ATHABASCA UNIVERSITY

CONCURRENT PROGRAMMING:
A CASE STUDY ON DUAL CORE COMPUTERS

BY

HAROLD SHIP

A project submitted in partial fulfillment
Of the requirements for the degree of
MASTER OF SCIENCE in INFORMATION SYSTEMS

Athabasca, Alberta
April, 2008

© Harold Ship, 2008
Abstract

Recent innovations in computing technology have resulted in the development of multi core CPU's which are capable of performing multiple tasks simultaneously. On the other hand, many algorithms are implemented sequentially, even though some tasks lend themselves to being able to be performed more efficiently in parallel on multi core CPU's. This essay presents a review of current literature in concurrent programming techniques, including Shared State Concurrency, Message Passing, Concurrency, Software Transactional Memory and Declarative Concurrency. Several programming languages are compared with respect to how concurrent programming is supported. A parallel implementation of the introsort algorithm is provided as a case study. The sorting time was measured on single and dual core CPU's for data sets of various sizes, resulting in 32% improvement in total sort time for the concurrent version on the dual core computer. Although this demonstrates the benefits of programming for concurrency on multi core computers, it stands to be further improved. It is recommended that the techniques discussed be applied in practice in libraries of algorithms.
# TABLE OF CONTENTS

## INTRODUCTION

- Objective .................................................................................................................. 1
- The Potential of Concurrency .................................................................................. 1

## LITERATURE REVIEW

- CPU Architecture ...................................................................................................... 2
- Concurrency in Computing ........................................................................................ 3
- Implementation of Concurrent Sorting Algorithm ..................................................... 6

## CONCURRENT PROGRAMMING TECHNIQUES

- The Problem of Concurrency .................................................................................... 8
- Shared-State Concurrency .......................................................................................... 8
- Message Passing Concurrency ................................................................................... 11
- Software Transactional Memory ............................................................................... 13
- Declarative Concurrency ............................................................................................ 14
- Functional Programming .......................................................................................... 15
- Summary .................................................................................................................... 15

## CONCURRENCY IN PROGRAMMING LANGUAGES

- Overview ................................................................................................................... 17
- POSIX Threads ........................................................................................................... 18
- OpenMP ..................................................................................................................... 22
- Concurrency in Java ................................................................................................ 25
LIST OF TABLES

Table 1: OpenMP For Loop Results on Dual-Core.................................................................24
Table 2: Value of C for Quicksort on Single Core.................................................................39
Table 3: Value of C for Quicksort on Dual Core.................................................................40
Table 4: Quicksort Sort Times, Single Core, Pathological Data........................................42
Table 5: Quicksort Sort Times, Dual Core, Pathological Data...........................................42
Table 6: Value of C for Introsort on Single Core Computer...............................................43
Table 7: Value of C for Introsort on Dual Core Computer..................................................44
LIST OF FIGURES

Figure 1: Quicksort on Single Core.........................................................................................41
Figure 2: Quicksort on Dual Core..............................................................................................41
Figure 3: Introsort on Single Core............................................................................................43
Figure 4: Introsort on Dual Core...............................................................................................45
CHAPTER I

INTRODUCTION

Objective

The objective of this work is to research and review the concept of concurrency as it relates to multi-core CPU's. A comparison will be made of several concurrency models and concurrent programming techniques.

As part of this research, the concept and techniques of concurrency are demonstrated with a case study that applies concurrent programming methods to a well known algorithm and compares the results on single and dual-core CPU's.

The Potential of Concurrency

Recent innovations in computing technology have resulted in the development of CPU's capable of performing multiple tasks simultaneously. This is called concurrency.

This development has essentially placed a “tool” in the work box of the programmer which is currently under-exploited. Even more so, the technical limitations of processor speed require that this tool must be used in order to make full use of today's computing' power.

