg` tuple on the first position of the array and forms the left subtree.

DeltaNodeInt is used as the intermediate node of the Delta tree and sorts out the tuples whose the key field are declared as the integer type. It stores the tuples with the `TreeMap<Integer, DeltaNode>` collection, indexing by the tuple field. For example, the `GenTuples` tuple uses the *index* field as the key. When a new `GenTuple` tuple is inserted into the sorted `TreeMap`, the insertion method of the `DeltaNodeInt` uses its index value as key and check whether this key has been existed in the map. If not, then the method associates this key with a new `DeltaNode` object, which is created with the tuple. As shown in Figure 2.1, each `GenTuples` tuple is put into the `DeltaNodeSet` object (depth=2) next to the `DeltaNodeInt` (depth=1).

In addition, this `DeltaNodeInt` node creates a `DeltaNodeNamed` node, so that the `PvWatts` tuples are sorted on the next `DeltaNodeInt` node (depth=3). As the `PvWatts` tuple uses two integer keys, the JStar runtime creates two layers of `DeltaNode` nodes, which one layer contains the `DeltaNodeName` node and the other has the `DeltaNodeSet`. With this structure, all the `PvWatts` tuples are moved into the same leaf node of the right subtree.

DeltaNodeObject is used as the intermediate node of the Delta tree and sort out the tuples whose the key field are declared as the object type. Like the **DeltaNodeInt**, the **DeltaNodeObject** stores tuple with the **TreeMap<Object, DeltaNode>** implementation. The key is the object itself. For example, if the **PvWatts** tuples are ordered by *PvWatts* object, then the position of a tuple is dynamically determined by all of its key fields: the **PvWatts** tuples are sorted in a ascending order of year, month, day, time and watts values. When a new tuple is inserted to this tree map, the order of tuples must be re-ordered again and each tuple's position needs to be changed as well. Thus, both of the **DeltaNodeObject** and the **DeltaNodeInt** objects needs to dynamically change their storage size. The **TreeSet** is preferable to the **Array** because it provides a total ordering on tuples and quickly resizes the capacity[16].

DeltaNodeSet is used as the leaf node of a subtree. Because of the branch structure, the tuples of the same kind have been sorted and ordered before they are moved into the **DeltaNodeSet**. Thus, the **DeltaNodeSet** node can directly move these tuples to the storage.

The storage of the **DeltaNodeSet** must be implemented with an efficient data structure. Inserting a tuple requires to check whether the tuple exists in this storage. As the number of tuples increases , this check takes more time to compares the new incoming tuple with the old ones in the data storage. An efficient data structure is needed to ensure the performance of insertion operation. The **HashSet** set is used to store the tuples in the **DeltaNodeSet** for its stable insertion speed. It offers a constant-time performance over the basic operations (add, remove, contains and size)[15].

2.3.2 The Gamma Database

The Gamma database needs to be efficient at adding new tuples, searching for individual tuples and searching for a subset of tuples with some common field values. The default implementation of each table in the sequential Gamma database is `TreeSet`. When a new tuple is moved to Gamma database, JStar runtime uses the `compareTo` method defined in the `orderBy` declaration to check if this tuple has been present before. This comparison method only compares the key fields and each of the operations depends on the data type of the compared key field. If the key field is an integer, then the comparison uses the `greater than (>)` operator and the `less than (<)` operator to determine the equality of two tuples. As for the String values, the comparison uses the `compareTo` method of Java `String` class to compare two strings. If the comparison results of all the key fields are the same, then the method returns zero. Otherwise, it returns a non-zero value. The new or non-existing tuples are moved to the Gamma database; the duplicate tuples, whose result is zero, are discarded.

Consider the PvWatts example. The following procedure is used to insert a new PvWatts tuple to the table in Gamma database. First, the year value of this tuple is used to compare with one tuple in the Gamma database. If the year value is greater than that of the other tuple, then the method returns the positive one (+1). If the value is less than the other, then the negative one (-1) is returned. When the result is equal, the next key field (the month value) is used to compare these two tuples. This comparison method continues until the result is returned or it has used all the key fields for comparison. Then another tuple in Gamma is chosen to compare with this tuple and repeat the above procedure until all of the tuples have been compared. If the final result is zero, then the Gamma database adds this tuple. Otherwise, it ignores this tuple without doing any action.

Chapter 3

Related Work

The parallel computing divides a great deal of computation into many tasks which can be carried out in parallel. Having many computers to work on the same problem can shorten the completion time and achieve the same or better performance than using a single unit computer. To provide a large number of the computing resources, the High Performance Computing (HPC) facility are either the multi-core machines or clusters of small machines that are linked together through the local network or interconnect to work together. And currently the clusters of multi-core machines are preferable because their prices are more affordable and have more computational power than the single-unit machines [26].

The HPC parallel programming model employs the **distributed** or **shared** memory design to parallelize the computation across the multiple processors. The distributed memory programming model distributes the tasks and data over the multi-core machine, where each processor owns one private memory space to store the data locally. If the processor requires the non-local data, it must communicate with other processors and move the remote data to its local memory. Referencing the remote data takes some extra time to find the data location and thus could lead to an unexpected delay. Instead of data distribution, the shared memory programming model keeps all the data with one public memory space where the processors can retrieve data from.

The distributed memory offers the memory locality to allow each processor to store the data in the closest (private) memory location. As each processor mostly uses the local data, the distributed memory model can avoid the race conditions but produce inevitable communication costs. Message Passing Interface (MPI) is the communication library for the distributed memory model. The MPI can be called directly from **C** and **Fortran**, or packaged as a library and imported into a **Java** project. The MPI program is efficient and portable as the MPI interface has been widely adapted in every distributed memory systems and also optimized to provide the good performance.

The shared memory system uses one single memory space to store the data and provides the unified global memory addresses to quickly locate and retrieve the data. As the same data are occasionally synchronized by more than one processors, the shared memory program may have the performance problems, such as the race conditions.

3.1 Library-based Shared Memory Programming Languages

Java language is considered as a option for programming on parallel hardware. With built-in multi-threading and networking APIs, Java programmers can write a parallel application to utilize the computing power of the multi-core machines in a cluster. But writing a low-level multi-threading application is hard and buggy as the data synchronization requires the external mechanism and the incorrect design of parallel tasks may lead to deadlock. Therefore, Java from 1.5 specification supports the high-level shared memory programming with several concurrency utilities, including the thread pools, the concurrent collections and the atomic variables.[26]

Using Java for HPC may have some difficulties. Although the performance of JVM has been continuously improved by experts and engineers, the performance of Java HPC solutions may still be reduced by some unpredictable

factors. For example, the excessive objects allocation increases the overhead costs for the Java garbage collector and thus results in a longer execution time. The poor cache performance from many object references may also increase the running time.

Despite the above issues, Java is still an important parallel programming language for some HPC application. The reasons are that: *a)* there has been a number of projects developed in Java for HPC; *b)* Java programs can support both of sequential and parallel implementations and achieve a good performance; *c)* the recent development on Java, e.g. the low-latency communication, has overcome some performance issues. [26]

As OpenMP standard supports the multi-platform shared memory programming (in *C*, *C++* and *Fortran*), it provides the portability to the multi-threading code across the heterogeneous hardware and operating systems. But the OpenMP standard is not included in Java, and therefore, most Java OpenMP-like projects are implemented in the form of a Java library to be imported in the Java project. JOMP, for example, is a library for Java to achieve the OpenMP-like parallelism.[25]

3.2 Partitioned Global Address Space Languages

3.2.1 Titanium and X10

Titanium uses Java as its base and adds extra features for high performance parallel programming.[26] Titanium adopts Partitioned Global Address Space (PGAS) programming model: all processes use one single memory space and reference objects with global addresses. Each process allocates one region of this spaces and stores all its data objects in its local region. Through sharing the same memory, a process can read and write the data objects that reside on others. Moreover, Titanium adopts lightweight synchronization to ensure that single-value variables are only read and written by one process, and have consistent values in all processes. Benchmark results show that

Titanium implementations can have or often have the better performance than the standard Fortran/MPI ones.[7]

X10 is also another Java-based shared memory programming language. It uses APGAS (Asynchronous Partitioned Global Address Space) as its execution model for distributed processing.[20] The global memory space is split into several *places*, each place is implemented by one instance of JVM. By having multiple JVMs running on different computers and connecting them through the network, *X10* forms the distributed system. Thus, a large computation work can be divided into many tasks, each of which can be distributed and processed concurrently.

Regarding memory referencing, each JVM creates and stores its local data objects in a specific location of the memory space. Thus, each object has a global address and be remotely referenced from other places, using a mechanism named *GlobalRef*. That is, all referenced objects can not be collected as garbage even if there is no local reference to it.[20]

Both Titanium and *X10* use the syntax of JAVA language, so they inherit its imperative program paradigm: using statements to define the computation and assign values to variables. But JStar is a declarative language which expresses the program's computation without assignments. Regarding parallelism, Titanium and *X10* programmers need to explicitly specify where the parallelism should go whereas JStar parallelism is implicitly determined by the JStar compiler. Like the PGAS programming model, JStar uses a global DeltaNode and Gamma database to store all local data objects in one memory space. While it is possible to transparently distributed the JStar database across multiple computers.[6] This thesis will focus on the shared memory implementations.

3.2.2 Chapel

Chapel is an emerging parallel programming language that originated under the DARPA High Productivity Computing Systems (HPCS) program.[3]

Chapel also supports the PGAS memory model, which all the variables are regarded as the local ones although some of them are stored in the global memory space. And the Chapel runtime and compiler will implement the network communication for these remote variables. Chapel also supports the programmers to introduce the explicit parallelism on the single-core machine or execute the code sequentially on the multithreaded machine.

The difference between JStar and Chapel is the way of how the users specify the task parallelism. Chapel allows the programmers to specify the task parallelism explicitly with the *sync* statements, the atomic variables and structured parallelism.[3] But the JStar language use an implicit style of programming to implement its parallel tasks. Regarding the data parallelism, as the explicit parallel programs would cause the race conditions or deadlock, Chapel and JStar both uses similar and implicit features, such as *forall* loop, and reducers.

3.3 Intel Concurrent Collections

Intel Concurrent collections (CnC) aims to provide users with high level parallelism, so that users are able to write their algorithms without detailed parallelism knowledge.[2] Writing an efficient parallel program is difficult. Parallel programs may introduce new kinds of bugs which have never been found in sequential programming. One of the potential bugs is race conditions. Another bug is when one thread is not able to lock the state of a shared resource, it may infinitely block others from accessing the resources and lead to the deadlock. Besides, an inefficient parallel program may have performed once worse than the sequential one. As many parallel programs usually divide one task into a number of subtasks, they need to use a barrier to force all threads to wait for each other until all results have been synchronized. But this barrier results in higher synchronizing time as finished processes have to wait for unfinished ones. Sometimes the synchronization overhead costs may dominate

the benefits of parallel programming and cause longer execution time.

The concept of CnC is to make parallel programming accessible to domain experts and tuning experts.[2] Because the CnC model hides the details of parallelism, domain experts do not need to write low-level parallel programs, but only need to identify the dependencies which need to be run in parallel. The CnC program model specifies these relationship in a **CnC specification graph** which defines these relationship graphically and statically.

Tuning experts can be involved in the team for improving the performance of CnC programs. By mapping the CNC graph on a target parallel architecture, tuning experts can improve the performance of CnC program without needing to understand the application. And since the deterministic semantics—the same inputs producing the same outputs—is adopted by CnC, the correctness of the optimized CnC programs can also be ensured.

JStar and CnC have some similarities. They both use the determinism program model to ensure the correctness of the program. They also provide the dependency graph to visualize their programs, but the JStar graph is dynamic. They separate the roles of domain experts and parallelism experts. This allows the development team to include the people of different profession to work together but still can make use of their expertises. For example, the application developers can focus on the business logics without thinking about the implementations while the programmers can merely concentrate on the coding and debugging.

3.4 Domain-Specific Languages

Heterogeneous computing hardware is becoming an important trend in the computer industry and can provide significant performance increase. But in order to interact with these heterogeneous devices, application developers have to learn a variety of programming models. These incompatible models make the applications more complicated to deploy on different platforms and hard to

maintain their source code. Therefore, a parallel heterogeneous programming model is needed to help programmers to deal with different computing devices across systems.

3.4.1 DSL Characteristics

The Delite framework developed by Stanford University's Pervasive Parallelism Laboratory (PPL) supports this goal and proposes the following characteristics for a parallel domain specific language:[12]

Productivity. The application programmer can easily write the programs without the use of explicit parallel constructs. DSLs (Domain-Specific Languages) are used to satisfy these goals. Each DSL is a programming language with high-level abstractions. For example, LaTeX is the DSL for academic papers and SQL is for database querying. Because each DSL is designed for a particular domain, application writers are familiar with its notation and constructs. In addition, it can provide a sequential-like programming model for writing parallel code and using heterogeneous computing resources. So the productivity of application writers can be improved.

Performance. Application writers often take a lot of effort to write low-level code for better performance. But the new programming model should not decrease their performance. The DSL approach can achieve both productivity and performance because its compiler can trade off the generality to generate the high performance code. General-purpose compilers have to impose some restrictions on programmers to guarantee the correctness of generated code. This sometimes leads to low performance code. But the DSL uses implicit parallelism in a limited domain to prevent programmers from writing inefficient programs; therefore, the DSL compiler can use aggressive implementation of parallel patterns to optimize the code without causing any safety issues.

Portability and Scalability. The application should be able to run on various computing resources across different systems.

3.4.2 The Delite Framework

The Delite is implemented to run the same DSL programs on heterogeneous systems without changing the source code. The workflow chart is described in the following steps:

1. The application developers write their programs in a DSL and submit to the Delite framework.
2. The Delite compiler starts to build IRs of all operations. IR is defined by Lightweight Modular Stage (LMS), the framework designed for DSL embedded in Scala.
3. When the DSL programs are compiled, the LMS translate each operation into an IR node and forms symbolic representations of the original DSL programs.
4. The Delite compiler then applies static optimizations to achieve high performance on the IR nodes. For example, the Common Subexpression elimination (CSE) could remove redundant operations by reusing existing ones. And the linear algebra simplification, a domain specific optimization, is also applied in this process.
5. After optimizing the IR trees, the Delite compiler generates the kernel code for any available target hardware and forms all IR nodes into a Delite execution graph (DEG). The Delite framework also generates kernels for different types of hardware while building the DEG.

The Delite runtime uses the machine specifications (the number of CPUs and GPUs), DEG and DSL data structure to schedule the execution of this application. The scheduling algorithm is designed to enable implicit parallelization; therefore, a deferred execution model is proposed. The Delite run-

time delays ops being received so that more "run-ahead" ops are allowed to be submitted. Then these ops are formed into a dynamic task graph and dispatched to a thread pool by a heuristic, which minimizes the costs of data communication and scheduling overhead.

The Delite runtime provides the data-parallel operations and classes, which can be extended. For example, the OptiML is a DSL aimed at Machine learning and provides various domain-specific control structures, such as a sum construct. This function sums up the result of a block for each iteration and is implemented by extending the `DeliteOpMapReduce` parallel pattern. Moreover, the Delite runtime assists in generating the GPU code if DSL authors use the `@GPU` annotation to specify the operations that they want to ship to graphics processing units (GPU) on a single machine.

In order to minimize the overhead of execution on heterogeneous hardware, the Delite runtime generates execution plans for available computing resources and compiles them with the respective kernels to create executable files. The data transfers can be minimized because DEG provides detailed dependency information. On the other hand, memory allocation is well managed by the Delite runtime. For a CPU kernel, the Delite runtime uses the Java Virtual Machine to manage the memory. For GPUs, the runtime pre-allocates all data structures to address the GPU memory allocation issue.

The Delite framework shares similar goals with JStar. They use a runtime environment to compile and execute parallel programs on different platforms. And they both provide a graphic representation tool to optimize the compiled programs. Delite framework generates a DEG to optimize the execution operations on the heterogeneous parallel hardware and JStar can assist parallel experts to tune the performance with a task dependency graph. Delite and JStar[6] both support the GPU compilation features to generate code running on GPUs. However, they have different purposes: JStar is a general-purpose language based on Datalog with negation whereas Delite is a machine-learning domain-specific programming language. JStar uses a declarative programming

style and immutable data objects to guarantee the thread-safety of any JStar operation but the Delite uses a mutable programming model, plus reducers to concurrently sum up the values in an array.

Chapter 4

Generating Parallel Code

The JStar compiler provides several compiler flags to configure the code generation. The most important flags are the sequential and parallel flags, which determine the kind of data structures used in the Gamma set and the Delta Tree. Using non-thread-safe data structures in the parallel programs not only affects its performance, but causes the race conditions, or using locks incorrectly can lead to the deadlock. Thus, this chapter introduces the compiler flags and the performance of different data structures. In addition, we explore the optimization options of JStar compiler for improving the performance of JStar programs.

4.1 Sequential (-seq flag) versus Parallel Code (-par flag)

The Delta tree and the Gamma database are the two main tuple storages during the execution of a JStar program. When a new tuple is created by the rule, it is not processed immediately by the JStar runtime but queued in Delta tree in the defined order. After the JStar runtime removes a tuple from the Delta tree, it inserts this tuple into the table in Gamma, and triggers all of its associated rules. When the JStar runtime executes the rules, it may make queries into the Gamma tables and return the query result. The result

contains an iterable object, so that the rule can use a *for* statement to traverse each tuples of the resulting set.

The **Fork/Join** framework is selected to execute the JStar programs in parallel because it provides a good match for the semantics of the Delta tree. To be able to execute tasks concurrently, we want to have the only one thread to traverse all the Delta nodes in the tree, and then the other threads to process the tasks on the behalf of the Delta nodes. Besides, by recursively breaking down the size of tuples that needs to process at each iteration, the Fork/Join framework can improve the efficiency of the JStar runtime: the working threads in the ForkJoinPool can split large tasks into smaller tasks, and the available threads which have finished their tasks can run the tasks from other busy threads. The work stealing strategy used in the Fork/Join framework can reduce the overhead costs of creating new threads and improve the utilization of the existing threads in the pool.

4.1.1 Tuple Lifecycle

The lifecycle of a tuple has five phases: **Created**, **Queued**, **Processed**, **Stored** and **Retired** (Figure 4.1) After a tuple is created and put into the Delta tree, its status is moved from the **Created** phase to the **Queued** phase. In the **Queued** phase, the tuples are queued in different or the same level of the Delta tree, and waiting for their turn to be processed by the JStar runtime. The JStar runtime takes a set of *minimal* tuples from the Delta tree and executes these tuples in parallel, using the Fork/Join framework. When this tuple is removed from the Delta tree, it is moved to the **Processed** phase.

In **Processed** phase, the JStar runtime uses this tuple to trigger and execute its applicable rules, and then inserts it to its own table in the Gamma database. Those tuples in the Gamma table are kept until the end of the program, and could be queried by other rules. When one rule needs some tuples, it makes a query request to the table in the Gamma database. After receiving the request, the JStar runtime calls the corresponding query method with the

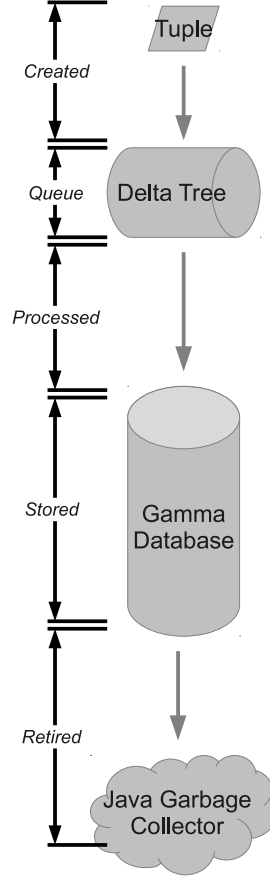


Figure 4.1: The lifecycle of a JStar tuple.

values, which are parsed from the request. The query results are returned as a *SortedSet* object, so that the rule can iterate through each tuple in the collection. These queried tuples are in **Stored** phase. At the end of the program or when a tuple will never be queried again, the tuples enters the **Retired** phase. The Java garbage collector can reclaim the retired tuples.

4.1.2 Optimisations

The optimisation strategies include *noDelta* and *noGamma* options. Applying *noDelta* optimisation on a specific tuple kind can omit the Delta tree insertion. Thus, instead of entering the **Queued** phase, a *noDelta* tuple is move to the **Processed** phase immediately after the **Created** phase (Figure 4.2 (a)). Similarly, the *noGamma* option omits the **Stored** phase. As shown in the Figure 4.2 (b), those tuple applied with *noGamma* option enters the **Retired** phase after the **Processed** phase. The details of optimisation strategies

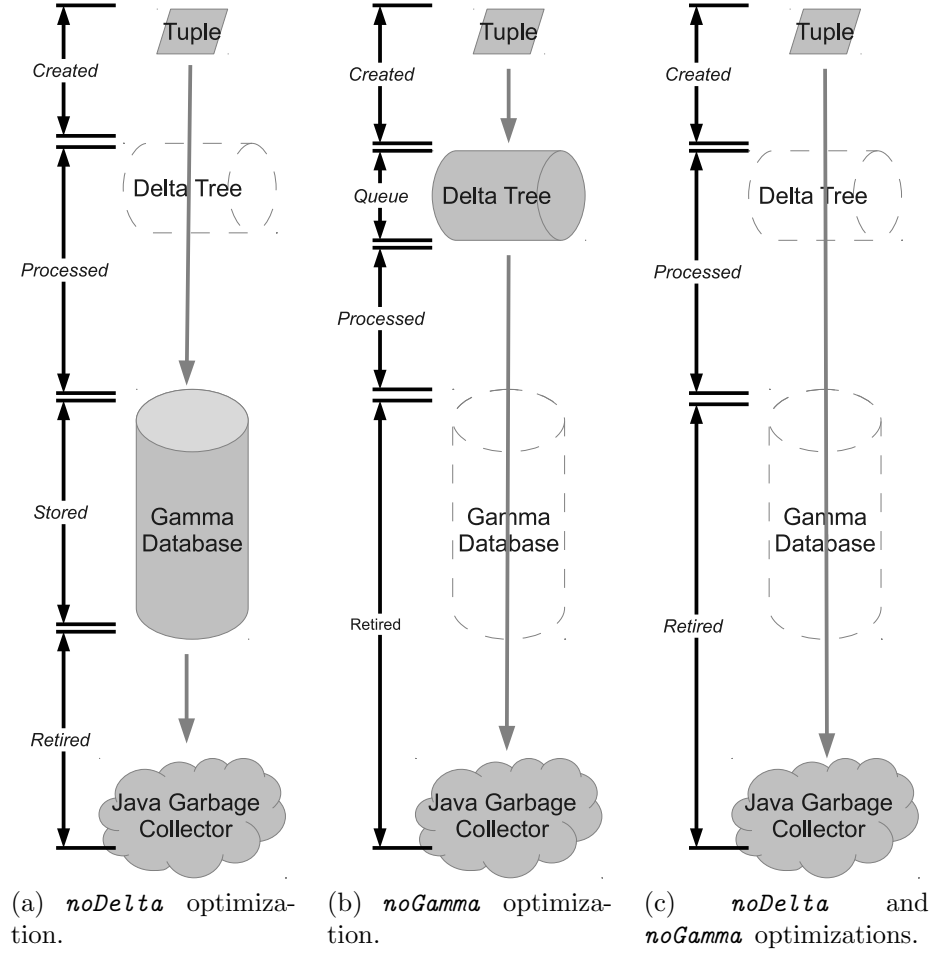


Figure 4.2: The lifecycle graphs of a JStar tuple inlined with three optimizations.

will be described in Section 4.4.

4.1.3 Code Generation

The JStar compiler can generate different kinds of implementation with the parallelism compiler flag and two optimization strategies. The parallelism flag of the JStar compiler is to specify what kind of data structures that the JStar compiler should use to create data storage during execution. When the JStar compiler is given the *-seq* flag, it uses the non-synchronized `TreeSet` class to create the tables in Gamma database and the primitive `DeltaNode` implementations to create Delta node in Delta tree. If the compiler generates Java code with *-par* flag, it uses `ConcurrentSkipListSet` class to construct the Gamma table and the parallel version of the `DeltaNode` implementations

to build up the Delta Tree.

The sequential version of a JStar program implements the Gamma database and the Delta Tree with primitive Java collection classes: `TreeSet` and `DeltaNode`. The advantages of these data structures is that they provide the guaranteed $\log(n)$ time cost for the basic operations (add, remove and contains). Thus, inserting a tuple to the Delta tree or adding it to the Gamma database does not cause a long delay as the number of tuples increases. But these two classes are not synchronized. If `TreeSet` or `DeltaNode` is accessed by multiple threads without the external data synchronization, it would cause race conditions and possible data corruption.

The parallel version of a JStar program uses `ConcurrentSkipListSet` and `ParallelDeltaNode` as the data structures for the Gamma database and the Delta tree respectively. The basic operations for these two collection classes take the $\log(n)$ time on average and the operations are guaranteed to be executed atomically.

4.2 Speed of the Gamma Database: PvWatts-Gamma Results

The JStar PvWattsGamma program is a testing program to understand the JStar system and provide a way of measuring the performance of the JStar implementation. This example generates 16 million PvWatts tuples in total and inserts each one of them to the table in Gamma database, and then ends the program. To measure the execution time of Gamma database, PvWatts-Gamma program must be compiled with **noDelta** optimisation on PvWatts tuples. Besides, JStar PvWattsGamma program generates PvWatts tuples with a number of parallel tasks, which each of them has its own number range to produce a set of date time values for each PvWatts tuple. Thus, the key values of any PvWatts tuple is unique in the whole Gamma table so that the Gamma table does not filter out any tuple whose key values are the same as

existing one. The source code is listed in Listing 4.1.

Listing 4.1: Source Code of the JStar PvWattsGamma Program.

```

1 package jstar.examples.pvwattsgamma;
2 /**
3  * This program measures the time taken to insert a given number of tuples
4  * into the Gamma set.
5  * To measure Gamma speed only, it should be compiled with --noDelta PvWatts.
6  * Arguments: --threads=1 --benchmark=12 NNN
7  * (NNN is the number of GenTuple tasks)
8  */
9 val TOTAL_TUPLES = 16000000;
10 table PvWatts(
11     int year, int month, int day, String time, int watts)
12     orderby (PvWatts)
13     inv 1000 < year && 1 <= month && month <= 12 && 1 <= day && day <= 31;
14 table CmdLineArg(int index, String value)
15     orderby (CmdLineArgs)
16     inv 0 <= index;
17 /**
18  * A request to generate tuples: from (inclusive) .. to (exclusive)
19  */
20 table GenTuples(int index, int from, int to)
21     orderby (GenTuples)
22     inv 0 <= index;
23 order CmdLineArgs < GenTuples < PvWatts < Int;
24 foreach (CmdLineArg arg) {
25     val num = Integer.parseInt(arg.value);
26     for (i : 0 .. num-1) {
27         val from = TOTAL_TUPLES / num * i;
28         val to = TOTAL_TUPLES / num * (i + 1);
29         put new GenTuples(i, from, to)
30     }
31 }
32 foreach (GenTuples gt) {
33     val time = "group" + gt.index;
34     for (v : gt.from .. gt.to - 1) {
35         val year = 1980 + (v % 9);
36         val month = v % 12 + 1;
37         val day = (v * 3) % 32;
38         put new PvWatts(year, month, day, time, v)
39     }
40 }

```

In the PvWattsGamma example, all PvWatts tuple are inserted into the same Delta tree, and then moved to the PvWatts table in Gamma database. Because of the large number of PvWatts, insertion and query operation on Gamma database take up most of the total execution time, and result in a bottleneck and limit the speedup. Therefore, we need to have an efficient Gamma database to improve the performance of JStar PvWatts program.

4.2.1 The Gamma Table Data Structure Choices

The performance of a Gamma database depends on its table implementation, which uses `NavigableSet` interface. Thus, the data structure choice for each table affects the performance of overall Gamma database. As described in previous section, the table implemented from different kinds of data structures has different insertion time and query time. This section explores several JAVA collection classes and choose an appropriate data structure for the Gamma database.

TreeSet is implemented from `NavigableSet` interface, which is a sorted set and provides several subset methods that are very useful for potential key queries.[5] A new tuple is added to the TreeSet if and only if it has not been present before. The nature orderings is used to determine whether a new tuple is equal to the existing tuples in a TreeSet. If PvWatts tuple are inserted to the TreeSet, then they are first sorted by their year field, then by the month, and then by the remaining primary key fields (from the left to the right).

ConcurrentSkipListSet is the concurrent implementation of `NavigableSet` class.[5] Like the TreeSet, PvWatts tuples in this set are sorted with their natural ordering. The Insertion, removal and query operation of the PvWatts tuples can be executed by multiple threads without causing any Thread-safe issue. Although the `size` method cause delay in operation as it needs to iterate through all tuples in the set.

HashMap is a Hash table based on `Map` interface.[5] It takes a constant time to insert a new tuple and retrieval an existing tuple from its table. Through its `Hash` function, the retrieval of tuples in a `HashMap` is similar to an array. For example, PvWatts tuples can be keys in a `HashMap` with the `Boolean.True` being used as the value (Figure 4.3). It is also possible to use the `HashSet` class directly, but not all of the `Map` classes in the Java library have the `Set` equivalents. So we prefer to consistently

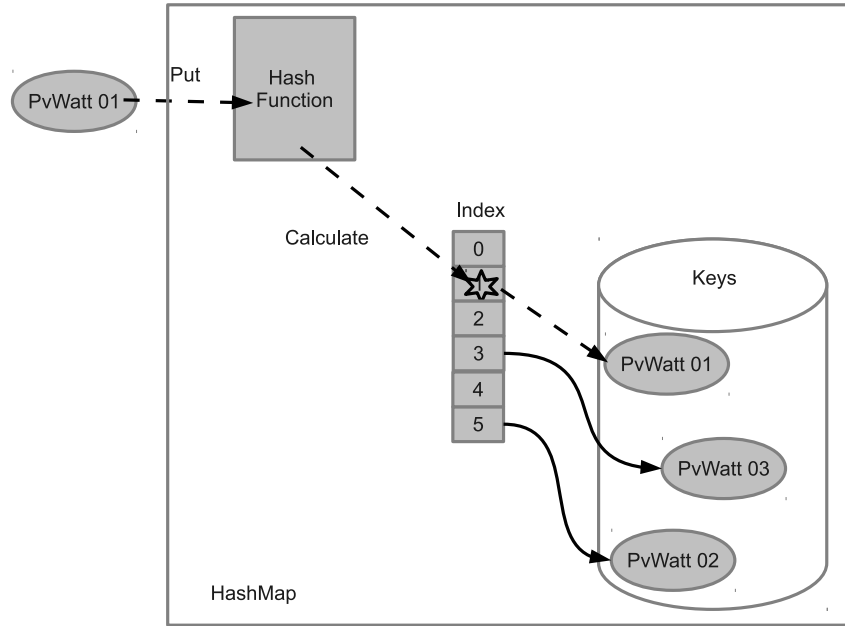


Figure 4.3: The `HashMap` insertion behaviour diagram.

use the `Map` implementations when generating parallel code. When a new `PvWatts` tuple is put into a `HashMap`, the `Hash` function finds an index in the array and then places it in the *Key* set. Instead of iterating all elements in the set, the array index provides an easy and direct way of accessing a tuple. But if there are the queries that return a subset of the table, the `HashMap` iteration time is proportional to its total number of elements, so a `HashMap` is unlikely to be a good choice of data structure in this case. If there are too many elements, the total searching time could be increased and thus the overall performance would be degraded. Besides, another factor which affects the performance of the `HashMap` is the initial capacity. When a large number of tuples are inserted into a small-sized `HashMap`, it will frequently spend the time resizing its table and therefore its performance will be slowed down. Like `TreeSet`, `HashMap` is also not thread-safe, so they are only suitable for sequential situation.

