

Approaches And Experiences In Multicore Software Engineering: bzip2 Refactoring

Sayan Bhattacharyya – u4594519

Comp8790

Software Engineering Project

Abstract

In this report, I explore some of the challenges involved with software engineering for multicore architectures. I discuss some background research, including three standard software engineering approaches for application parallelization. I also discuss some related work that has been done based on earlier efforts to parallelize large sequential applications for multicore platforms. I then focus on my effort to parallelize bzip2, the open source compression application. In particular, the focus is on the key issues that I faced, such as the problems with removing data dependencies in the sequential code that hinder the process of parallelism, the challenge of synchronising several parallel threads of execution and other issues. These challenges were non-trivial and I took a significant amount of time to refactor the application and fix the bugs that I detected during testing. I discuss the reasons for choosing the approach that I chose, including the choice of the design and the Application Programming Interfaces (API) that I used for implementing parallelism. I also discuss the application profiling that I did on the sequential application. This was one of the key tasks that helped me understand and re-factor the design. I also report and discuss the results obtained from the various tests that I conducted. It was found that the choice of the design resulted in an application whose performance gain over the sequential version of bzip2 improves with the number of threads of execution, although the rate of improvement slows down beyond a certain number of threads.

Table of Contents

1. Introduction	3
2. Background	3
2.1 Difference between concurrency and parallelism	4
2.2 Hardware Level Parallelism	4
2.3 Simultaneous multi-threading and multicore	5
2.4 Software design considerations on multicore and single-core platforms	5
2.5 Performance gain from parallelism	6
2.6 Contention issues	6
2.7 Programming language support for parallelism	7
2.8 Software Engineering challenges associated with Multicore Programming	8
2.9 Related work	9
2.10 Some Techniques For Multicore Software Engineering	10
3. Project Requirements	12
4. Project Timetable	13
4.1 Initial Schedule	13
4.2 Final Schedule:	14
5. Design	15
5.1 Existing design	15
5.2 Initial Naïve Approach	15
5.3 A note on gprof analysis	16
5.4 Re-factoring:	17
6. Implementation	24
6.1 Code changes in bzip2.c	24
6.2 Code changes in bzlib.c	24
6.3 Code changes in compress.c	25
6.4 Code changes in the Makefile	26
6.5 Software Engineering challenges faced	26
7. Testing and Results	28
7.1 Testing for race conditions using Sun's Performance Analyzer tool	28
7.2 Testing the performance gain as compared to the sequential version of bzip2	30
8. Discussion, Conclusion and Future Work	33
8.1 Results of the performance gain test	33
8.2 Conclusion	34
8.3 Future Work	35
9. Glossary	36
10. Appendix 1-GProf Analysis Flat Profile	37
11. Appendix 2-Race Conditions	39
12. Appendix 3-Speedup values	41
13. Bibliography	42

1. Introduction

“...when we start talking about parallelism and ease of use of truly parallel computers, we’re talking about a problem that’s as hard as any that computer science has faced. ...I would be panicked if I were in industry.” – John Hennessey, President, Stanford University (Asanovic et al., 2009) [14]

This project builds on earlier multicore software engineering research which has showed that a more coarse grained approach to parallelism through code re-factoring and applying parallel design patterns is more effective in terms of performance gain when it comes to large applications. The resultant application gives much higher speed-up (with respect to the non-parallel version) than when sequential code is parallelized by using naïve techniques and by taking the help of Application Programming Interfaces (APIs) for fine-grained parallelism. The aim of this project is to investigate experiences and methodologies in re-factoring a non-trivial existing software application for efficient implementation on multicore processors. In this project, I undertake a 'parallelization' exercise in re-factoring, detailing and analysing the experiences generated, and comparing them with available literature.

For this project, the application that I chose for the purpose of parallelization is bzip2, the Open Source compression software. My approach to parallelization was decided after research and analysis. I measured the speedup of my application by comparing the time taken to compress files with the time taken to compress the same files with the sequential bzip2 application.

In addition to the parallelization exercise described above, the scope of this project also includes an analysis of the software engineering challenges that are involved with parallelization of a large sequential application that has been written and has been optimised for sequential performance. The analysis covers the some of the possible traps that an application programmer can fall into, and some of the approaches that can be adopted to avoid such traps and pitfalls. Discussion of these issues is based on my personal experience of parallelizing bzip2 for this project. I also explore some of the available software engineering models for parallelization such as Parallel Application Design Layers (PADL).

Success in this case will be determined by the design and implementation of a scalable solution for the bzip2 parallelization problem. It will also be determined by the ability to list Software Engineering challenges faced, and Software Engineering approaches adopted to solve this problem.

2. Background

The traditional computer architecture, also known as the Von Neumann architecture, is 'sequential'. A program is stored as a set of instructions sequentially in memory and is executed one instruction at a time by the processor. As computing platforms became more advanced, computer users started to expect more performance from computers. One of the expectations was the ability to multi-task or carry out several operations at the same time.

Such expectations gave rise to the demand for more processing capability. However, it is widely accepted today that the Moore's Law does not hold any longer (Manek Dubash, 2005) [16]. To increase the processing power of a computer, the only way today is to increase the number of available processors. Multicore Computing has revolutionised this trend by allowing multiple processors on a single PC chip. Inexpensive multicore chips are pushing parallel computing into the mainstream. In 2005, affordable dual-core laptops, quad-core PCs, and eight-core servers were available on the market (V. Pankratius et. al., 2008) [2] This has enabled the average programmer to take advantage of hardware level parallelism. However, "*The emergence of inexpensive parallel computers powered by multicore chips combined with stagnating clock rates raises new challenges for software engineering.*" (V. Pankratius et. al., 2008) [2].

'Concurrency' is the primary technique that software developers adopted to design and develop resources that could multi-task and share resources. "*Concurrency allows for the most efficient use of system resources. Efficient resource utilization is the key to maximizing performance of computing systems. Unnecessarily creating dependencies on different components in the system drastically lowers overall system performance.*" (Roberts and Akhter, 2008) [4] Concurrency is essentially a software design solution for implementing algorithms that are 'parallel' because the execution can be divided into several tasks that can run concurrently.

2.1 Difference between concurrency and parallelism

Concurrency can apply to multiple tasks running in parallel on multiple hardware execution units or multiple tasks sharing a single hardware resource. In the latter case, the application developer experiences an illusion of parallelism. In reality, the single hardware unit will be able to execute only one instruction at a time and the instructions from the different tasks get "interleaved". This is not true parallel execution, since in reality only one instruction is executed at a time. Parallelism on the other hand refers to actual parallel execution of different tasks on different hardware units. "*In order to achieve parallel execution in software, hardware must provide a platform that supports the simultaneous execution of multiple threads.*" (Roberts and Akhter, 2008) [4] This is achieved by increasing the number of processors. This is actual parallelism as multiple threads run on multiple processors at the same time. However, such hardware is costlier than hardware that supports concurrency but not parallelism.

Most modern computers are Single Instruction Multiple Data (SIMD) or Multiple Instructions Multiple Data (MIMD) machines. In SIMD, all processors execute the same instruction in a given clock cycle. However, each processor operates on a different chunk of data. In MIMD, different processors execute different instructions on different data elements in one clock cycle. (Barney)[5] These models allow software developers to implement parallelism at two levels–

1. Data parallelism
2. Task parallelism

2.2 Hardware Level Parallelism

Super-scalar processors can execute multiple instructions in one clock cycle. "*In an effort to make the most efficient use of processor resources, computer architects have used instruction-level*

parallelization techniques to improve processor performance. Instruction-level parallelism (ILP), also known as dynamic, or out-of-order execution, gives the CPU the ability to reorder instructions in an optimal way to eliminate pipeline stalls. The goal of ILP is to increase the number of instructions that are executed by the processor on a single clock cycle." (Roberts and Akhter, 2008) [4] This parallelization technique is implemented by the hardware. Software developers do not need to implement any specific functionality in code to activate hardware parallelism.

2.3 Simultaneous multi-threading and multicore

In simultaneous multi-threading, a single physical processor is divided into multiple logical or virtual processors. Each virtual processor has its own CPU state and interrupt logic. However, the execution units and the cache are shared between the different virtual processors. From the software developer's perspective, there are multiple processors per physical processor. Intel refers to this as 'Hyperthreading' (Roberts and Akhter, 2008) [4]. Multicore architecture, on the other hand, uses the concept of 'Chip Multiprocessing' or CMP. In this, within a 'single' processor, there are multiple execution units. This essentially means multiple physical processors on the same die.

Hyperthreading technology "*literally interleaves the instructions in the execution pipeline. Which instructions are inserted when depends wholly on what execution resources of the processor are available at the execution time.*" (Roberts and Akhter, 2008) [4]. "*The performance benefits of HT Technology depends on how much latency hiding can occur in your application. In some applications, developers may have minimized or effectively eliminated memory latencies through cache optimizations. In this case, optimizing for HT Technology may not yield and performance gains. On the other hand, multi-core processors embed two or more independent execution cores into a single processor package. By providing multiple execution cores, each sequence of instructions, or thread, has a hardware execution environment entirely to itself. This enables each thread run in a truly parallel manner*" (Roberts and Akhter, 2008) [4].

2.4 Software design considerations on multicore and single-core platforms

Software engineers have different design consideration while writing multi-threaded applications on single core platforms compared to writing multi-threaded applications on multicore platforms. Some of them are–

- In many multicore platforms, the different cores do not share cache. Hence, if two different threads run on two different cores and they modify neighbouring memory regions, the application will not be able to get the maximum benefit from caching. This is not a problem with single-core machines, since a single cache is shared between threads.
- If the application developer decides to assign priorities to different threads such that when the higher priority thread runs, it will not get interrupted by the lower priority thread. This might be a problem in multicore machines, since the two different threads will be able to run independently on different execution units without interfering with each other. If the design of the application is based on such thread priorities, the application will run properly on single-core machines, but will show non-deterministic behaviour on multicore machines.

2.5 Performance gain from parallelism

Performance gain by parallelizing an application can be mathematically obtained by applying Amdahl's Law. The modern version of Amdahl's Law (Hill and Marty, 2008) [6] states that if you enhance the fraction f of a computation by a speedup S , the overall speedup is:

$$\text{Speedup}_{\text{enhanced}}(f,S) = 1 / \{(1-f) + f/S\}$$

For multicore systems, the equation can be written as (Barney) [5]

$$\text{Speedup} = 1 / \{S + (1 - S)/n\}$$

Where S is the time spent in executing the serial portion of the parallelized version and n is the number of processor cores.