Many algorithms are performed sequentially. Some tasks lend themselves to being able to be performed more efficiently in parallel, thus reducing processing time. This essay will show an example of an algorithm (introsort) that will be re-written to take advantage of a multi-core processing capability.
CHAPTER II

LITERATURE REVIEW

CPU Architecture

From the advent of the Intel 8008 in 1972 until around 2006, computer CPU's improved according to Moore's Law, doubling in speed approximately every 18 months. (Sutter, 2005) discusses how CPU designers have used the three techniques of increasing clock speed, execution optimization, and caching instructions and data to achieve faster processing speeds. The first two of these techniques try to maximize the number of instructions per second for a single CPU.

Today the situation has changed. Designers can no longer squeeze out any more clock cycles, and optimization has also for all practical purposes reached its limit. Nevertheless, computer users still demand faster performance from their CPU's.

Sutter describes how these demands are being met in new ways. In particular, “hyperthreading and multicore architectures.”

Hyperthreading and multicore are new paradigms. “These dual core chips ... should in fact be regarded as the most substantial advance in processor development for many years.” (AMD, 2005). The seeds were sown by Intel in 2002 with Hyperthreading, which simulates two “logical” processing units on a single Pentium 4. It was AMD however, who released the first multicore CPU for desktop computers, the Athlon 64 X2.

(AMD, 2005) describes the architecture of the Athlon 64 X2, why multicore is necessary, and compares the AMD64 to Intel's alternative Pentium D.
Sutter believes that multicore CPU’s will revolutionize how we write software and that concurrent programming is the key. However, because of problems such as locking, and because many problems are difficult to parallelize, and especially because concurrent programming is difficult, concurrent programming is not yet common.

Concurrency in Computing

“Concurrency is a term that refers to a family of policies and mechanisms that enable one or more threads or processes to execute their service processing tasks simultaneously.” (Schmidt, 2000).

A brief history of concurrency in computing is given by (Goetz, 2006). The motivating factors for allowing multiple programs to execute simultaneously as individual processes include resource utilization (utilizing one program's wait time to run another), fairness (fine-grained time-sharing of processor time and resources for different processes), and convenience (multiple smaller simpler programs in cooperation instead of a monolithic program).

These individual processes each had their own memory space and resources and executed instructions sequentially. Threads evolved from processes in order to utilize the convenience
of multiple execution sequences while sharing the same resources as a single process.

Some of the benefits of threads, as described in (Goetz, 2006) are exploiting multiple processors, simplicity of modelling, simplified handling of asynchronous events, and more responsive user interfaces. He also describes some risks, such as safety hazards (because the order of execution of the threads is unpredictable), liveness hazards (deadlock) and performance hazards (stalling or poor responsiveness).

There are many algorithms that are usually performed sequentially. In fact most common algorithms are performed sequentially due to the “simplicity of modelling” mentioned in (Goetz, 2006). However, some tasks lend themselves to being able to be performed more efficiently in parallel, especially in the true parallel execution environment of a multicore CPU.

The most common model of concurrency in software is known as Shared State Concurrency. Most multi-threaded programs are written using shared state, since the sharing of memory and other resources is the primary difference between threads and processes. Shared State Concurrency is characterized by multiple concurrent tasks having simultaneous access to the same memory. There are several techniques of ensuring the correctness and consistency of this state.

The shared state model is not without problems, though. One such problem occurs when attempting to distribute the concurrency to multiple computers. Each computer has it's own memory, so sharing state is “often not practical.” (Lee, 2006). However, the essence of Lee's argument against multithreaded programming is that multithreading is effectively non-deterministic. “... a folk definition of insanity is to do the same thing over and over again and
to expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane. Were they sane, they could not understand their programs.” (ibid).

Another of Lee's arguments against this model is drawn from the Java Memory Model. This is a formal description of how reads and writes to shared state behave under different conditions (Goetz, 2006). This model describes situations where “even astonishingly trivial programs produce considerable debate about their possible behaviors.” (Lee, 2006).

An alternative model, Message Passing Concurrency, is used by the Erlang programming language. Erlang’s model of concurrency is based on the fact that “parallel activities (processes) can be programmed directly in Erlang and that the parallelism is provided by Erlang and not the host operating system.” (Armstrong, 1996). Note that Erlang's processes are not operating system processes, but rather concurrent sequential tasks that communicate through message passing. In Erlang, “spawn” is used to start a process, “send” to send a message to a process, and “receive” to receive messages into a process.