The performance of a `HashMap` depends on two factors: *initial capacity* and *load factor*. The initial capacity is to set up the estimated size of a `HashMap` instance. The load factor determines how full the `HashMap`

instance should be before its capacity is increased. For example, the `PvWatts` table is created with 8 million initial capacity and 0.75 load factor. That is, the capacity threshold is 6 (8×0.75) million tuples. When the total number of tuples in the `PvWatts` table exceeds this threshold, then the `HashMap` has to double its bucket size (16 million) to accept more insertion requests. Resizing a `HashMap` requires rebuilding its internal data structure, and thus causes a delay in execution time.

`ConcurrentHashMap` is implemented from `HashMap`.^[5] Its retrieval and update operations act like `HashMap` but provide the thread-safety. That is, `ConcurrentHashMap` can be accessed by multiple threads at the same time and still behaves correctly, but allow finer-grained concurrent access than `HashMap`.

`ConcurrentHashMapV8` is developed by JSR-166.^[21] Its goal is to improve the `ConcurrentHashMap` class and provide higher efficiency and low memory usage. And its implementation obeys the method specifications of `HashMap`.

4.2.2 The `PvWattsHashTable` Gamma Table

The JStar built-in Gamma table implements the `NavigableSet` interface. Those classes based on the `Map` interface can not be directly used to create a `PvWatts` Gamma table. The `PvWattsHashTable` class is written as a container to make use of all the `Map` implementations. It uses an array of `Map` instances, each `Map` instance storing the `PvWatts` tuples of one month. For example, the first `Map` instance whose array index is 0 stores the invalid tuples. The second one (index=1) retains the tuples whose month field is January, and followed by February, March, and so on. To measure the effects of *initial capacity* on the performance, the `PvWattsHashTable` constructor can initialize each `Map` instance with a specific capacity value. If the capacity is not set or the `Map` implementation does not support to an initial capacity, the `Map`

instance is constructed with the default configuration. Besides, this array is implemented using the `AtomicReferenceArray` so that the `PvWattsHashTable` can be guaranteed to be thread-safe and each `Map` instance can be read or written atomically.

4.2.3 Benchmark Result of the Sequential Tuple Insertion

Benchmarking the `PvWattsGamma` program requires several JStar runtime flags, including *table*, *debug*, *benchmark* and *threads*. The *table* flag takes a string which contains the property settings of a table and each property is separated by a comma. The program arguments in the following list are one example of the `PvWattsGamma` benchmarks.

Listing 4.2: The Benchmarking Program Arguments.

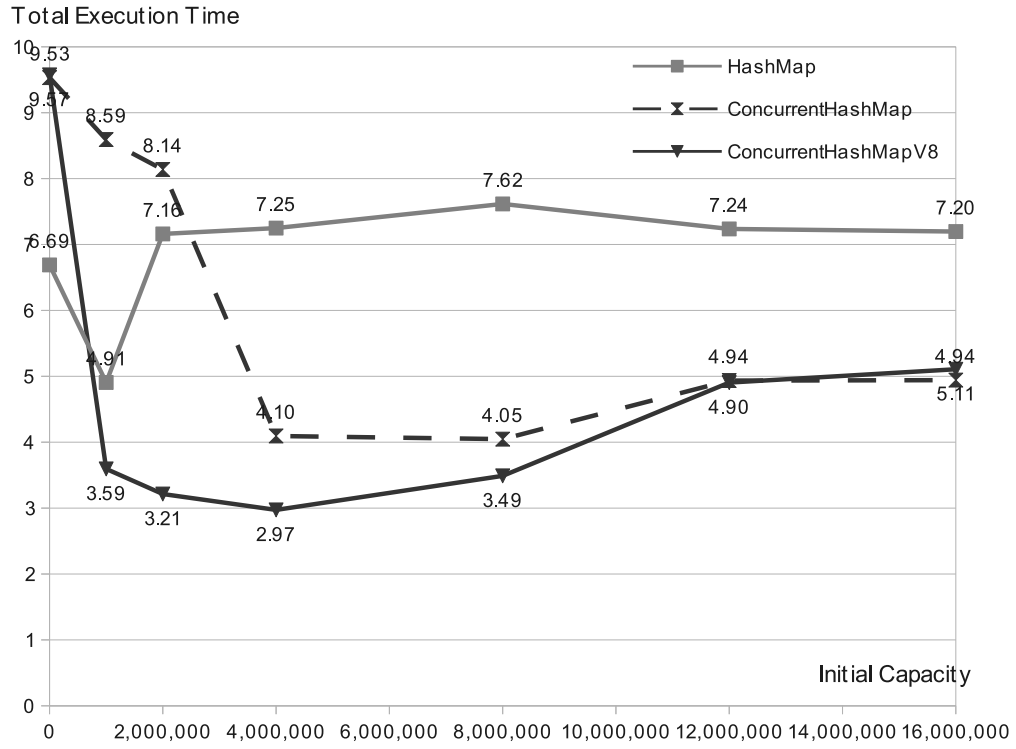
```

1 --table,GenTuples,group=1
2 --table,PvWatts,estimated=16000000,gamma=ConcurrentHashMap
3 --benchmark=30 --threads=4 24

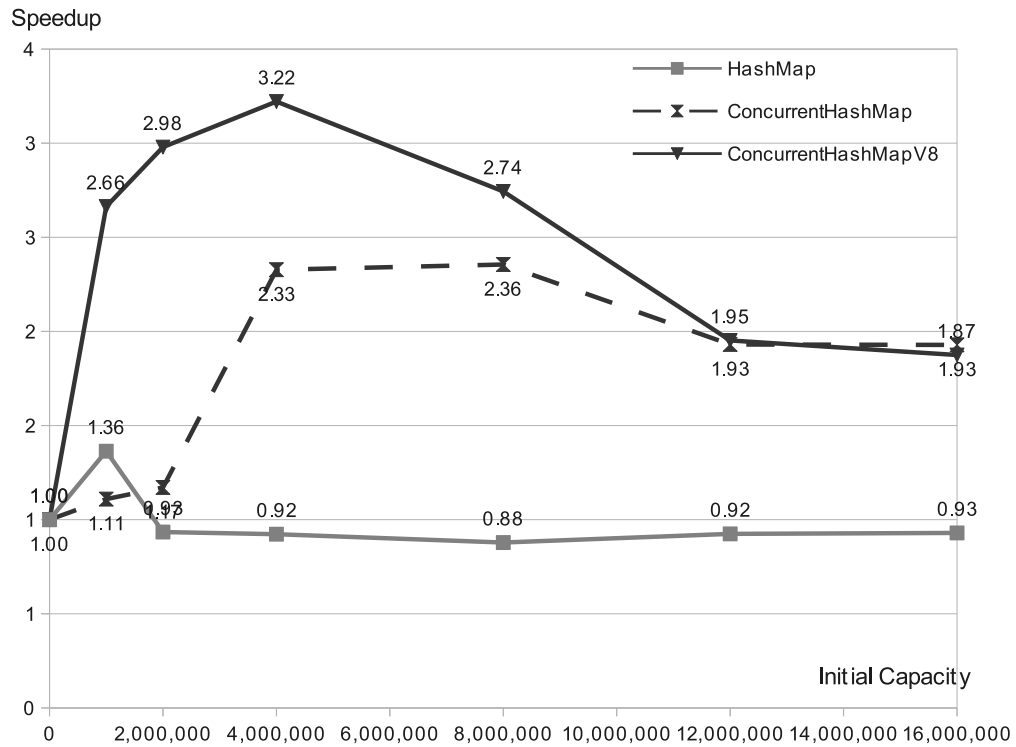
```

The first argument means that each `GenTuples` will be treated as a separate task rather than several tuple being grouped into one group. This is the optimal setting for this program because each `GenTuple` triggers a rule that does quite a lot of work, so it is best handled as a separate task. The second *table* flag specifies `ConcurrentHashMap` as the data structure of the `PvWatts` table in Gamma. And the initial capacity of `PvWatts` table is set to be 8 million. The *benchmark* flag specifies the number of repeated experiments. The *Threads* flag creates 4 threads in the thread pool. The number *24* means that JStar runtime creates 24 `GenTuples` separate tasks.

Figure 4.4 shows that the performance of the sequential version of the JStar `PvWattsGamma` program. The JStar `PvWattsGamma` program is compiled with the *-seq* flag to generate the Java code and use both of the synchronized and non-synchronized Java collection classes, such as the `HashMap`. The experi-



(a) Total execution time with varying initial capacity.



(b) Speedup with varying initial capacity.

Figure 4.4: Performance of inserting 16 million PvWatts tuples in sequential into the Gamma table with three data structures and varying the initial capacity on dual-CPU Intel Xeon W5590 @ 3.33GHz (total of 8 cores.)

ment creates the Gamma PvWatts table with different kinds of data structures, including `HashMap`, `ConcurrentHashMap` and `ConcurrentHashMapV8`.

To realize the effect of resizing the capacity on the performance of a data structure, the experiment vary the initial capacity from 0 to 16 million at seven levels: *i*) 0(default), *ii*) 1 million, *iii*) 2 million, *iv*) 4 million, *v*) 8 million, *vi*) 12 million, *vii*) and 16 million. Each experiment is repeated 30 times. The average execution time skips the first 6 runs and averages the remaining ones (from 7th to 30th). The speedup measures how many times faster than the average time of default capacity (initial capacity = 0) for each data structure.

Figure 4.4(a) shows that `ConcurrentHashMapV8` has the shortest total execution time at most initial capacity, except for the default value and 16 million. The `HashMap` outperforms the other two data structures at the default value, and the `ConcurrentHashMap` has the shortest running time at the 16 million. The speedup graph is shown in Figure 4.4(b). The `ConcurrentHashMapV8` achieves the maximal speedup of 3.22 when the initial capacity is 4 million, but the speedup decreases as the capacity increases from 4 million to 16 million. The speedup of `ConcurrentHashMap` has the second best results, with the maximal speed of 2.36 at 8 million. Similarly, the speedup decreases from 8 million to 16 million. The `HashMap` has the poorest speedup of 1.1 only when the initial capacity is 1 million, and does not have any improvement on the speed at the other levels. This speedup illustrates that the resizing capacity can affect the performance both on the `ConcurrentHashMap` and the `ConcurrentHashMapV8` but makes a little change to the `HashMap`.

The effect of resizing capacity may cause the bias on the benchmarking results, so a suitable capacity for all the `Map` based data structures is needed to avoid this resizing effect. From the Figure 4.4, the total execution time and the speedup of the `ConcurrentHashMapV8` are approximately close to those of the `ConcurrentHashMap` at both of 12 million and 16 million. The `HashMap` also has nearly the same total execution time and speedups at these two levels. Therefore, to fairly compare the performance of these three `Map` implemen-

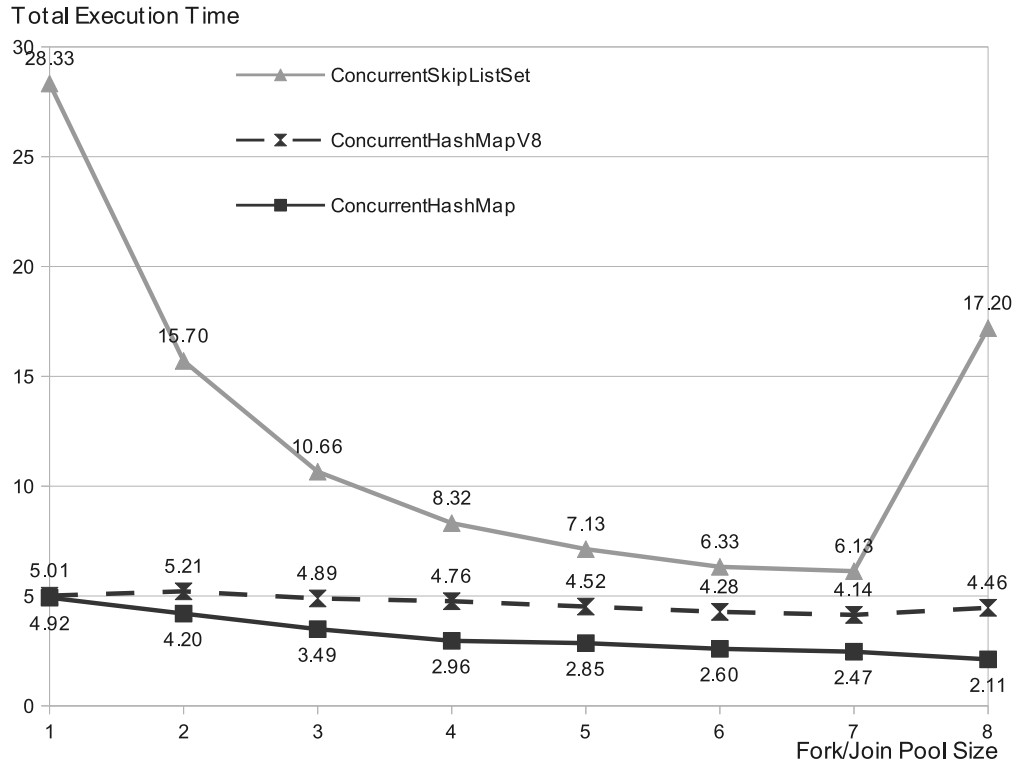
tations and avoid the resizing effect, the appropriate initial capacity should be either 12 million or 16 million. That is, when the **12** or **16 million** is used as the initial capacity for the PvWatts table in the Gamma database, the `HashMap`, `ConcurrentHashMap` and `ConcurrentHashMapV8` will have the stable performance.

4.2.4 Benchmark Result of the Parallel Tuple Insertion

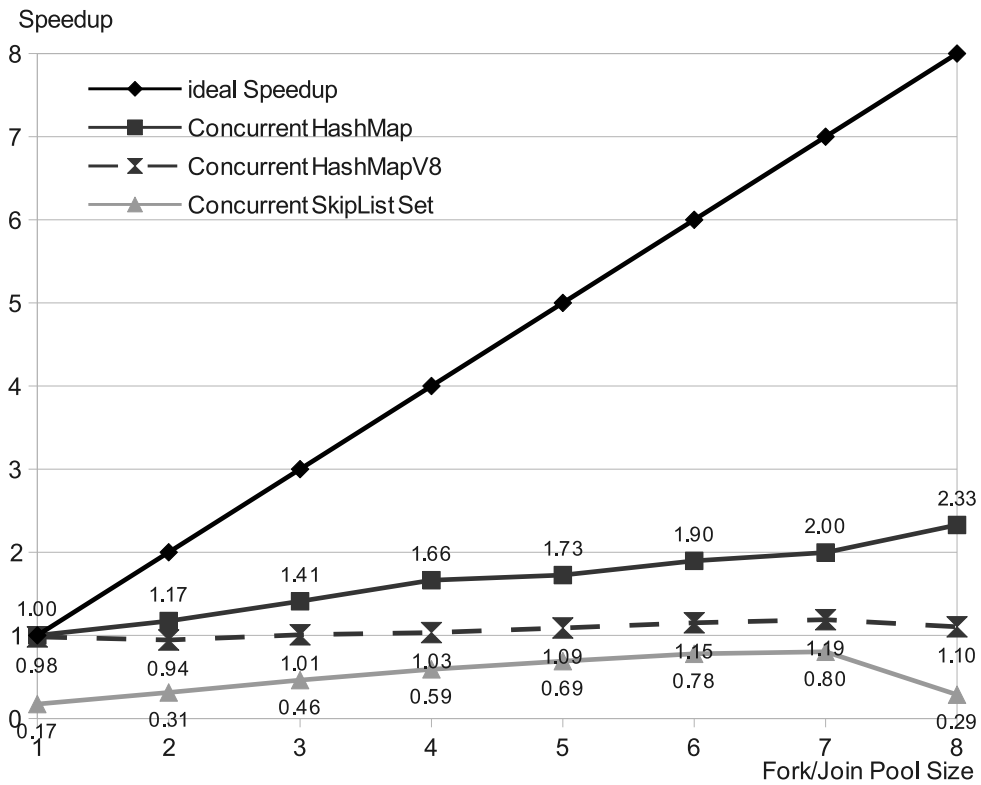
The benchmarking experiment for the parallel version of JStar PvWattsGamma program uses three kinds of concurrent data structures (that is, `ConcurrentSkipListSet`, `ConcurrentHashMap` and `ConcurrentHashMapV8`), and vary the number of threads in the pool for each one of the structures. From the benchmarking results of the sequential PvWattsGamma programs, the parallel experiment uses 16 million as the fixed initial capacity for both of the `ConcurrentHashMap` and the `ConcurrentHashMapV8`. The `HashMap` is not included in this experiments because it requires the external synchronization to be used in the mult-threading environment.

The Figure 4.5(a) shows that the `ConcurrentHashMap` outperforms both of the `ConcurrentHashMapV8` and the `ConcurrentSkipListSet` on the total execution time and the speedup. When the number of threads increases, the total execution time of `ConcurrentHashMap` decreases and achieves the shortest time with 8 threads. Followed by the `ConcurrentHashMapV8`, the `ConcurrentSkipListSet` has the longest time from 1 to 8 threads. Therefore, With the initial capacity of 16 million, the `ConcurrentHashMap` is the fastest data structure for the PvWatts Gamma table. And its running time can be regarded as the baseline to obtain the speedups for the parallel JStar PvWattsGamma program.

Figure 4.5(b) is the absolute speedup graph, related to the single-threaded time of the `ConcurrentHashMap`. It shows that the performance of `ConcurrentHashMap` and `ConcurrentHashMapV8` can be scaled from 1 thread upto 8 threads whereas the `ConcurrentSkipListSet` is scalable upto 7 threads.



(a) Total execution time with varying Fork/Join pool size.



(b) Speedup with varying Fork/Join pool size.

Figure 4.5: Performance of inserting 16 million PvWatts tuples in parallel into the Gamma table with three data structures and varying the Fork/Join pool size on dual-CPU Intel Xeon W5590 @ 3.33GHz (total of 8 cores.).

We conclude that the `ConcurrentHashMap` with the initial capacity of 16 million and within the `PvWattsHashTable` is the best choice of concurrent data structure for implementing the tables in Gamma database. (It would have been better to do these experiments from 8 to 15 threads if possible. Maybe `ConcurrentHashMapV8` has the better scalability with more threads.)

4.3 Speed of the Delta Tree

The insertion time into the Delta tree is an important factor to consider for the Delta tree data structure. As each tuple needs to be inserted into this tree, longer insertion times will cause a delay in processing tuples and thus increase the total execution time. Currently the Delta tree is implemented as a tree structure with fixed depth. The *orderby* clause in the *Table* statements of a JStar program defines the tree depth and the order of tuples.

The JStar Dijkstra program is used to measure the speed of Delta tree data structures. The detailed program will be discussed in Chapter 7. This program first generates a random graph of 1 million vertices connected with 2 million edges where each edge is assigned with a random length ranging from 1 to 10. Then it uses the *Dijkstra* algorithm to find the shortest path from the root node (vertex=0) to the end node (vertex=1,000,000). This program uses 7 kinds of tuples, but there is only one tuple kind inserting into the Delta tree after the optimisation. Therefore, using this program to measure the speed of the Delta tree is more accurate than other JStar programs. The Delta tuple is declared in the follows:

```

1  /** The estimated shortest-path distance from the origin to the given vertex. */
2  table Estimate(int vertex, int distance) orderby (Int, seq distance, Estimate);

```

The Delta tree behaves like the `priority queue` in this test case. The `Estimate` tuples are inserted into the Delta tree at the position in accordance with the *distance*, which is what we need in the JStar Dijkstra program - a priority queue ordered by distance from the starting vertex. The `Estimate`

with the smallest *distance* value will be put in front of all the others. When the JStar runtime takes out the tuples from the Delta tree, it will firstly process the root node (distance=0) and then spans all the other nodes. The *seq distance* expression defines their position in a subtree of the Delta tree: the tuple with the small distance will be placed before the one with the large distance. As the tuples in the front position will be processed earlier than those in the back, the *orderby* clause can ensure that the distance of the **Estimate** tuple in the Delta tree acts like the *priority* and turns the Delta tree into a *priority queue*. Note that a sorting flag, the *par e* expression, allows tuples to be unsorted in the subtree, and thus the delta subtree can be executed in parallel.

4.3.1 The DeltaNode Data Structure Choices

The performance of the Delta tree in the JStar Dijkstra program is determined by the **Estimate** Delta nodes, which belong to the **Integer** data type. Three kinds of **Integer** Delta tree data structures have been implemented in the JStar runtime, as described in the follows:

DeltaNodeInt is the sequential implementation of the nodes of the Delta tree. It stores the Delta nodes with a Java `TreeMap<Integer,DeltaNode>`, which is indexed by the integer field. For example, the **Estimate** tuple uses the *distance* field as its index, and thus the subtrees in the **DeltaNodeInt** are also indexed by the *distance*. When a **Estimate** tuple is inserted into the **DeltaNodeInt** node, the JStar runtime checks this tuple's distance and determines whether a subtree has been created in the `TreeMap`. If so, then the runtime inserts this tuple to the existing subtree. Otherwise, a new subtree is created and put into this `TreeMap` with the tuple's distance as the key.

The `process` method of the **DeltaNodeInt** retrieves and processes the first Delta node from the `TreeMap`, and then removes the node from the `TreeMap`. In the JStar Dijkstra program, the first delta node in the

`DeltaNodeInt` is the `Estimate` tuple which has the shortest distance among the others.

`ParallelDeltaNodeInt` is the parallel Delta node implementation, which stores the Delta nodes with a Java `ConcurrentSkipListMap<Integer, DeltaNode>`. Its insertion and processing behaviors are similar to those of the `DeltaNodeInt`, except that it uses the thread-safe `putIfAbsent` method to put a subtree into its data storage.

`ParallelDeltaNodeIntRange` is another parallel Delta node implementation, which stores the Delta nodes with a Java `AtomicReferenceArray`.

4.3.2 Benchmark Result

The benchmarking experiment measures the total execution times of the JS-tar Dijkstra program, varying the data structures for the Delta tree. As the `DeltaNodeInt` is not thread safe, we use its result as the baseline to calculate the speeds for the two parallel Delta node implementations in a multi-core machine. As shown in Section 7.4.3, benchmark results on a dual-CPU Intel Xeon E5-2680 (total of 16 cores) show that `ParallelDeltaNodeInt` reaches the absolute speedup of 6.37 and `ParallelDeltaNodeIntRange` has the absolute speedup of 4.67 with 15 threads. We conclude that `ParallelDeltaNodeInt` has a slightly better performance than `ParallelDeltaNodeIntRange`.

4.4 Optimisation Strategy

Tuples have different roles in a JStar program. In the `PvWattsGamma` example, `CmdLineArg` tuples are considered as the trigger-only tuples. Their role is to pass the information from command line arguments to JStar `PvWattsGamma`. As the `CmdLineArg` table in the Gamma database is never queried by other rules, there is no need to move the `CmdLineArg` tuples from the Delta tree to the table in Gamma. On the other hand, `PvWatts` tuples should be

moved into the Gamma set directly without needing any Delta tree insertion, because the **PvWatts** tuples do not trigger any rule in this example. The current version of JStar compiler supports two kinds of optimization strategy: *noDelta* and *noGamma*.

4.4.1 *noDelta* Optimisation

The *noDelta* optimisation usage is to run the JStar compiler with the *-noDelta* *T* flag to translate a JStar program to Java source code. For example, translating the PvWattsGamma JStar program with the *-noDelta PvWatts* flag will generate the Java code, which puts each of the **PvWatts** tuples to the table in the Gamma database and immediately fires any rules that use it as the input.[28]

After the JStar runtime parses the program arguments, it labels the **PvWatts** as one of the tuple kinds whose Delta tree insertion should be omitted. When the JStar compiler translates the statements which inserts the **PvWatts** tuples into Delta tree, the compiler will replace the statement, which inserts the **PvWatts** to the Delta tree, with the direct-processing statement. So then the JStar runtime executes **PvWatts** tuples directly instead of putting them into the Delta tree and being executed later.

The advantages of the *noDelta* optimisation is that it reduces the total number of Delta nodes in the Delta tree and improves the speedup. This faster speedup results from the less workload of Delta tree and faster response time of the comparison operation. As the Delta tree contains the smaller number of Delta nodes, the JStar runtime spends less time on traversing each level of the Delta nodes and searching for each new tuple. Thus, the total execution time is shortened. This improvement becomes more significant when one table greatly outnumbered the others in a program. For example, the PvWattsGamma program generates over 16 million of **PvWatts** tuples in total. Applying the *noDelta* flag on the **PvWatts** tuples avoids the insertion of all **PvWatts** Delta nodes and reduces the total number of tuples in the Delta tree

from millions to 24 (the total number of **GenTuples**).

The effect the *noDelta* optimisation is shown on the sequential JStar PvWattsGamma program. The speedup is to measure how much the JStar program with *noDelta* flag runs faster than the native one. Each program was run 30 times. The results of first 6 runs are ignored because of the JVM HotSpot warm-up and the remaining ones are take to calculate the averaged execution time. The benchmark result reaches the 1.4X speedup with the *noDelta* flag on the sequential JStar PvWatts program.

4.4.2 *noGamma* Optimisation

The *noGamma* optimisation usage is similar to the *noDelta* option. That is, compiling a JStar program with *-noGamma T* flag can generate the Java code, which omit the insertion of tuples from table T into Gamma.[28]

Consider the Dijkstra case, which finds the shortest path for the single-source graph and will be described in Chapter 7. The **Estimate** tuples should be applied with *noGamma* optimisation as they are never been queried by other rules. With the *noGamma* optimisation, the JStar runtime can also reduce the time processing the **Estimate** tuples. Before applying the *noGamma* optimisation, the JStar runtime has to take two actions for each **Estimate** tuple: moving the tuple to the Gamma database and executing the next rule. After compiling the program with *noGamma Estimate* flag, the generated Java source code is optimized to directly trigger the rule associated with the **Estimate** tuple without any insertion in Gamma when the JStar runtime processes each **Estimate** delta node from the Delta tree.

The benchmark experiments measure the total execution of the sequential JStar Dijkstra program with/without the *noGamma* optimisation. The benchmark results on the Intel(R) Xeon(R) CPU W5590 (total of 8 cores) shows that the speedup is 1.6X. That is, the sequential JStar Dijkstra program with the *noGamma Estimate* optimisation can run 1.6 times faster than its native implementation.

4.5 Summary

Compiler Flag	Description	Example of Usage
Output Folder	specifies the output folder where JStar compiler generates the Java source code.	<i>-d < folder ></i>
Verbose Mode	details the process of JStar compilation.	<i>-v</i>
Sequential Code	generates sequential code and uses the non-synchronized data structure.	<i>-seq</i>
Parallel Code	generates parallel code and uses concurrent data structure.	<i>-par</i>
noDelta Optimisation	generates the Java code, which omit the insertion of the Delta tree.	<i>-noDelta < tuple ></i>
noGamma Optimisation	generates the Java code, which omit the insertion of the Gamma database.	<i>-noGamma < tuple ></i>

Table 4.1: The JStar compiler option list.

Chapter 5

Performance Tuning Process

This chapter introduces a series of steps aiming to improve the performance of a JStar program. The process basically follows the quality improvement cycle: assess-measure-modify-evaluate. First, *assess* the program and establish the performance index. Second, *measure* the performance and analyze the problems to find the bottleneck. Third, *modify* the implementation and conduct trial experiments. Last, *evaluate* the performance after the modification and check whether the change has improved the performance. Repeat these above four steps until the performance becomes better.

5.1 Performance Tuning Procedure

After experimenting with this process for a number of times, we summarize the strategy in the following steps:

1. Analyze the task dependencies and identify tuple options.
 - (a) Generate the task dependency graph.
 - (b) In-line the Delta tuples : Look at the graph and add a *-noDelta* T flag to for every table T that is NOT used as a trigger.
 - (c) In-line the Gamma tuples: Look at the graph and add a *-noGamma* T flag to every table T that is never queried.

2. Translate the JStar program to the equivalent Java implementation.

Compile the JStar program into a Java parallel program by specifying the above options and the sequential (*-seq*) / parallel (*-par*) mode to the JStar compiler. Then the Java compiler is used to convert all the Java source code(.java) into the byte-code(.class).

3. Reduce the amount of the messages.

Make sure that the program does not display the unnecessary messages as the `println` method bottlenecks the performance. Reduce the output to a few lines/second if possible.

4. Repeat the experiment.

Make sure that the program runs 30 times (*--benchmark=30*) or more to let the Hotspot compiler settle down, and get the reliable average by ignoring at least the first 6 measurement.

5. Vary the number of threads.

Run the program by varying the Fork/Join pool size (*--threads=N*) from 1...8 or 1...15. Plot the speedup curve and stop if the speedup is good enough.

6. Avoid the performance bottleneck from Java garbage collection.

Run the Java program with the *-verbose:gc* option to print out the information of Java garbage collector. Look at the GC times to see if they take up a large percentage of the total running time. If so, try running the program with a larger heap. Or think about how to change the program so that fewer tuples are generated or garbage collected.

7. Tune the performance of Java virtual machine (JVM).

Run the program with a single thread and the *HotSpot* profiler (*-Xprof*). And Investigate what part of the program takes up most of the CPU time.

(a) If it is in one of the data structure methods, think about how to

make that generic data structure more efficient for this particular program, e.g. using a fixed size array instead of the `HashTable`.

- (b) If it is in one of the rules, check whether that rule is being executed in parallel, or is it a sequential bottleneck. If the latter, rewrite the JStar program so that rule is done in smaller parts, so that it can be done in parallel.

5.2 Performance Tuning Tools

We developed some tools to ease the burden of the performance tuners. As investigating the bottlenecks is not easy and sometimes time-consuming, our tools provides some graphs or text messages to shorten the time of finding out the tuple strategy and improving the performance of a JStar program. These tools are described in the following sections.

5.2.1 Task Dependency Graph

Tuple transactions during the execution of a JStar program can be logged in a plain text file by enabling the `log` option (`--log=log.txt`). There are two types of transactions: *PUT* and *GET*. After a tuple is inserted into the Delta tree, the logger writes out a *PUT* message. When a rule queries tuples from the Gamma database, the logger writes a *GET* message. The *GET* and *PUT* messages contain the triggered rule name, input and output tuple names. The following list is the snippet of the log file of the JStar PvWattsGamma program.

```

1  PUT,CmdLineArg,Gamma
2  rule1,CmdLineArg
3  PUT,GenTuples,Delta
4  ...
5  PUT,GenTuples,Delta
6  PUT,GenTuples,Gamma
7  ..