Important points to note are:

- The maximum benefit obtained from parallelization depends on the serial part of the application.
- An increase in the number of processor cores only affects the parallel part of the code.

Later work in the field of Multicore Computing has given more accurate expressions:

"Under Amdahl's law, the speedup of a symmetric multicore chip (relative to using one single-BCE core) depends on the software fraction that is parallelizable (f), the total chip resources in BCEs (n), and the BCE resources (r) devoted to increase each core's performance. The chip uses one core to execute sequentially at performance $\text{perf}(r)$. It uses all n/r cores to execute in parallel at performance $\text{perf}(r) \times n/r$. Overall, we get:

$$\text{Speedup}_{\text{symmetric}}(f, n, r) = \frac{1}{\frac{(1-f)}{\text{perf}(r)} + \frac{f \cdot r}{\text{perf}(r) \cdot n}} \quad "$$

(Hill and Marty, 2008) [6]

2.6 Contention issues

Parallel programming is generally considered to be a non trivial task, especially when design involves issues of contention management. Some of the most important contention issues that designers of parallel programs encounter are (Hughes and Hughes) [1]:

- *Race Condition* – If two or more tasks attempt to change a shared piece of data at the same time and the final value of the data depends simply on which task gets there first, then a race condition has occurred.

- *Deadlock* – If two concurrently executing tasks have access to some shared modifiable resource, and they must wait for each other to finish using before they can access, both tasks get stalled.
- *Indefinite Postponement* – If one or more tasks are waiting for a piece of communication before they can execute and the communication never comes, comes too late, or is incomplete, then the tasks may never execute.

2.7 Programming language support for parallelism

Since the scope of this project is limited to development in C, I will limit my discussion to C and C++. C and C++ do not have 'language support' for parallelism. However, vendor Application Programming Interfaces (APIs) can be used to implement parallelism in C and C++. Some of these are:

- a) **OpenMP**- OpenMP "is an API, jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. The API supports C/C++ and Fortran on multiple architectures" (Barney) [7]. OpenMP is used to implement multi-threaded shared memory parallelism. The API consists of directives for the C/C++ compilers, library routines and environment variables. It is an explicit parallelization model, providing the programmer greater control over parallelization. OpenMP uses the Fork-Join model (Figure 1) of parallelism where there is a main master thread, which creates a team of threads to execute the section of code that needs to be parallelized and once the execution is complete, they synchronise and join with the master thread.

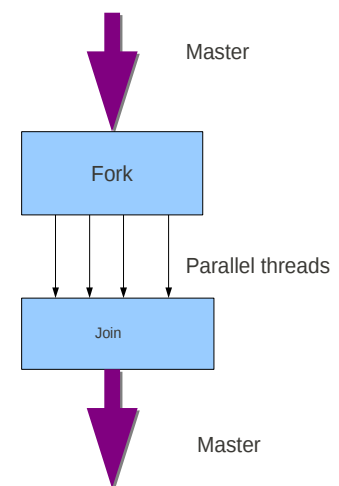


Figure 1

- b) **POSIX**- POSIX (Portable Operating Systems Interface) is a "standard that defines a standard operating systems interface and environment, including a command interpreter (or "shell") and common utility programs to support applications portability at the source code level. The standard is intended to be used by both applications developers and system implementers." (Hughes and Hughes) [1] Parallelism using POSIX can be achieved by using a process as a unit of execution (Figure 2) or a thread as a unit of execution (Figure 3). When a process is used as a unit of execution, the individual processes are mapped to the individual cores. When the thread is used as a unit of execution, the application level threads are mapped to threads in the operating system's kernel (kernel threads are also called 'Light Weight Processes'). These kernel threads are then scheduled as multiple execution units in the multicore architecture

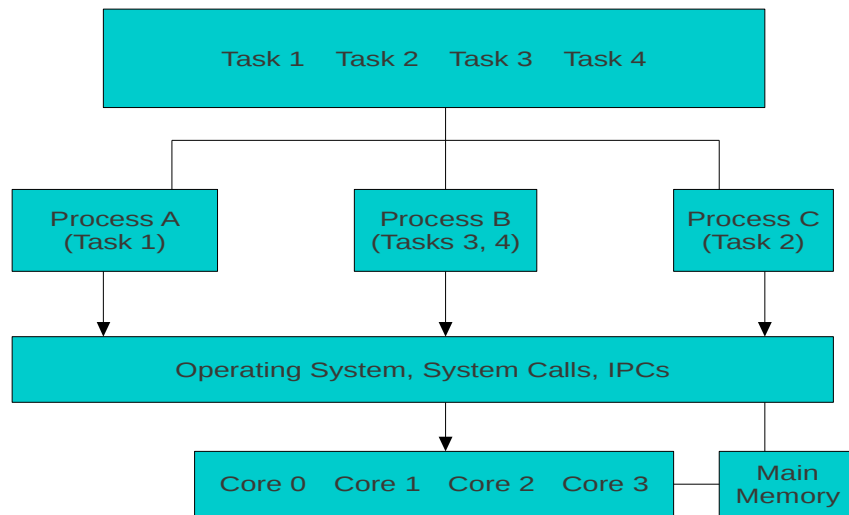


Figure 2 (Hughes and Hughes) [1]

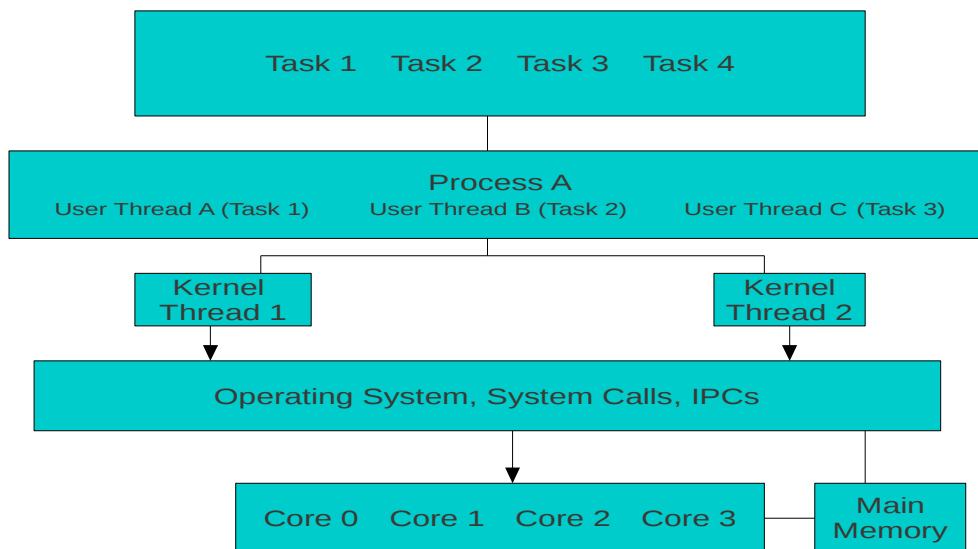


Figure 3 (Hughes and Hughes) [1]

2.8 Software Engineering challenges associated with Multicore Programming

David Geer points out some challenges (Geer, 2007) [11] associated with multicore programming:

- **Dividing activities into smaller parts** – Dividing an activity into smaller parts, such that each part can run independently on a separate core of a multicore computer is often a non

trivial exercise. Some programming activities (e.g. graphics) cannot be divided into smaller sub-functions easily without performing complex transformations.

- **Data dependency** – A certain data item that is processed by a task might be needed as input by another task running in parallel. Access to such data items have to be synchronised. If the amount of processing that is needed to synchronise is more than what parallelism saves, then it is not a good idea to parallelize.
- **Data splitting** – If a certain data set is divided into multiple parts and the different parts are sent to different cores and if the task executing on one core needs data that is on a different core, the result might not be correct.
- **Balance** – *“If one task designated to run as a separate subfunction turns out to be not all that important to the overall application, it might waste the resources of the core on which it is run.”*
- **Testing** – Parallelized applications that run on multicore machines have several execution paths. Testing all execution paths might be difficult.

2.9 Related work

Empirical findings from previous research in the field of multicore software development by Pankratius and others (Pankratius et. al., 2008) [2] indicate that:

- High speedups can be achieved through parallelization in multicore systems. However, the method that the software engineer uses to parallelize software is as important (if not more) as the choice of the right compiler or the right operating system.
- Currently, multithreading is a more popular way to achieve parallelism than multiprocessing, and many application programming languages such as C# and Java have inherent support for threads. However, it is a tedious and error prone process to manually manage multiple threads.
- Explicit thread management can be eliminated through special compiler directives such as OpenMP. OpenMP allows for incremental parallelism and threaded programs written using OpenMP are more portable. However, it does not integrate very well with the programming language so some of the error messages are not helpful. Debugging is also difficult. If future versions of existing programming languages can have in-built parallel constructs like the ones provided by OpenMP, it would be better.
- For large applications, it has been seen that code refactoring is needed to achieve a high degree of parallelism. Refactoring is also needed to ensure efficient use of thread safe libraries. Currently, almost all techniques to ensure consistent modification of data structures or to find correct access patterns are manual. There is scope for research in this field to

introduce more automation in these techniques.

- There can be different levels of parallelism within an application. Sequential applications can be parallelized at all these levels to achieve high speedup. One can have a parallel design where each unit of parallelization can have tasks that can be further parallelized.
- Parallel design patterns such as Master-Worker, Producer-Consumer, Peer-to-peer and others can be very useful for parallelizing large applications. If a pattern can be configured at design time and the right parameters can be set at run time, application tuning can be easy and different architectures can be tried.
- Autotuning is indispensable for multicore software engineering. We definitely need future work in this area, especially on intelligent heuristics that reduce the parameter space.
- If the sequential code has been written to get the maximum performance improvement, “*just exchanging library calls with parallel implementations*” and using compiler flags for auto parallelization might not yield acceptable performance improvement. (Pankratius et. al., 2008) [3]

Pankratius et. al have reported [3] the experience of groups of students who have tried to parallelize bzip2 in the University Of Karlsruhe. It was reported that the students found the exercise to be significantly difficult. Even though all members of all the groups had attended and passed a course on Multicore Software Engineering, one of the groups failed to deliver a solution with performance gain over the original sequential version of bzip2, one of them failed to deliver a bug free solution within the given time limit and one of them abandoned their initial approach and based their work on a previous version of parallel bzip2 code that was made available to them. While the paper describing this [3] makes some suggestions, it does not explain in detail the exact challenges that the students faced in an attempt to carry out the exercise. In order to get a first hand experience of the Software Engineering challenges involved, I carried out this exercise myself. The whole exercise, including the application design, code refactoring, testing, analysis of the results and documentation of the Software Engineering challenges faced is my original work.