There are other known concurrency models. For example, (Jones, 2007) describes Software Transactional Memory, and (Van Roy, 2003) describes Declarative Concurrency.

These models are discussed in depth in CHAPTER III.

Amdahl's Law is used for estimating the performance improvement that can be obtained from parallelization, per CPU core. A definition and example use of Amdahl's Law can be found in (Goetz, 2006).
Implementation of Concurrent Sorting Algorithm

(Goodrich, 2001) is a basic text on algorithms with implementations in Java. Specifically, it contains an analysis and implementation of quicksort, mergesort and heapsort. All of these algorithms are $O(n \log n)$ in the average case.

Quicksort is a recursive algorithm which at each recursion divides it's input dataset into two subsets. One subset contains items less than a specific value (known as the pivot) and the other subset contains the items greater than or equal to the pivot. In most cases quicksort is the fastest sorting algorithm, but in certain pathological cases is $O(n^2)$. There is a need for an improvement.

One such improvement is known as Introsort (Musser, 1997). This algorithm optimistically begins to run a quicksort, and introspectively examines its running time to detect a pathological $O(n^2)$ input. In such a case the algorithm switches over to heapsort. The algorithm in essence works as follows:

1. Choose a pivot for Quicksort
2. Run Quicksort one level
3. If the depth is more than threshold then
   a) Run heapsort or another $O(n \log n)$ sort on the subset
4. else
   a) split the subset in based on pivot
   b) Run step 2 for each subset

However, creating a parallel version of introsort depends on parallelization of quicksort,
heapsort and the introspection. From (Hong, 1989) it can be determined that parallelization of heapsort is a daunting task. Therefore we will attempt to modify introsort to use mergesort instead of heapsort.

In (Garcia, 2005) a specialized sorting algorithm based on quicksort and mergesort algorithms was measured on simultaneous multi-threading and symmetric multiprocessor platforms. The results showed significant improvement of parallelization under the right conditions.
The Problem of Concurrency

The problem of concurrency is to divide a task into sub-tasks which can run in parallel. In terms of scheduling, in a single processor environment, these tasks can be interleaved by a scheduler. In contrast, a multiprocessor or multi core environment, several tasks can truly be executed in parallel by different processors or cores. Thus the limitation that compels the scheduler to interleave threads is at partially offset by increasing the number of available processors.

However, the problem is not only scheduling. There may be other obstacles to sub-tasks running freely in parallel.

Shared-State Concurrency

The predominant method of implementing concurrency involves creating and starting multiple threads of execution, with program state stored in common memory that is accessible to all threads. Communication between threads is accomplished by reading and writing to common memory. This is known as shared-state concurrency. It is the most common method of implementing concurrency in Java, C# and other imperative and object-oriented languages.

This is simple to conceptualize, and works well as long memory read operations are concerned. However, in all but the most trivial applications, there will be memory that is...
written to by one thread and read or written to by another. In such an application, precautions must be taken to prevent race conditions. A race condition occurs when the access of 2 or more execution threads to a resource results in an error in the program (Goetz, 2006).

For example, suppose we have a dynamic web site, implemented using shared-state, and each page access updates a global counter. The code for such a function might look like this:

```java
req.setContentType('text/html');
page_accesses += 1;
req.write("<html>");
...
```

The problem here is with the statement `page_accesses += 1;` which appears to be a single statement but in fact once compiled may result in a sequence of statements. When compiled from Java, the resulting byte code of the single Java statement is the following 4 statements:

1. `getstatic #2; //Field page_accesses:I`
2. `iconst_1`
3. `iadd`
4. `putstatic #2; //Field page_accesses:I`

Now suppose

1. `page_accesses` starts equal to 0
2. Two requests from the web are received at the same time.
3. The first request is handled in one thread. In Line 1, the value of `page_accesses` is read from memory. In lines 2-3 the constant value 1 is added to it, resulting in 1. However, line 4 has not been executed so the new value has not yet been stored in memory. The current value of `page_accesses` seen by other threads is therefore still 0.
4. The scheduler preempts the thread and starts running the second request. Since `page_accesses` was not yet updated with the incremented value, the second request also reads a value of 0. When the second thread gets to line 4, it will write 1 in the location for `page_accesses`.