```

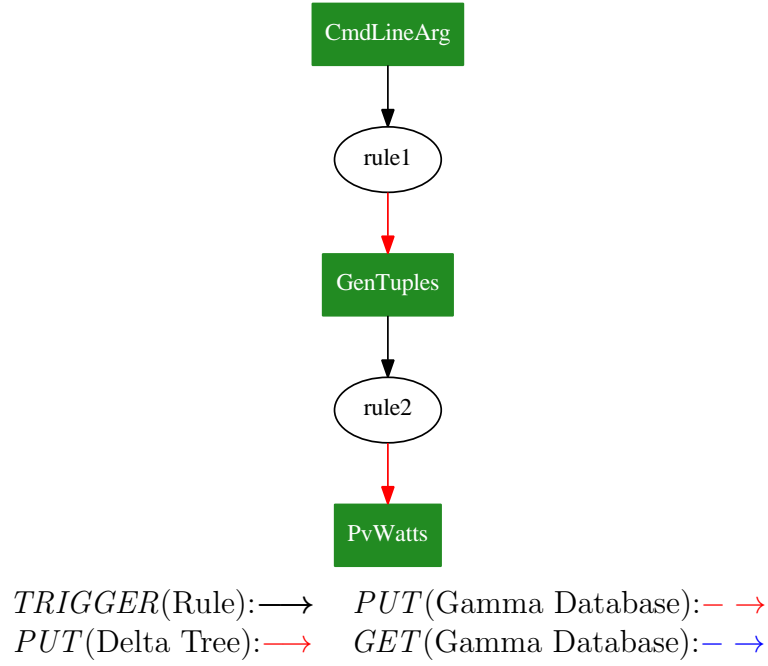


Figure 5.1: The task dependency graph of the JStar PvWattsGamma Program.

The JStar dependency program reads this log file and outputs the task dependency graph (output.dot), which shows the table usage. As the task dependency graphs are written in the *DOT* language, the **Graphviz** is used to view these graphs and to convert them to other graphics file formats, such as EPS (Encapsulated PostScript) file.[11] The following command can generate a task dependency graph from the log file (log.txt), and convert it from the dot file (output.dot) to a EPS file (dependency.eps). Figure 5.1 is the task dependency graph of the JStar PvWattsGamma program.

-
- 1 **java** -cp Dependency.jar jstar.example.dependency.Main log.txt
 - 2 **dot** -Tps output.dot -o dependency.eps
-

5.2.2 Debug Trace Output for Delta Tree, CPU Usage and JStar Tuple Timing Graph

Debugging is commonly used in any programming language. By printing out messages, programmers can easily track the value of a variable in each state and verify the correctness of the program. But a multiple threading programs introduces many potential bugs. For example, the JStar Delta tree is shared

among all threads in the ForkJoinPool, but is allowed only one task (thread) to add the Delta nodes at one time. If a task incorrectly locks the Delta tree, then other tasks have to spend lots of time waiting for *stall* (a delay in the execution time) to be free. To avoid this potential bug, we can implement the synchronization of the Delta tree by using the Java *Lock* object or by retrying the lock-free data structure. Two debug flags have been implemented on the JStar runtime to display the Delta tree information, such as the delta tree size and the CPU usage. In addition, we developed the tuple timing graph to visualize the tuple processing time.

Delta Tree Size (`--debug=delta`) flag enables the JStar runtime to display the total number of tuples in each leaf node of the Delta tree.

CPU Usage (`--debug=cpu`) flag enables the JStar runtime to display the time that a tuple has been processed by a thread and its CPU time. This flag outputs the tuple timing for each tuple in the Delta set, including the *CPU usage*, the *starting and ending wall clock time*, and *the tuple*. The starting wall clock time of a tuple is the time which the JStar runtime takes it out from the Delta tree and starts to process its rules; the ending time is when the JStar runtime finishes its task and moves this tuple to the table in the Gamma database. The CPU usage is a ratio of the time which the thread spends on executing the tuple's tasks and moving to Gamma, which tells us the computation time versus the wall time. The formula is shown in the follows:

$$CPU_Usage(\%) = \frac{CPU_Time}{Wall_Clock_Time}$$

The CPU usage is a performance index, indicating how efficient the CPU processes a tuple in the Delta tree. Thus, the cpu usage of a totally efficient task should be very close to 1.0. The example output line of the

`--debug=cpu` message is shown as follows:

```
DeltaTime: 1220/4424 1369351017581..1369351022005 GenTuples(3,
1999998, 2666664)
```

This message contain 4 parts, separated by the *Tab*. The first is the the message title, followed by the CPU usage, the starting and ending wall time (separated by the `...`), and the tuple. All of debug messages can be printed out to the console or output into a log file. To investigate the tuple time, the log file will be analyzed with the tuple timing graph tool, described next.

JStar Tuple Timing Graph is a visualization tool for displaying the activities of JStar runtime on a time-line. It is implemented with the Google Apps Engine¹, HTML 5 and Java script. Since the JStar tuple timing graph tool is a web-based application, users can access this service on the internet without any installation. By deploying the tools to Google Apps market place or to a local machine (<http://localhost:8888/>), the JStar tuple timing graph tool is able to display the tuple timing results from a JStar log file in a timeline graph. To generate this graph, the CPU debug flag is enabled to print out the real clock time and CPU time, and then the generated log file is dragged onto the Tuple Timing Graph web site.

Figure 5.2 is the tuple timing graph of the JStar PvWattsGamma program with the `PvWatts` table implemented by the `PvWattsHashTable` plus the `ConcurrentHashMap` and 4 threads. Each task was plotted on the time line with its duration. The graph shows that 4 tuples were taken out from the Delta tree each time and executed in parallel. Then the program ended when the last tuple was processed and finished.

¹<https://developers.google.com/appengine/>

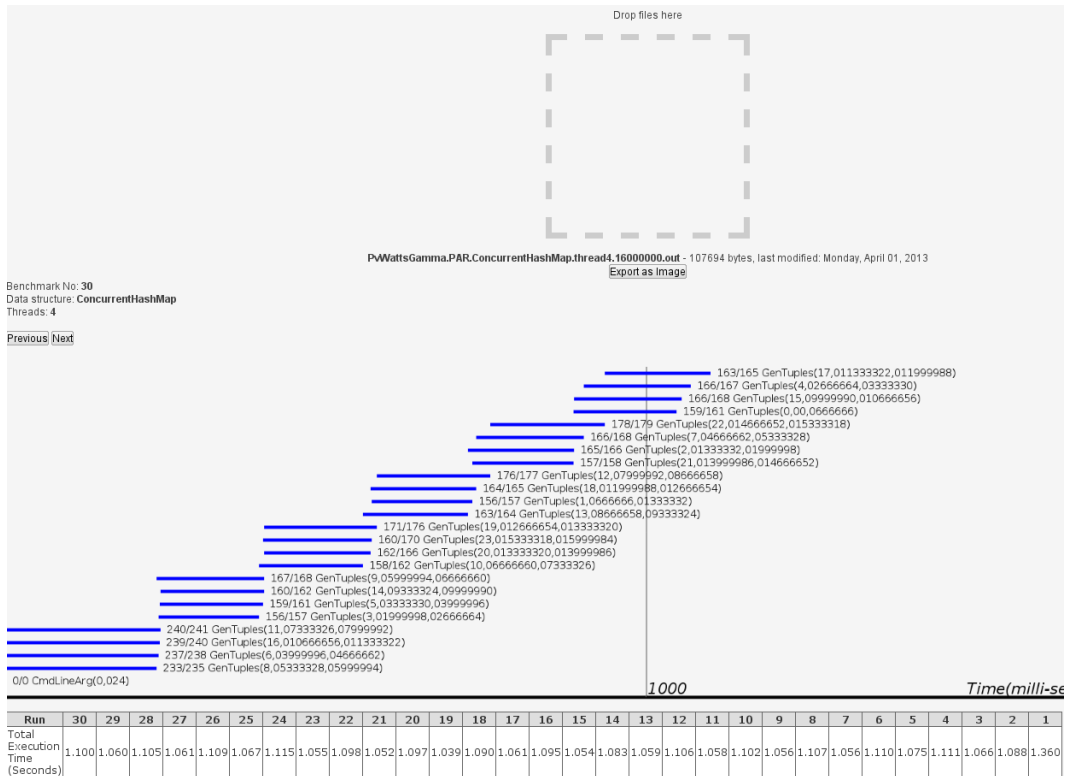


Figure 5.2: The JStar tuple timing graph of the JStar PvWattsGamma program with PvWattsHashTable Gamma table (ConcurrentHashMap) and 4 threads in the Fork/Join pool.

5.3 Summary

The JStar runtime supports several program options to assist the JStar programmers to do benchmarks or tune the performance easily, plus several tools for analyzing the log files. Table 5.1 lists these options and their usages.

Option	Description	Example of Usage
Benchmark	repeats the program for a fixed number of times and prints out each execution time. For benchmarking purposes.	<code>--benchmark=30</code> runs the same program 30 times.
Thread	creates a specified number of threads in the Fork/Join pool.	<code>--threads=2</code> runs the program with two threads.
Debug	prints out debug messages.	<code>--debug=delta</code> prints out the time which the JStar runtime spends on a set of tuples in the Delta tree. <code>--debug=cpu</code> prints out the time which JStar runtime spends on a group of tuples in the Delta tree.
Log	logs the transaction messages among rules, the Delta tree and the Gamma database.	<code>--log=PvWatts.log</code>
Data structure	enables the JStar runtime to choose the data structure and set up its initial capacity (if supported) for a specified table in the Gamma database.	<code>--table,PvWatts, gamma=ConcurrentHashMap, estimated=16000000</code> constructs the PvWatts table in the Gamma database with a ConcurrentHashMap and sets its initial capacity to be 16 million.
Group Size	sets up the group size for a specified tuple type.	<code>--table,SumMonth, group=1</code> sets the group size of SumMonth tuples to be 1. Since each group is processed as one recursive task, this means that every SumMonth tuple in the Delta tree will be a separate Fork/Join task.

Table 5.1: The JStar program option list.

Chapter 6

Case Study: PvWatts

The goal of the PvWatts program is to average the power generation (in WATTS) for each month of a year in Brisbane. All of the hourly power data are generated from NREL's PVWATTS programs. As the amount of records is not very large (only 8760), the parallel program would finish the computation within few mills-seconds and thus the effects of its parallelism would be hardly visible. So we replicated each hourly record one thousand times and stored those records in CSV file. This is the same as analyzing the hourly solar powers output of 1,000 different solar system for a period of one year. Then we write a JStar program to read this file and calculate the average monthly production. This case study was chosen because it involves large-scale data input from a file, plus complex dependencies for the analysis of that data (using several reducers), so speedup is challenging.

6.1 JStar PvWatts Program

To implement the JStar version of PvWatts case, we use the `CsvReaderTask` library to read all records in a CSV file. `CsvReaderTask` provides a `parallelRead` method to create a number of Fork/Join tasks to process one CSV file in parallel. The procedure of `parallelRead` is described in the follows. First, the CSV file is split into N individual segments. Then the `CsvReaderTask` creates the same amount of Fork/Join tasks so that each segment of the file

is read by one task. Each task uses an efficient **FastCSVReader** to read the file segment. The **FastCsvReader** process the files with a reading-on-demand pattern: reading a record at request. Thus, its performance is improved as it creates few objects and avoids unnecessary **String** conversions. The source code of JStar PvWatts program is shown in List D.1 of Appendix D.

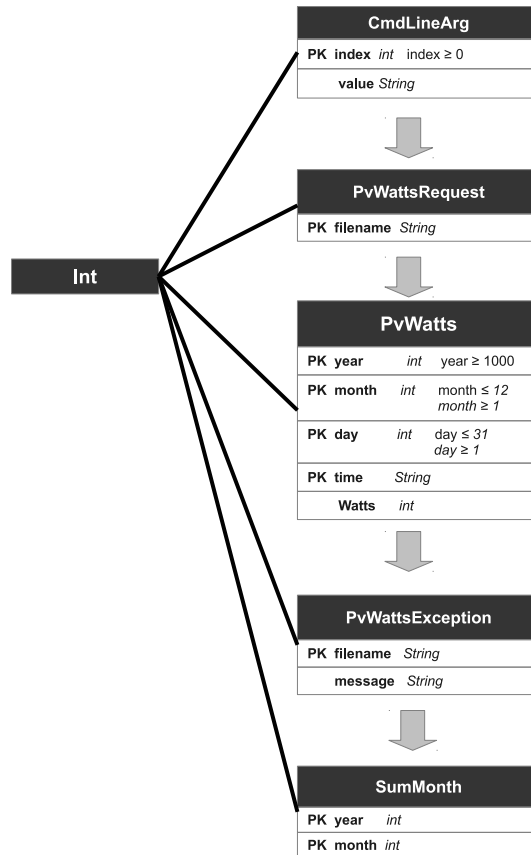


Figure 6.1: The table schema of the JStar PvWatts Program.

The JStar PvWatts program contains five tables: **CmdLineArg**, **PvWattsRequest**, **PvWatts**, **PvWattsException**, and **SumMonth**. The table schema is shown in Figure 6.1. After reading arguments from the command line, the JStar PvWatts program creates one **CmdLineArg** tuple for each argument and inserts it into Delta tree. Before the JStar runtime moves these **CmdLineArg** tuples to the Gamma database, it executes the rule associated with them. This rule creates **PvWattsRequest** tuple with the corresponding argument values and puts all of them into the Delta tree. Then, when each **PvWattsRequest** is processed by the JStar runtime, the file path information is sent to the next rule. Thus,

by creating **PvWattsRequest** tuples with the values of **CmdLineArg** tuples, the information can be passed from the command line arguments to the JStar **PvWatts** program.

When the rule associated with **PvWattsRequest** tuple is triggered, multiple readers are used to read the CSV file in parallel and then output **PvWatts** tuples. First, by querying the **CmdLineArg** tuples, the rule gets the given number of readers and creates the same number of **CsvReaderTasks**. Since each task is responsible for one separate segment of the input file, the total work of reading one CSV file can be executed by multiple threads. Each reader reads one record at one time and parses its fields as primitive data types, such as an integer or a string. Then one **PvWatts** tuple is created with these fields and inserted into the Delta tree. Then the reader reads the next record and repeats the above procedure to output **PvWatts** tuples until all of records in its region of the file have been read. Note that each reader reads slightly past the end of its region to ensure that each record is read exactly once.

SumMonth tuples are sorted by yearly and monthly fields in the Delta tree. The **PvWatts** rule processes each **PvWatts** tuple and creates a corresponding **SumMonth** tuple with its yearly and monthly values. So a number of redundant **SumMonth** tuples are generated in this loop as most of them have the same yearly and monthly values. Since the Delta tree uses tuples as its key values, tuples with identical values are discarded immediately when they are added to the Delta tree, which avoids triggering duplicate rules. By inserting **SumMonth** tuples into Delta tree, all the months that require calculation are recorded for processing later, after all input records have been read.

Reducers are used to calculate the average monthly power. Each **SumMonth** rule queries a collection of **PvWatts** tuples with its yearly and monthly values. Even though the tuples in this collection might be randomly sorted, the mean calculation will not change its value because it is associative:

$$\forall a, b, c, n \in int, \frac{((a + b) + c)}{n} = \frac{(a + (b + c))}{n}$$

As the mean calculation is an associative operation, the `Statistic` object implemented from the `Reducer` class of the JStar runtime can be used to average the power generation for each month and print out the final results.

6.2 Benchmark of the Naive JStar PvWatts Program

Symphony node CN-191	
CPU	2 x Intel [®] Xeon [®] quad-core W5590 CPUs (total of 8 cores @ 3.33GHz)
L1 cache	32K
L2 cache	256K
L3 cache	8192K
RAM	48GB RAM (@1333 MHz)
Disk	2 x 1TB hard disc
OS	64-Bit Linux operating system (kernel version 2.6.38)
JAVA	64-Bit JRE version 1.7.0_17

(a) Hardware specification

JVM options
<code>-Xmx8G</code> sets maximum Java heap size to be 8 GB.
<code>-verbose:gc</code> enables verbose garbage collector.
<code>-XX:+PrintCompilation</code> prints the message when a method is compiled.[14]
<code>-XX:+PrintTenuringDistribution</code> prints the tenuring age information.[14]
<code>-Xbatch</code> stops the program while the hot spot compiler is recompiling/optimising the code.

(b) Java Virtual Machine options

Table 6.1: Benchmark configuration of the JStar PvWatts program.

The CN-191 computing node of Symphony cluster is selected to run the benchmarks of JStar PvWatts program and its specification is shown in the Table 6.1(a). As CN191 is one of the Symphony cluster nodes and its computing resources are managed by the *Torque* and *Maui scheduler*, we followed standard operating procedures shown in Appendix B to conduct the JStar

PvWatts benchmarking experiments.

The performance of the JStar program would partially depend on JVM. To avoid IO communication dominating total execution time, the output of JStar PvWatts program have been reduced to 40 lines as shown in the List D.2 of Appendix.D. Besides, tuning the JVM options can also avoid the Java garbage collection spending too much time reclaiming memory.

Table 6.1(b) is the configuration of the JVM options used to benchmark the JStar PvWatts program. 8G of heap is required to run the JStar PvWatts program as the JStar runtime needs sufficient memory spaces to store tuples in the Gamma database and the Delta tree which are both in-memory databases. If the heap size is not large enough, the garbage collector(GC) would be frequently called to free the memory space, and this would cause delays and degrade the performance.

The benchmark experiment measures the total execution time of both sequential and parallel versions of the JStar PvWatts program, with no optimisation options. Each experiment is repeated 30 times. The average execution time ignores the results of first 6 runs and takes only the later experiments (7th-30th) into account. The number of threads ranges from 1 to 8. To illustrate the parallelism of JStar program, the absolute and relative speedup are plotted on the graph. And their formula are defined as follows:

$$S_{abs} = \frac{T_s}{T_p}$$

$$S_{rel} = \frac{T_1}{T_p}$$

where:

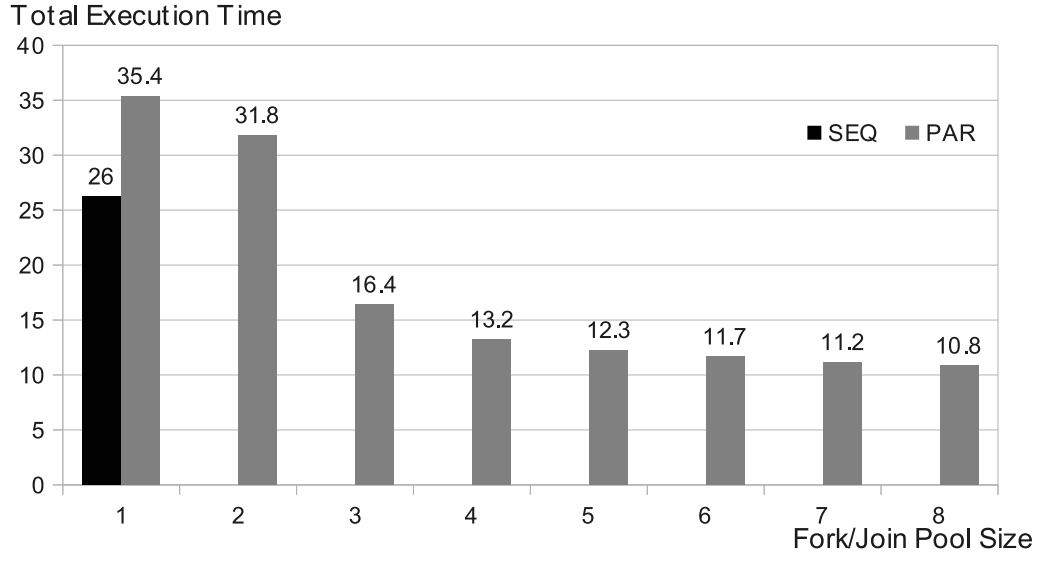
S_{abs} is the absolute speedup.

S_{rel} is the relative speedup.

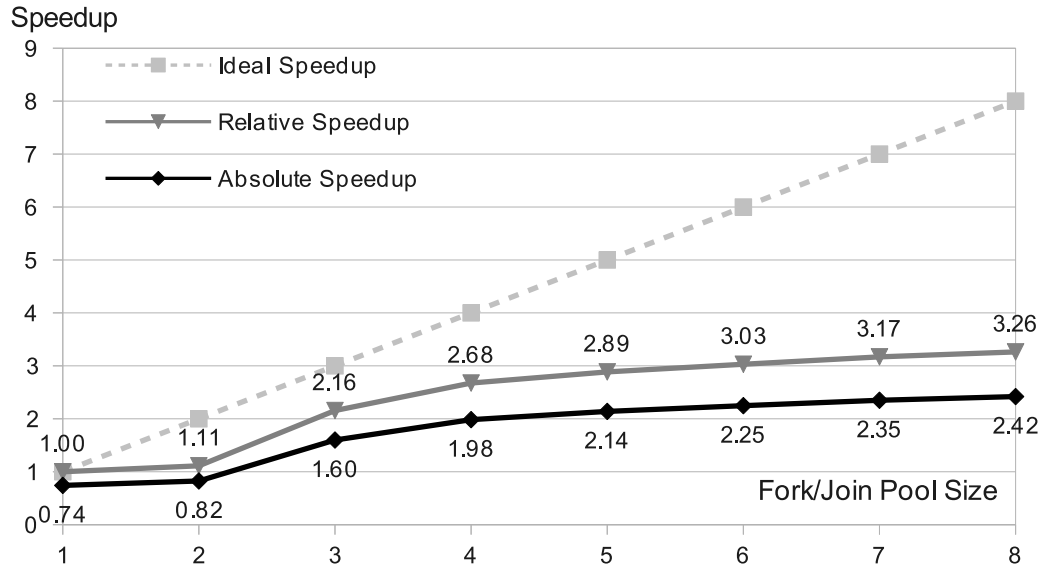
T_s is the average execution time of the sequential JStar program.

T_1 is the average execution time of the parallel JStar program with one thread.

T_p is the average execution time of the parallel JStar program with p threads.



(a) Total execution time with varying Fork/Join pool size.



(b) Speedup with varying Fork/Join pool size.

Figure 6.2: Performance of the naive JStar PvWatts program with varying the Join/Fork pool size on dual-CPU Intel Xeon W5590 @ 3.33GHz (total of 8 cores)

Figure 6.2 is the benchmarking results of the native JStar PvWatts program on *sorted1000X.csv*. It shows that the JStar parallel PvWatts program runs faster than the sequential one with more than 3 threads. Both the relative and absolute speedup graph shows that the time of the parallel program decreases as the number of threads increases to 8.

6.3 Performance Tuning Process

As described in Chapter 5, the PvWatts case is optimized by applying the performance tuning process. The steps are shown in the following subsections:

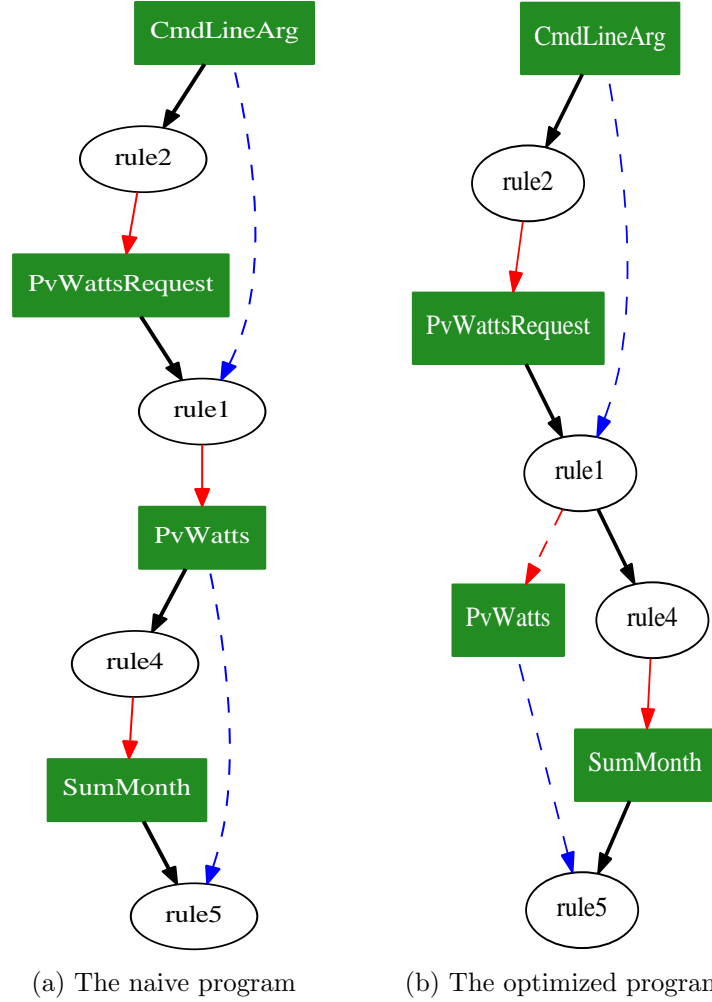


Figure 6.3: The task dependency graphs of the native and optimized JStar PvWatts programs.

6.3.1 Analyzing Tuple Dependency

The JStar PvWatts program uses a map-reduce programming model for processing a large input file (*sorted1000X.csv*). The program uses one program parameter to locate the input file, then by reading this file with one or more

readers, the program outputs **PvWatts** tuples to the Delta tree and Gamma database. After the reader finishes its task, the reducers query these tuples from the Gamma and calculate the power generation during each month. The task dependency graph of the native JStar **PvWatts** implementation is shown in Figure 6.3(a). However, this program execution is inefficient because all of the tuples must be inserted into two in-memory data structures (Delta set and Gamma database). In the **PvWatts** program, the number of **PvWatts** tuples is in proportion to the input file size. When the number of **PvWatts** tuples increase, the **PvWatts** program dynamically requests JVM to allocate more heap memory space. If the maximal heap memory space is not sufficient, then the JVM frequently calls garbage collector to free and reclaim the memory space. As the garbage collection takes time, the overall performance of the JStar **PvWatts** program slows down as well.

6.3.2 Inlining Tuples

PvWatts and **SumMonth** are the tuples which should be inlined to improve the speed of the JStar **PvWatts** program. By analyzing the task dependency graph, we see that the **SumMonth** tuples are never queried during the execution of the program. Thus, there is no need to move **SumMonth** tuples to Gamma database, that is, the *noGamma* optimisation can be applied to the **SumMonth** tuples.

Another optimisation strategy is to shift the summation request rule (Rule 4) to the reading phase. After the reader loop creates a **PvWatts** tuple, it immediately triggers early execution of the **PvWatts** rule and outputs a **SumMonth** tuple. That is safe, because the rule does not perform any database query.[6] With this change, the **PvWatts** tuples are no longer used as the trigger of rules but only for the query in the reducer loop. By applying *noDelta* optimisation on **PvWatts** table, the reader loop does not insert a **PvWatts** tuple into the Delta tree but directly puts it in Gamma and triggers the Rule 4. This new strategy improves the performance of the Delta tree and avoids the

unnecessary work, and thus increases the speedup of the JStar program.

And the optimized program execution is shown in Figure 6.3(b). The differences are:

- The *noDelta* optimisation: the *PvWatts* tuples are no longer put into the Delta tree but directly moved to the Gamma database.
- The *noGamma* optimisation: the *SumMonth* tuples are put into the Delta tree only.

Tuple	Delta Tree	Gamma Database
CmdLineArg	✓	✓
PvWattsRequest	✓	✓
PvWatts	×	✓
SumMonth	✓	×

Table 6.2: The inline tuple list of the JStar PvWatts program.

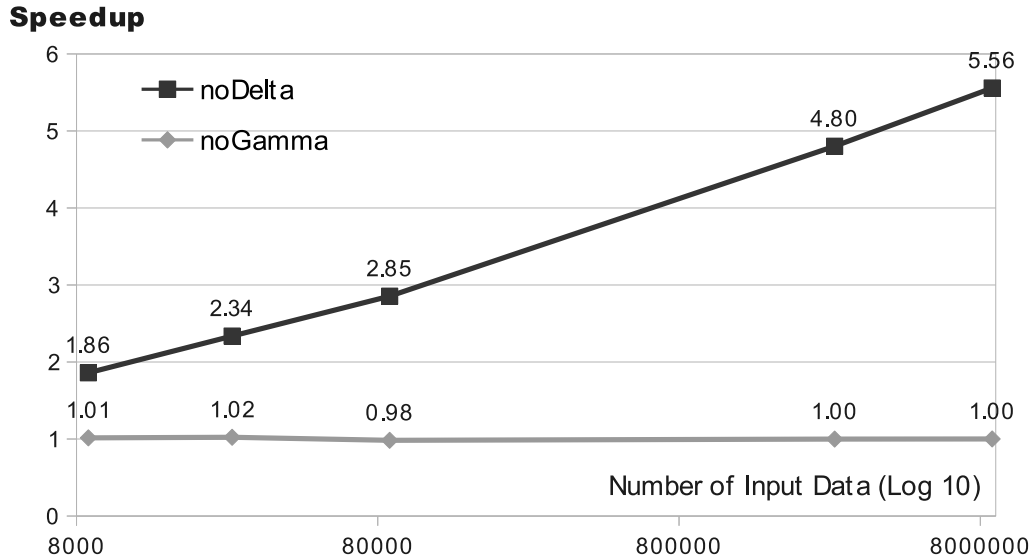


Figure 6.4: Speedups of the sequential JStar PvWatts program with varying the optimisation strategies on quad-CPU Intel E7-4870 @2.40GHz (total of 32 cores).

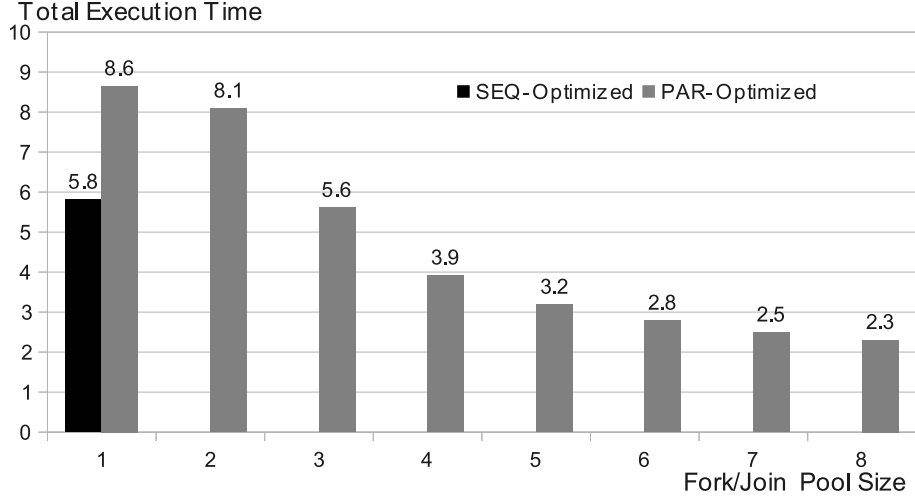
Table 6.2 is the tuple list of the optimized JStar PvWatts program. The check symbol (✓) symbol means the tuple is inserted into the data structure and the cross symbol (×) means that the tuple is not put into the data

structure. To illustrate the effects of the above optimisation strategies on the JStar PvWatts program, we benchmarked the program by applying with one optimisation separately and varying input files sizes.