2.10 Some Techniques For Multicore Software Engineering

These techniques can be used to incorporate parallelism when applications are designed and developed from scratch.

PADL – PADL stands for Parallel Application Design Layers. “*Starting with the top layer, each lower layer contains more detail and is one step closer to operating system and compiler primitives. PADL is meant to be used during the requirements analysis and software design activities of the SDLC.*” (Hughes and Hughes) [1]

The five layers of PADL are:

- **Layer 5** – In this layer, the application architecture is chosen. This layer is independent of design patterns, programming language, multi-threading etc. In fact, in this layer, one does not have to consider anything to do with the computer. The two main architectures that are used in PADL are Multi-agent Architecture and Blackboard Architecture. Detailed explanation of these architectures is beyond the scope of this project.
- **Layer 4** – This layer involves the choice of the correct concurrency model or design pattern. The model should be able to support the architecture in Layer 5 and vice versa. The main models that are used in this layer are:
 - ◆ Delegation Model.
 - ◆ Peer-to-Peer Model.
 - ◆ Pipeline Model.
 - ◆ Producer-Consumer Model.
- **Layer 3** – This layer involves the choice of the correct application framework, including the libraries, algorithms, and other software solutions to implement the design choices made in layers 4 and 5. This is the first layer that introduces software in the whole five layered approach.
- **Layer 2** – This layer involves the choice of the interface to the Operating Systems API. For PADL, POSIX is generally chosen as it is a standard.
- **Layer 1** – This layer involves the actual kernel execution units such as system calls, signals, drivers etc.

Par Lab solutions (Asanovic et. al., 2009) [14] – This is a project at the University of California, Berkeley. This project has several aspects, some of them are:

- Combination of Structural and Computational design patterns to produce complex architectures. These “*sit on top of the algorithmic structures and implementation structures*”.
- Splitting the application into “productivity” and “efficiency” layers. “*Productivity-layer programmers will compose libraries and programming frameworks into applications with the help of a composition and coordination language.*” The language will have inbuilt protection against races and other concurrency bugs. “*The general-purpose programmer will work largely with the frameworks and stay within what we call the productivity layer.*”
- Generation of code with search-based “autotuners” instead of compilers, since code that is automatically parallelized by compilers suffers from issues such as efficiency, scalability and portability. Autotuner code on the other hand is often faster than vendor libraries that supply manually tuned code for specific platforms.

Stanford Pervasive Parallelism Laboratory solutions (Olukotun, 2009) [15] – The Stanford Pervasive Parallelism Laboratory aims to make parallel application development practical for the average Software Engineer by 2012. They want to move away from applications developed for High Performance Scientific Computing and focus on non-scientific applications instead. They also want to focus on ease of use of the solution rather than on the efficiency of parallelization. They want to move away from approaches like automatic parallelization because of the lack of scalability and also from explicit parallelization tools such as OpenMP and Pthreads because of their 'low productivity' and the difficulty involved in finding parallelism in the application. They are focussing on **Domain Specific Languages** which increase development productivity and abstracts system information away from the programmer. Code written in these Domain Specific Languages will run on portable **Parallel Runtime** environments. The Parallel Runtime environment will interface with the underlying hardware architecture.

3. Project Requirements

The main requirements for this project are:

- Parallelization of the bzip2 compression application. *"bzip2 compresses files using the Burrows-Wheeler block-sorting text compression algorithm, and Huffman coding. Compression is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors and approaches the PPM family of statistical compressors. bzip2 is built on top of libbzip2, a flexible library for handling compressed data in the bzip2 format."* (Seward) [8]
- Testing the parallel application on Sun UltraSPARC™ T2. *"The UltraSPARC™ T2 is the industry's first "system on a chip", packing the most cores and threads of any general-purpose processor available, and integrating all the key functions of a system on a single chip: computing, networking, security and input/output (I/O), plus tight integration with the Solaris Operating System."* (Sun Microsystems) [9] This machine has 8 cores with 8 threads per core. In the student system, 7 out of the 8 cores are available for software development and testing.
- Documentation of the challenges, experiences and mistakes made for the above steps. This should explain (from personal experience) why parallelizing large applications on a multicore computer is a hard and non-trivial software engineering challenge.
- Literature survey, including exploration of previous software engineering activities related to application parallelization on multicore platforms and the lessons learnt from such activities.
- Exploration of multicore software engineering techniques such as PADL.

4. Project Timetable

The initial project schedule that I had planned for myself was highly sequential. I wanted to finish one project activity and then take up another. I had planned this for myself in Week 4. However, with greater understanding of the project, I had to alter my approach. I realised that some of the individual tasks are more challenging than what I had assumed initially, and some tasks could run in parallel, hence the sequential schedule is not very efficient.

4.1 Initial Schedule

Week 5-6

Analyse the bzip2 application – read the manual and look up the corresponding code in the code base and make notes on options to parallelize.

Week 7-8

Parallelize the code and start testing for speed-ups. Write the introduction for the report. Also note down parallelization approaches and challenges. These will be included in the final report.

Week 9-10

Fix bugs in code. Adopt alternate parallelization options if desired results are not obtained after testing.

Week 11-13

Finalise the report. Prepare for final presentation.

Week 14

Deliver the final presentation.

Two main problems with the above schedule are:

1. Writing and testing the code and fixing bugs are more time-consuming activities than what I had estimated initially. The above coding approach is naive, which I rejected later based on further analysis. Instead, I adopted a software re-factoring based approach which is more challenging and time consuming.
2. Report preparation and construction of the artefact are two tasks that can run in parallel to a certain extent. It is possible to work on the report to a greater extent in weeks 7-10 than what I had planned in the schedule above.

For these reasons, I rejected the above schedule after a few days and adopted the schedule below.

4.2 Final Schedule:

Week 5-6

Analyse the bzip2 application. Read the manual and look up the corresponding library functions to understand their behaviour. Learn gprof (Stallman and Fenlason) [10] and run the analysis tool to understand the overall performance of the tool and the amount of time taken by each function.

Week 7

Prepare a design for re-factoring the bzip2 application. Write the introduction of this report.

Week 8-10

Write the code to implement the design. Fix bugs as they come up. This essentially has two parts.

1. Re-factoring the existing code so that parallelization can be implemented.
2. Parallelize the result obtained from the above step.

Write the background, requirements and the timetable section of this report. Read about PADL and other standard Multicore Software Engineering techniques .

Week 11

Test the application and run performance analysis to explore opportunities for further parallelism. Try to implement those opportunities and fix any new existing bug that comes up.

Week 12-13

Continue with report writing and prepare final presentation. Perform more tests and document results.

Week 14

Deliver the final presentation.

5. Design

5.1 Existing design

The bzip2 application compresses files by dividing them into 'blocks' of default size 900KB. The application starts by reading the file 5000 bytes at a time. Once it reads a block full of data, it calls the compression routines which compress it by running three algorithms on it:

1. The block-sort algorithm which is also known as the Burrows-Wheeler transform (Manzini, 1999) [17].
2. The “move to front” algorithm.
3. Huffman coding.

Once this finishes, the compressed data is written to an output stream and then the next block of data is read from the file. The process is entirely sequential.

5.2 Initial Naïve Approach

The initial attempt to parallelize the application was a naïve approach. I was going through the bzip2 source code and the bzip2 programmers' manual and within the body of each function I was trying to identify loops that can be parallelized by using OpenMP directives.

There are several problems with this approach. Most of the *for* loops in the bzip2 source code are small in size. As a part of this project, I learnt and practised loop parallelization with OpenMP. The two lessons that I had learnt from the loop parallelization exercises (by logging the time taken) are:

1. If the number of iterations in the loop is small, it would mean that not much work is being done by the loop. In that case, parallelization will not yield great performance benefits.
2. If the number of iterations is small, the time overhead involved in the creation (and synchronisation at the end of all iterations) of threads can turn out to be more than the speed-up obtained from parallel execution of these threads. In such a situation, parallelization of a loop with OpenMP might actually make the loop slower than when it is allowed to run sequentially.

Also, the literature survey ([1], [2] and [3]) and careful code analysis showed that the loops themselves are not doing significant amount of work. By studying Amdahl's Law (section 2.5), I understood that the serial portion of parallel software also determines the maximum performance gain from parallelism. Hence, if the amount of work that is done by the parallel portion of the software is significantly lower than the amount of work done by the serial portion, increasing the number of cores in the hardware and the number of threads in the software will not yield satisfactory performance benefit.

Previous work (Pankratius et. al., 2008) [3] has shown that there are several **Software Engineering challenges** involved with the parallelization of such big applications. Some of them are:

- Fine granular parallelism (parallelizing code sections with OpenMP constructs etc.) might not always give acceptable speedup. Even if we want to parallelize an application using only fine granular parallelism, we have to refactor the code quite a bit.
- If we just find the critical execution path and parallelize along this path, many parallelization opportunities will not be exploited, as the critical path will change after initial parallelization.
- Coarse grain parallelism by re-factoring the code and using thread design patterns like producer-consumer or master-worker gives better results but it involves significantly more effort than fine-grained parallelism.
- If the sequential code has been written to get the maximum performance improvement, exchanging library calls with parallel implementations might not yield acceptable performance.
- It might be beneficial to look for opportunities of parallelism in the algorithm and implement them in the code by refactoring it, than to look for opportunities of parallelism directly in the code.

My code analysis (done manually) had to be verified, hence I used the **GNU profiler tool (gprof)** to profile the application and understand its performance. The gprof tool gives us a lot of useful information about the application that is being profiled, including the amount of time the application spent in a function and in the functions it called. This helped me understand two things:

1. The execution sequence of the serial application better than what I could understand by reading the source code and the programmer's manual.
2. The parts of the code that were doing maximum work and were taking the maximum amount of time.

5.3 A note on gprof analysis

I ran the gprof (GNU Profiler) tool on the bzip2 application with a sample file. For this, I had to alter the bzip2 makefile. All compilation and linking instructions had to be issued with the -pg flag. The gprof analysis data was written to a file called gmon.out when the bzip2 application (compiled with the altered makefile) was used to compress a sample file.

To carry out the analysis, I ran the following command to generate a file containing the profile analysis data:

```
gprof gmon.out > analysis
```

This command made the gprof program write the interpretation of the profile analysis data in the file called 'analysis'.