5. When the first thread continues, it too stores its calculated value of 1 in `page_accesses`.

One of the most common precautions taken against race conditions is locking. Various schemes such as mutexes and semaphores can be used to restrict access to a resource to a single execution thread. Once a thread acquires a lock to a resource, other threads must wait for it to release the lock before they can access the resource. For example, the above Java bytecode example with locking might look like this:

```
1:  monitorenter
2:  getstatic       #3; //Field page_accesses:I
3:  iconst_1
4:  iadd
5:  putstatic       #3; //Field page_accesses:I
6:  aload_1
7:  monitorexit
```

Despite it's widespread use, shared state with locking has several issues that limit its utility. The first issue, is that the implementation of locking of shared resources is difficult and error-prone. Even worse, there may exist errors which are exposed only very rarely or only under certain platforms and conditions. Such problems can be extremely difficult to find and fix.

Another issue with locking, is that locking does not scale well. For example, consider what happens when read accesses greatly outnumber writes. Because of the writes, access to some resource is locked. Suppose 100 threads now want to access this resource, all of them for
read-only. They can not avoid the lock, in case a write request exists or is about to occur.

However, the lock grants exclusive access to just one thread at a time, effectively running the 100 requests in series.

The significance of this is that concurrency is destroyed, at least partially. Therefore, locking is not an effective option for highly concurrent systems.

Message Passing Concurrency

An alternative to multiple threads sharing state, is to maintain no shared program state among threads. Threads may communicate with each other by passing (sending and receiving) asynchronous messages. (Van Roy, 2004) refer to this as message passing concurrency.

The following example adapted from (Armstrong, 1996) demonstrates the basics of message passing concurrency in Erlang:

```
-module(counter).
-export([start/0, loop/1]).
start() ->
    spawn(counter, loop, [0]).
loop(Val) ->
    receive
        increment ->
            loop(Val + 1);
        stop ->
            true
        {Sender, value} ->
            Sender ! Val,
            loop(Val);
        Other ->
            loop(Val)
    end.
```

Calling the function counter:start/0 will use the built-in spawn/3 (line 4) to create and return
a new process. This process will start running counter:loop/1 with starting value 0. The resulting process can send and receive messages. However, it will only respond to the message “increment”, “{Sender, value}” or “stop” (lines 7, 9).

When the process receives the message “increment”, it will increment the counter by calling counter:loop/1 with the current value plus 1 (line 8). When it receives the message {Sender, value} (line 11) it will send the current value of the counter to the Sender process, and when it receives the message “stop” it will stop receiving messages.

Some of the most important aspects of this style of concurrency are:

1. Each thread of execution (in Erlang called a process) can send messages to other threads.
2. Messages are sent asynchronously. That is, the sender sends the message and forgets.
3. When the sender expects a reply, it sends its Pid (process identifier) as part of the message and the receiver expects it.
4. No thread has access to the internal data of other threads. In Erlang, this is because as a functional language it maintains no state outside of a function definition.

There are several advantages to the message passing concurrency model. When shared access to global state is eliminated, it takes with it the potential for race conditions. Since the threads do not share any resources, there can not be any errors of this type. Identifying and guarding against race conditions is the single largest problem in concurrent implementations.

Further, there is no need for locking, since the motivation for locking was to prevent race
conditions. There are many reasons why locking is a less than ideal solution to the race condition problem. For example, missing a lock can result in a race condition, while locking too often can serialize what was supposed to run concurrently. Taking locks in the wrong order can cause deadlock, and ensuring that locks are released for all error conditions can be difficult. See (Jones, 2007) for several other problems with locking.