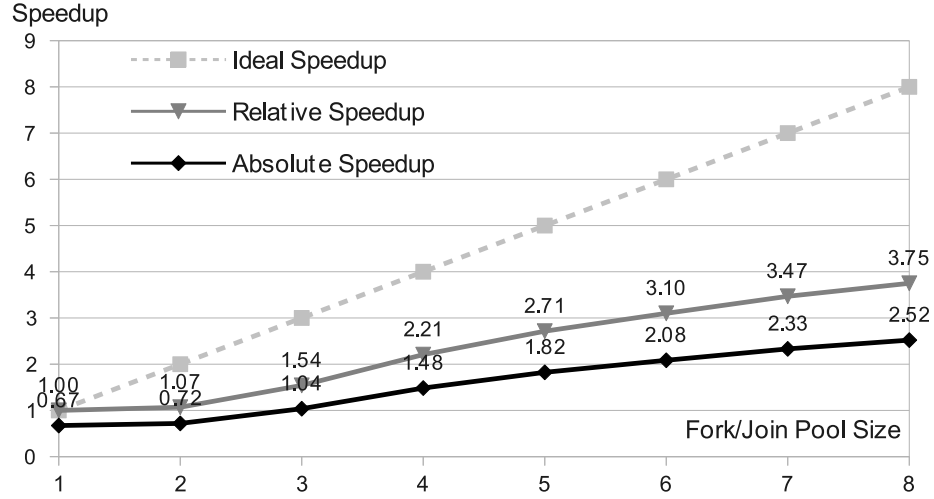
Figure 6.4 is the benchmark results on one of the computing node in the NeSI cluster (The submission procedure to the NeSI cluster is shown in Appendix C and the submission job configuration is in Appendix C.2). The speedups in this figure show the effect of applying one optimisation strategy on the JStar PvWatts program for 5 kinds of input file sizes ($1\times$, $3\times$, $10\times$, $300\times$ and $1000\times$). The speedup is the relative speedup to the naive sequential JStar program (without any optimisation). The results show that as the number of tuples increases, the *noDelta* optimisation scales up the performance with the maximum speedup of 5.56 on the largest input file (192Mb, 8,760,000 records). But the *noGamma* optimisation does not increase the speed and provides a little or no scalability on the JStar PvWatts program. We conclude that the effective optimisation strategy for the sequential JStar PvWatts program is the *noDelta PvWatts* option.

The largest input file is used to benchmark the optimized parallel JStar PvWatts program, using the *-noDelta PvWatts* optimisation and *-noGamma SumMonth* optimisation and varying the number of threads.

Figure 6.5 is the benchmark results on the CN-191. From the absolute speedup chart, the optimized strategy shortens the total execution time of the parallel implementations by 2.52 times, compared to the sequential optimized one. And the relative speedups show that the optimized parallel program can improve the performance as the number of threads increases from 1 to 8, with a maximum relative speedup of 3.75. That is, the parallel optimized program with 8 threads can run 3.75 times faster than the program using one thread. From the above two benchmark results, we conclude that the optimisation strategies not only improves the speedup but also provides the scalability for both of the sequential and parallel JStar PvWatts program.



(a) Total execution time with varying Fork/Join pool size



(b) Speedup varying with varying Fork/Join pool size

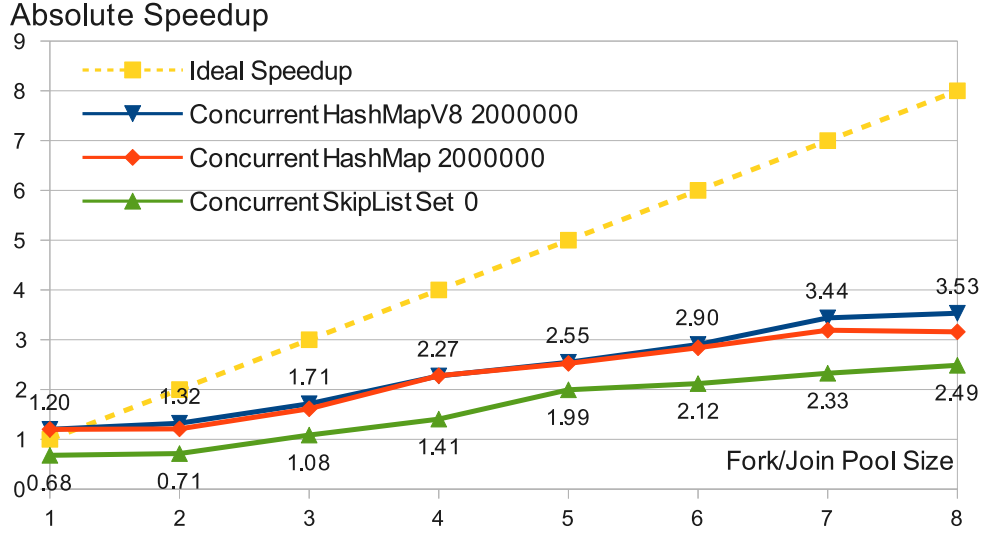
Figure 6.5: Performance of the optimized parallel JStar PvWatts program with two optimisations and varying the Fork/Join pool size on dual-CPU Intel Xeon W5590 (total of 8 cores).

6.3.3 Data Structures of the PvWatts Gamma Table

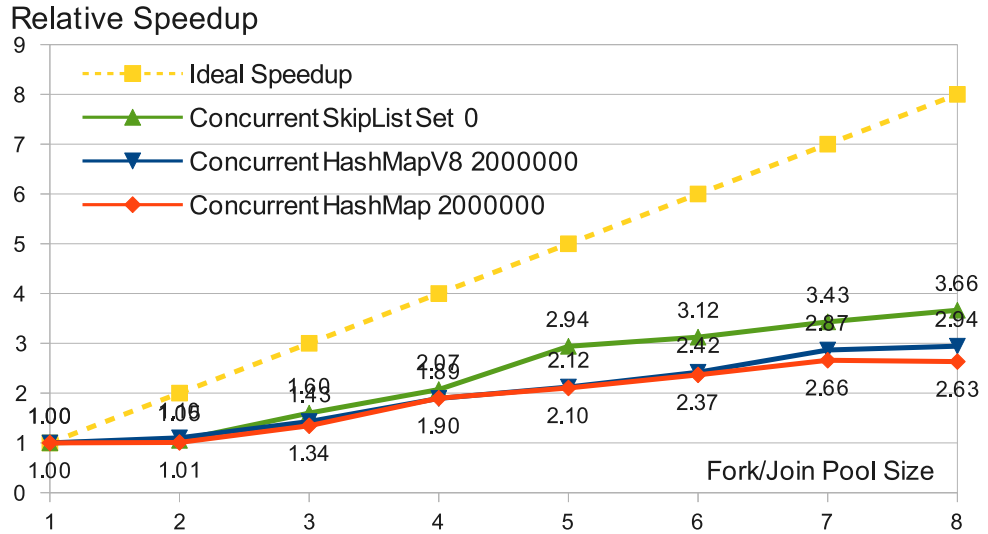
Regarding the data structures, the `PvWattsHashTable` is a customized implementation of the PvWatts table in Gamma. It uses the *category* concept to design the PvWatts table data structure. As the `PvWattsHashTable` use the month as its index to insert the PvWatts tuples, the query of PvWatts tuples in one month could be performed efficiently.

The naive JStar program uses `ConcurrentSkipListSet` to create the PvWatts table in Gamma. We could improve its parallelism by using other thread-safe

Map implementations, such as `ConcurrentHashMap` or `ConcurrentHashMapV8`. By using a customized `PvWattsHashTable` and overriding the `createGammaSet` method of the `JStarMain` class, these two types of Map data structures could be used to instantiate the `PvWatts` Gamma database.



(a) Absolute speedup with varying Fork/Join Pool Size.



(b) Relative speedup with varying Fork/Join Pool Size.

Figure 6.6: Speedups of the optimized parallel JStar PvWatts program with varying the PvWatts Gamma table data structure and the Fork/Join pool size on dual-CPU Intel Xeon W5590 (total of 8 cores).

Figure 6.6 are the absolute and relative speedup graphs of the optimized parallel version of JStar PvWatts program, varying the number of threads and the data structures of `PvWatts` tables in the Gamma database: `ConcurrentSkipListSet`, `ConcurrentHashMap` and `ConcurrentHashMapV8`. Each Gamma table is con-

structed with initial capacity of 2,000,000, so that the effect of resizing buckets of Gamma table is avoided, which results in a better performance.

The optimized parallel program with `ConcurrentHashMapV8` with 8 cores runs 3.66 times faster than the optimized sequential program, and has the slightly better relative speedup than the `ConcurrentHashMap` (2.94 relative speedup versus 2.63). As the `ConcurrentHashMap` with initial capacity of 2 million has slightly slow but similar speedups as the `ConcurrentHashMapV8`, the resizing side-effect can be removed by setting up an appropriate initial capacity. The naive parallel program (with `ConcurrentSkipListSet`) with 8 cores has the lowest absolute speedup of 2.49 but has the best relative speedup of 3.66 because it has the slowest execution time with one thread. Therefore, after applying the optimisations, we can still increase the speed of the parallel program over its native program by using one of the concurrent data structures (`ConcurrentHashMap` or `ConcurrentHashMapV8`) with 2 million capacity.

6.4 Phase Experiment

Before attempting further parallelism, we need to analyze the workflow of the optimized JStar PvWatts program and find out what rule/phase mainly limits the improvement on the program speed, so that we could use other tool to solve this bottleneck problem.

From the Figure 6.7, we can see that the program is executed in two phases: the reader loop and the reducer loop. The reader loop reads the input file and inserts the `PvWatts` tuples into the Gamma database and then puts the `SumMonth` tuples into the Delta tree. The reducer loop gets one `SumMonth` tuple from the Delta tree, and averages the `PvWatts` tuples of one month by making a query to the Gamma database. The reducer loop must be started after the reader phase. As the reader and reducer are implemented with parallelism, it is not easy to investigate the bottleneck by looking at the graph merely.

The phase experiment is designed to determine whether the bottleneck of

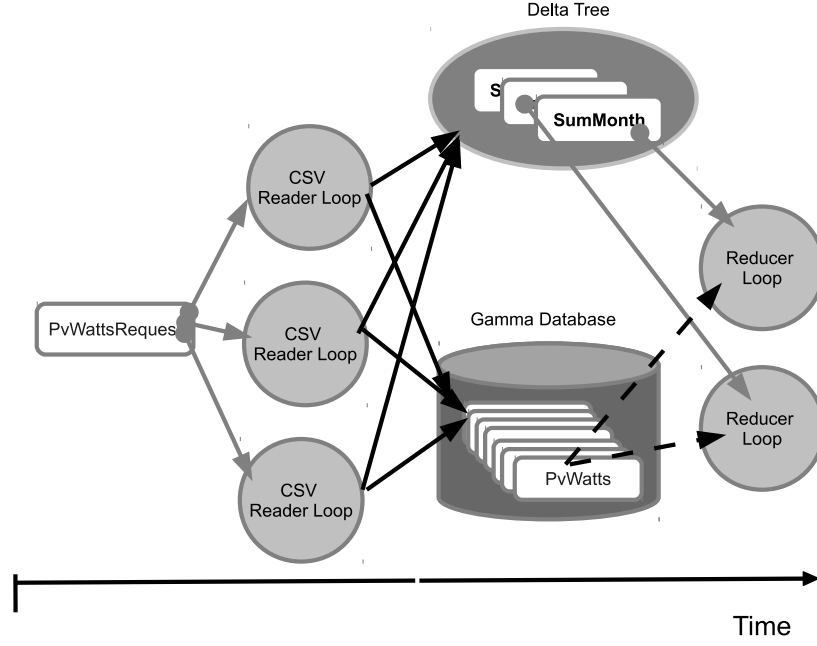


Figure 6.7: The workflow of the two-phase JStar PvWatts program.

the JStar PvWatts program is the reader or the reducer. We compiled the JStar PvWatts program with `-seq` flag to generate the sequential Java source code. Then the `Main` class is overridden to execute the program phase-by-phase, so that we could benchmark the reader phase and the reducer phase individually. And we use the same data structures in all the phase experiments (the `DeltaNodeInt` for the `SumMonth` and the `TreeSet` for the `PvWatts`). The phase experiment design is described in the following list:

Phase 0 benchmarks the time which JStar PvWatts program reads and parse the input file. At this phase, the program only uses a single reader to read the file, and counts the total number of records. Hence, no tuples is inserted in either Gamma database or Delta tree. At the ending, the program prints out the number of records to ensure that JVM actually executes the codes. Note that as JVM uses Just-In-Time compilation for improving the performance of *Hot Spot*, JVM would sometimes optimize the program and skip some codes which are not critical for the whole program. Printing out the variable values can avoid this JVM optimisation.

Phase 1 benchmarks the *Phase 0*, plus creating the `PvWatts` tuples and put them into the Gamma database. No `SumMonth` tuples are created in this phase.

Phase 2 benchmarks the *Phase 0*, plus creating the `PvWatts` tuples, plus creating the `SumMonth` tuples and insert them into the Delta tree. The `PvWatts` and `SumMonth` tuples are both created in this phase.

Phase 3 benchmarks the *Phase 0*, plus creating the `PvWatts` tuples and put them into the Gamma database, plus creating `SumMonth` tuples and insert them into the Delta tree, plus doing the reducer loop. This phase is running the JStar `PvWatts` program.

Phase 10 benchmarks the *phase 0*, plus creating the `SumMonth` tuples and inserting them to the Delta tree. No `PvWatts` tuples are created in this phase.

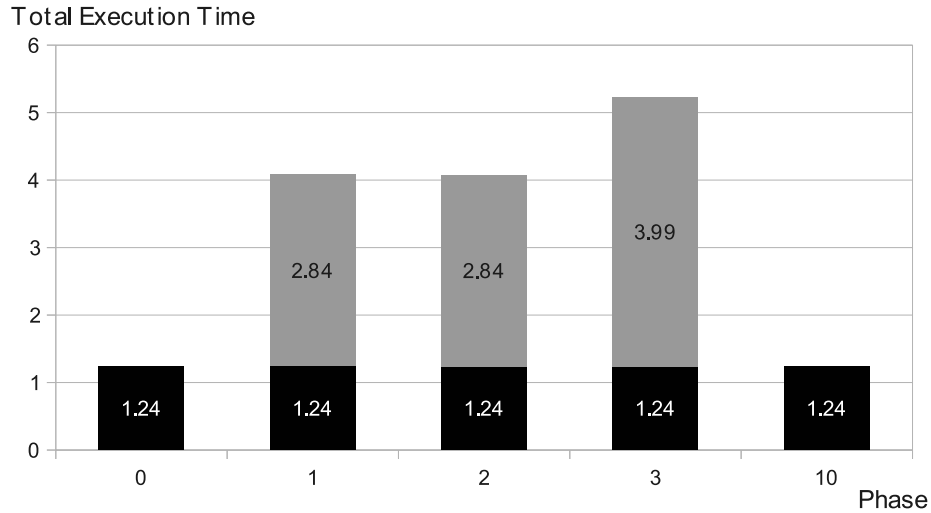


Figure 6.8: Performance of the phase experiment with varying the phase on dual-CPU Intel Xeon W5590 (total of 8 cores).

Figure 6.8 is the benchmark results of the phase experiments. *Phase 0* results show that the `FastCsvReader` takes 1.24 seconds to finish reading the input file. We will use this execution time to calculate the phase time difference. *Phase 10* has the same time as *Phase 0*, so putting the `SumMonth`

tuples into the Delta tree does not take up significant time and slows down the speed. And this observant result is also consistent to that of the time difference between *Phase 1* and *Phase 2* (*Phase 1* is roughly equivalent to *Phase 2*).

The time difference between *Phase 1* and *Phase 0* (2.84 seconds) shows that creating all the **PvWatts** tuples and inserting them into the Gamma database takes 2.5 times of the reading task. The difference between *Phase 3* and *Phase 2* (1.15 seconds) indicates that the reducers take roughly the same amount time as the reading task.

Based on the phase results, the execution time of the JStar PvWatts program can be split into three parts: the reading task (23%), the **PvWatts** tuple insertion(55%) and the reducers (22%). Inserting the **PvWatts** tuples the Gamma database and the reducers are the bottlenecks of the JStar PvWatts program. In the next section, we will introduce *Disruptor* to help us to remove the bottlenecks and improve the speedups.

6.5 Disruptor Version

Disruptor is an order-matching, real-time and in-memory transaction processing system. Compared with other data exchange approaches, Disruptor has less write contention, a lower concurrency overhead and a more friendly cache mechanism. It implements a queue approach between concurrent threads that has low latency and high throughput. It also provides low levels of jitter, using new designs of producers, consumers and data storage. A key aspect of their design philosophy is to get the best caching behaviours by having only one thread (core) writing to any memory location.[27]

6.5.1 RingBuffer

The *RingBuffer* design of Disruptor solves the data content problem and improves the efficiency of disruptor applications. By preallocating a fixed size of

the memory, disruptor instance uses a block of memory space and forms an array-like queue to provide the storage for exchanging data among threads. When the consuming threads (*Consumer*) are too busy to take out the data from the ring buffer, the producing threads (*Producers*) still can put data onto the buffer without blocking, until it becomes full.

The *RingBuffer* automatically retains a **sequence number** which links to the last item in the buffer, and is incremented atomically after the producer places a new entry onto the ring buffer. This sequence claiming principle does not cause any lock if only one producer finds the next available slot on the buffer. On the other hand, the consumers are given a sequence number which determines the slot to be read from the ringbuffer. But consumers cannot access this slot and reads data from it until its status becomes available. Sequence number coordinates the producers and consumers to work together with minimal contention in a multi-threading environment.

The *RingBuffer* uses a low-latency memory design to improve the performance of Java garbage collector. As the *RingBuffer* is a pre-allocated and fixed-size memory space, the JVM tries to use a contiguous area in the main memory space or possibly the cpu cache line, which can gain a fast access speed by utilizing the *cache striding*. Besides, each slot in the *RingBuffer* might be overwritten with new data several times during the execution of a disruptor program. That is, most of the data in the *RingBuffer* are the short-lived and *immortal* objects, which will be referenced once in the program and then never be used again. Because the garbage collector does not need to move these immortal objects to the tenured memory space, the memory space of these immortal objects can be quickly reclaimed back and returned to the JVM. The preallocated design and shorted-lived objects of *RingBuffer* reduces effectively the burden of the Java garbage collector and works efficiently.

The *RingBuffer* also ensures that there is no message loss among producers and consumers. For producers, the *RingBuffer* acts as a queue and allows them to place data in batches without any interruption. For consumers, the

RingBuffer is a buffer that helps them to deal with busy traffic situations without missing any data. For example, if the producer outpaces the consumers and puts so many events that consumers could not process immediately, then these new upcoming events are retained in the ring buffer until the consumers start to process them. The Disruptor system provides optimized implementations for single/multiple producers and single/multiple consumers, which are described in the next two subsections.

6.5.2 Producer

The *single-producer* mode provides mutual exclusion and maintains the execution order. In this case, the *RingBuffer* is always accessed by only one thread and no other threads are able to write data on the ring buffer. No writing contention occurs during the execution and thus there is no need to ensure mutual exclusion with locks or *CAS (Compare and Swap)* instructions. This makes the single-producer mode fast and efficient.

The single producer claims events in an ordered and sequential manner. Thus, the consumers will see events in the same order as the producer adds the events. Figure 6.9 illustrates the procedure of publishing an event to the ring buffer. The single producer first asks the *producer barrier* for the next slot. As the ring buffer keeps track of the current sequence number(2), the barrier quickly finds the next slot by locating the slot adjacent to the current slot. Before the producer writes data to the slot, the barrier needs to check the availability of this new slot. If the slot is still occupied by one of the consumers, then the producer barrier will wait until none of the consumers accesses it.

When this slot is ready for writing data, the producer barrier updates the sequence number to the next sequence number (3) and the producer can start to write data onto it. After completing the writing, the producer tells the barrier to commit the changes to the *RingBuffer* and consumers. The producer barrier updates the sequence number to 3. And it also publish the

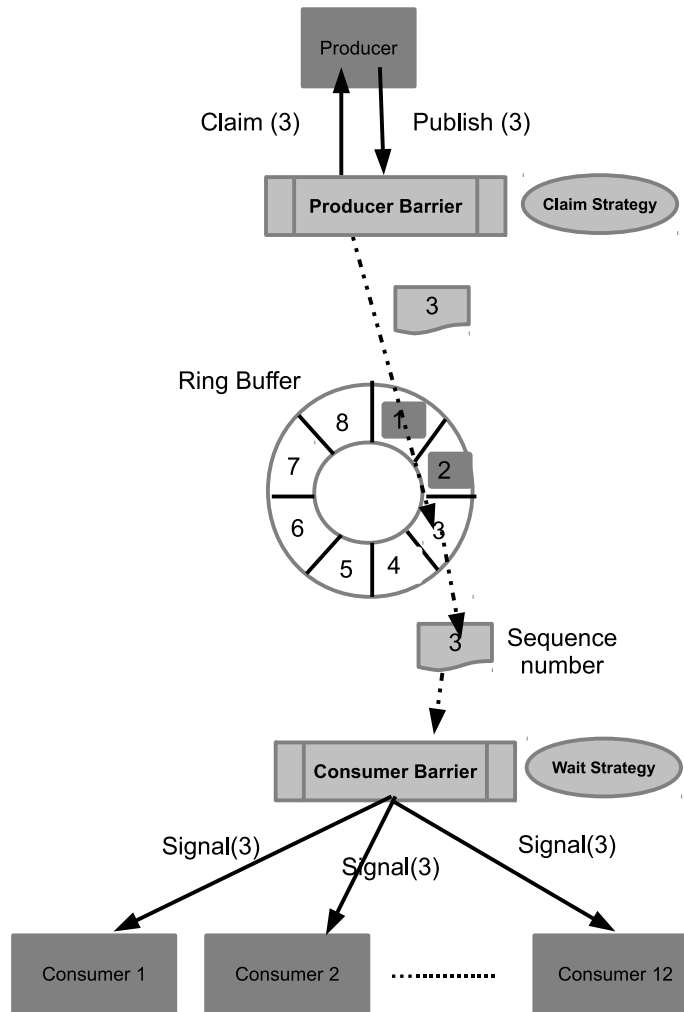


Figure 6.9: The workflow between one producer and *RingBuffer*.

slot (3) and notifies all the consumers that the new data on the slot (3) is ready for reading.

Disruptor also supports multiple producers publishing events onto the *RingBuffer* by using a concurrent version of the claim strategy. By default, the Disruptor uses `SingleThreadedStrategy` claim strategy for only one producer and applies the `MultiThreadedStrategy` strategy to coordinate the multiple producers. The difference between these two claim strategies is the type of variables that they use to avoid the wrapping of the *RingBuffer*. The single-producer uses a `long` variable as there is no needs for CAS operation. But the multi-producer one uses an `AtomicLong` variable, which has been implemented with lock-free and thread-safe programming to ensure the atomicity of the sequence number.

Producers can claim a batch of slots and publish many events in one step. This batching effect reduces the overhead costs of the producer and producer barrier, and helps consumers to regain the pace with producers so that the whole concurrent system is balanced. This effect increases the throughput and helps the Disruptor remain low and flat in latency.

6.5.3 Consumer

Each consumer processes every event put into the *RingBuffer*, so Disruptor can run multiple consumers concurrently. When a slot is claimed by the producer, the sequence number for that slot becomes unavailable and none of the consumers can read any data from it. Once the producer publishes a slot, the *RingBuffer* notifies all the consumers that that sequence number is ready and then consumers can either all read in parallel or take turns to read data from it.

To guarantee that any change in a slot would be visible for all consumers, Disruptor uses the *sequence barrier* to force all the consumers to wait until the *RingBuffer* changes its status. And For example, when a slot is being written by the producer, its sequence number is blocked from every consumer. After the producer publishes it, the sequence barrier gives out the reading notification to all consumers. All of the consumers can read the slot concurrently.

The *consumer sequences* allow consumers to coordinate work on the same entry in an ordered manner.[27] Consumers wait for the next sequence number to become available before they read the event from it. Consumers do not directly interact with the each consumer but use the *consumer barrier* (that is, a coordinator of consumers), which tracks the current available reading slot. As each consumer has a separate *consumer sequence*, the Disruptor can assign the consumer to read the slot with respect to its own sequence.

Figure 6.10 shows the procedure when the barrier grabs three events from the *RingBuffer* and sends them to consumers. After the producer publishes the event onto slot 3, the *RingBuffer* updates the sequence number to 3 and makes

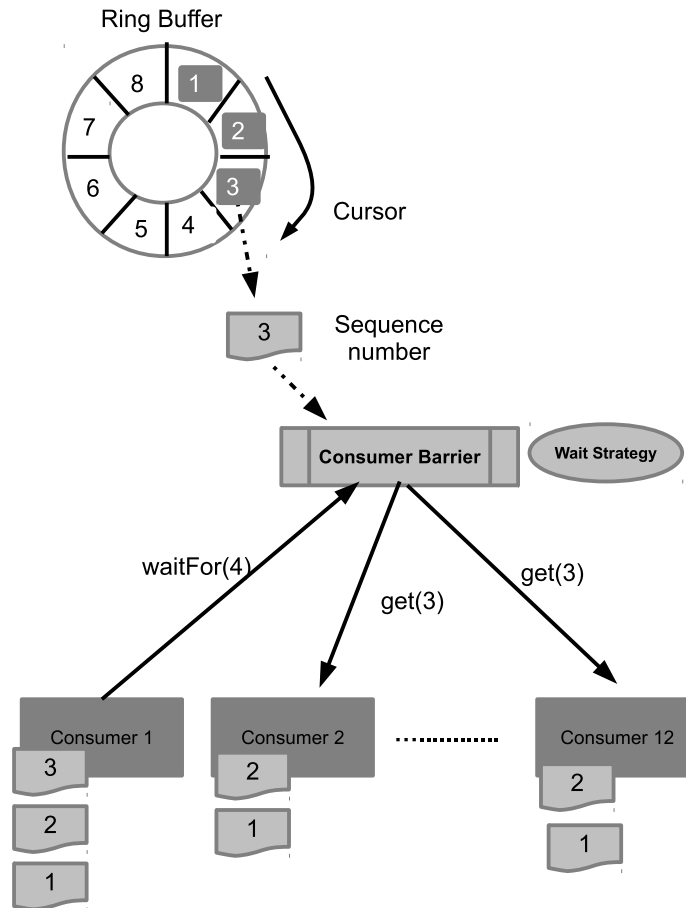


Figure 6.10: The workflow among *RingBuffer* and 12 Consumers.

the slots 1-3 all visible to the consumers. When a consumer asks its consumer barrier about the next sequence number in the *RingBuffer*, the barrier tells that consumer the highest sequence number and the number of available slots in the *RingBuffer* (slot 1, 2 and 3 in this case). Instead of directly querying the status of each slot in the *RingBuffer*, consumers passively wait for the barrier which tells them what they should read data from. After requesting the barrier to fetch events from these available slots, each consumer starts to process the events individually. The advantage of the single consumer barrier is that it allows the *RingBuffer* to be read without needing any multi-reader lock. Therefore, disruptor has a lower latency performance than other readers-writer framework.

A consumer barrier sometimes forces consumers to wait for the next sequence number when the *RingBuffer* is too busy to return the number. It uses a waiting strategy to define how consumers should behave when they await

the number. The following introduces each one of these waiting strategies and describes the work flow with the example shown in Figure 6.10.

- **BlockingWaitStrategy** uses a lock and condition variable to force the consumer to wait for the next entries. When consumer 1 awaits the next sequence number, the barrier locks up consumer 1 and causes it to await until the cursor moves onto the sequence 4. Since the lock and condition are used to control the waiting time for consumers, chances are that this strategy causes a high latency during the execution and thus slows down the system performance. As a result, this strategy is not suitable for high performance applications but is suitable for the limited CPU resource applications, which demand more threads than the cores on the machine.
- **BusySpinWaitStrategy** uses a busy spin loop within each consumer while the cursor is moving onto the next sequence number. Instead of stopping the consumers working, this strategy keeps the awaiting consumers busy as much as possible, and thus consumes more CPU resources than other strategies. But when sufficient cores are available, it is the best strategy for the CPU-bounded application which needs high performance and low latency.
- **YieldingWaitStrategy** yields the threads occupied by the awaiting consumers and gives them back to the thread pool, so that other busy consumers can take up these new threads to speed up the overall progress. But this strategy occasionally causes latency spikes after some regular intervals.
- **SleepingWaitStrategy** is similar to **YieldingWaitStrategy** but takes a different action when the consumers have been waiting for a short duration. Initially this strategy keeps the awaiting consumer spinning. After a regular interval, if the cursor still has not moved to the next sequence, then the system scheduler disables the consumer temporarily

(a few nano seconds). When the consumer becomes disabled, it can not accept any command but must sleep until the sleep time elapses or the barrier interrupts the consumer.

`BlockingWaitStrategy` is the only waiting strategy we use in the disruptor version of the JStar PvWatts program. The disruptor PvWatts program uses only one producer to output PvWatts tuples, and 12 consumers to concurrently calculate the average power for each month of the year. As a result, there are 13 worker threads and one main thread running in the thread pool. `BlockingWaitStrategy` is designed to coordinate these consumers to work well with limited CPU resources. `BusySpinWaitStrategy` has bad performance when the worker threads outnumber the CPU cores; thus, it is not a suitable strategy for benchmarking the disruptor program.

Instead of spinning the awaiting consumers, both of `YieldWaitStrategy` and `SleepingWaitStrategy` free up their resources and give them to those who are busy and in need of threads. But they achieve this goal by calling an inappropriate method (`Thread.Yield` method), which sometimes fails to give the throughput. For example, if only one consumer needs to do a large workload job and others have finished their tasks, then the scheduler pauses the threads and give one thread to the busy one. But in this case, yielding threads does not shorten the total execution time as all the other consumers still have to wait for the bottleneck consumer to finish the job.

6.5.4 Disruptor PvWatts Program

The Disruptor version of PvWatts program parallelizes the two-phased workflow of JStar PvWatts program. It uses a single producer and multiple consumers to process all tuples during the execution. Its work flow chart is shown in Figure 6.11 and described as follows.