I am reproducing a section of the 'flat profile analysis table (from the 'analysis' file) which shows the running time of the different functions that get called when the bzip2 program compresses a file (please see Appendix 1 for the full table):

% time	cumulative seconds	self seconds	total calls	ms/call	ms/call	name
53.3	0.08	0.08	1	80	130	BZ2_compressBlock
33.3	0.13	0.05	1	50	50	BZ2_blockSort
13.3	0.15	0.02	35	0.57	4.29	handle_compress
0	0.15	0	3155	0	0	add_pair_to_block
0	0.15	0	35	0	4.29	BZ2_bzCompress
0	0.15	0	24	0	0	BZ2_hbMakeCodeLengt hs

This table indicates that out of the 0.15 seconds that bzip2 took to compress the file, 0.13 seconds were spent on just two functions. The 0.00s in the table do not indicate that the calls took no time, but it shows that the calls took so little time that the analyser could not measure accurately.

Form this, I could focus on a smaller subset of the code for parallelization. Since loops were not doing enough work, it was evident that parallelization had to be done at a higher level. The code needed re-factoring and rewriting according to a parallel design pattern for greater performance benefit.

5.4 Re-factoring:

Design attempt 1:

Based on the lessons learnt from the above approach and the issues involved, I initially considered a pipelining model.

"The pipeline model is characterized by an assembly-line approach in which a stream of items is processed in stages. At each stage, work is performed on a unit of input by a thread. When the unit has been through all the stages in the pipeline, then the processing of the input has been completed and exits the system. This approach allows multiple inputs to be processed simultaneously. Once data has been processed at a certain stage, it is ready to process the next data in the stream. Each thread is responsible for producing its interim results or output and making them available to the next stage in the pipeline. The last stage or thread produces the result of the pipeline." (Hughes and Hughes) [1]

The existing sequential design of bzip2 is:

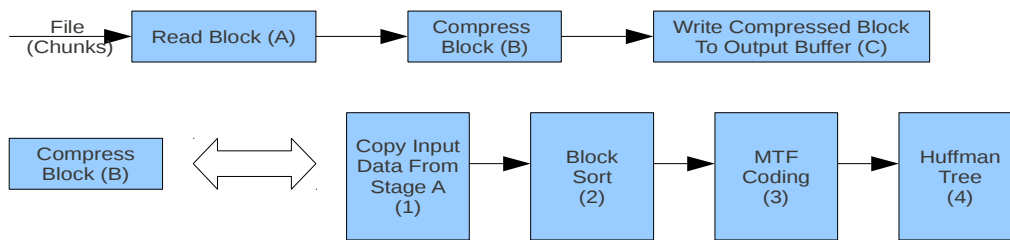


Figure 4

The input file is read in chunks of 5000 bytes till a 'block' (900 KB by default) is read. The block is then compressed and written to the output stream.

I designed the pipeline approach as:

1. Re-factor the code into several modules such that each module represents each of the stages indicated in figure 4 above.
2. Apply **Pipelining** for the stages above.
3. Use synchronisation mechanisms at stage 'n' to synchronise with stages 'n-1' and 'n+1'.
4. If beneficial, apply fine-grain parallelism techniques within each stage (loop parallelization etc.)
5. When stage C finishes its work, it signals stage A to read the next block.

Issues with design attempt 1

The main issue with this design is that because of the pipelining approach, the maximum amount of parallelization that can be achieved is 6, as there are 6 stages. This is not a very efficient model since the maximum possible performance gain is fixed and does not scale with the number of available processors or the number of threads. For this reason, I rejected this design and adopted a more scalable design.

Design attempt 2 (This was finally used):

I adopted a design pattern called the **Delegation Model**.

Delegation Model: "In the delegation model, a single thread (boss) creates other threads (workers) and assigns each a task. It may be necessary for the boss thread to wait until each worker thread completes its task before it can continue its executing its code. Its code may be based on the results of

the worker thread. The boss thread delegates the task each worker thread is to perform by specifying a function. As each worker is assigned its task, it is the responsibility of each worker thread to perform by specifying a function. As each worker is assigned its task, it is the responsibility of each worker thread to perform that task and produce output or synchronize with the boss or other thread to produce output.” (Hughes and Hughes) [1]

This is a more scalable design. In this design, the number of parallel units scale with:

- The number of available CPUs (where the Operating System scheduler can put each parallel unit on a separate core). This allows the programmer to implement more parallel units in the application to take the advantage of multiple available CPUs. Compared to this, the earlier pipelined design could have a maximum number of six parallel units as the design has six stages. This would not allow the programmer to take advantage of the fact that there are more than six available cores in many multicore architectures.
- The number of blocks, if each block is processed independently by a separate parallel unit

The boss thread creates a pool of worker threads which remain suspended until they are invoked. The basic functionality of the boss thread (Hughes and Hughes) [1] is to:

1. Create the worker threads.
2. Place the work/data to be processed in a storage location from where the worker threads and pick up the work/data.
3. Signal worker threads to do the work when there is work to be done.

The basic functionality of the worker threads (Hughes and Hughes) [1] is to:

1. See if there is work to be done.
2. Do the work.
3. Wait for the boss's signal if there is no work to be done.

A sequence diagram for task creation is shown below (note: the sequence diagram below is a simplified version of the actual design which has multiple workers. This diagram shows just one worker):

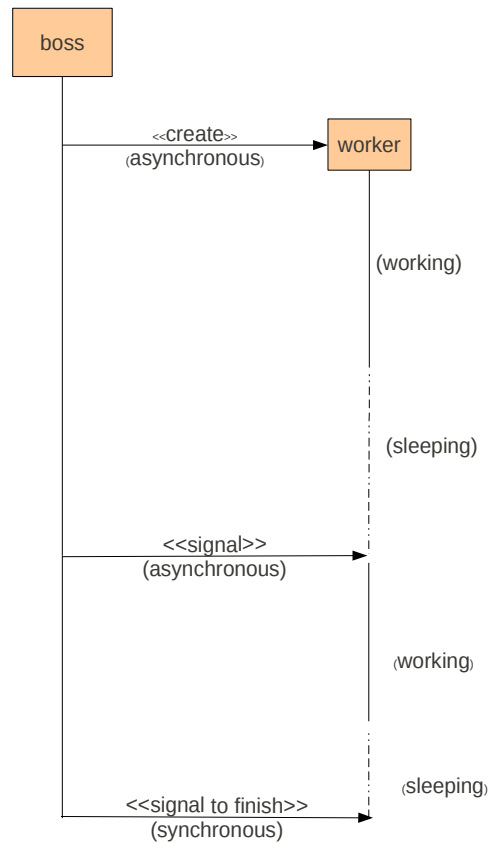


Figure 5

In this simplified diagram, the boss creates a worker and continues. When it has work for the worker, it wakes up the suspended worker with a signal. The worker finishes the task and suspends itself again. When there is no more work, the boss signals to worker to terminate and waits. Once the worker terminates, the boss terminates as well.

In reality, there can be several workers and each worker can be signalled by the boss to perform the task multiple times.

The following diagram (Figure 6) shows the overall model using threads as parallel task units. The boss thread creates a pool of worker threads. There is a global array of blocks. It writes data read from the source file in this global array of blocks. Each thread picks up an uncompressed block and processes it. Inter-thread communication mechanisms such as mutexes and condition variables are used to synchronise access.

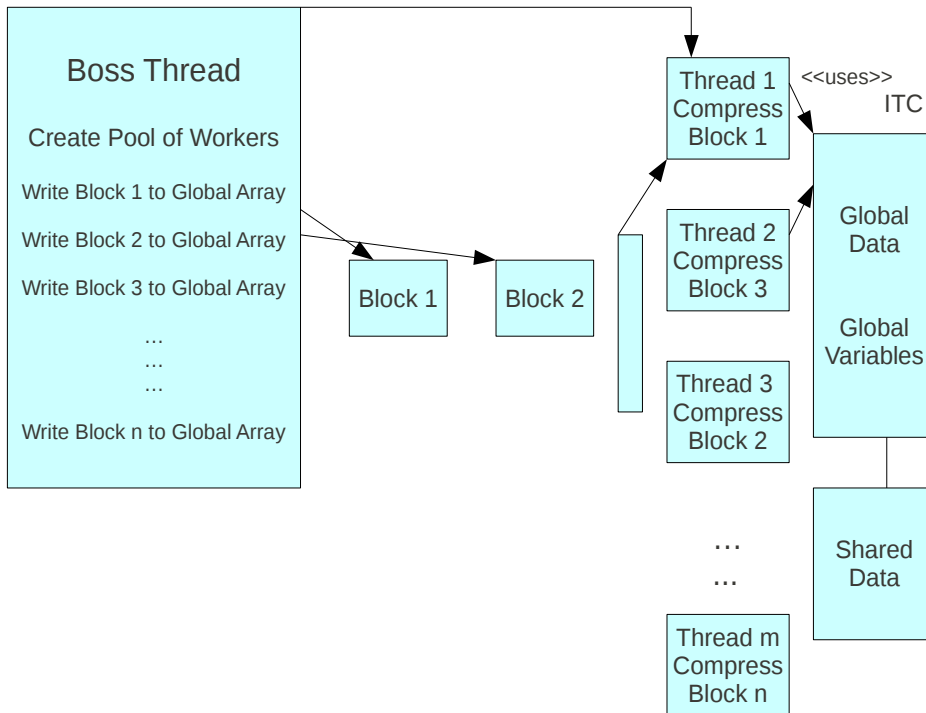


Figure 6 (Based on diagrams in Hughes and Hughes [1])

I decided to incrementally proceed with the design and implementation (implementation details and issues will be covered in the implementation section of the report). I started by writing and implementing an initial version of the algorithm with just one boss and one worker. Because of this simplification, contention issues between workers do not get reflected in the following algorithm.

(NOTE : The part of the serial code that takes up the maximum amount of time as per the gprof analysis has been made a separate worker thread.)

One boss and one worker (simplified version of the main design)

Boss thread –

```
create worker
loop
  loop
    read 5000 bytes
    copy input until stop
    if a block full of data has been read, or if end of
      file is reached then break
  end loop
  signal worker thread
  if end of file then break
end loop
set global thread termination flag
wait for worker to end
```

Worker thread –

```
loop
  if global thread termination flag set then break
  wait for signal from boss
  compress block
  write compressed data to output handle
end loop
```

I implemented and tested this algorithm . Please refer to the Implementation section of the report for the Software Engineering challenges that I had to face to re-factor the code. Once the tests were successful, I wrote a more complete algorithm which incorporated several workers processing blocks in parallel. This includes contention issues between blocks.