The program initializes the Disruptor instance by specifying the number of consumers, the number of producers and waiting strategy for the *RingBuffer*. After the disruptor starts up the ringbuffer, one producer and 12 consumers,

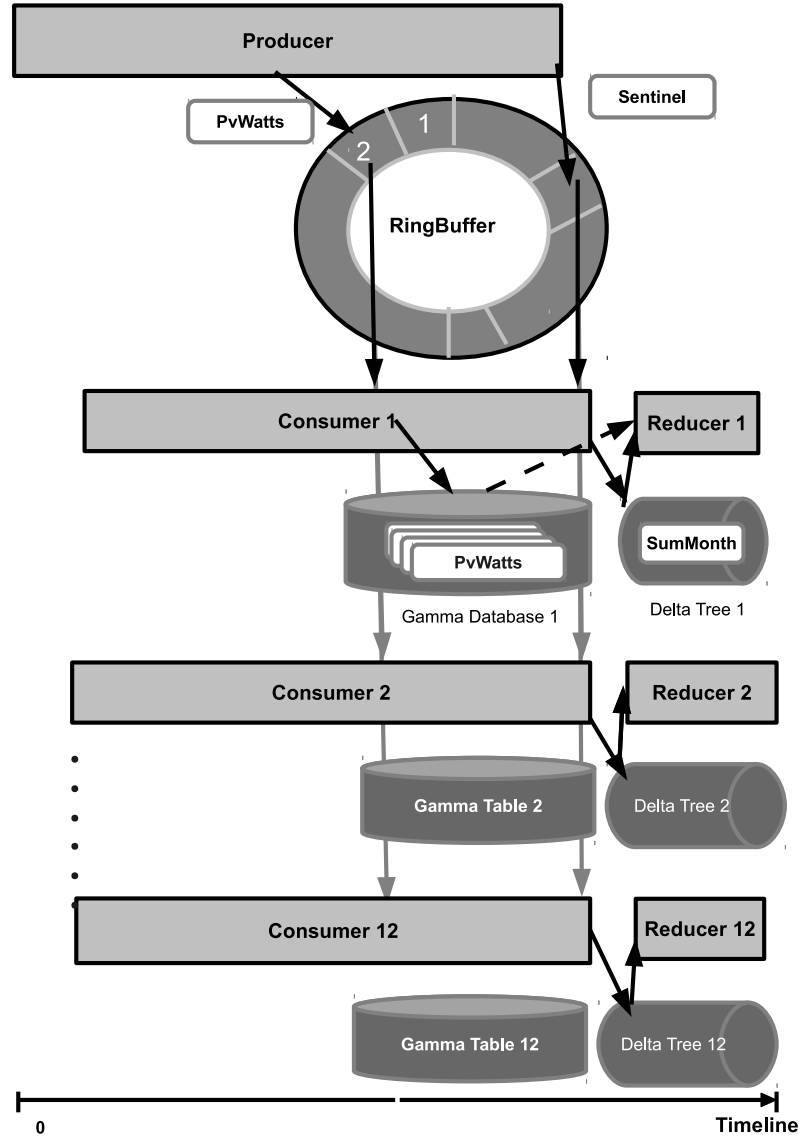


Figure 6.11: The workflow of the Disruptor PvWatts program.

the producer starts doing the *CSV read loop* tasks: reading the large input file, generating all **PvWatts** tuples, publishing these tuples in batch mode and sending out a sentinel tuple after all lines in the input file have been processed. At the same time, each consumer starts to claim the **PvWatts** tuples from the *RingBuffer*.

To reduce the workload of *reducer loop* and improve the parallelism, the Disruptor PvWatts program assigns a separate month value to each consumer. Thus, each consumer just needs to process the **PvWatts** tuples of one month and put these tuples to its own and local Gamma database. Besides, the consumer also creates one corresponding **SumMonth** tuple for each **PvWatts** tuple

and inserts this tuple to a local Delta tree. When a consumer receives the sentinel tuple, it processes the **SumMonth** tuple in its own Delta tree and triggers the reducer loop to output the average monthly power generation. Then the reducer queries the tuples in the Gamma table, sums up the watts values and prints out the averaged monthly power generation.

The producer places tuples onto the ring buffer in batch mode: claiming and publishing upto 256 events each time. Since there is only one producer in the program, the **SingleThreadedStrategy** strategy is used to claim the slots in the *RingBuffer*. But consumers read one event from the ring buffer each time. Thus, each consumer gets the **PvWatts** tuple in the same slot when they receive notification from *sequence barrier*. As the same slot could be accessed by 12 consumers concurrently, **BlockWaitStrategy** is set up for defining the behaviour of multiple reading operation on the *RingBuffer*. According to the sequence number, each consumers reads one slot in parallel and the publisher is blocked from writing to the slot until all the consumers finish their reading.

Category	Parameter	Value
RingBuffer	Event	PvWatts(Builder)
RingBuffer	Size	1024
RingBuffer	Wait Strategy	BlockingWaitStrategy
RingBuffer	Claim Strategy	SingleThreadedClaimStrategy
Producer	Number of Producer	1
Producer	Batching Size	256
Producer	Task	Read input file, create PvWatts tuples and place tuples onto the ring buffer .
Consumer	Number of Consumers	12
Consumer	Batch Size	256
Consumer	Task	Put PvWatts tuples to Gamma database and process the SumMonth tuple from Delta Tree.

Table 6.3: The configuration of the Disruptor PvWatts program.

The single-producer and multiple-consumer design removes the possibilities of contention occurring in the procedure of JStar PvWatts program, provides data locality and pipelines the reducer and the consumers. The summarized Disruptor configuration is shown in Table 6.3.

6.5.5 Benchmark Result

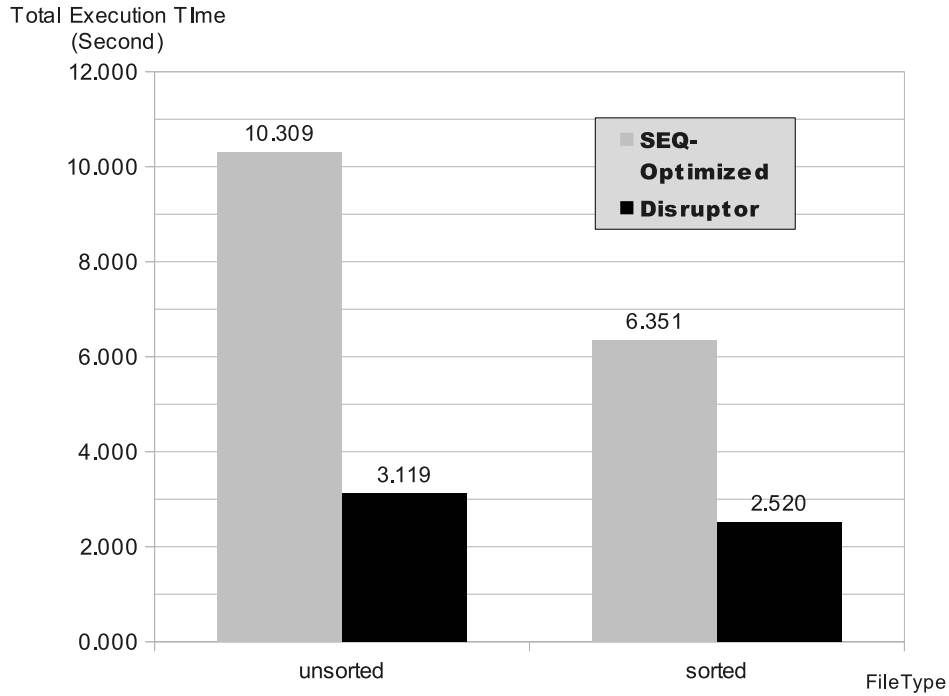


Figure 6.12: Performance of the sequential optimized JStar and Disruptor PvWatts Programs on two kinds of input files (*sorted* and *unsorted*) on an 8-core machine (Intel i7-2600 with 4 cores + hyperthreading).

The benchmark experiments were carried out on an Intel i7-2600 quad-core machine to evaluate the speed up of PvWatts program between the Disruptor version and the sequential optimized version. The performance of this experiments is measured by using speedup, which is computed on the basis of the total execution time of sequential inlined JStar PvWatts program. According to the *Amdahl's Law*, the theoretical maximum speedup with one producer and N consumers is:

$$Speedup(N) = \frac{1}{B + \frac{1-B}{N}}$$

Assume that B is the percentage of the program that must be run in sequential. From the phase experiments, the reading phase takes 1.24 seconds. The sequential percentage of JStar PvWatts program is 19.5%(1.24/6.351). The maximum speedup with 12 consumers is 3.8 ($\frac{1}{0.195 + \frac{0.805}{12}}$). Figure 6.12 shows that the Disruptor PvWatts program has the speedup of 3.3 on the unsorted input file (*large1000X.csv*) and the speedup of 2.5 on the sorted input file. This speedup is fairly good, compared to the theoretical speedup.

6.6 Conclusion

In the PvWatts study, we learned:

- The optimized JStar parallel code gives a reasonably good speedup upto 4 cores with the maximal speedup of 3.53 (8 cores), compared to the sequential optimized JStar PvWatts program.
- This program is a two-phase program and hard to parallelize as it has the bulky I/O communication and uses the complex data structures. And it is Gamma database dependent because at least 8.7 million PvWatts tuples must be inserted in or queried from the Gamma database.
- The Disruptor PvWatts program obtains a fairly good speedup of 3.3 with a single *RingBuffer* (size of 1024), one producer and 12 consumers. This speedup is very closed to the theoretical speedup from the Amdahl's Law.

Chapter 7

Case Study: Dijkstra's Shortest Path Algorithm

7.1 Dijkstra's Shortest Path Algorithm

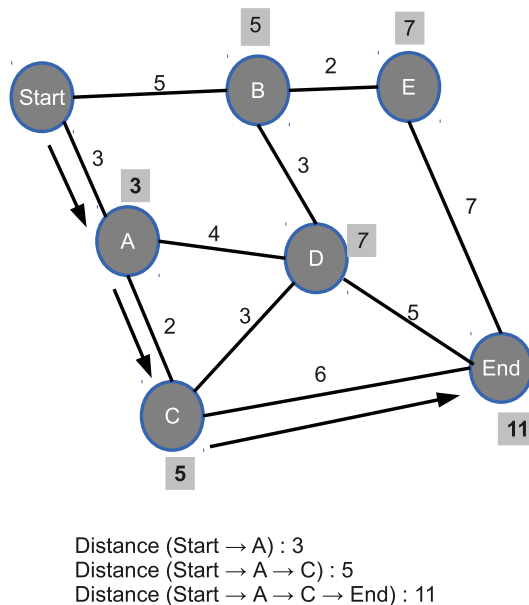


Figure 7.1: The shortest path solved by the Dijkstra's algorithm.

The Dijkstra's shortest path algorithm solves the shortest path problem that has one starting point and non-negative path costs. The input graph contains one starting node, one ending node and other intermediate nodes. Each edge connects two nodes in the graph and has a distance cost. The algorithm finds a path with the lowest cost from the starting node to the

ending one. Figure 7.1 illustrates how Dijkstra’s algorithm solves the shortest path problem for a graph with 7 nodes and 10 edges.

1. Set the node *Start* to be the first node. From this node, calculate the distance to node *A* (path cost = 3) and the distance to node *B* (path cost = 5). As node *A* has the shortest distance, it is set to be the next node.
2. Start from node *A* and calculate the distances of its neighbouring nodes. The distance to node *D* (through node *A*) is 7 ($3 + 4$), and the distance to node *C* is 5 ($3 + 2$). As the path to node *C* through node *A* has the shortest path cost, node *C* is set to be the next node.
3. From node *C*, the direct path to node *End* (path cost = 11) has lower cost than the indirect path through node *D* (path cost = 13). Thus, the shortest path from node *Start* to node *End* is through node *A* and node *C* with the lowest path cost of 11.

7.2 JStar Dijkstra Program

The JStar Dijkstra program implements the Dijkstra’s algorithm to find the shortest path to every node of a random connected graph. The source code of the JStar Dijkstra program is shown in the Listing E.1 of Appendix E. This program is a typical two-phase task: the graph generation and the shortest-path algorithm. We chose this case study because the shortest path phase has dynamically varying amounts of available parallelism that are dependent on the shape of the graph and the lengths of edges, so static scheduling strategies are not adequate. The graph generation creates a directed graph with one million vertices and two million edges where each edge has a random cost ranging from 1 to 10. The Dijkstra’s algorithm finds the shortest path from the vertex (0) to every vertex of the graph. The procedure of the JStar Dijkstra program is described as follows:

1. Graph Generation Phase:

For each vertex $V_1 \in 1 \dots 1,000,000$, randomly choose another vertex $V_2 \in 0 \dots (V_1 - 1)$. Then create an edge (V_1, V_2) with a random length $(1 \dots 10)$. Repeat this step until the graph forms a connected tree with one million vertices and one million edges. Generate another one million random edges between the vertices. Randomly choose two vertices from the tree and connect them with a directed and random length edge. Repeat this step until another one million edges are added to the tree. Note that the tree contains one million vertices and two million edges.

2. Shortest Path Phase:

Use Dijkstra's algorithm to find the shortest path from the initial node (Vertex = 0) to the every other node.

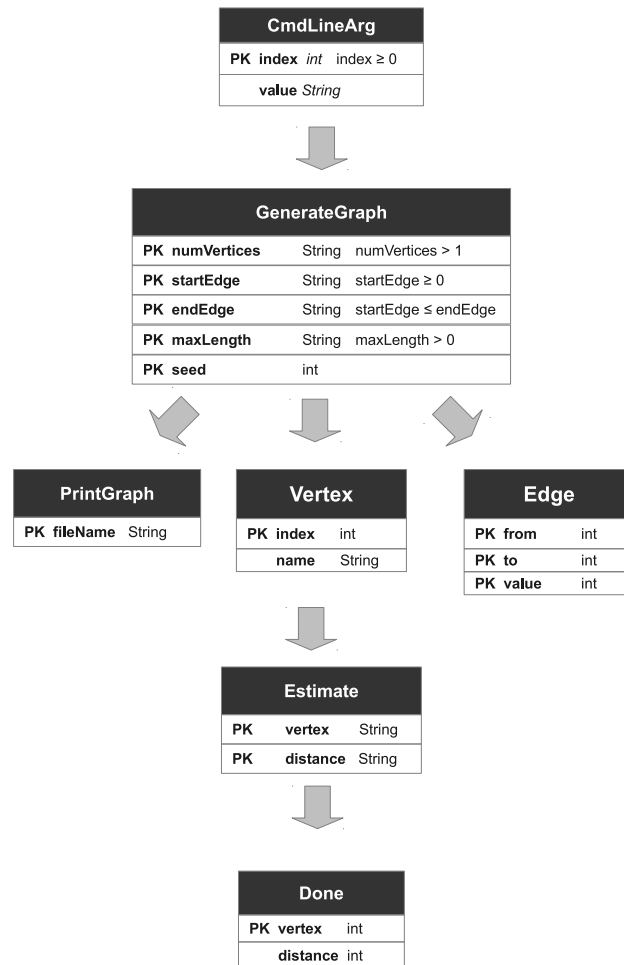


Figure 7.2: Table schema of the JStar Dijkstra program.

The table schema of the JStar Dijkstra program is shown in Figure 7.2. The **CmdLineArg** tuple defines the graph size and creates the **GenerateGraph** tuples, generating the random connected graph with one million vertices and two million edges. The **Vertex** and the **Edge** tables store the vertices and the edges respectively, and are read-only during the shortest path phase. Each **Done** tuple stores the current node for one iteration. For each current node, the **Estimate** tuple calculates the distances for all its neighboring and unvisited nodes and sets the next current node which has the lowest path cost. The **PrintGraph** tuple prints out the graph.

7.3 Benchmark Configuration

The benchmark experiments were conducted on one of computing nodes (compute-b1-002-p) in the NeSI Pan cluster. The detailed hardware specification is shown in Table 7.1(a). And the JVM Arguments for benchmarking the JStar Dijkstra program are shown in the Table 7.1(b). The standard operation procedure of job submission to the Pan cluster is described in Appendix C. To improve the performance of the Java garbage collector, we benchmarked the parallel JStar Dijkstra program with two additional JVM options: *UseCondCardMark* and *BiasedLockingStartupDelay*.

7.4 Performance Tuning Process

This section describes how we tuned the performance of the JStar Dijkstra program and implemented the efficient data structures. All the speedups are based on the average execution time of the sequential optimized JStar Dijkstra program.

NeSI compute-b1-002-p	
CPU	Intel® Xeon® CPU E5-2680 @ 2.70 GHz (total of 16 cores)
L1 cache	32K
L2 cache	256K
L3 cache	20480K
RAM	126GB RAM
Disk	200 TB shared GPFS
OS	64-Bit Linux operating system (kernel version 2.6.32-279.14.1.el6.x86_64)
JAVA	64-Bit JRE version 1.7.0_17

(a) Hardware specification

JVM options
<i>-Xmx8G</i> sets the maximum Java heap size to be 8 GB.
<i>-verbose:gc</i> enable verbose garbage collector.
<i>-Xbatch</i> stops the program while the hot spot compiler is recompiling/optimising the code.
<i>-XX:+PrintCompilation</i> prints the message when one method is compiled.[14]
<i>-XX:+PrintTenuringDistribution</i> prints the tenuring age information.[14]
<i>-XX:+PrintGCDetails</i> print messages at the garbage collection.[14]
<i>-XX:+UseCondCardMark</i> avoids the false sharing at the card tables in the garbage collection.[9]
<i>-XX:BiasedLockingStartupDelay=0</i> enables the objects in the HotSpot by default to be created with biased locking at the JVM startup.[8]

(b) Java Virtual Machine options

Table 7.1: Benchmark configuration of the JStar Dijkstra program.

7.4.1 In-lining Tuples

The task dependency of the naive JStar Dijkstra program is shown in Figure 7.3(a). The **Estimate** tuples are the only tuple kind which will trigger the other rule (Rule4), and thus the *noDelta* optimisation can be applied on all the other tables, including **CmdLineArg**, **GenerateGraph**, **Vertex**, **Edge** and **Done**. The *noGamma* optimisation can also be applied to all the tables, except for the **Done** and the **Edge** tables because they will be served as the query tables for the rules in the program. The task dependency of the optimized JStar Dijkstra program is shown in Fig. 7.3 (b). Note that the **PrintGraph**

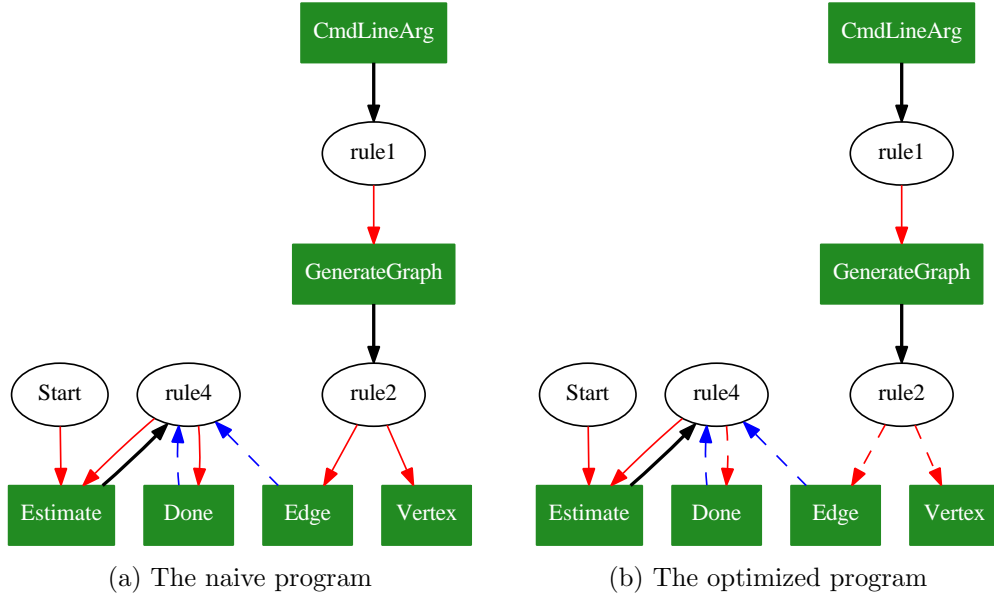


Figure 7.3: Task dependency graph of naive and optimized JStar Dijkstra programs.

tuples are not shown in the graph as they are served as the debugger in this program. And we will ignore all the debuggers and tracers to get the *unbiased* results when benchmarking the JStar program.

Tuple	Delta Tree	Gamma Database
CmdLineArgs	✓	×
GenerateGraph	×	×
Vertex	×	×
Edge	×	✓
PrintGraph	×	×
Estimate	✓	×
Done	×	✓

Table 7.2: The inline tuple list of JStar Dijkstra program.

Table 7.2 is the tuple table list after we apply both of the *noDelta* and the *noGamma* optimisation on the JStar Dijkstra program. It shows that the optimized program will put the **Estimate** tuples into the Delta tree, and insert the **Edge** and the **Done** tuples into the tables in the Gamma database. All the other tuples will not kept in the data storage but discarded immediately after triggering their associative rules.

7.4.2 Improving the Parallelism

The JStar Dijkstra program is composed of two phase: the graph generation and the shortest path. The graph generation phase is a bottleneck that downgrades the overall performance of the JStar Dijkstra program. By printing out the JVM information with the profile flag (*-Xprof*), We found out that the sequential naive implementation of the JStar Dijkstra program spent most of its running time generating the random graph. Thus, at this phase we parallelized the graph creation task with 24 separate **GeneratingGraph** tuples, each generating $1/24$ of the whole graph.

The shortest path phase is implemented by putting the **Estimate** tuples recursively. The **Estimate** tuple takes the current node to calculate all the path costs of its adjacent nodes. Then it chooses the node with the lowest cost to be the next starting node and then puts the current node with the path cost to the **Done** table. If the next node is not the ending node, then it puts another **Estimate** tuple to continue finding the shortest path. In this phase, the **Estimate**, **Done** and **Edge** are the only three tuple kinds which are inserted or queried during the execution of the program. According to the task dependency graph in Section 7.4.1, the parallelism of these tuples can be optimized by inlining the **Estimate** tuples in the Gamma database and omitting the insertion of the **Done** and **Edge** tuples into the Delta tree.

We applied the above two optimisation strategies on the JStar Dijkstra program and measured the speedups which are compared with the naive sequential implementation. The benchmark results in Figure 7.4 show that our strategy can improve and scale the performance with the maximum speedups of 6.37 (15 cores) that are not very scalable.

The goal of the next benchmark experiments is to find the most efficient data structures to make the optimized JStar Dijkstra program to gain the best speedup. The experiments design focuses on the data structure of the Delta tree and that of the Gamma database. The benchmark experiments measure the speedup of the optimized JStar Dijkstra program with/varying the data

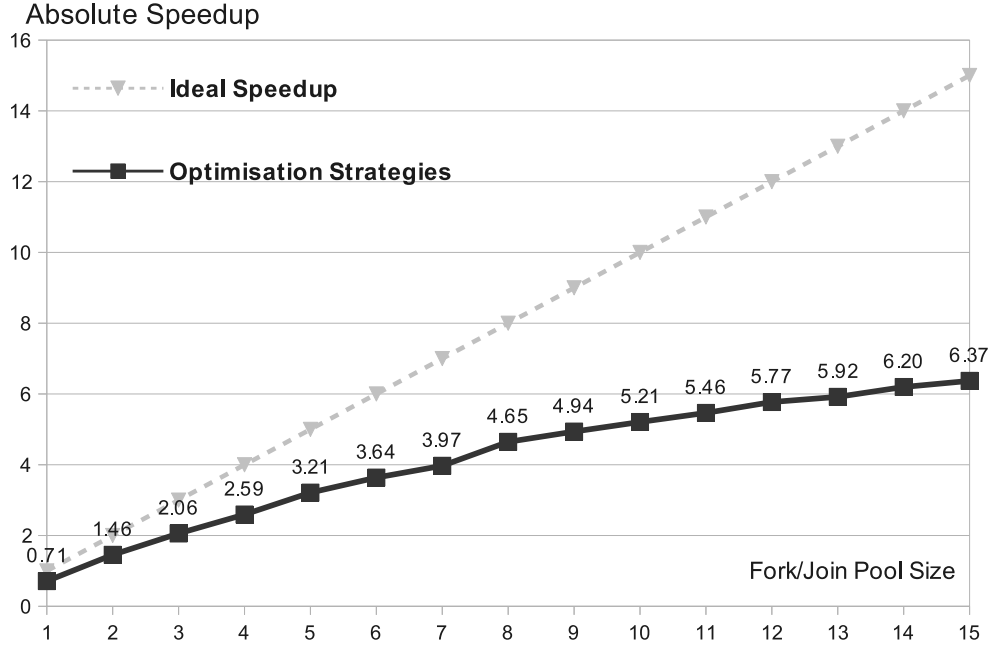


Figure 7.4: Speedups of the JStar Dijkstra program with the *noDelta* and *noGamma* optimisations on a dual-CPU Intel Xeon E5-2680 (total of 16 cores).

structures and the number of threads. The experiments are described in the following subsections:

7.4.3 Optimizing the Delta Tree Data Structures

This benchmark experiment finds the efficient data structure for the Delta tree. After optimizing the program, we found out that the `Estimate` tuple is the only one tuple kind in the Delta tree. When a `Estimate` tuple is put into the Delta tree, it will insert this tuple to the data storage of the `DeltaNodeInt` node with its *vertex* value as the key. As the number of tuples increases, the efficiency and scalability of the *DeltaNodeInt* implementations can limit the speedup of the Delta tree. As a result, we may upgrade the performance of the Delta tree by varying the data structures of the `DeltaNodeInt` nodes.

The benchmark experiment creates the integer Delta nodes with three kinds of implementations: the `DeltaNodeInt`, `ParallelDeltaNodeInt` and `ParallelDeltaNodeIntRange`. As the `DeltaNodeInt` does not support the multi-threading, its average execution time is used as the base to calculate the

absolute speedup for the other two implementations. The relative speedup is the speedup relative to the parallel JStar Dijkstra program running with one thread.

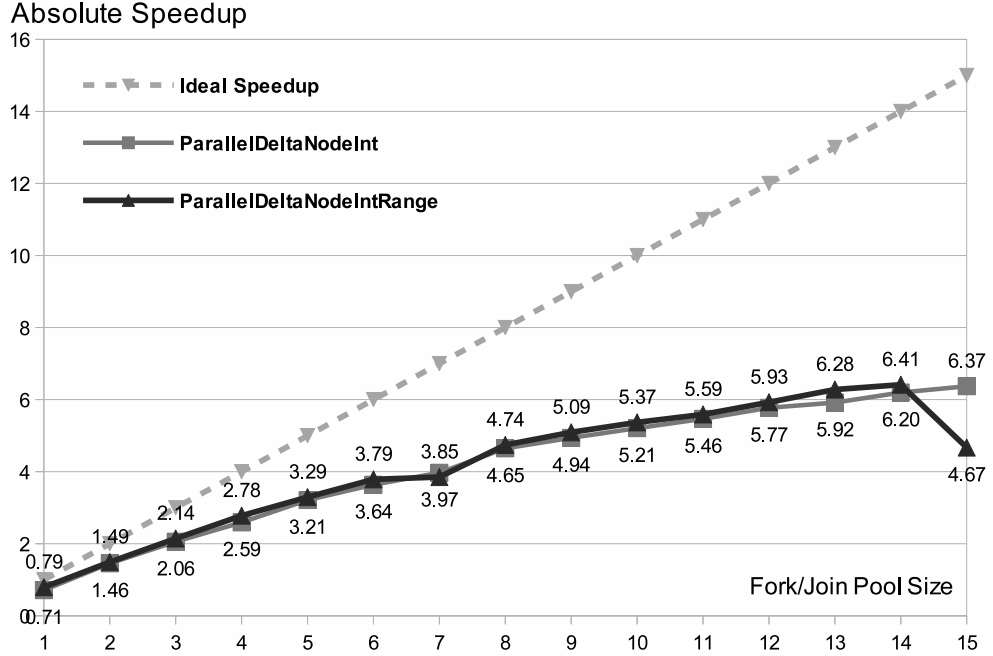


Figure 7.5: Speedups of the optimized JStar Dijkstra program with the *noDelta* and *noGamma* optimisations and varying the *DeltaNodeInt* data structure on a dual-CPU Intel Xeon E5-2680 (total of 16 cores).

Figure 7.5 is the benchmark results of the Delta tree data structures. It show that the *ParallelDeltaNodeIntRange* has a slightly good but similar speedup as the *ParallelDeltaNodeInt* with maximum absolute speedup of 6.41 (14 cores). But it has the poorer scalability and worse performance on the 15 cores with the absolute speedup of 4.67. As the *ParallelDeltaNodeInt* is the naive data structure for the Delta tree, varying the Delta tree data structures in this case does not improve the performance.

7.4.4 Optimizing the Done Gamma Table

After profiling the optimized JStar Dijkstra program, we found out that most of the time was spent on processing the queries of the *Done* and the *Edge* tuples. The *Edge* Gamma table stores the path cost for each edge in the graph. And the *Done* table stores the *visited* nodes whose shortest path has

been found. Thus, the JStar Dijkstra program will use the **Done** and **Edge** tables to estimate the shortest path for each *unvisited* node in the graph. As the graph contains a large number of nodes and edges, querying tuples from these two Gamma tables puts a heavy burden on the program.

The **Done** table contains only two integer field values, so we can simply use a one-dimensional **Array** to store the **Done** tuples in Gamma. But as the Java array is not supported by the naive Gamma table (uses the **NavigableSet**), we created an implementation to make use of our new data structure. The benchmark experiments in this subsection uses the **CHMDoneTable**, an alternative **Done** Gamma table, to improve the efficiency of the Gamma database. And optimizing the data structures for the **Edge** table will be discussed in the following subsection.

Listing 7.1: The Source Sode of **CHMDoneTable**

```

1 package jstar.examples.dijkstra;
2 import ...
3 public class CHMDoneTable extends AbstractDoneTable{
4     private int[] mGamma;
5     ...
6     public Done moveToGamma(final Done done) {
7         int vertex = done.getVertex();
8         assert vertex != Integer.MIN_VALUE; // cannot store this special value
9         mGamma[vertex] = done.getDistance();
10        return done;
11    }
12    ...
13    @Override
14    public Done queryUnique(final int vertex) {
15        int val = mGamma[vertex];
16        if (val == Integer.MIN_VALUE) {
17            throw new RuntimeException("CHMDoneTable.queryUnique(" + vertex + ")");
18        }
19        return new Done(vertex, mGamma[vertex]);
20    }
21    ...
22 }

```

The **CHMDoneTable** is the customized **Done** table to enhance the performance of the JStar Gamma database. Instead of using the Java **NavigableSet** implementation, the **CHMDoneTable** uses a one-dimensional integer array to store the **Done** tuples in Gamma. The index of this array is defined as the node number and thus the array length is equal to the total number of the nodes in the program. Each *element* is the shortest distance (from the starting

node) to its node number (the index). The source code of the `CHMDoneTable` is shown in the List 7.1. When a `Done` tuple is moved to the `CHMDoneTable`, the JStar runtime accesses this array and assigns the tuple's *distance* to the element whose index is the tuple's *vertex*. And the shortest distance of a node can be retrieved from the array by its *vertex*.

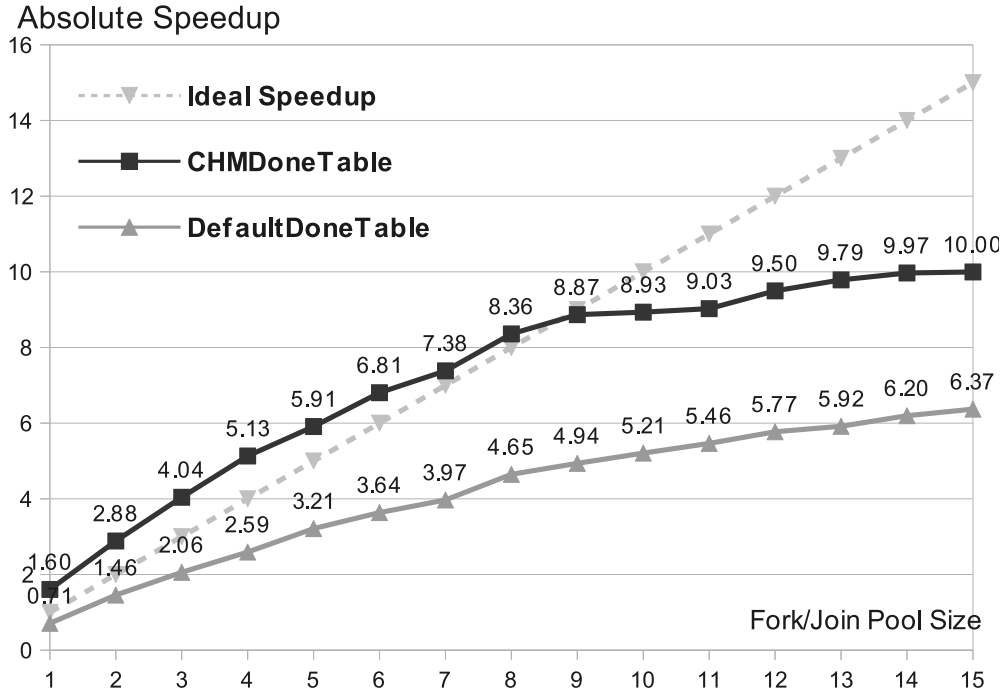


Figure 7.6: Speedups of the optimized JStar Dijkstra Program with the `CHMDoneTable` and `ParallelDeltaNodeInt` data structures on a dual-CPU Intel Xeon E5-2680 (total of 16 cores).

Figure 7.6 is the benchmark results of the `CHMDoneTable` and the naive implementation (`ConcurrentSkipListSet`). The speedup is relative to the total execution time of the sequential optimized JStar Dijkstra program. The results show that `CHMDoneTable` has the better performance than the naive one with maximum speedup of 10 (15 cores) and it also provides the scalability upto 9 threads.

7.4.5 Optimizing the Edge Gamma Table

We found out that the `Lookup` method of the `Edge Gamma` table slowed down the performance after profiling the JStar Dijkstra program. To improve the

efficiency of Edge Gamma table, we implemented the `EdgeHashTable` to make use of Java array. Because the `Edge.from` are dense ($0 \dots 999,999$) and because every query specifies `Edge.from`, we can use an array to store the `Edge.from` as the index. But we can have several `Edge` tuples with the same `from` values, so each entry in this array must be a `Set`.

This experiment creates the `Edge` table with 3 Java concurrent implementations: `ConcurrentSkipList`, `ConcurrentHashMap` and `ConcurrentHashMapV8`. Based on the previous experiments, the `ParallelDeltaNodeInt` is chosen to be data structure of the Delta tree and the `CHMDoneTable` is used to create the `Done` table in the Gamma database.

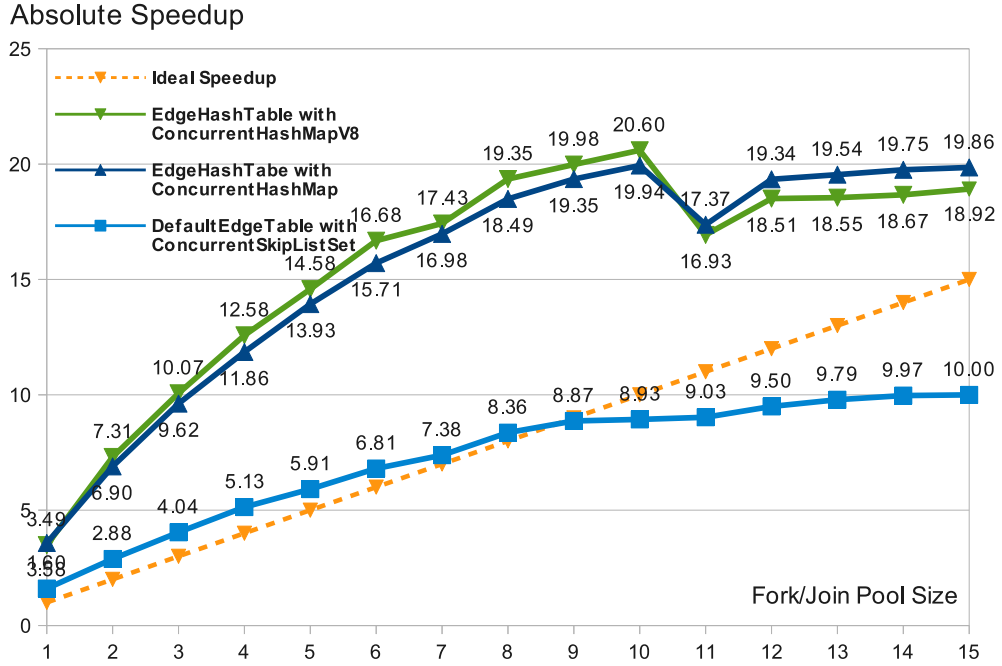


Figure 7.7: Speedups of the optimized JStar Dijkstra program with the `CHMDoneTable` and `ParallelDeltaNodeInt` and varying the Edge Gamma table data structures on a dual-CPU Intel Xeon E5-2680 (total of 16 cores).

The speedup results are shown in Figure 7.7. The speedup is the absolute speedup relative to the sequential optimized JStar Dijkstra program, *a*) inlining the tuples, *b*) using the `TreeSet` to create the `Done` and `Edge` table in Gamma, and *c*) using the `DeltaNodeInt` (`TreeSet`) to create the Delta nodes for the `Estimate` tuples. Benchmark results show that the speedups of `ConcurrentHashMap` and `ConcurrentHashMapV8` are double those of `ConcurrentSkipList`

from 1 to 10 cores. And `ConcurrentHashMapV8` has the maximum speedup of 20.6 with 10 cores while the `ConcurrentHashMap` has the best speedup of 19.94 with 10 cores. Regarding the scalability, these data structures fail to scale up the performance from 11 to 15 cores, except that `ConcurrentSkipList` slightly increases the speedups. To achieve the best performance, `ConcurrentHashMapV8` is recommended to implement the `Edge` table in the Gamma database.

7.5 Conclusion

In the Dijkstra's shortest path case study, we learned:

- By using appropriate data structures for the Gamma tables (hand-written in this case, but with the potential to be automatically generated), the JStar Dijkstra program achieves a good speedup up to 10 cores with a maximum speedup of 20.6 compared to the sequential optimized JStar Dijkstra program.
- This program has complex structures to parallelize. It is not embarrassingly parallel but Delta tree dependent because at least 2 million `Estimate` tuples must be sorted in the Delta tree.

Chapter 8

Case Study: Median-Finding

The median is the number that splits a collection of numbers into two groups: the higher group and the lower group. That is, all the numbers in the higher group are greater than or equal to the median, and the numbers in the lower group are less than the median. To find the median, we could sort the numbers in order and find the middle one. For example, the median of the 5 numbers {15, 3, 1, 12, 8} is 8 because after sorting we have {1, 3, 8, 12, 15}. And if there are an even amount of numbers, then the median is the average of the middle pair (e.g. the median of {1, 3, 8, 12, 15, 23} is 10). However, using a sequential program to sort one million numbers from the lowest values to the highest ones will take up most of the running time and block the speedup of finding the median.

Iteration	Pivot Value	Task 1	Task 2
0	—	{15, 3}	{1, 8, 12}
1	7.5	{3} {15}	{1} {8, 12}
2	11.25	{15}	{8} {12}
3	—		{8}

Table 8.1: An example of the finding-median iterative algorithm.

An iterative algorithm finds the median for a huge amount of numbers (N) by using separate and small-sized tasks, instead of one sequential task. This algorithm is described with the above example. First, we split the 5 numbers into two tasks. *Task 1* gets the first 2 numbers {15, 3} and *Task 2* gets the

next 3 numbers {8, 12, 1}. Second, in each iteration Task 1 and Task 2 both use the same *pivot value* to split their numbers into two groups: the numbers that are greater than or equal to the pivot number are put into one group, and the others are put into the other group. Each group size is reported back to the controller to determine the next pivot value, and the smaller group is discarded. Repeat this procedure on the remaining groups until the median or a pair of middle numbers is left. The iterations are shown in Table 8.1. Note that the pivot number is half of the maximal number and minimal number.

The parallel tasks have less computation as each one of them processes only one part of the whole numbers. Besides, their partition results are compared with the same pivot value at each iteration, so the final result is guaranteed to be the global median. The JStar Median-Finding program implements this iterative algorithm.

8.1 JStar Median-Finding Program

The JStar Median-Finding program finds the median of 100 million random doubles, ranging from 0 to 100 millions (the List F.1 of Appendix F is the source code). It is a typical two-phase program: generating the number and finding the median and its table schema is shown in Figure 8.1. The `CmdLineArgs` and `InitRequest` are used to generate the 100 million doubles and the rests are used for iterative median algorithm.

To ensure that all the numbers are generated in advance of finding the median, tuple orders of the `InitRequest` and `CmdLineArg` tables are declared to come before the others, including the `Int`. The `Data` table stores the pivot numbers that we need to find the median at each iteration. The `PartitionRequest` table requests the start of an iteration to split the numbers into two partitions by comparing them against the pivot number. The `PartitionResult` table stores the partition results for each iteration. The `Controller` table gets the partition results and decides the next pivot number,

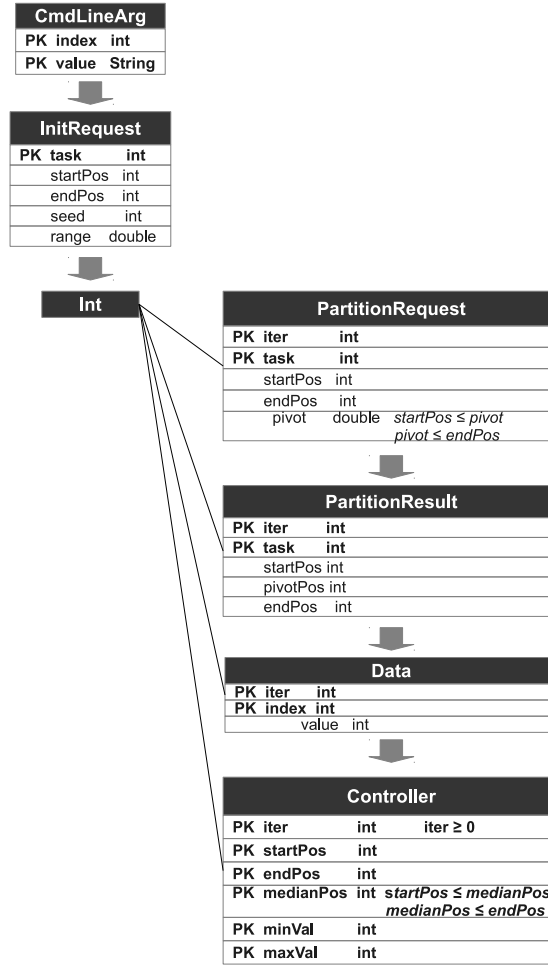


Figure 8.1: Table schema of the JStar Median-Finding program.

or stops the searching when the median is found. So the pattern of communication is essentially a master-slave situation, where the master **Controller** rule sends **PartitionRequest** tuples off to N parallel processes and then gets N **PartitionResult** tuples back and decides what to do for the next step. So this application has a lot of parallelism but is not embarrassingly parallel. since it has a central controller which can become a bottleneck.

8.2 Benchmark Configuration

The benchmark experiments were conducted on the *Gaia* computing node of the Symphony cluster in the Waikato University. Table 8.2(a) is the hardware specification of the Gaia, and Table 8.2(b) lists the JVM options used to benchmark the JStar Median program. Unlike other case studies, we increased

Symphony node CN-192	
CPU	4 x Intel® Xeon® 8 core E7- 8837 CPUs (32 cores @ 2.67GHz)
L1 cache	32K
L2 cache	256K
L3 cache	24576K
RAM	512GB RAM (@1066 MHz)
Disk	24 x 300GB 10k rpm SAS-2 hard disc 2 x 73GB 15k rpm SAS-2 hard disc
OS	64-Bit Linux operating system (kernel version 2.6.38)
JAVA	64-Bit JRE version 1.7.0_17

(a) Hardware specification

JVM options
<i>-Xmx64G</i> sets maximum Java heap size to be 64 GB.
<i>-verbose:gc</i> enables the verbose garbage collector.
<i>-Xbatch</i> stops the program while the hot spot compiler is recompiling/optimising the code.
<i>-XX:+PrintCompilation</i> prints the message when one method is compiled.[14]
<i>-XX:+PrintTenuringDistribution</i> prints the tenuring age information.[14]
<i>-XX:+PrintGCDetails</i> print messages at the garbage collection.[14]
<i>-XX:+UseCondCardMark</i> avoids the false sharing at the card tables in the garbage collection.[9]
<i>-XX:BiasedLockingStartupDelay=0</i> enables the objects in the HotSpot by default to be created with biased locking at the JVM startup.[8]

(b) Java Virtual Machine options

Table 8.2: Benchmark configuration of the JStar Median-Finding program.

the maximum heap size to 64 GB as the unoptimised program requires a large heap of memory space to store the numbers and search results.

8.3 Performance Tuning Process

8.3.1 In-lining Tuples

Figure 8.2 (a) is the task dependency graph of the naive JStar Median program. As the `Data` and `PartitionResult` tuples are never used as triggers in the program, we can use the *noDelta* optimisation to omit their Delta

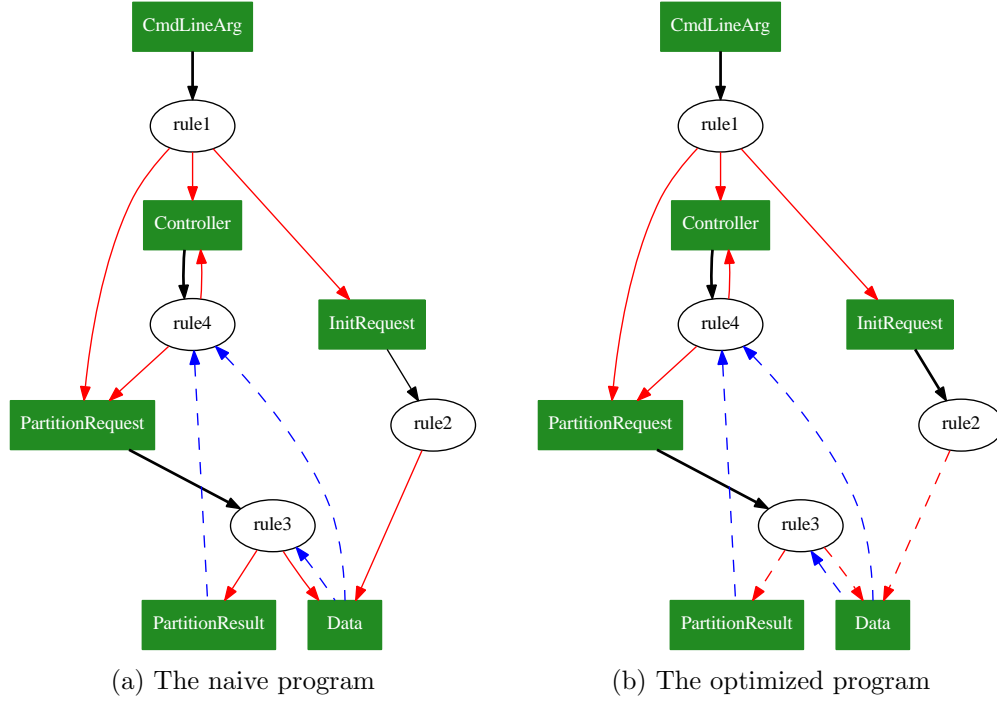


Figure 8.2: Task dependency graph of the JStar Median-Finding program.

node insertion and put their tuples straight into the Gamma database. The **CmdLineArg** and **InitRequest** tuples are used for the random number generation only, and never queried by other rules. Thus, we can apply the *noGamma* optimisation on these two tables. Similarly, the **Controller** tuple is never used by other rules and acts as a trigger, which starts an iteration to partition the tasks' numbers and stores their results. We can also apply the *noGamma* optimisation on the **Controller**. Table 8.3 is the inlined tuple list.

Tuple	Delta Tree	Gamma Database
CmdLineArg	✓	×
InitRequest	✓	×
Data	×	✓
PartitionRequest	✓	✓
PartitionResult	×	✓
Controller	✓	×

Table 8.3: The inline tuple list of JStar Median-Finding program.

The **Data** table has the most transactions in the JStar Median-Finding program. It is queried by two rules (*Rule 3* and *Rule 4*) and also accepts insertion requests from *Rule 3* and *Rule 4*. These recursive insertions and queries on the **Data** table may cause a performance issue as the naive **Data** table stores tuples with a **Set** implementation, which has log-time for most operations.[19] That is, the JStar Median-Finding program generates 100 million doubles and all of these numbers at each iteration are stored in the **Data** table, so the **Data** Gamma table might contain more than thousands of millions tuples before the median is found. This huge number of tuples makes the Java Garbage Collector busy allocating the memory space and slows down the performance. Therefore, we designed a new and efficient implementation of the **Data** Gamma table.

8.3.2 Optimizing the Data Gamma Table

The **CHMDataTable** is the alternative **Data** table in the Gamma database. It uses one two-dimensional Java array to store the **Data** tuples. Each of the tasks **T** works on one part of the numbers **N** and produces an array of the numbers that will need to find the median in the next iteration. As the **PartitionRequest** at each iteration just needs the **Data** tuples from the previous one iteration, two copies of one Java array are enough to keep all the **Data** tuples during all of the iterations. The data storage of the **Data** tuples can be reduced to a fixed-size amount by recursively overwriting this 2D array. That is, the tuples in the current iteration are placed into one Java array and the tuples in the previous iteration are put into the other. And the index of the array is determined by modulo 2 of the iteration number.

Figure 8.3 is one example that illustrates the 2D array in the **CHMDataTable**. The **Data** tuples in the 7th iteration are put in the *Iter 1* ($7\%2$) 1D array. When the program starts the 8th iteration, it looks up the numbers in the array of *Iter 1*, and then produces an array of numbers and places them in the array of *Iter 0*.

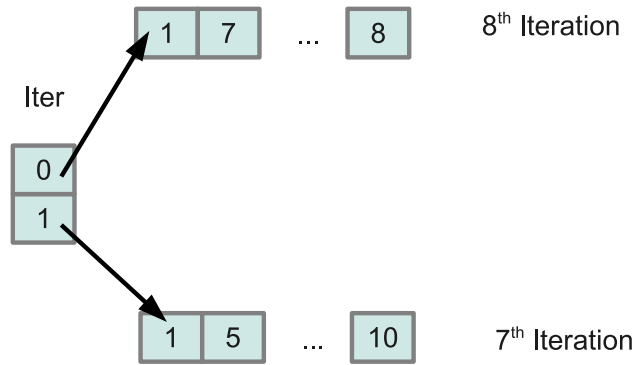


Figure 8.3: An example of the `CHMDataTable` data structure.

8.3.3 Benchmark Results

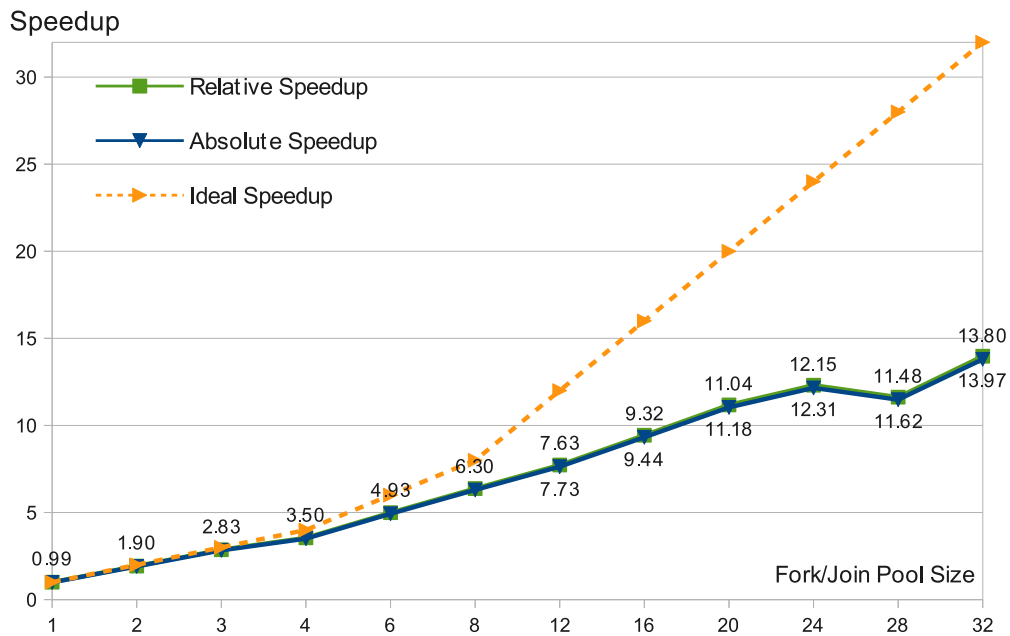


Figure 8.4: Speedups of the optimized JStar Median-Finding program with `CHMDataTable` and varying fork/join pool size on a quad-CPU Intel Xeon E7-8837 @ 2.67GHz (total of 32 cores).

Figure 8.4 shows the absolute and relative speedup graphs of the optimized JStar Median-Finding program, using the `CHMDataTable` and the naive implementations (`ConcurrentSkipListSet`) for other Gamma tables. The benchmark results show that the speedups scale well up to 8 cores with a good speedup of 6.30, and then becomes gradual with the maximum speedup of 13.8 (32 cores).

8.4 Conclusion

In the Median-Finding case study, we learned:

- The JStar parallel implementation can achieve a reasonably good speedup (with efficiency greater than 0.5 over 1 . . . 24 cores) for an algorithm with a central **Controller** bottleneck.
- For tables with billions of tuples, it is important to use the efficient Gamma data structures (e.g. Java native array) and reuse the space from previous iterations.

Chapter 9

Case Study: Matrix Multiplication

The JStar Matrix Multiplication program multiplies two matrices of the same size ($N \times N$) and produces the resulting matrix. In this test case, the size of the matrices is 1,000. Assume that we have two 1,000 – by – 1,000 matrices: matrix A and matrix B. Matrix A is an anti-diagonal matrix and Matrix B is a square matrix. The sequential matrix multiplication algorithm is to multiply one row in matrix A by one column in matrix B, and sum up these product results to get one element in the final matrix. Repeat this step until all the elements in the product matrix have been calculated. Matrix A and Matrix B are specified in the following notations:

$$A_{N,N} = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{i,1} & a_{i,2} & \cdots & a_{i,j} & \cdots & a_{i,N} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}$$
$$a_{i,j} = \begin{cases} 1 & \text{if } (i+j) = N-1 \\ 0 & \text{otherwise} \end{cases} \quad \forall i, j \in \{1, \dots, N\}$$

$$B_{N,N} = \begin{pmatrix} 0 & 1 & \cdots & b_{1,j} & \cdots & N-1 \\ 1 & 2 & \cdots & b_{2,j} & \cdots & N \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & b_{i,j} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ N-1 & N & \cdots & b_{N,j} & \cdots & 2N-2 \end{pmatrix}$$

$$b_{i,j} = (i-1) + (j-1) \quad \forall i, j \in \{1, \dots, N\}$$

The matrix multiplication of Matrix A and Matrix B are:

$$AB_{N,N} = A_{N,N} \times B_{N,N}$$

$$= \begin{pmatrix} ab_{1,1} & ab_{1,2} & \cdots & ab_{1,j} & \cdots & ab_{1,N} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ ab_{i,1} & ab_{i,2} & \cdots & ab_{i,j} & \cdots & ab_{i,N} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ ab_{N,1} & ab_{N,2} & \cdots & ab_{N,j} & \cdots & ab_{N,N} \end{pmatrix}$$

$$ab_{i,j} = \sum_{m=1}^N a_{i,m} \times b_{m,j} \quad \forall i, j \in \{1, \dots, N\}$$

$$= \begin{pmatrix} N-1 & N & N+1 & \cdots & 2N-3 & 2N-2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 2 & 3 & 4 & \cdots & N & N+1 \\ 1 & 2 & 3 & \cdots & N-1 & N \\ 0 & 1 & 2 & \cdots & N-2 & N-1 \end{pmatrix}$$

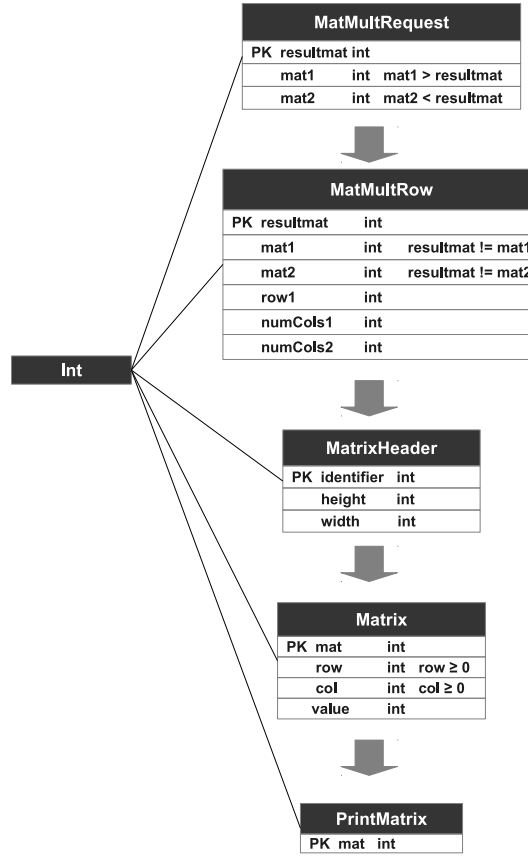


Figure 9.1: The table schema of the JStar MatrixMulti program.

9.1 JStar MatrixMult Program

The JStar MatrixMult program contains five tables: **MatMultRequest**, **MatMultRow**, **MatrixHeader**, **Matrix** and **PrintMatrix**, as shown in Figure 9.1. To effectively parallelize the matrix multiplication, the program computes each row of the product matrix separately and create the result matrix. Each row of product matrix can be conducted concurrently by taking one row from Matrix A and multiplying it with Matrix B, as described in the following formula:

$$\begin{aligned}
 AB_i = A_i \times B &= \begin{bmatrix} a_{i,1} & a_{i,2} & \cdots & a_{i,j} & \cdots & a_{i,N} \end{bmatrix} \times \begin{pmatrix} b_{1,1} & \cdots & b_{1,j} & \cdots & b_{1,N} \\ b_{2,1} & \cdots & b_{2,j} & \cdots & b_{2,N} \\ \vdots & & \vdots & & \vdots \\ b_{N,1} & \cdots & b_{N,j} & \cdots & b_{N,N} \end{pmatrix} \\
 &= \begin{bmatrix} ab_{i,1} & ab_{i,2} & \cdots & ab_{i,j} & \cdots & ab_{i,N} \end{bmatrix}
 \end{aligned}$$

The JStar MatrixMult program starts up with one `MatMultRequest` tuple, generating one `MatMultRow` tuple for each row in the product matrix. Each `MatMultRow` tuple performs one row multiplication rule which takes a row from Matrix A, loops over all columns in Matrix B and uses a nested reducer that sums up the product results for each column, and places the results in the corresponding row of the final matrix. The source code of the JStar Matrix Multiplication is shown in Listing G.1 of Appendix G. This case study was chosen because, unlike the previous case studies, it is embarrassingly parallel, so should be a good candidate for a parallel implementation with high speedup.

9.2 Benchmark Configuration

Gaia (Symphony node CN-192)	
CPU	4 x Intel® Xeon® 8core E7- 8837 CPUs (8 cores @ 2.67GHz)
L1 cache	32K
L2 cache	256K
L3 cache	24576K
RAM	512GB RAM (@1066 MHz)
Disk	24 x 300GB 10k rpm SAS-2 hard disc 2 x 73GB 15k rpm SAS-2 hard disc
OS	64-Bit Linux operating system (kernel version 2.6.38)
JAVA	64-Bit JRE version 1.7.0_17

(a) Hardware specification

JVM options
<i>-Xmx7G</i> sets maximum Java heap size to be 7 GB.
<i>-verbose:gc</i> enables the verbose garbage collector.
<i>-Xbatch</i> stops the program while the hot spot compiler is recompiling/optimising the code.
<i>-XX:+PrintCompilation</i> prints the message when one method is compiled.[14]
<i>-XX:+PrintTenuringDistribution</i> prints the tenuring age information.[14]
<i>-XX:+PrintGCDetails</i> print messages at the garbage collection.[14]

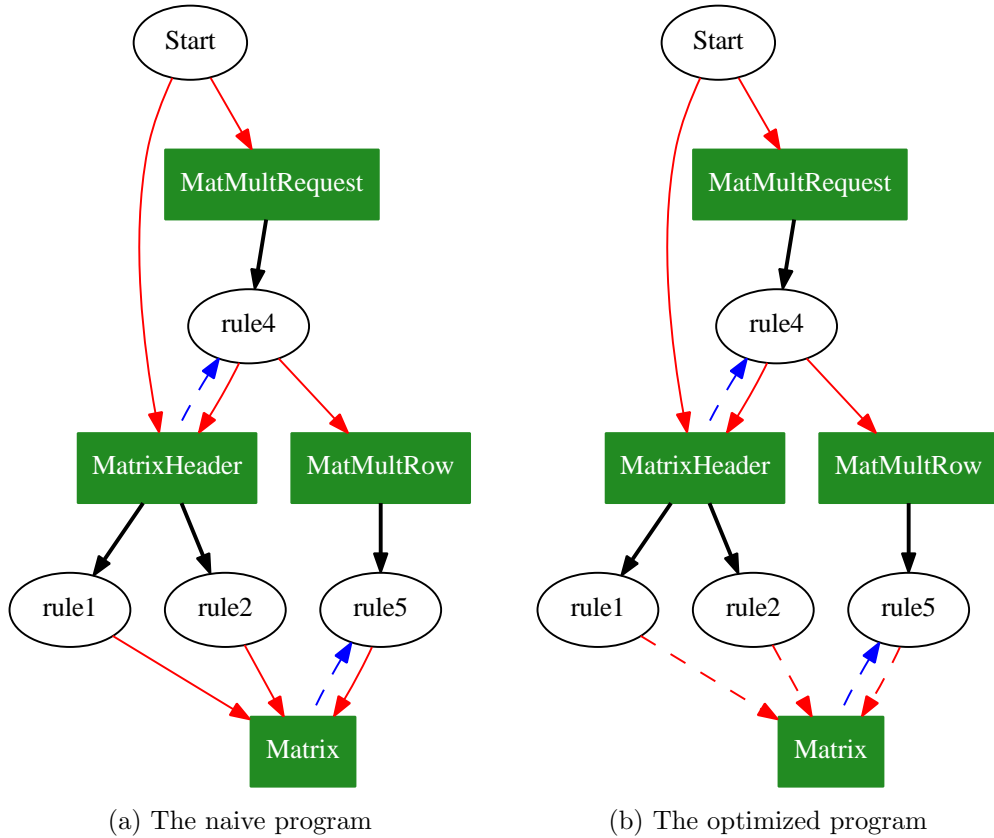
(b) Java Virtual Machine options

Table 9.1: Benchmark configuration of the JStar MatrixMult program.

Table 9.1 is the benchmark configuration for the JStar MatrixMult Program. The benchmark experiments were conducted on a 32-core machine, as shown in Figure 9.1 (a). As the MatrixMult program takes two of the same sized $(1000 - by - 1000)$ square matrices and produces another square matrix, the total amount of memory the program uses is fixed and predicable. This program does not require a large amount of heap space, but we set the maximum heap space to be 7GB to avoid garbage collection during the benchmark.