One boss and multiple workers (contention management)

Worker thread –

```
loop
  if global thread termination flag set then break
  check the global array of blocks for the first block that is
    ready to be compressed
  if no new uncompressed block in the array then wait for signal
    from boss
  read the block number
  mark the block as compressing
  compress block
  if NOT first block and NOT last block
    wait for the previous block to be written
  write the current block
```

```

        signal the worker waiting for this worker to write the next
        block
    end if
end loop

Boss thread –

create n workers
loop
    loop
        read 5000 bytes
        copy input until stop
            if a block full of data has been read, or
            if end of file is reached then break
    end loop
    set the block number in the structure passed to the worker as
    a parameter
    mark the block as ready to be compressed and signal the next
    worker thread
    if end of file then break
end loop
set global thread termination flag
wait for workers to end

```

Figure 7 shows the behaviour of 4 threads following the above algorithm. (**Note:** The diagram is not to scale. The proportions of the lengths of the synchronised and parallel sections do not indicate the ratio of the actual times spent in the synchronised and parallel sections of the code.)

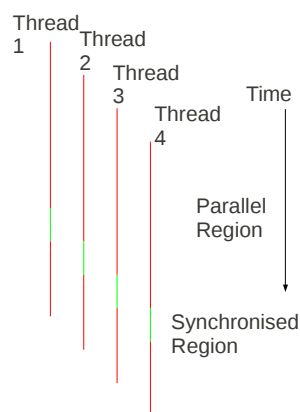


Figure 7

6. Implementation

The following description is a brief overview of the code changes that I have made in the files mentioned:

Code added –

6.1 Code changes in bzip2.c

- Code to call the boss method for block compression. A new method called `BZ2_bzMasterCompress` has been written in `bzlib.c` which is called from here.

Code removed –

- The old code to read from the file and call `BZ2_bzWrite()`.

6.2 Code changes in bzlib.c

Code added –

- Addition of new header files for UltraSPARC T2 specific code. (This is for the test described in section 7.2. For the purpose of portability, this can be removed).
- Global declaration of mutexes, condition variables, storage buffers and other global variables needed for the code.
- A **structure** called **HackBlock** to properly copy the last byte of a given block (which was being dropped when the code was initially refactored).
- A new method called **updateNewBlock()**. This method initialises each block independently. It allocates heap memory for data storage, initialises important block parameters and CRC checksum values per block. CRC checksum is a checksum to detect incorrect data alteration.
- A new method called **combinedCRCupdate()**. For block 0, the combined CRC value starts at 0. Block 0 updates the value and the updated value is needed by block 1 as an initial seed. It then updates the value and block 2 needs the value as an initial seed. This calculation is done before the blocks are actually processed and compressed, hence, this method is called after block 'n' computes the value so that it can be given to block 'n+1' as an initial seed.
- A new method called **worker()**. This will be executed by the different worker threads. It has an infinite loop which executes unless the boss asks the worker to stop. The worker thread checks the blocks to find the next block that is ready to be compressed. If it finds a block, it proceeds with the remaining code, otherwise it waits for the boss to wake it up. When it is in

a position to proceed further, it performs CRC calculations, calls `combinedCRCUpdate()` (described above) and then calls `BZ2_compressBlock()` defined in `compress.c`. The compressed data is written to the output stream. This part is synchronised with condition variables to make sure that block 'n+1' never gets written to the output stream before block 'n'. Heap memory is then freed. As mentioned earlier, this keeps happening in an infinite loop, unless a Boolean flag called "allWorkersRetire" is set by the boss to "True" and the worker is signalled by the boss to exit. This flag is checked by the workers on receiving the signal.

- A new inline method called `ADD_CHAR_TO_BLOCK_THREAD()`. This adds the next read byte to the block that has to be compressed. This is almost same as the macro `ADD_CHAR_TO_BLOCK` in the original code. However, it has been converted from a macro to an inline function and special code has been added to handle the last character of each block. When the original code was re-factored to process the data as independent blocks, the last byte of every block was being dropped because of a bug. The structure `HackBlock` has been used in this method `ADD_CHAR_TO_BLOCK_THREAD()` to fix the bug.
- A new function called `BZ2_bzMasterCompress()`. This is the boss of the Boss-Worker (Delegate) pattern that I have implemented. This is not executed by a separate thread but by the main thread of execution. The mutexes, condition variables and other local variables are initialised first and then the worker threads are created. This is followed by an infinite loop which terminates only when the input file is completely read. In this, the block to be filled is initialised with `updateNewBlock()` and then 5000 bytes are read at a time, until 900 KB (default size of a block) of data has been read. Each read byte is copied to the block to be compressed by calling `ADD_CHAR_TO_BLOCK_THREAD()`. A special fix is used for the last byte of each block (see above). When a full block gets read, the boss signals one of the workers to process the block while it starts reading the next block. When it finishes reading the whole input file, it exits the infinite loop and sets a global variable called "allWorkersRetire" to "True". This helps the "workers" to exit but not before they finish what they are doing. Once this variable is set, the boss broadcasts all the workers and waits for them to "join" the main thread of execution.
- Code to accept the desirable number of threads as an environment variable. If the user does not set this variable, the number of threads defaults to 8.

Code removed –

- Some initialisations removed from `BZ2_bzCompressInit()` as they will be handled in `prepareNewBlock()`.

6.3 Code changes in `compress.c`

Code added –

- Some of the global synchronisation constructs that get initialised in `bzlib.c` are used here, so I have declared them in this file with the C 'extern' keyword.

- A new method called **bitShiftUpdate()**. This is not needed for the first block. For all other blocks, the compressed data gets written to an array using bit shifts and the unary OR operation. The final value of the bit shift variables after block 'n' gets compressed and bit-shift written to the array is used as a starting seed for block 'n+1'. This is the design in the original sequential bzip2 application and if this is not obeyed, the decompression fails. Hence this had to be implemented in my design. This actually affects independent parallel processing to a certain extent. In my design, block 'n' and block 'n+1' are compressed independently using the Burrows-Wheeler transform, the Move To Front coding and the Huffman coding. After this, the thread processing block 'n+1' waits till block 'n' finishes its bit shift write. Once block 'n' finishes, bitShiftUpdate() is called and the final value stored by the bit shift variable is copied to block 'n+1' as an initial seed. Block 'n+1' is then allowed to proceed. This affects parallelism to a very small extent, since the amount of processing that requires synchronisation is very small compared to the amount of processing that can be done independently (Figure 7).
- I have moved all the code for writing to the output array and bit shift updates to the end so that the main algorithms of Block-sort, Move To Front coding and Huffman coding are executed independently in parallel threads. The last section is synchronised with the help of condition variables such that the thread processing block 'n+1' cannot execute till the thread processing block 'n' finishes all writes and bit shift updates.

Code removed –

- Some of the code for bit shift updates have been removed and have been moved to the end (after the algorithms for Block-sorting, Move To Front coding and Huffman coding).

6.4 Code changes in the Makefile

Code added –

- The **-lpthread** flag has been added after the main linking instruction. This has been added because the code needs to use the Pthread library.

Code removed –

- Some of the small tests have been removed as they are unnecessary. I have performed required tests separately.

6.5 Software Engineering challenges faced

I faced several challenges because of data dependencies in the original bzip2 code. These made the whole re-factoring exercise non-trivial. Some of these challenges were:

1. Choosing the right library to implement parallelism: For this project, I learnt Pthreads and

OpenMP. However, this project needed a lot of code refactoring. For this purpose, I needed greater control over the code structure for implementing parallelism. Also, the library chosen had to structurally support the implementation of the Delegation Model. Because of this, I chose Pthreads over OpenMP. I found literature to support my choice. *“Both APIs are portable, but Pthreads offers a much greater range of primitive functions that provide finer-grained control over threading operations. So, in applications in which threads have to be individually managed, Pthreads or the native threading API (such as Win32 on Windows) would be the more natural choice.”* (Binstock, 2009) [18]

2. After the creation of each new block by the “boss”, my code had to initialise the block. The original code did not treat each block as a completely independent data structure, hence the initialisation was done only once. I had to write a function called `updateNewBlock()` and call it separately for each block to set some of the block parameters, initialise memory, and more.
3. There were some initialisations that were specific to the first block in the original code. However, since the blocks were not designed as independent data structures in the original code, this was not obvious. After a careful scrutiny of the original code, I worked out this requirement and initialised those parameters.
4. After redesigning each block as a separate data structure, I faced a new problem. Input data is read from the input file 5000 bytes at a time. Once a given block is read and copied to the data structure that has to be compressed, if the read buffer still had some data (e.g. 2000 remaining bytes), it was getting lost. With the start of the construction of the next block, 5000 new bytes were getting read and the 2000 bytes remaining in the read buffer were getting lost forever. I had to fix this bug. This problem is not in the original bzip2 code because the code is sequential and the blocks had not been designed as completely independent data structures.
5. Each block calculates a CRC checksum before compression. The checksum value calculated by block 'n' is used by block 'n+1' as a starting seed. After redesigning each block as a separate data structure which would be processed independently, the checksum values were not being calculated correctly. I had to write code to transfer combined CRC values from block 'n' to block 'n+1'. This dependency did not affect parallelization since CRC calculations are performed before the actual compression process.
6. After separating each block as an independent data structure (and after fixing issue 4 mentioned above), a new bug showed up. The byte at the boundary between two consecutive blocks was being dropped. This was a design issue. I had to write a 'hack' fix for this.
7. The biggest issue with re-factoring was that (in the original sequential design) each block, during the process of compression gets compressed and written to an output array using bit shift writes and unary OR and the final numeric value obtained after all bit shifts and unary ORs for block 'n' is used as an initial seed for block 'n+1'. This is a huge sequential constraint. I had to solve this by moving all bit shift operations to the end of the compression process and synchronising them between block 'n' and block 'n+1' using condition variables. This affects independent parallelization to a certain extent, however, the amount of time that a thread spends in the synchronised region is much smaller than the amount of time that it spends in the parallel region.

8. Several bugs crept up during testing with multiple threads. These were thread synchronisation issues and had to be fixed by using thread synchronisation constructs properly in proper places. I will discuss one such bug here. Let us consider the situation where thread 'i' has to wait for thread 'j' to finish executing a certain section of code before it can execute that section itself. Waits and signals have been implemented with POSIX condition variables. If thread 'j' executes that code and signals thread 'i' even before thread 'i' reaches the barrier for waiting, the signal will be lost and thread 'i' will wait forever when it reaches the barrier. To solve this, I had to maintain a global shared variable. When thread 'i' finishes executing the section that needs synchronisation, it sets the global variable to indicate that it has done its task. When thread 'j' reaches the section in question, it checks the variable. If it has been set by thread 'i', thread 'j' bypasses the wait.

7. Testing and Results

Testing the code involves two parts:

1. Testing for data races using Sun's Performance Analyzer tool [12].
2. Testing the performance gain as compared to the sequential version of bzip2.

7.1 Testing for race conditions using Sun's Performance Analyzer tool

Sun's Performance Analyzer tool reveals a lot of information about an application. The key information that I wanted from the analysis was detection of race conditions. Concurrent modification of variables can lead to race conditions and non-deterministic behaviour. The Performance Analyzer shows us possible race conditions in our code.

The Performance Analyzer consists of several tools. Two of them are of special importance here. As per Sun Microsystems [12],

1. *“The Collector tool collects performance data using a statistical method called profiling and by tracing function calls. The data can include call stacks, microstate accounting information, thread synchronization delay data, hardware counter overflow data, Message Passing Interface (MPI) function call data, memory allocation data, and summary information for the operating system and the process. The Collector can collect all kinds of data for C, C++ and Fortran programs, and it can collect profiling data for applications written in the Java programming language. It can collect data for dynamically-generated functions and for descendant processes.”*

2. *“The Performance Analyzer tool displays the data recorded by the Collector, so that you can examine the information. The Performance Analyzer processes the data and displays various metrics of performance at the level of the program, the functions, the source lines, and the instructions. These metrics are classed into five groups:*

- *Clock profiling metrics*
- *Hardware counter metrics*
- *Synchronization delay metrics*
- *Memory allocation metrics*
- *MPI tracing metrics”.*

To perform this analysis, I took the following steps:

- a) I created a separate directory and copied the entire modified source code of bzip2 to that directory.
- b) I changed the makefile to replace GCC with Sun's C compiler for the performance analysis. The bzip2 makefile uses the GNU C compiler (GCC) to compile and link the application.
- c) I also altered the makefile to specify POSIX thread library flags as per Sun's specifications.
- d) One of the methods in the code was declared **inline**. This method was calling a **static** method. GNU C compiler does not consider this to be a problem. Sun's compiler follows a C standard that considers **inline** as **extern inline** by default. A static method cannot be called from an extern inline method. Hence, I fixed this through a preprocessor directive:

#define inline static inline

- e) I had to insert the following flags in the makefile for code compilation and linkage as per Sun's User's Guide [13]

-g -xinstrument=datarace -xvpara

- f) I ran the executable with the Collector tool (the “collect” command) to generate experiment data.
- g) I ran the Performance Analyzer tool (the “analyze” command) to analyse the experiment data and generate results.

Result and analysis:

Ten race conditions were detected. Please see Appendix 2 for screen shots showing the race conditions. One of them is because of a debugging “printf” statement (all debugging printf statements will be removed). The others will be analysed and fixed in the artefact if required. The tool indicated that some of the variables that I have declared in my code are being written to and/or read from in multiple threads. However, if this tool detects a race condition, it does not necessarily mean that it will occur. A sequence diagram of the threads can indicate if the particular race condition will occur or not. Preliminary analysis of the results showed that most race conditions that the tool detected (such as access to variables like 'allWorkersRetire' and 'zipWritten') are either of the following:

- Not important, because I had already written protective code to ensure that the result is independent of which thread accesses the variable first.
- Not possible, given the time lag between threads. Even though the time lag is small, it is enough to ensure that the race condition does not get triggered.

However, it is a good Software Engineering practice to run such tests and analyse the results. The sequence diagram should be studied to confirm whether the race detected by the tool can actually occur or not.

7.2 Testing the performance gain as compared to the sequential version of bzip2

I conducted these tests with several threads. To carry out the test with 'n' threads, the threads were placed in such a way that there would be one thread per execution unit (UltraSPARC T2, the underlying hardware, has 8 CPU cores, 8 hardware threads and 2 execution units per core. 7 of the 8 cores are available for this project).

As an example, to run with 4 threads, thread 1 would be placed on virtual CPU 0, thread 2 would be placed on virtual CPU 4, thread 3 would be placed on virtual CPU 8 and thread 4 would be placed on virtual CPU 12. The reason for this is that virtual CPUs (hardware threads) 0 to 7 belong to core 1, virtual CPUs 8 to 15 belong to core 2 and so on. Since each core has 2 execution units, virtual CPUs 0 to 3 would belong to the first execution unit of core 1 and virtual CPUs 4 to 7 would belong to the second execution unit of core 1.

To place threads on specific virtual CPUs, architecture specific code had to be used. For the Sun platform, the library method to do this is:

```
int processor_bind(idtype_t idtype, id_t id,
                  processorid_t processorid, processorid_t *obind);
```

For 'n' threads, I carried out the test by placing n threads on the execution units as described above. I ran the application to compress files. For 'n' threads, I chose a file large enough to make use of of all 'n' threads. The total execution time to compress the file was obtained. I ran the same test on the same file with the sequential version of bzip2. The ratio of the execution times for the parallel

version and the sequential version is **speedup** for 'n' threads.

To obtain the execution time, I used the following Perl script:

```
use Time::HiRes;
my $start = [ Time::HiRes::gettimeofday( ) ];

`program`;

my $elapsed = Time::HiRes::tv_interval( $start );
print "Elapsed time: $elapsed seconds!\n";
```

Where **program** is the program (including the arguments) that has to be timed. I obtained the Perl script from the gamedev.net online forum [19]. This was posted on 13th September, 2004 in the forum.

I conducted the experiment twice. First time, I used files of different sizes, such that a file with 'n' complete blocks will be processed by 'n' threads. Hence, I used larger files to test the effect of more threads. Second time, I ran the experiment with different threads on the same file. The file chosen this time was large enough to ensure that no created thread sits idle during the execution of the program.

The speedup data that was obtained from these experiments can be found in Appendix 3.

Figure 8 shows the number of worker threads on the X axis and the speedup values on the Y axis when the experiment was conducted the first time with different input files. This experiment was conducted with **1-10 threads**. In this, each thread processes one block as the number of threads equals the number of complete blocks. Hence, as the number of threads were increased, the input size was proportionately increased as well.

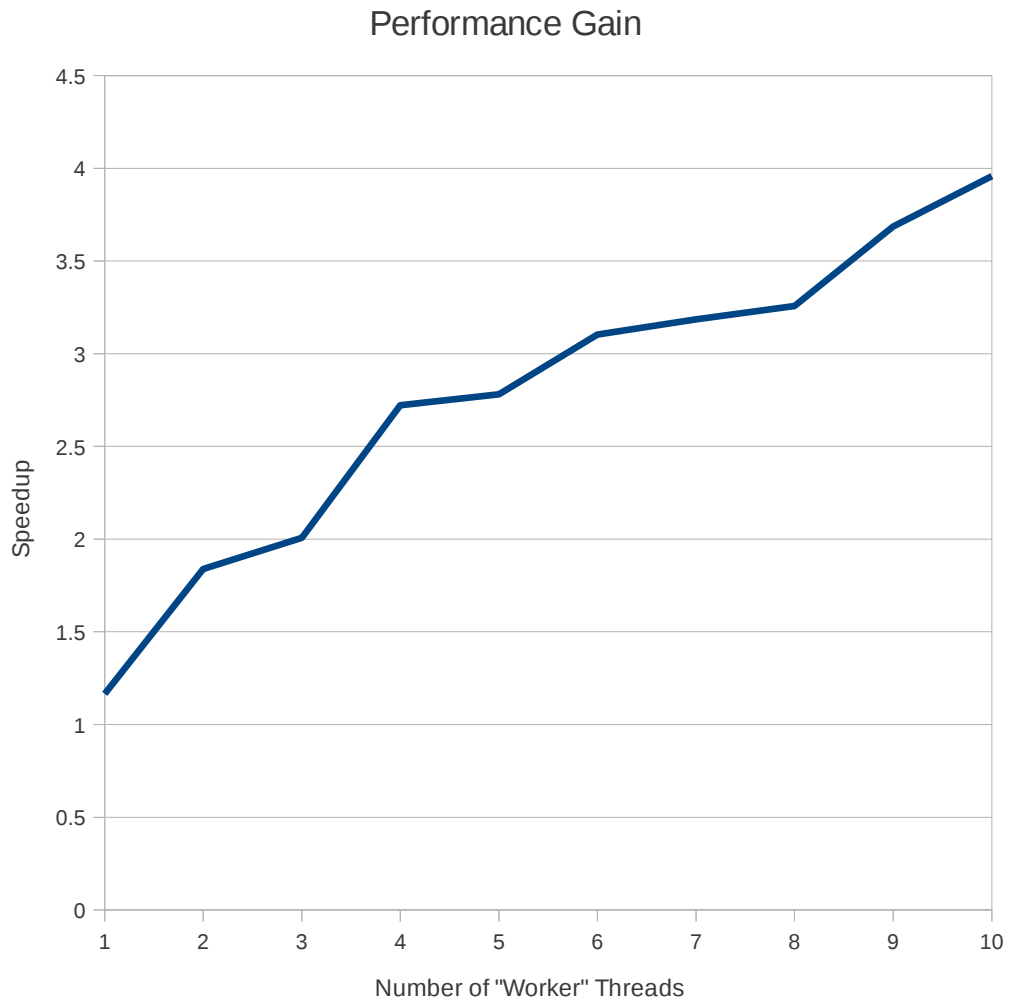


Figure 8

Figure 9 shows the number of worker threads on the X axis and the speedup values on the Y axis when the experiment was conducted the second time with the same input file. An input file of 12.8 MB was used for this. In this, each thread processes several blocks of data, if the number of threads is much lesser than the number of blocks. This experiment was conducted with 1-14 threads.