9.3 Performance Tuning Process

9.3.1 In-lining Tuples



TRIGGER(Rule): \longrightarrow PUT(Gamma Database): $\text{red} \longrightarrow$
 PUT(Delta Tree): $\text{red} \longrightarrow$ GET(Gamma Database): $\text{blue} \longrightarrow$

Figure 9.2: Task dependency graphs of the JStar MatrixMult programs.

Figure 9.2 (a) is the task dependency graph of the JStar MatrixMult program. It shows that **Matrix** tuples are never used as triggers of rules in this

Tuple	Delta Tree	Gamma Database
MatMultRequest	✓	×
MatrixHeader	✓	✓
MatMultRow	✓	×
Matrix	×	✓
PrintMatrix	×	×

Table 9.2: The inline tuple list of JStar MatrixMult program.

program and served as the query of **Rule5**. Thus, **Matrix** tuples can be put into the Gamma table directly. Besides, as the **MatMultRequest** and **MatMultRow** tuples are never queried by any rule, they can use the *noGamma* optimisation to skip the insertion of tuples from delta tree to Gamma table. Figure 9.2(b) is the task graph of the optimized JStar MatrixMult program.

The summaried tuple tables are listed in Table 9.2. The **PrintMatrix** tuples are not inserted into either the Delta tree or the Gamma database, because printing out the matrix on the terminal may bias the actual execution time. Thus, we omit the requests that prints out the whole result matrix but display the entry in the right bottom corner of the matrix to ensure the correctness of the program.

9.3.2 Optimizing the Matrix Gamma Table

Matrix3D is an customized implementation of the **Matrix** Gamma Table. The matrix basically functions like a two-dimensional array of integers, which consist of rows and columns. Thus, we implement the Gamma set of the **Matrix** table with the customized **Matrix3D** class, which uses a three-dimensional array to store all the matrices, including the Matrix A, Matrix B and Matrix AB (result matrix).

The **Matrix** table is created with 3D array of integers whose first index represents the matrix number and each element is a 2D array of the $N \times N$ matrix which the **row** and **col** indices vary from 0 to $N - 1$. When a **Matrix** tuple is moved to the Gamma Database, it assigns its value to the element by

specifying three field values, including the *Mat*, *Row* and *Col*. Similarly, to retrieve a matrix value, we can use three fields to get the entry. The source code is shown as follows:

```

1  /** Defines the contents of each matrix. */
2  package jstar.examples.matrixmult2;
3  import ...;
4  public class Matrix3D extends AbstractMatrixTable implements Table<Matrix> {
5      final int[][] data;
6      ...
7      public Matrix moveToGamma(Matrix t) {
8          data[t.getMat()][t.getRow()][t.getCol()] = t.getValue();
9          return t;
10     }
11
12     public Matrix queryUnique(int mat, int row, int col) {
13         return new Matrix(mat, row, col, data[mat][row][col]);
14     }
15     ....
16 }

```

9.3.3 Benchmark Results

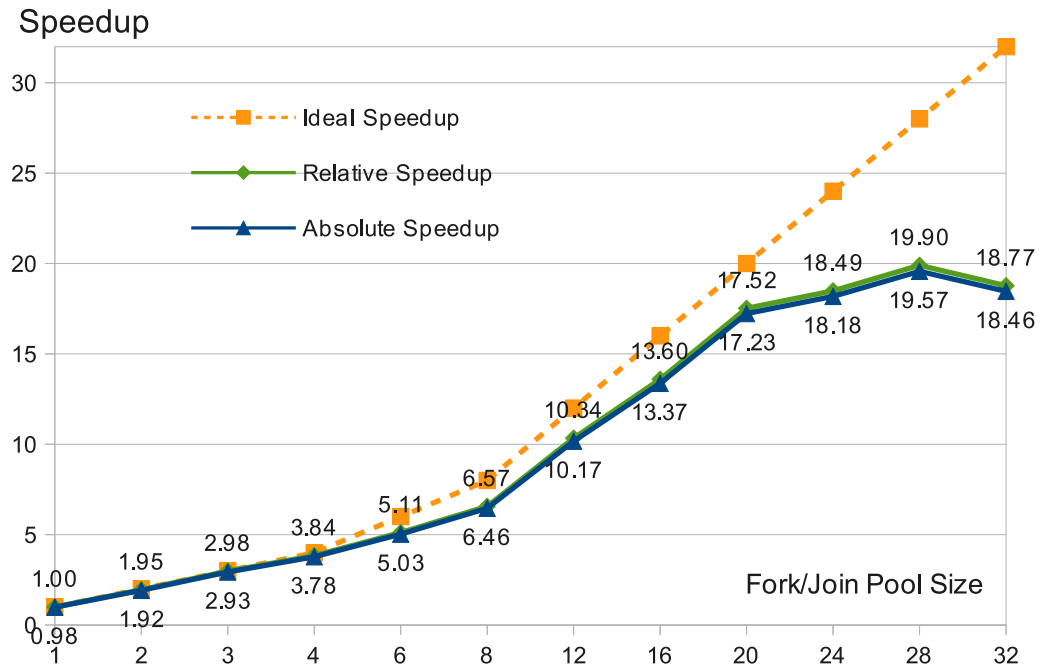


Figure 9.3: Speedups of the optimized JStar MatrixMult program with the Matrix3D and varying Fork/Join pool size on a quad-CPU Intel Xeon W5590 (total of 32 cores).

Figure 9.3 shows the speedups of the optimized JStar MatrixMult program, using **Matrix3D** and varying the Fork/Join Pool size. Benchmark results show that this program is an embarrassingly parallel program with a scalable good speedup of 17.52 (20 cores) and a maximum speedups of 19.90 (28 cores).

This high performance is due to the **Matrix3D** and the optimisation strategies. As the **Matrix3D** is a fixed-sized data storage, the JVM may use an adjacent memory space to store all the **Matrix** tuples and speed up the program. In addition, the optimisation strategy reduces the total number of tuples to 1000 in the Delta tree.

After applying the optimisations, the Delta tree are inserted with three tuple kinds, including the **MatMultRequest**, **MatrixHeader** and **MatMultRow**. There is only one **MatMultRequest** tuple and **MatrixHeader** are the request tuples which trigger the rules to generate the matrix and start the matrix multiply. The **MatMultRow** tuple is also a request to produce one row of the result matrix, and thus the total number of this tuple kind is one thousand (each row is a **MatMultRow** tuple.)

The JStar MatrixMult program would get a better speedup if the **queryUnique** method of the **Matrix3D** could be simplified to return the value, instead of the **Matrix** object. This optimisation requires several changes to the Java code generation in the JStar compiler, it has not been implemented yet.

9.4 Conclusion

In the matrix multiplication case study, we learned:

- The JStar parallel implementation can achieve a good speedup up to 20 cores with the best speedup of 19.90 (28 cores), compared to the JStar MatrixMult program with the optimisations and sequential data structures. We believe that the gradual reduction in speedup from 20-32 cores is probably due to the memory bandwidth becoming saturated by many memory-intensive tasks.

- This program is an embarrassingly parallel program (after we applied the optimisation). And it has a high computation to communication ratio because only 1,000 `MatMultRow` (per row of the result matrix) tuples need to be sorted in the Delta tree and thus result in a low latency between the tuple insertion and rule trigger.

Chapter 10

Conclusions and Future Work

We have briefly introduced the JStar in-memory data architecture and described new JStar compiler options to generate a JStar program into a naive parallel Java implementation, which uses general-purpose concurrent data structures to construct the Delta tree and the Gamma database. As the default implementation inserts every tuple in both of these two data storages, the waiting time in the Delta tree and the efficiency of the Gamma database often limits the speedup of the JStar program. Thus, we developed several inlining optimisations to reduce the total number of tuples in the Delta tree and the Gamma database. Inlining the Delta nodes (the *noDelta* optimisation) can avoid the waiting time that a tuple is queued in the Delta tree and trigger the tuple rules immediately when the tuple is created. Avoiding tuples inserting into the Gamma database (the *noGamma* optimisation) can reduce memory usage as the JVM does not need to create those tables in the Gamma database.

This thesis also defines the performance tuning process for JStar programs. By analyzing the task dependency graphs and applying the corresponding optimisation strategies, the performance of the parallel and sequential JStar program can be improved to a certain extent which is shown by the case studies. But further parallelism requires customized data structures, e.g. light-weight and fixed-sized Java arrays can be used to store the tuples in the Gamma

database in some programs. In this process, we can test the optimisation strategy or the data structure choice by following the tuning procedures without needing to change the JStar source program.

We followed the performance tuning process to improve the speedups on four case studies: *PvWatts*, *Dijkstra's shortest path*, *Median-Finding* and *Matrix Multiplication*. Their benchmark experiments were conducted on different multi-core machines as the Symphony cluster or the NeSI cluster use an automatic scheduler to dispatch the parallel tasks. Benchmark results show that three case studies have very good speedups, but the *PvWatts* case study has low speedup. These good speedups result from the combination of the optimisation strategies and the choice of efficient Gamma table data structure. But the *PvWatts* case is more complicated than the others, because it involves bulk file input communication and data summation, which may cause race hazards if the reducers perform their parallel tasks before the readers. Thus, we implemented the *Disruptor* version of PvWatts program to pipeline the readers and the reducers to get a better performance. Compared to the theoretical speedup (Amdahl's law), the Disruptor PvWatts program has a fairly good speedup and a slightly higher speedup than the standard JStar strategy of using the Delta tree to send tuples from one rule to other rules. This shows that the Disruptor ring buffer can be an efficient way of sending tuples between rules and a useful alternative to the Delta tree for some programs (e.g. when reordering of tuples is not required.) It also shows that JStar allows a wide variety of parallel implementation strategies without changing the JStar source code.

Future Work Our work in this thesis can display the data dependency explicitly and provides a way to customize the data structures used in the Delta tree or the Gamma table. Possible areas for future work include:

- allow the compiler to introduce more aggressive parallelism, for example, by parallelizing *for* loops and *for* loops with reducers.

- allow the users to view the parallelism of the program with more kinds of graphs.
- automate the generation of a wider range of the Gamma data structures, involving `Hashtable` and Java `arrays`.

After the work done in this thesis, we have shown that the JStar compiler with appropriate optimisation options can generate efficient parallel Java code with reasonably good speedup on multi-core computers. These suggestions for future work have the potential to make the tuning process easier and allow even more parallelism in programs with complex rules.

References

- [1] James Bridgewater. JStar - Logical Parallelism. ENGG482 Report, Department of Computer Science, The University of Waikato, Hamilton, New Zealand, June 2012.
- [2] Michael G. Burke, Kathleen Knobe, Ryan Newton, and Vivek Sarkar. The Concurrent Collections Programming Model. Technical Report TR 10-12, Department of Computer Science, Rice University, Houston, Texas, U.S.A, December 2010. Available from https://wiki.rice.edu/confluence/download/attachments/5210519/cnc_encyc_TR.pdf?version=1&modificationDate=1322637549211.
- [3] Bradford L. Chamberlain. A Brief Overview of Chapel. This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Defense Advanced Research Projects Agency. Available from <http://chapel.cray.com/papers/BriefOverviewChapel.pdf>, January 2013.
- [4] John G. Cleary, Mark Utting, and Roger Clayton. Datalog as a Parallel General Purpose Programming Language. Working Paper 06/2010, Department of Computer Science, The University of Waikato, Hamilton, New Zealand, June 2010. Available from <http://researchcommons.waikato.ac.nz/handle/10289/4486>.
- [5] Oracle Corporation. Java Platform, Standard Edition 7 API Specification. Specification, Oracle Corporation, May 2013. This document is the API specification for the Java Platform, Standard Edition. Available from <http://docs.oracle.com/javase/7/docs/api/>.
- [6] Simon Graeme Crosby. Parallelization of JStar Programs on a Distributed Computer. Master thesis, Department of Computer Science, The University of Waikato, Hamilton, New Zealand, May 2012. Available from <http://researchcommons.waikato.ac.nz/handle/10289/6607>.

- [7] Kaushik Datta, Dan Bonachea, and Katherine Yelick. Titanium Performance and Potential: an NPB Experimental Study. In *Proceedings of the 18th international conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 200–214, Berlin, Heidelberg, 2006. Springer-Verlag.
- [8] David Dice. Biased Locking in HotSpot. Available from https://blogs.oracle.com/dave/entry/biased_locking_in_hotspot, May 2013.
- [9] David Dice. False Sharing Induced by the Card Table Marking. Available from https://blogs.oracle.com/dave/entry/false_sharing_induced_by_card, May 2013.
- [10] Eclipse.org. Xtext 2.3 Documentation. Technical report, The Eclipse Foundation, June 2012. Available from <http://www.eclipse.org/Xtext/documentation/2.3.0/Documentation.pdf>.
- [11] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and Dynagraph Static and Dynamic Graph Drawing Tools. In *GRAPH DRAWING SOFTWARE*, pages 127–148. Springer-Verlag, 2003.
- [12] Lee HyoukJoong, K.J. Brown, A.K. Sujeeth, H. Chafi, K. Olukotun, T. Rompf, and M. Odersky. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *Micro, IEEE*, 31(5):42–53, Sep. - Oct. 2011.
- [13] Intel Inc. Intel Core i7-2600 Processor. Available from <http://ark.intel.com/products/52213>, June 2013.
- [14] Oracle Inc. Java HotSpot VM. Available from <http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html>, May 2013.
- [15] Oracle Inc. Java Platform Standard Edition - HashSet. <http://docs.oracle.com/javase/6/docs/api/java/util/HashSet.html>, May 2013.
- [16] Oracle Inc. Java Platform Standard Edition - TreeSet. <http://docs.oracle.com/javase/6/docs/api/java/util/TreeSet.html>, May 2013.
- [17] Oracle Inc. The Java Tutorial. <http://docs.oracle.com/javase/tutorial>, May 2013.
- [18] Oracle Inc. The Java Tutorial - Processes and Threads. <http://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html>, May 2013.

-
- [19] Oracle Inc. The Java Tutorial - Set Implementations. <http://docs.oracle.com/javase/tutorial/collections/implementations/set.html>, May 2013.
 - [20] Kiyokuni Kawachiya, Mikio Takeuchi, Salikh Zakirov, and Tamiya Onodera. Distributed Garbage Collection for Managed X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 5:1–5:11, New York, NY, USA, 2012. ACM.
 - [21] Doug Lea. Concurrency JSR-166 Interest Site. Available from <http://g.oswego.edu/dl/concurrency-interest/>, May 2013.
 - [22] Calvin Lin and Larry Snyder. *Principles of Parallel Programming*. Addison-Wesley Publishing Company, USA, 1st edition, 2008.
 - [23] Calvin Lin and Larry Snyder. *Principles of Parallel Programming*, chapter 3. Addison-Wesley Publishing Company, 2008.
 - [24] B. Marion, M. Anderberg, R. George, P. Gray-Hann, and D. Heimiller. PVWATTS Version 2 — Enhanced Spatial Resolution for Calculating Grid-Connected PV Performance. In *NCPV Program Review Meeting*, Colorad , U.S.A, 2001. Available from <http://www.nrel.gov/docs/fy02osti/30941.pdf>.
 - [25] Jan Obdrzalek and Mark Bull. JOMP Application Program Interface Version 0.1 (draft). Available from http://www2.epcc.ed.ac.uk/computing/research_activities/jomp/papers/jompAPI.pdf, August 2000.
 - [26] Guillermo L. Taboada, Sabela Ramos, Roberto R. Expsito, Juan Tourio, and Ramn Doallo. Java in the High Performance Computing Arena: Research, Practice and Experience. *Science of Computer Programming*, 78(0):425 – 444, 2011. Special section: Principles and Practice of Programming in Java 2009/2010 Special section: Self-Organizing Coordination.
 - [27] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. Disruptor: High Performance Alternative to Bounded Queues for Exchanging Data between Concurrent Threads. Technical paper, LMAX, May 2011. Available from <http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf>.
 - [28] Mark Utting, Min-Hsien Weng, and John G. Cleary. The JStar Language Philosophy. In *Proceedings of the 2013 International Workshop on Pro-*

gramming Models and Applications for Multicores and Manycores, PMAM '13, pages 31–41, New York, NY, USA, 2013. ACM.

Appendix A

JStar Tuple Timing Graph Installation Guide

JStar Tuple Timing Graph requires the Eclipse, JDK 1.7, and Google Apps Engine eclipse plugin for Eclipse 4.2. The instructions are:

1. Download and install JDK 1.7 and Eclipse 4.2.
2. Install **Subversion Team Provider** from Eclipse.
 - (a) From the menu bar, select Help > **Install New software**.
 - (b) Add Juno eclipse update site. Click the *Add* button. When the *Add Repository* dialog shows up, enter *Juno* at the name textarea and copy the Juno URL¹ to the location textarea.
 - (c) On **Work with:** textbox, choose *Juno*. Click **Collaboration > Subversion SVN Team Provider**. Select 'OK' on the 'Security-Warning' and 'License Agreement' windows.
 - (d) When the connector installation dialog pops up, choose **SVN Kit 1.7.5-v1** as the SVN connector. Click 'Finish' to restart Eclipse.
3. Install **Google Plugin for Eclipse** and **Google App Engine Java SDK**.
 - (a) Add 'Google plugin update site' to the repository. Enter 'Google Plugin' at the name textarea and copy the google plugin URL² to the location textarea.

¹<http://download.eclipse.org/releases/juno/>

²<http://dl.google.com/eclipse/plugin/4.2>

- (b) Choose *Google Plugin* and select **Google Plugin for Eclipse (required)** and **SDKs/Google App Engine Java SDK**. Click *Next* and review the item and agree the installation. Click *Finish* and restart the eclipse.
4. Import JStar Tuple Timing Graph project from the SVN. As the JStar Tuple Timing Graph source code are stored in the Waikato SVN repository, the developers need to email the administrator to create a Symphony account.
 - (a) Import project from the SVN repository. Start the eclipse. Select **File > Import**. After the dialog pops up, choose **SVN Project from SVN**.
 - (b) Check out the project from the SVN repository. Enter the SVN URL³ on the check-out windows and click *Next*.
 - (c) Check out as a project with name specified: **graph_tuple_times**. Click 'Finish'.
5. Run the JStar tuple timing graph tool.

The Google App engine SDK includes a web server (Jetty), so the apps developers can test the application on their local machine. We can start the server and run the application inside the Eclipse.

 - (a) Right click the **graph_tuple_times** project and choose **Run As > Web application**. The google apps web server is running on the local machine at 8888 port number.
 - (b) Start the **Chrome** browser and go to the **http://localhost:8888**.
6. Drag-and-drop the output file to the container. The graph is displayed on the HTML 5 canvas. Click 'Next' to display the next benchmark results. Click 'Previous' to show the previous one.

³https://svn.cms.waikato.ac.nz/svn/starlog/trunk/tools/trunk/graph_tuple_times

Appendix B

Symphony Benchmark S.O.P

Symphony is a computer cluster hosted by The University of Waikato. Having 92 computing nodes, the Symphony cluster enables researchers to perform a variety of computing tasks. As all the computing resources of the Symphony cluster are considered as a pool, a queue is used to accept the different requests from users. Each computing job usually defines the programs and required computing resources. Based on the cluster status, the Symphony scheduler submits the jobs to the queues and then execute them simultaneously. The following steps are used to benchmark a JStar program on the Symphony cluster.

1. Prerequisites

(a) Register as an Symphony account. As the Symphony computer nodes are Linux machines, the *SSH (Secure Shell Client)* is necessary to copy our JStar files to/from the Symphony head node.

(b) Install JDK 1.7. Running the JStar program requires Java version 1.7 or above. The installation is described as follows:

- Download the latest JDK for 64-bit linux from Oracle Java website.¹
- Unpack the tarball file to the HOME directory.²
- Add `JAVA_HOME` variables to the profile file (`/.profile`) with the following commands:

```
export PATH="/home/youraccount/jre1.7.0_09/bin:$PATH"
```

¹http://www.java.com/en/download/linux_manual.jsp

²`tar zxvf jdk-7u-version-linux-x64.tar.gz`

- Save change to the profile and run the 'source' command (`source /.profile`).

2. Compile the JStar program.

After compiling the JStar program to the Java source code, we archive these generated Java source code into a JAR file and upload the required libraries to the `lib` folder at the same directory of the Symphony head node by using the `scp` command.

3. Use the same benchmark configuration.

Reliable experimental results are critical to analyze the performance of a JStar program, so each benchmark experiment is repeated 30 times with the same JVM options to get steady results.

4. Submit the benchmark experiments to the Symphony cluster.

A shell script is employed to run all the benchmark experiments. To submit this shell script to the Symphony queue we will use the `qsub` command and specify the required resources. The following example is the configuration of the 8-core machine with 8GB of memory.

```
qsub -N PVWATTS -m abe -l walltime=03:00:00,nodes=1:ppn=8,pmem=8g
run.sh
```

5. Summarize benchmark results to a CSV file.

Benchmark results are output to the same plain text file. And we use an AWK program to read the file, extract the execution time for each benchmark and writes the time out a CSV file, sparated with Tab value.

6. Plot the speedup graph.

The CSV file is imported to the `LibreOffice Calc`. As the JVM needs the time to warm up its HotSpot, we ignore the results of initial 6 runs and average the execution times in the later runs. Then we plot the absolute and relative speedups versus the number of threads in the pool on a chart.

Appendix C

NeSI Benchmark S.O.P

NeSI (New Zealand eScience Infrastructure) provides HPC facilities to support the researches in New Zealand. We have applied for the NeSI account and conducted our benchmark experiments on the Pan cluster, which at this moment 80 16-core machines and 80 12-core machines with more than 90 GB RAM per node. The center of eResearch wiki website describes how to get started with the NeSI Pan Cluster (<https://wiki.auckland.ac.nz/display/CERES/Getting+started>). But to benchmark the JStar programs on the NeSI requires the following steps:

1. Prerequisite

- Fill out the application form on the NeSI web site (<https://www.nesi.org.nz/>).
- Apply for the research project on the NeSI Pan Cluster. By using **Puttygen** on Windows, the SSH public key and private key are generated on the local machine. And email the NeSI administrator with the username and public key for the account access. Install the **WinSCP**¹ to upload the local files to and download the files from the Pan cluster.
- Install JDK 1.7 on the home directory. Upload and uncompress the latest version of JDK 1.7 to the HOME directory.
- Set up the JAVA_HOME and PATH environment variables by editing the bash profile (**.bash_profile**) on the HOME directory with

¹<http://winscp.net/eng/index.php>

the following commands: `export JAVA_HOME=$HOME/jdk1.7.0_17`
`PATH=$JAVA_HOME/bin:$PATH:$HOME/bin`

- Open the Putty to activate the new settings and verify the java with the following command:

```
$ . ~/.bash_profile
$ which java
```

2. Run the Hello World LoadLeveler job, printing out the “Hello World” and the java version.

- Write a LoadLeveler job file. and name it as the helloworld.11.

Listing C.1: The LoadLevel Job of the Hellow World Program.

```
1 #@ shell = /bin/bash
2 #@ job_name = test
3 #@ class = default
4 #@ group = pd
5 #@ account_no = nesi00061
6 #@ wall_clock_limit = 00:01:00
7 #@ resources = ConsumableMemory(10240mb) ConsumableVirtualMemory
   (10240mb)
8 #@ job_type = serial
9 #@ output = $(home)/stdout.txt
10 #@ error = $(home)/stderr.txt
11 #@ notification = never
12 #@ queue
13 # Enforce memory constraints for jobs running on single nodes.
14 # Value is in KB
15 ulimit -v 10485760 -m 10485760
16 sleep 30
17 echo "Hello, world!"
18 java -Xmx8000m -version
```

- Submit the job file to the Pan cluster through the following commands:

```
$llsubmit helloworld.11
```

- Check the job status by using the following commands:

```
$llq -u whoami
```

- Verify the output files (`stdout.txt` and `stderr.txt`) by the Java version.

3. Upload the required libraries and the Jar file (the JStar Java source code) to the NeSI cluster.

4. Submit the LoadLeveler job file by using the `lsubmit` command.

Listing C.2: The LoadLevel Job of the JStar PvWatts Program.

```

1  #@ shell = /bin/bash
2  #@ job_name = pvwatts
3  #@ class = default
4  #@ group = pd
5  #@ account_no = nesi00061
6  #@ wall_clock_limit = 00:01:00
7  #@ resources = ConsumableMemory(10240mb) ConsumableVirtualMemory(10240
   mb)
8  #@ output = $(home)/pvwatts/stdout.txt
9  #@ error = $(home)/pvwatts/stderr.txt
10 #@ initialdir = $(home)/$(job_name)
11
12 ##@ job_type = serial
13 #@ job_type = parallel
14 #@ total_tasks = 15
15 #@ node = 1,1
16
17 #@ executable = $(home)/$(job_name)/run.sh
18 #@ notification = never
19 #@ queue
20 # Enforce memory constraints for jobs running on single nodes.
21 # Value is in KB
22 ulimit -v 10485760 -m 10485760

```

5. Download the benchmark results and plot the speedup graphs with the AWK program and LibreOffice Calc.

Appendix D

Case Study: PvWatts

Listing D.1: The Source Code of JStar PvWatts Program.

```
1  package jstar.examples.pvwatts;
2
3  import java.io.FileInputStream;
4  import nz.ac.waikato.fastcsv.FastCsvReader;
5  import nz.ac.waikato.fastcsv.CsvReaderTask;
6  import nz.ac.waikato.jstar.runtime.reduce.impure.Statistics;
7  import java.io.File;
8  import java.io.FileWriter;
9  import java.io.FileReader;
10 import java.io.BufferedReader;
11 import java.io.IOException;
12 import java.util.ArrayList;
13 import java.util.concurrent.RecursiveAction;
14 import nz.ac.waikato.jstar.runtime.IOHelper;
15 import jstar.examples.pvwatts.ReaderTask;
16 import nz.ac.waikato.jstar.runtime.JStarProgram;
17
18 /**
19  * This program measures the time taken to read a CSV file of hourly solar energy,
20  * and calculate the averaged monthly production.
21  *
22  *
23  * Arguments: --threads=1 --readers=1 --benchmark=12 [large1000X.csv]
24  * where:
25  *   --benchmark=<value>
26  *     set the number of repeated experiments.
27  *   --threads=<value>
28  *     set the number of threads.
29  *   --readers=<value>
30  *     set the number of parallel readers.
31  *   [path]
32  *     specify the (relative) path of input file
33  */
34
35 table CmdLineArg(int index, String value)
36     orderby (CmdLineArgs, seq index)
37     // key (index)
38     inv 0 <= index;
39
```

```

40
41 table PvWatts(
42     int year, int month, int day, String time, int watts)
43     orderby (PvWatts)
44     //orderby (Int, seq year, Int, seq month, Int, seq day, Int, seq time, PvWatts)
45     inv 1000 < year && 1 <= month && month <= 12 && 1 <= day && day <= 31;
46
47 table PvWattsRequest(String filename) orderby (PvWattsRequest);
48 table PvWattsException(String filename, String message) orderby(PvWattsException)
49     ;
49 table SumMonth(int year, int month) orderby (SumMonth);
50 order CmdLineArgs < PvWattsRequest < PvWattsException < PvWatts < SumMonth
51
52 foreach (PvWattsRequest req) {
53     val startTime = System::currentTimeMillis();
54     val arg = get uniq? CmdLineArg([value.startsWith("--readers=")])
55     val numReaders = if (arg == null) 1 else {
56         val pos = arg.value.indexOf("=")
57         val numStr = arg.value.substring(pos + 1);
58         Integer::parseInt(numStr)
59     }
60     unsafe {
61         try {
62             println("opening " + req.filename + " with " + numReaders + " reader tasks
63                 .")
64             CsvReaderTask::parallelRead(req.filename, null, null, numReaders)
65                 [csv |
66                     val year = csv.getIntField(0, 0);
67                     val month = csv.getIntField(1, 0);
68                     val day = csv.getIntField(2, 0);
69                     //remove the blank space from the time string
70                     val time = csv.getStringField(3).trim();
71                     val watts = csv.getIntField(4, 0);
72                     put new PvWatts(year, month, day, time, watts);
73                 ]
74             // Alternative code for multiple readers.
75             // val length = new File(req.filename).length;
76             // var pos = 0L;
77             // val readers = new ArrayList<ReaderTask>();
78             // for (r : 0 .. numReaders-1) {
79             //     val endPos = pos + length / numReaders + 2;
80             //     val r1 = new ReaderTask(req.filename, pos, endPos, null);
81             //     // r1.setDelta((this_program).deltaTree) // add tuples to delta tree.
82             //     r1.setProgram(this_program) // skip the delta tree.
83             //     readers.add(r1);
84             //     if (r > 0) {
85             //         r1.fork();
86             //     }
87             //     println("forking reader " + r + ": " + pos + " .. " + endPos)
88             //     pos = endPos;
89             // }
90             // readers.^get(0).invoke(); // ask this task to do it.
91             // (this_program).forkJoinPool.invokeAll(readers);
92             // for (r : 0 .. numReaders-1) {
93             //     // we call join on reader 0 too, because the IntegerRange cannot cope with empty
94             //     // ranges!
95             //     println("join reader " + r)
96             //     readers.^get(r).join();
97             //     println("done! " + r)

```

```

97 // }
98
99 // Sequential code for reading the whole file.
100 // val istream = new FileInputStream(req.filename);
101 // val csv = new FastCsvReader(istream);
102 // while (csv.readNextRecord()) {
103 // if (!csv.commentLine && !csv.getStringField(0).equals("Year")) {
104 // put new PvWatts() [
105 // year = csv.getIntField(0, 0);
106 // month = csv.getIntField(1, 0);
107 // day = csv.getIntField(2, 0);
108 // //remove the blank space from the time string
109 // time = csv.getStringField(3).trim();
110 // //time = csv.getStringField(3);
111 // watts = csv.getIntField(4, 0);
112 // ]
113 // }
114 // }
115 // istream.close();
116     } catch (java.io.IOException ex) {
117         put new PvWattsException(req.filename, ex.message)
118     }
119 }
120
121 val endTime = System.currentTimeMillis();
122 println("Reading time: " + (endTime - startTime)/1000.0 + " s.");
123 }
124
125 foreach (CmdLineArg arg) {
126     println("arg=" + arg.value)
127     if(!arg.value.contains("--")) {
128         put new PvWattsRequest(arg.value);
129     }
130 }
131
132 foreach (PvWattsException ex) {
133     println("Exception: " + ex.message)
134 }
135
136
137 foreach (PvWatts pv) {
138     put new SumMonth(pv.year, pv.month)
139 }
140
141 foreach (SumMonth s) {
142     val stats = new Statistics();
143     for (record : get PvWatts(s.year, s.month)) {
144         stats += record.watts
145     }
146
147     //Print out the valid results.
148     if(s.year > 0){
149         println(" " + s.year + "/" + s.month + ": " + stats.mean);
150     }
151
152
153 }