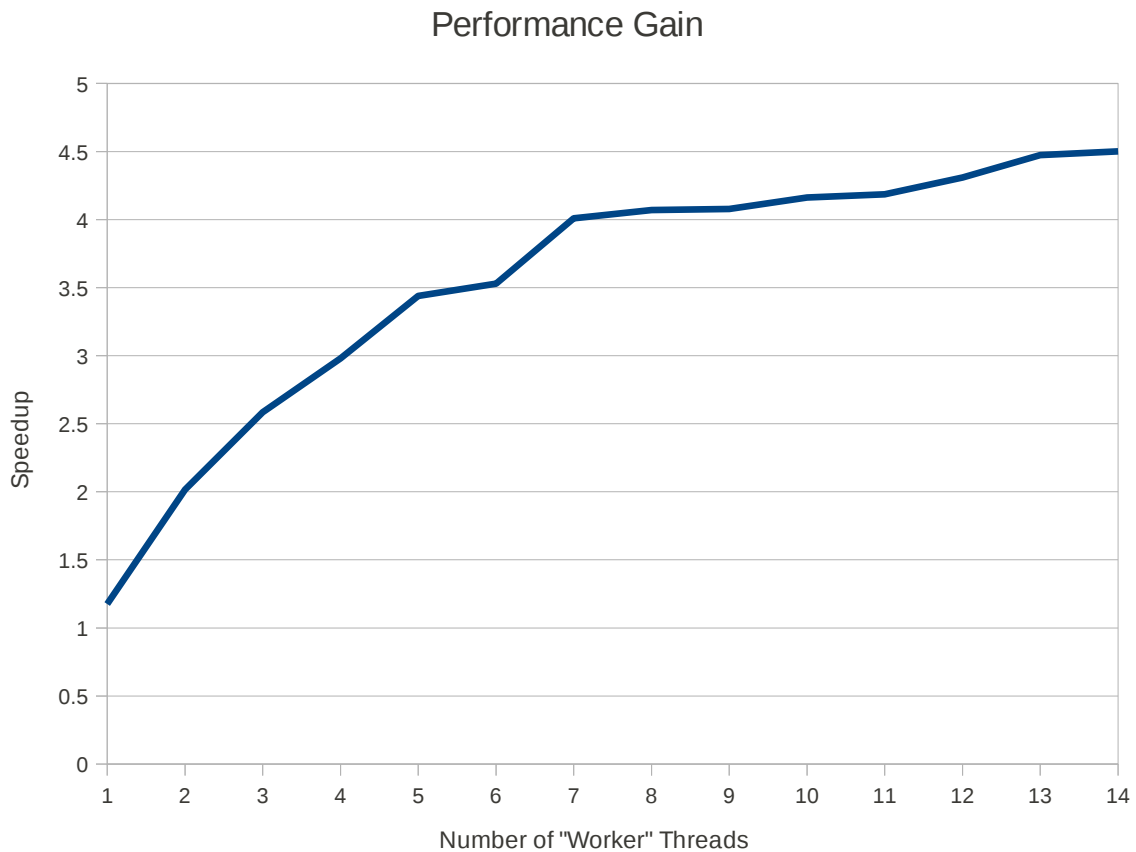


Figure 9

I uncompressed the compressed files to ensure that they have not been damaged by my version of bzip2.

8. Discussion, Conclusion and Future Work

8.1 Results of the performance gain test

The graphs that have been plotted in section 7.2 are not straight lines. The possible reasons are:

1. The code has a sequential section and a synchronised section along with the independent parallel section (although the parallel section is much bigger in terms of execution time). The sequential and the synchronised sections affect linear speedup.
2. Thread creation and synchronisation across multiple cores has overheads. This overhead increases with the number of threads and slows down the application. This effect is possibly more visible in Figure 12.
3. File size affects the amount of data cached in L2 caches and this might affect the outcome of the experiment to a certain extent.

However, the overall trend of the graphs are upwards, indicating that the design is scalable with the number of threads and processor cores. This is because of the design that I adopted to solve this problem. The delegation model allows for scalability with respect to the available resources

8.2 Conclusion

The overall approach that I took and the activities that I carried out for the parallelization exercise is:

- A thorough study of the code and relevant documentation to understand the work done by different methods and the implementation of different algorithms. This gave me an overall idea of how the code was behaving, how the methods that implement the compression algorithms are called, and more.
- The use of a tool such as gprof to understand the execution path better and to find out the parts of the code that take the maximum amount of time.
- Preparation of a design (based on the above step) using an existing design pattern for re-factoring the code. This design had to be **scalable** with the number of cores in a multi-core computer.
- Re-factoring the code as per the design and tests of the code for bugs.
- Initial test of the code with a single “worker” thread to confirm stability and correctness of re-factored code. I identified and fixed several bugs in this step.
- Introduction of multiple threads and synchronisation mechanisms.
- Test of the code for race conditions using Sun's Profile Analyzer tool.
- Test of the code for speed-ups for several threads.
- Generation of graph for the data collected.

In this case, the following steps enabled me to produce a scalable solution.

Software Engineering lessons learnt:

I learnt the following Software Engineering lessons from this exercise:

- A tool to profile the original sequential application is very helpful in identifying 'zones' in the serial code that take up the maximum amount of time. Code refactoring now becomes a simpler exercise, because the programmer can focus on a smaller area of the code.

- There are several parallel design models that can be used to refactor the code. The advantage that these models have over fine grained parallelism techniques such as loop parallelization with OpenMP is that they allow the programmer to model as much task as she wishes to as a parallel unit of execution. This gives the programmer greater control over the code.
- Some design models allow scalability of parallelism with the number of parallel units and the number of processor cores. These models should be chosen if the program has to make good use of future hardware.
- Applications such as bzip2, which have been a lot of data dependencies can be very hard to parallelize. Parallelization breaks data dependencies and this can give rise to several bugs. The programmer has to code very carefully and test several possibilities to eliminate bugs.
- The programmer has to understand and implement synchronisation of parallel task units very carefully. A lot of unnecessary synchronisation can beat the whole purpose of parallelization whereas insufficient synchronisation can lead to logical errors.

8.3 Future Work

Further analysis of the artefact, including a search for opportunities of lower level parallelism within each parallel task, can be carried out. This will allow parallelism at several levels of abstraction. Analysis can also be carried out to reduce the amount of synchronisation without breaking the requirements of the application. To study the effectiveness of one design pattern over the other, the application bzip2 could also be re-factored as a Peer-to-Peer model. Unlike the Delegation model, in the Peer-to-Peer approach, each of the several threads of execution perform the same task. One thread creates all the other threads, but it does not delegate work to other threads (like the boss thread in the Delegation model). In a Peer-to-Peer approach for bzip2, each thread reads a block of data from the source, processes the data and writes the result to the output. In this case, the read and the write have to be synchronised.

When the requirement is to design and develop a parallel application from scratch and not re-factor an existing application, there are several Software Engineering solutions that can be used. Many of these integrate with several stages of the Software Development Life Cycle. Please see section 2.10 for a discussion of these techniques.

9. Glossary

Fine grained parallelism: The individual task units such as threads or processes perform small amounts of work and there is significant communication overhead between the task units.

Coarse grained parallelism: The individual task units such as threads or processes perform large amounts of work. There is a lot of computation between communication events.

Moore's Law: The number of transistors that can be placed on a chip doubles every two years.

Super-scalar processors: Processors that can support more than one instruction per clock cycle (by implementing parallelism at the hardware level).

CPU State: The state of the current process that is being executed.

CPU Interrupt Logic: This handles interrupts that require the current task execution to be paused.

Thread safe library: A library is thread safe if its methods show correct behaviour when multiple threads try to access them simultaneously.

Autotuning: Auto-generation of platform-specific efficient parallel code.

Parameter Space: The set of values of parameters.

10. Appendix 1-GProf Analysis Flat Profile

% time	cumulative seconds	self seconds	total calls	ms/call	ms/call	name
53.3	0.08	0.08	1	80	130	BZ2_compressBlock
33.3	0.13	0.05	1	50	50	BZ2_blockSort
13.3	0.15	0.02	35	0.57	4.29	handle_compress
0	0.15	0	3155	0	0	add_pair_to_block
0	0.15	0	35	0	4.29	BZ2_bzCompress
0	0.15	0	24	0	0	BZ2_hbMakeCodeLengths
0	0.15	0	23	0	0	myfeof
0	0.15	0	22	0	4.29	BZ2_bzWrite
0	0.15	0	16	0	0	bsPutUChar
0	0.15	0	6	0	0	BZ2_hbAssignCodes
0	0.15	0	6	0	0	myMalloc
0	0.15	0	5	0	0	copyFileName
0	0.15	0	4	0	0	default_bzalloc
0	0.15	0	4	0	0	default_bzfree
0	0.15	0	4	0	0	hasSuffix
0	0.15	0	3	0	0	snocString
0	0.15	0	2	0	0	addFlagsFromEnvVar
0	0.15	0	2	0	0	bsPutUInt32
0	0.15	0	2	0	0	fileExists
0	0.15	0	2	0	0	uInt64_from_UInt32s
0	0.15	0	2	0	0	uInt64_toAscii
0	0.15	0	2	0	0	uInt64_to_double
0	0.15	0	1	0	0	BZ2_bsInitWrite
0	0.15	0	1	0	0	BZ2_bzCompressInit
0	0.15	0	1	0	55.71	BZ2_bzWriteClose64
0	0.15	0	1	0	0	BZ2_bzWriteOpen
0	0.15	0	1	0	0	applySavedFileAttrToOutputFile
0	0.15	0	1	0	0	applySavedTimeInfoToOutputFile
0	0.15	0	1	0	150	compress

% time	cumulative seconds	self seconds	total calls	ms/call	ms/call	name
0	0.15	0	1	0	0	containsDubiousChars
0	0.15	0	1	0	0	countHardLinks
0	0.15	0	1	0	0	fopen_output_safely
0	0.15	0	1	0	150	main
0	0.15	0	1	0	0	notAStandardFile
0	0.15	0	1	0	0	pad
0	0.15	0	1	0	0	saveInputFileMetaInfo

11. Appendix 2-Race Conditions

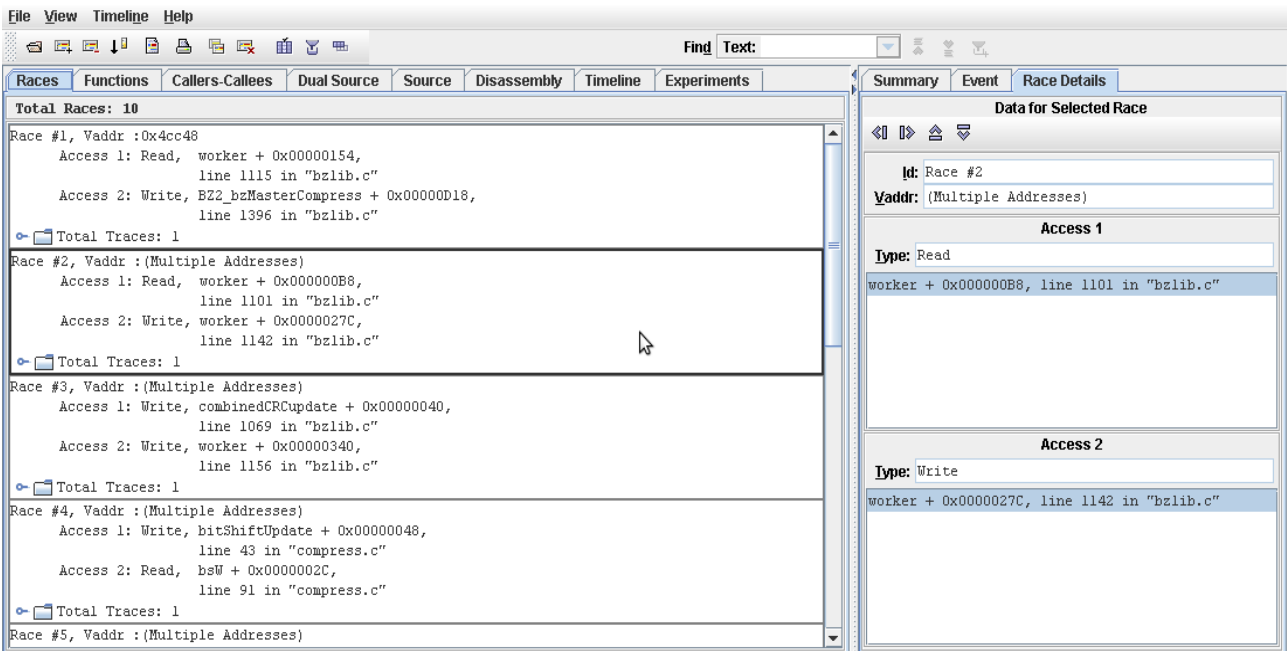


Figure 10: Race Conditions in bzlib.c and compress.c

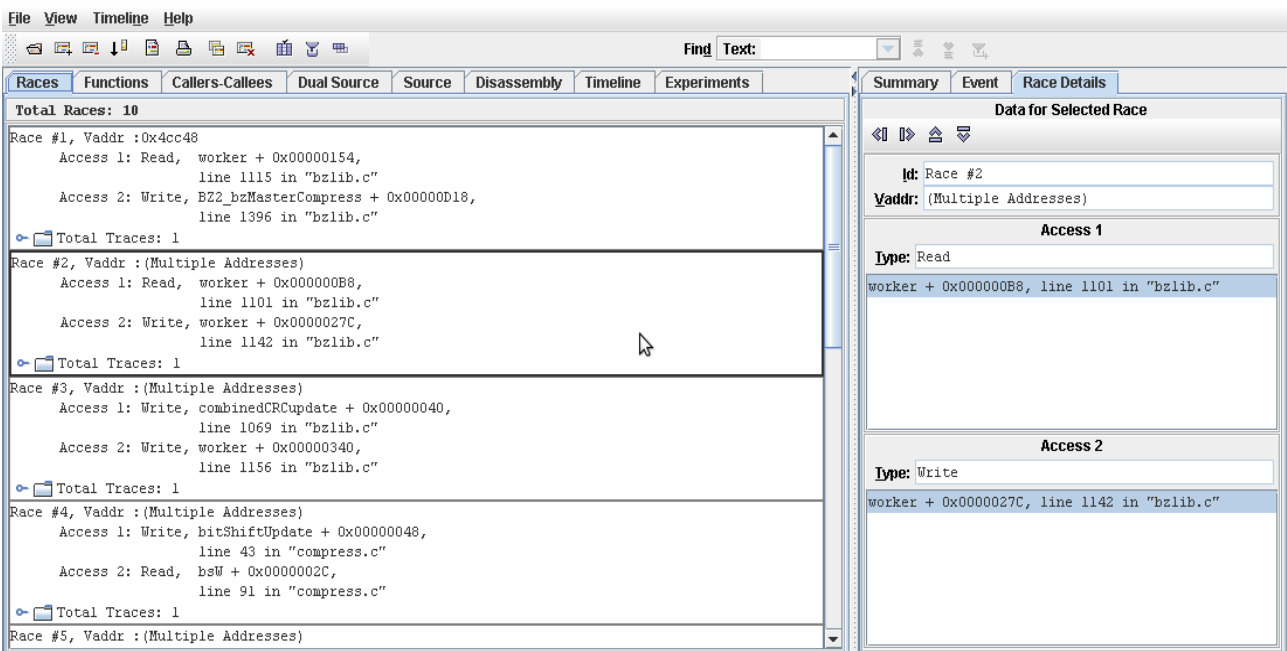


Figure 11: Race Conditions in bzlib.c and compress.c

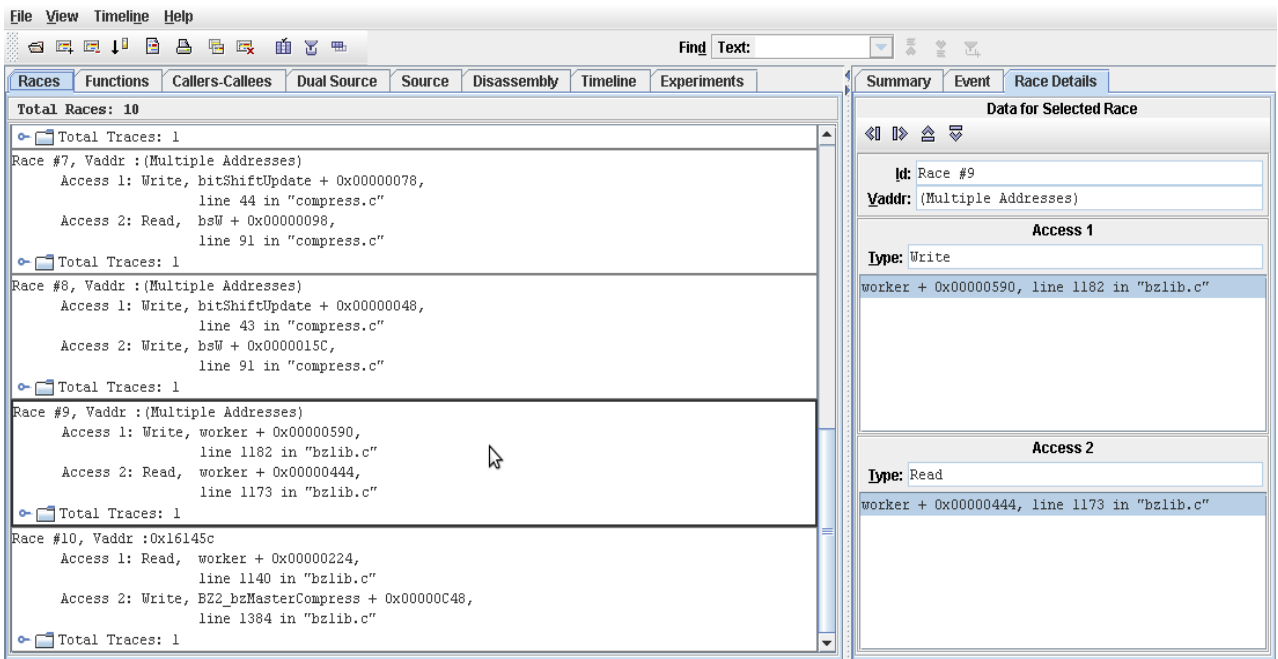


Figure 12: Race Conditions in bzlib.c and compress.c

12. Appendix 3-Speedup values

For Figure 8, the speedup values are:

#Number of threads	#Speedup
1	1.165941 663
2	1.838216 135
3	2.006910 314
4	2.722494 458
5	2.780437 962
6	3.103622 087
7	3.185928 337
8	3.257531 034
9	3.685347 641
10	3.957896 884

For Figure 9, the speedup values are:

#Number of threads	#Speedup
1	1.175952167
2	2.016630792
3	2.584555797
4	2.980347101
5	3.439198764
6	3.529041645
7	4.009126284
8	4.070108751
9	4.078413297
10	4.16241794
11	4.186149717
12	4.308579183
13	4.472923781
14	4.500746619

13. Bibliography

- [1] C. Hughes and T. Hughes, *Professional Multicore Programming, Design and Implementation for C++ Developers*.
- [2] V. Pankratius, C. Schaefer, A. Jannesari, and W.F. Tichy, "Software engineering for multicore systems: an experience report," *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, New York, NY, USA: ACM, 2008, pp. 53–60.
- [3] V. Pankratius, A. Jannesari, and W.F. Tichy, "Parallelizing bzip2 A Case Study in Multicore Software Engineering."
- [4] J. Roberts and S. Akhter, *Multi-Core Programming: Increasing Performance through Software Multithreading*.
- [5] B. Barney, *Introduction to Parallel Computing*, Lawrence Livermore National Laboratory: .
- [6] M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, 2008, pp. 33–38.
- [7] B. Barney, *OpenMP*, Lawrence Livermore National Laboratory: .
- [8] J. Seward, *bzip2 and libbzip2, version 1.0.5 A program and library for data compression*.
- [9] Sun Microsystems, *UltraSPARC T2 Processor – The world's first true system on a chip*.
- [10] R. Stallman and J. Fenlason, *GNU gprof*.
- [11] D. Geer, "For Programmers, Multicore Chips Mean Multiple Challenges," *Computer*, vol. 40, 2007, pp. 17–19.
- [12] Sun Microsystems, *Performance Analyzer, Sun Studio 10*.
- [13] Sun Microsystems, *Sun Studio 12: Thread Analyzer User's Guide*.
- [14] K. Asanovic et al., "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, 2009.
- [15] K. Olukotun, "Towards Pervasive Parallelism."
- [16] M. Dubash, "Moore's Law is dead, says Gordon Moore," 13-Apr-2005.

- G. Manzini, "The Burrows-Wheeler Transform: Theory and Practice," in *Mathematical Foundations of Computer Science 1999*, vol. 1672, M. Kutylowski, L. Pacholski, and T. Wierzbicki, Eds. Springer Berlin / Heidelberg, 1999, pp. 34-47.
- [17]
- [18] A. Binstock, "Threading Models for High-Performance Computing: Pthreads or OpenMP?," 05-Oct-2009.
- [19] "[Perl] How to get the current time in milliseconds??" [Online]. Available: http://www.gamedev.net/community/forums/topic.asp?topic_id=268800.