```

Listing D.2: The Output Result of JStar PVWATTS program.

```

1 [Full GC 1796706K->2588K(4213760K), 0.0680930 secs]
2 arg=-readers=8
3 arg=large1000X.csv
4 opening large1000X.csv with 8 reader tasks.
5 forking reader 0: 0 .. 24093633
6 forking reader 1: 24093633 .. 48187266
7 forking reader 2: 48187266 .. 72280899
8 forking reader 3: 72280899 .. 96374532
9 forking reader 4: 96374532 .. 120468165
10 forking reader 5: 120468165 .. 144561798
11 forking reader 6: 144561798 .. 168655431
12 forking reader 7: 168655431 .. 192749064
13 join reader 1
14 done! 1
15 join reader 2
16 done! 2
17 join reader 3
18 done! 3
19 join reader 4
20 done! 4
21 join reader 5
22 done! 5
23 join reader 6
24 done! 6
25 join reader 7
26 done! 7
27 1984/11: 264.081944444433
28 1996/4: 220.4347222222115
29 1992/12: 269.1868279569903
30 1993/6: 189.5194444444803
31 1992/7: 213.3749999999968
32 1987/10: 255.8306451612951
33 1987/5: 180.03091397849596
34 1992/3: 245.93413978494277
35 1982/2: 242.97321428571257
36 1999/9: 277.9527777777831
37 1999/1: 264.38440860215184
38 1995/8: 248.5900537634419
39 Execution time: 11.182 secs

```

Appendix E

Case Study: Dijkstra's Shortest Path Algorithm

Listing E.1: Source Code of the JStar Dijkstra Program.

```
1  package jstar.examples.dijkstra
2
3  import java.util.Random;
4  import java.util.BitSet;
5  import java.io.File;
6  import java.io.PrintWriter;
7  import java.io.IOException;
8  import java.lang.System;
9
10 /**
11  * Number of tasks to use during generation of random edges.
12  * Must be at least 2.
13  */
14 val EDGE_TASKS = 24;
15
16 /** Currently not used. */
17 val RANDOM_SEED = 16;
18
19 /** We get less parallelism as this increases. */
20 val MAX_PATH_LENGTH = 10;
21
22 /**
23  * Dijkstra is a graph search algorithm that solves the single-source shortest path
24  * problem for
25  * a graph with nonnegative edge path costs, producing a shortest path tree.
26  * (http://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm)
27  *
28  * Arguments: --threads=1 --benchmark=12 --graph=1000000,1000000
29  * where:
30  * --benchmark=<value>
31  *   set the number of repeated experiments.
32  * --threads=<value>
33  *   set the number of threads.
34  * --graph=VVV,EEE
```

```

35  *   VVV is the number of vertices, and EEE is the number of EXTRA
36  *   random edges to be added to a random tree with VVV vertices.
37  *   So the total number of edges will be VVV+EEE.
38  *
39  */
40  table CmdLineArg(int index, String value) orderby (CmdLineArgs);
41
42  /**
43  * Each one of these tuples creates endEdge – startEdge edges.
44  * The first edges (numbered less than the number of vertices)
45  * form a random tree with root vertex 0 connected to all the other vertices.
46  *
47  * The higher edges are added between random vertices.
48  *
49  * Each edge is generated with a length of between 1..maxLength.
50  */
51  table GenerateGraph(int numVertices, int startEdge, int endEdge, int maxLength, int
    seed)
52      orderby(GenerateGraph)
53      inv 1 < numVertices && 0 <= startEdge && startEdge <= endEdge && 0 <
    maxLength;
54
55  table Vertex(int index, String name) orderby(Vertex);
56
57  table Edge(int from, int to, int value) orderby(Edge);
58
59  /** Add one of these tuples to print the graph to a *.dot file. */
60  table PrintGraph(String fileName) orderby (PrintGraph);
61
62  /** The estimated shortest–path distance from the origin to the given vertex. */
63  table Estimate(int vertex, int distance) orderby (Int, seq distance, Estimate);
64  put new Estimate(0, 0); //Set the origin.
65
66  /** The final shortest–path distance to each node. */
67  table Done(int vertex -> int distance) orderby (Int, seq distance, Done)
68
69  order CmdLineArgs < GenerateGraph < { Vertex, Edge } < Int;
70  order GenerateGraph < PrintGraph;
71  order Estimate < Done;
72
73  /**
74  * Initialize the graph
75  *
76  *
77  * +-----+ 7 +-----+ 1  +-----+
78  * | S +----->+ B +-----+----->+ C |
79  * +-----+ +-----+ ^ +---+---+
80  * | ^ | | | ^
81  * | | | | |
82  * 2 | 3 | | 2 | 8 4 | | 5
83  * | | | | |
84  * | | V | V |
85  * | +---+---+---+ | +---+---+---+
86  * +----->+ A +-----+----->+ D |
87  * +-----+ 5  +-----+
88  *
89  * The starting point is S, and the destination is D.
90  *
91  */
92  /**

```

```

93     put new Vertex(0,"S");
94     put new Vertex(1,"B");
95     put new Vertex(2,"A");
96     put new Vertex(3,"C");
97     put new Vertex(4,"D");
98
99     put new Edge(0,1,7);
100    put new Edge(0,2,2);
101    put new Edge(1,3,1);
102    put new Edge(1,2,2);
103    put new Edge(2,1,3);
104    put new Edge(2,3,8);
105    put new Edge(2,4,5);
106    put new Edge(3,4,4);
107    put new Edge(4,2,5);**/
108
109    foreach (CmdLineArg arg) {
110        if (arg.value.matches("--graph=[0-9]+,[0-9]+")) {
111            // val rand = new Random(Random.SEED);
112            val rand = new Random(); // no seed, so each run will be a different graph.
113            val seed = rand.nextInt();
114            val comma = arg.value.indexOf(",")
115            val vertices = Integer.parseInt(arg.value.substring(8, comma));
116            val extraEdges = Integer.parseInt(arg.value.substring(comma + 1));
117            val totalEdges = vertices + extraEdges;
118            for (task : 0 .. (EDGE_TASKS - 1)) {
119                val startEdge = totalEdges * task / EDGE_TASKS;
120                val endEdge = totalEdges * (task + 1) / EDGE_TASKS;
121                put new GenerateGraph(vertices, startEdge, endEdge,
122                                     MAX_PATH_LENGTH, seed + task);
123            }
124            // Comment out the next line when benchmarking, so we do not print the graph.
125            // put new PrintGraph(arg.value.substring(2) + ".dot");
126        }
127    }
128    foreach (GenerateGraph graph){
129        val startTime = System.currentTimeMillis();
130        val rand = new Random(graph.seed);
131        if (graph.startEdge < graph.endEdge) {
132            for (edge : graph.startEdge .. (graph.endEdge - 1)) {
133                if (edge < graph.numVertices) {
134                    //Generate the Vertex tuples
135                    put new Vertex(edge,"S"+edge);
136                    if (edge > 0) {
137                        // These edges form a random tree that spans all vertices, with node
138                        // 0 at the root.
139                        val fromVertex = rand.nextInt(edge); // from 0 .. edge-1
140                        val len = 1 + rand.nextInt(graph.maxLength);
141                        put new Edge(fromVertex, edge, len);
142                    }
143                } else {
144                    val fromVertex = rand.nextInt(graph.numVertices);
145                    val toVertex = rand.nextInt(graph.numVertices);
146                    val len = 1 + rand.nextInt(graph.maxLength);
147                    put new Edge(fromVertex, toVertex, len);
148                }
149            }
150            val endTime = System.currentTimeMillis();

```

```

150     println("generated edges " + graph.startEdge + ".." + graph.endEdge + " time: " +
151           startTime + " .. " + endTime + " = " + (endTime - startTime));
152 }
153
154
155 foreach (PrintGraph req) {
156     // NOTE: we could print the edges in parallel if we had a better output handler.
157     // Instead, we use unsafe code to write a sequential loop and handle the IOException.
158     unsafe {
159         try {
160             val out = new PrintWriter(new File(req.fileName));
161             // Display Edge tuples in a DOT-compatible format.
162             out.write("digraph DAG {\n");
163             for (edge: get Edge()) {
164                 val fromVertex = get uniq? Vertex(edge.from);
165                 val toVertex = get uniq? Vertex(edge.to);
166                 if((fromVertex != null) && (toVertex != null)){
167                     out.write(" " + fromVertex.name + " -> " + toVertex.name + " [label=\"\"
168                           + edge.value + "\"];\n");
169                 }
170             }
171             out.write("}\n");
172             out.close();
173         } catch (IOException ex) {
174             println("ERROR printing graph: " + ex)
175         }
176     }
177
178
179     /**
180     * We process the shortest-distance Dist tuples first.
181     * (The JStar delta set does the priority queue stuff for us automatically).
182     * For each Dist tuple, we look at all adjacent vertices and update their distances.
183     */
184     foreach(Estimate dist){
185         if (get uniq? Done(dist.vertex, [distance < dist.distance]) == null) {
186             // this is the first Dist tuple for this vertex, so must be the smallest.
187             if (dist.vertex % 100000 == 0) {
188                 // we print only about 10 of the results, so that output is not the bottleneck.
189                 println("shortest path to " + dist.vertex + " is " + dist.distance);
190             }
191             put new Done(dist.vertex, dist.distance);
192             // process all adjacent nodes that are not already finished.
193             //for (edge : get Edge([from == dist.vertex])) {
194             //The from field is the first field, so it is set to be indexed key by default.
195             for (edge : get Edge(dist.vertex)) {
196                 // Note: this if test is really just a minor optimisation to reduce the
197                 // number of Dist tuples going through the delta set. It might be better
198                 // to remove it (to avoid querying Done twice).
199                 if (get uniq? Done(edge.to) == null) {
200                     // println(" neighbour " + edge.to + " ... " + edge.value)
201                     put new Estimate(edge.to, dist.distance + edge.value)
202                 }
203             }
204         }
205     }

```

Listing E.2: Output of the JStar Dijkstra Program.

```

1  gamma DefaultCmdLineArgTable for CmdLineArg
2  gamma DefaultGenerateGraphTable for GenerateGraph
3  gamma DefaultVertexTable for Vertex
4  gamma ConcurrentHashMap
5  DEBUG: created EdgeHashTable(1000000) ConcurrentHashMap in 63.377298msecs.
6  gamma DefaultPrintGraphTable for PrintGraph
7  gamma DefaultEstimateTable for Estimate
8  gamma CHMDoneTable
9  delta ParallelDeltaNodeNamed for null 0
10 delta ParallelDeltaNodeSet for CmdLineArg(0, --graph=1000000,1000000) 1
11 delta ParallelDeltaNodeInt Estimate 1
12 delta ParallelDeltaNodeSet for GenerateGraph(1000000, 0, 83333, 10, 476065950) 1
13 generated edges 0..83333 time: 1366690675749 .. 1366690675787 = 38
14 generated edges 183333..191666 time: 1366690675747 .. 1366690675793 = 46
15 generated edges 1250000..1333333 time: 1366690675748 .. 1366690675794 = 46
16 generated edges 166666..250000 time: 1366690675747 .. 1366690675795 = 48
17 generated edges 1750000..1833333 time: 1366690675749 .. 1366690675795 = 46
18 generated edges 1166666..1250000 time: 1366690675748 .. 1366690675796 = 48
19 generated edges 1000000..1083333 time: 1366690675749 .. 1366690675796 = 47
20 generated edges 333333..416666 time: 1366690675749 .. 1366690675797 = 48
21 generated edges 833333..916666 time: 1366690675747 .. 1366690675797 = 50
22 generated edges 250000..333333 time: 1366690675748 .. 1366690675798 = 50
23 generated edges 583333..666666 time: 1366690675749 .. 1366690675799 = 50
24 generated edges 500000..583333 time: 1366690675748 .. 1366690675799 = 51
25 generated edges 750000..833333 time: 1366690675748 .. 1366690675800 = 52
26 generated edges 666666..750000 time: 1366690675748 .. 1366690675800 = 52
27 generated edges 916666..1000000 time: 1366690675748 .. 1366690675801 = 53
28 generated edges 1916666..2000000 time: 1366690675787 .. 1366690675831 = 44
29 generated edges 1333333..1416666 time: 1366690675794 .. 1366690675838 = 44
30 generated edges 833333..166666 time: 1366690675793 .. 1366690675839 = 46
31 generated edges 1583333..1666666 time: 1366690675795 .. 1366690675839 = 44
32 generated edges 1083333..1166666 time: 1366690675797 .. 1366690675841 = 44
33 generated edges 1666666..1750000 time: 1366690675797 .. 1366690675841 = 44
34 generated edges 1416666..1500000 time: 1366690675796 .. 1366690675843 = 47
35 generated edges 416666..500000 time: 1366690675795 .. 1366690675843 = 48
36 generated edges 1500000..1583333 time: 1366690675796 .. 1366690675844 = 48
37 shortest path to 0 is 0
38 shortest path to 900000 is 38
39 shortest path to 100000 is 41
40 shortest path to 600000 is 43
41 shortest path to 400000 is 49
42 shortest path to 700000 is 51
43 shortest path to 300000 is 51
44 shortest path to 200000 is 51
45 shortest path to 500000 is 52
46 shortest path to 800000 is 53
47 Execution time: 0.409 secs
48 Heap
49 PSYoungGen total 2150592K, used 1287676K [0x00000000755560000, 0
   x000000007ddd80000, 0x00000000800000000)
50 eden space 2064640K, 62% used [0x00000000755560000,0x000000007a3edf290,0
   x000000007d35a0000)
51 from space 85952K, 0% used [0x000000007d35a0000,0x000000007d35a0000,0
   x000000007d8990000)
52 to space 85952K, 0% used [0x000000007d8990000,0x000000007d8990000,0
   x000000007ddd80000)
53 ParOldGen total 1376128K, used 229186K [0x00000000600000000, 0x00000000653fe0000, 0
   x00000000755560000)

```

54 object space 1376128K, 16% used [0x0000000600000000,0x000000060dfd08b8,0
 x0000000653fe0000)
55 PSPermGen total 21248K, used 5466K [0x00000005fae00000, 0x00000005fc2c0000, 0
 x0000000600000000)
56 object space 21248K, 25% used [0x00000005fae00000,0x00000005fb356970,0
 x00000005fc2c0000)

Appendix F

Case Study: Median-Finding

Listing F.1: Source Code of the JStar Median Program.

```
1  /**
2   * This program finds the median of a sequence of N input values,
3   * using an iterative parallel algorithm.
4   *
5   * The input values are partitioned between T tasks.
6   * (eg. task 0 gets the first N/T values, task 1 gets the next N/T, etc.)
7   * In each iteration, the tasks are given the same pivot value,
8   * and they all split their input values into two groups: those that
9   * are less than the pivot and those that are greater or equal to the pivot.
10  * They report the sizes of the two groups back to the central controller,
11  * which discards the group that is smaller (globally), and starts the next
12  * iteration on the remaining group.
13  *
14  * To measure the speed of effective parallel program, it should be compiled with the
15  * following options.
16  * -noDelta : Data PartitionResult
17  * -noGamma : Controller CmdLineArg InitRequest
18  *
19  * Arguments: --benchmark=12 --tasks=2
20  * where:
21  * --benchmark=<value>
22  *   set the number of repeated experiments.
23  * --tasks=<value>
24  *   set the number of parallel tasks.
25  *
26  *
27  * TODO: this currently assumes the median value is distinct from
28  * its neighbouring values (so partitioning makes progress).
29  * To relax this assumption, each partition result probably
30  * needs to return the min/max values?
31  */
32  package jstar.examples.median;
33
34  import nz.ac.waikato.jstar.runtime.reduce.impure.Sum;
35  import java.util.Random;
36  import jstar.examples.median.BitOps;
37
38  /**
```



```

39  * Total number of elements in the initial Data array.
40  */
41  val SIZE = 100 * 1000 * 1000;
42  val RANGE = SIZE as double; // element values range from 0 .. RANGE.
43
44  table CmdLineArg(int index, String value) orderby (CmdLineArgs);
45
46  /** Request initialisation of a given segment of the Data array with random values. */
47  table InitRequest(int task -> int startPos, int endPos, int seed, double range)
48      orderby (InitRequest);
49
50  /** This is the array that we are finding the median of. */
51  table Data(int iter, int index -> double value)
52      orderby (Int, seq iter, Data, seq index);
53
54  /** Tells each partition task what pivot value to use. */
55  table PartitionRequest(int iter, int task -> int startPos, int endPos, double pivot)
56      orderby (Int, seq iter, PartitionRequest);
57
58  /**
59   * The result of each partition task is the number of values
60   * on each side of the partition.
61   */
62  table PartitionResult(int iter, int task -> int startPos, int pivotPos, int endPos)
63      orderby (Int, seq iter, PartitionResult)
64      inv startPos <= pivotPos && pivotPos <= endPos
65
66  // Hmm. had to move Data from first to just before Controller, so could see final iteration
67  // result.
68  // This required changing the +1/-1 in the code to control the ordering.
69  // But we should be able to do this in the orderby expression!
70  order CmdLineArgs < InitRequest < Int;
71  order PartitionRequest < PartitionResult < Data < Controller;
72
73  /**
74   * This is the global controller of the search.
75   * The desired median is at position medianPos, while startPos..endPos is
76   * the theoretical middle region of a sorted version of the array that
77   * contains medianPos, and minVal..maxVal is the range of values in that region.
78   */
79  table Controller(int iter, int startPos, int endPos, int medianPos, double minVal,
80      double maxVal)
81      orderby (Int, seq iter, Controller)
82      inv 0 <= iter
83          && startPos <= medianPos && medianPos < endPos
84          && minVal <= maxVal;
85
86  foreach (CmdLineArg arg) {
87      if (arg.value.startsWith("--tasks=")) {
88          val tasks = Integer.parseInt(arg.value.substring(8));
89          put new Controller(1, 0, SIZE, SIZE / 2, 0, RANGE);
90
91          for (i : 0 .. (tasks - 1)) {
92              val lo = (SIZE as long) * i / tasks;
93              val hi = (SIZE as long) * (i + 1) / tasks;
94              put new InitRequest(i, lo as int, hi as int, i, RANGE as double);
95              put new PartitionRequest(1, i, lo as int, hi as int, (RANGE as double) / 2);
96          }
97      }
98  }

```

```

97 }
98
99 /** Random number generation is slow, so we do it in parallel. */
100 foreach (InitRequest req) {
101     val rand = new Random(req.seed);
102     for (i : req.startPos .. (req.endPos - 1)) {
103         val value = rand.nextDouble * req.range;
104         //To increase the speed, the message is not printed out.
105         // println("data[" + i + "] = " + value)
106         put new Data(0, i, value)
107     }
108 }
109
110 //put new Data(0, 0, 10);
111 //put new Data(0, 1, 7);
112 //put new Data(0, 2, 22);
113 //put new Data(0, 3, 2);
114 //put new Data(0, 4, 33);
115 //put new Data(0, 5, 3);
116 //put new Data(0, 6, 43);
117 //put new Data(0, 7, 8);
118 //put new Data(0, 8, 1);
119 //put new Data(0, 9, 13);
120
121 foreach (PartitionRequest pr) {
122     if (pr.startPos < pr.endPos) {
123         val lowCount = new Sum();
124         val highCount = new Sum();
125         // NOTE: we are simulating a scan here, by reading the Sum values in the loop
126         // body.
127         for (index : pr.startPos .. pr.endPos - 1) {
128             val data = get uniq Data(pr.iter - 1, index);
129             if (data.value < pr.pivot) {
130                 put new Data(pr.iter, pr.startPos + lowCount.sum, data.value)
131                 lowCount += 1;
132             } else {
133                 highCount += 1; // predecrement because endPos is exclusive
134                 put new Data(pr.iter, pr.endPos - highCount.sum, data.value)
135             }
136         }
137         if (pr.endPos - pr.startPos != lowCount.sum + highCount.sum) {
138             println("ERROR: task " + pr.task + " has low " + lowCount.sum + " high "
139                 + highCount.sum)
140         }
141         val middle = pr.startPos + lowCount.sum;
142         //println(" done " + pr.task + "\t" + pr.startPos + "\t" + middle + "\t" + pr.
143         //    endPos)
144         put new PartitionResult(pr.iter, pr.task, pr.startPos, middle, pr.endPos)
145     } else {
146         //println(" TASK " + pr.task + " has no data so stops")
147     }
148 }
149
150 foreach (Controller control) {
151     val size = control.endPos - control.startPos;
152     if (size <= 1) {
153         for (res : get PartitionResult(control.iter)) {
154             val result = get uniq Data(control.iter, res.startPos);
155             println("MEDIAN Data[" + res.startPos + "] = " + result.value)
156         }
157     }
158 }

```

```

154     } else {
155         val lowSum = new Sum();
156         val highSum = new Sum();
157         for (res : get PartitionResult(control.iter)) {
158             lowSum += res.pivotPos - res.startPos
159             highSum += res.endPos - res.pivotPos
160         }
161         val pivotValue = (control.minVal + control.maxVal) / 2;
162         val pivotPos = control.startPos + lowSum.sum;
163         if (pivotPos > control.medianPos) {
164             // work on the left (lower) partition.
165             val newPivot = (control.minVal + pivotValue) / 2;
166             // println(" go left with pivot " + newPivot)
167             for (res : get PartitionResult(control.iter)) {
168                 put new PartitionRequest(control.iter + 1, res.task, res.startPos, res.
                    pivotPos, newPivot)
169             }
170             put control.copy [iter = control.iter + 1; endPos = pivotPos; maxVal =
                pivotValue];
171         } else {
172             // work on the right (higher) partition.
173             val newPivot = (pivotValue + control.maxVal) / 2;
174             // println(" go right with pivot " + newPivot)
175             for (res : get PartitionResult(control.iter)) {
176                 put new PartitionRequest(control.iter + 1, res.task, res.pivotPos, res.
                    endPos, newPivot)
177             }
178             put control.copy [iter = control.iter + 1; startPos = pivotPos; minVal =
                pivotValue];
179         }
180     }
181 }

```

Listing F.2: Output of the JStar Median Program.

```

1 CHMDataTable
2 [Full GC (System) [PSYoungGen: 800K->0K(22367808K)] [ParOldGen: 3127076K
   ->1564576K(4865024K)] 3127876K->1564576K(27232832K) [PSPermGen: 5131K
   ->5131K(21248K)], 0.1313500 secs] [Times: user=1.24 sys=0.00, real=0.13 secs]
3 MEDIAN Data[45416813] = 5.000602297827475E7
4 Execution time: 0.707 secs
5 Heap
6 PSYoungGen total 22367808K, used 20578230K [0x00002afce6d70000, 0
   x00002b023c230000, 0x00002b023c2c0000)
7 eden space 22366592K, 92% used [0x00002afce6d70000,0x00002b01ced5d898,0
   x00002b023bfd0000)
8 from space 1216K, 0% used [0x00002b023bfd0000,0x00002b023bfd0000,0
   x00002b023c100000)
9 to space 1216K, 0% used [0x00002b023c100000,0x00002b023c100000,0
   x00002b023c230000)
10 ParOldGen total 4865024K, used 1564576K [0x00002af23c2c0000, 0x00002af3651c0000, 0
   x00002afce6d70000)
11 object space 4865024K, 32% used [0x00002af23c2c0000,0x00002af29baa8178,0
   x00002af3651c0000)
12 PSPermGen total 21248K, used 5139K [0x00002af2370c0000, 0x00002af238580000, 0
   x00002af23c2c0000)
13 object space 21248K, 24% used [0x00002af2370c0000,0x00002af2375c4c90,0
   x00002af238580000)

```

Appendix G

Case Study: Matrix Multiplication

Listing G.1: Source Code of the JStar MatrixMult Program.

```
1 package jstar.examples.matrixmult2
2
3 import nz.ac.waikato.jstar.runtime.reduce.impure.*
4
5 val SIZE = 1000; // We use SIZE x SIZE matrices
6
7 /** Defines the shape of each matrix. */
8 table MatrixHeader(int identifier -> int height, int width)
9     orderby (Int, seq identifier, MatrixHeader)
10    // key (identifier)
11    inv identifier >= 0 && width >= 0 && height >= 0;
12
13 /** Defines the contents of each matrix. */
14 table Matrix(int mat, int row, int col -> int value)
15     orderby (Int, seq mat, Matrix)
16    // key (mat, row, col)
17    inv 0 <= row && 0 <= col;
18
19 /** Request that resultmat := mat1 * mat2 be calculated. */
20 table MatMultRequest(int resultmat -> int mat1, int mat2)
21     orderby (Int, seq resultmat, MatMultRequest)
22     inv mat1 < resultmat && mat2 < resultmat;
23
24 /** Request that row1 of the result matrix be calculated. */
25 table MatMultRow(int resultmat, int mat1, int mat2, int row1 -> int numCols1, int
    numCols2)
26     orderby (Int, seq resultmat, MatMultRow)
27     inv resultmat != mat1 && resultmat != mat2;
28
29 /** Request that the given matrix be printed. */
30 table PrintMatrix(int mat)
31     orderby (Int, seq mat, PrintMatrixRequest);
32
33 order MatMultRequest < MatMultRow < MatrixHeader < Matrix <
    PrintMatrixRequest;
```

```

34
35
36 // Create some sample matrices.
37 put new MatrixHeader(0, SIZE, SIZE);
38 foreach(MatrixHeader m | m.identifier == 0) {
39     for(x : 0..(m.height - 1)) {
40         for(y : 0..(m.width - 1)) {
41             val value = if (x + y + 1 == (m.width + m.height) / 2) 1 else 0;
42             put new Matrix(m.identifier, x, y, value);
43         }
44     }
45 }
46
47 put new MatrixHeader(1, SIZE, SIZE);
48 foreach(MatrixHeader m | m.identifier == 1) {
49     for (x : 0..(m.height - 1)) {
50         for (y : 0..(m.width - 1)) {
51             put new Matrix(m.identifier, x, y, x + y);
52         }
53     }
54 }
55
56 put new MatMultRequest(2, 0, 1);
57
58 //put new PrintMatrix(0);
59 //put new PrintMatrix(1);
60 //put new PrintMatrix(2);
61
62 foreach(PrintMatrix p) {
63     //println("Finish: " + System.currentTimeMillis())
64     val mat = get uniq? MatrixHeader(p.mat);
65     if(mat != null) {
66         println("Matrix " + mat.identifier + ":" );
67         //first pass – identify the largest (characterwise) element
68         val stats = new Statistics();
69         for(row : 0..(mat.height - 1)) {
70             for(col : 0..(mat.width - 1)) {
71                 stats += (get uniq Matrix(mat.identifier, row, col)).value.toString().length;
72             }
73         }
74         val padTo = stats.maximum.intValue();
75         for(row : 0..(mat.height - 1)) {
76             val line = new StringBuilder();
77             for(col : 0..(mat.width - 1)) {
78                 val matEntry = get uniq Matrix(mat.identifier, row, col);
79                 //Java doesn't support * in format strings for some reason
80                 // So we need to interpolate it into the format string manually?
81                 // It's really silly.
82                 line.append(String::format("%s%" + padTo + "d", ' ', matEntry.value));
83             }
84             line.append("]"); //end-of-row marker
85             //insert start-of-row marker and remove
86             // extraneous delimiter at start
87             line.replace(0, 1, "[");
88             println(line);
89         }
90     }
91 }
92
93 /*

```

```

94  * Matrix multiplication 101
95  * [a b c] [g h]
96  * [d e f] * [i j]
97  * [k l]
98  *
99  * [a*g+b*i+c*k a*h+b*j+c*l]
100 * [d*g+e*i+f*k d*h+e*j+f*l]
101 *
102 * x <= matrix 1 row } these are the row/column
103 * y <= matrix 2 column } of the cell in the result
104 * z <= m1col/m2row
105 *
106 * Then we need the products of m1[x,z]*m2[z,y]
107 */
108
109 foreach (MatMultRequest r) {
110     //println("Start: " + System::currentTimeMillis())
111     val mat1 = get uniq? MatrixHeader(r.mat1);
112     val mat2 = get uniq? MatrixHeader(r.mat2);
113     if(mat1 == null || mat2 == null || mat1.width != mat2.height) {
114         println("ERROR: cannot multiply " + mat1 + " and " + mat2);
115     } else {
116         put new MatrixHeader(r.resultmat, mat1.height, mat2.width);
117         for(row : 0 .. (mat1.height - 1)) {
118             put new MatMultRow(r.resultmat, r.mat1, r.mat2, row, mat1.width,
119                               mat2.width);
120         }
121     }
122 }
123
124 /** Produce one row of the output matrix. */
125 foreach (MatMultRow req) {
126     for (col2 : 0 .. (req.numCols2 - 1)) {
127         val sum = new Sum();
128         for (j : 0 .. (req.numCols1 - 1)) {
129             // println(" m(" + req.row1 + "," + col2 + ") += m1(" + req.
130                 row1 + "," + j + ") * m2(" + j + "," + col2 + ")");
131             val v1 = get uniq Matrix(req.mat1, req.row1, j);
132             val v2 = get uniq Matrix(req.mat2, j, col2);
133             sum += v1.value * v2.value;
134         }
135         put new Matrix(req.resultmat, req.row1, col2, sum.sum);
136     }
137 }
138 // print(" " + req.row1) // just to show progress
139 }

```

Listing G.2: Ouput of the JStar MatrixMult Program.

```

1 Gamma table [] for MatrixHeader
2 Gamma table Matrix3D[3][1000][1000] for Matrix
3 Gamma table [] for MatMultRequest
4 Gamma table [] for MatMultRow
5 Gamma table [] for PrintMatrix
6 [GC
7 Desired survivor size 5177344 bytes, new threshold 1 (max 15)
8 [PSYoungGen: 436405K->768K(457216K)] 452396K->28022K(1833728K), 0.0020960
   secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
9 [Full GC (System) [PSYoungGen: 768K->0K(457216K)] [ParOldGen: 27254K->14382K
   (1376512K)] 28022K->14382K(1833728K) [PSPermGen: 5394K->5394K(21248K)],
   0.0124840 secs] [Times: user=0.04 sys=0.00, real=0.01 secs]

```

```

10 [GC
11 Desired survivor size 4980736 bytes, new threshold 1 (max 15)
12 [PSYoungGen: 456448K->544K(446336K)] 470830K->15014K(1822848K), 0.0012100
    secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
13 .....
14 [GC
15 Desired survivor size 655360 bytes, new threshold 1 (max 15)
16 [PSYoungGen: 381696K->128K(373440K)] 397726K->16174K(1749952K), 0.0012240
    secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
17 Execution time: 2.901 secs
18 Bottom right entry = Matrix(2, 999, 999, 999)
19 Heap
20 PSYoungGen total 425024K, used 98278K [0x0000000755560000, 0x0000000776ab0000, 0
    x0000000800000000)
21 eden space 424896K, 23% used [0x0000000755560000,0x000000075b541818,0
    x000000076f450000)
22 from space 128K, 75% used [0x0000000776a90000,0x0000000776aa8000,0
    x0000000776ab0000)
23 to space 896K, 0% used [0x00000007768f0000,0x00000007768f0000,0x00000007769d0000)
24 ParOldGen total 1376512K, used 15998K [0x0000000600000000, 0x0000000654040000, 0
    x0000000755560000)
25 object space 1376512K, 1% used [0x0000000600000000,0x0000000600f9fa40,0
    x0000000654040000)
26 PSPermGen total 21248K, used 5402K [0x00000005fae00000, 0x00000005fc2c0000, 0
    x0000000600000000)
27 object space 21248K, 25% used [0x00000005fae00000,0x00000005fb346870,0
    x00000005fc2c0000)

```