There are also some disadvantages of message passing concurrency. The foremost is passing messages involves copying blocks of memory. This can be expensive in terms of time as well as space, and has implications as far as concurrency goes. Since modern CPUs all contain caches for instructions and data, the copying of different blocks of memory in different threads can result in performance degradation. See (Garcia, 2005) for details on a concurrent implementation of quicksort that takes this into consideration.

Software Transactional Memory

The Software Transactional Memory (STM) paradigm was originally described in (Shavit, 1995) and succinctly explained in (Jones, 2007). It is a method of ensuring integrity for shared memory concurrency. Similar to database transactions, each shared memory access or set of accesses by a thread in STM acquire a “snapshot” of memory, optionally modify it and commit the changes.

The commit operation is atomic, meaning that for all accesses for all threads, the transaction appears to have been committed in entirety or not at all. Should the commit operation succeed, any subsequent transactions on the same memory will see all of the changes. This is true whether the second transaction accesses all of the same memory as the first or not.

On the other hand, should a transaction commit operation fail, none of the changes are made
permanent. In case of failure, the transaction owner can decide to retry from the beginning, cancel, or whatever policy is appropriate to the application. The usual reason for failure is that another thread has changed some of the same memory during the transaction.

STM is generally lock free. One type of problem that is easier to solve with STM involves a transaction composed of 2 sub-transactions. This can be difficult or impossible to do efficiently with locking, but is straightforward with STM.

STM is a popular technique in the Haskell programming language, although implementations exist for a wide variety of platforms and programming languages.

**Declarative Concurrency**

Declarative Concurrency is described in (Van Roy, 2003), as are the other paradigms discussed in this chapter. A prerequisite of Declarative Concurrency is the single assignment store, which supports *declarative variables* (ibid, p 44). These variables can be bound at most once, but may also be used in their unbound state.

In this context, an unbound declarative variable is dereferenced causes the current thread to wait until the value is bound by another thread. For example, suppose one thread attempts the operation \( A=23 \), while another thread attempts \( B=A+1 \). In this model, it does not matter how the threads are scheduled. The result at will always be \( B=24 \), since the operation \( B=A+1 \) will wait until \( A \) is bound, in this case to 23. This property is referred to as *dataflow behavior* (ibid, p61).

In most programming languages, it is necessary that order that statements are executed in be deterministic. In fact, most of the time statements are executed in the order that they appear
in the source code. However, once a programming language has dataflow behavior, it can delay execution of a statement that binds a variable to a value until the value is needed. This allows the two statements to be scheduled to run at the same time on different processors. The dataflow behavior ensures the result will be correct. This property of delaying execution until the value is needed is known as lazy evaluation.

The dataflow property, together with multi-threading and lazy evaluation are the essence of Declarative Concurrency (ibid p239).

Functional Programming

There are aspects of functional programming which facilitate concurrency. The most important of these is referential transparency. This is the principle that the order of calling 2 or more functions does not affect the combined result. Referential transparency is a result of the property that a function's return value depends only upon its input parameters, and the property that functions do not have any side effects.

Functional languages often have single assignment property and immutable values. The first principle means that once a variable has been bound to a value, it can never be bound to another. The second principle can be demonstrated within the example of appending an item to a list. This creates a new list, which is a copy of the old one with the item appended. These properties by themselves remove the possibility of race conditions.

Summary

There are several paradigms of concurrency in use today. Shared state concurrency with locking is the most common, but is also the most error-prone and the least scalable on
multiple processors or cores.

Software transactional memory allows for sharing state among concurrently executing threads without locks. However, transaction management is still a potential source of errors.

Declarative concurrency also allows for sharing state among threads. Conceptually it is similar to Java's *Future* in that variables can be “read” only after they are “written”.

Message-Passing concurrency works by disallowing shared state between threads. All communication must occur by sending and receiving messages. This paradigm is easily extended to distributed, parallel processing.
CHAPTER IV

CONCURRENCY IN PROGRAMMING LANGUAGES

Overview

Algorithms must ultimately be implemented in a programming language. In this paper, three
programming languages are considered for the implementation: C++, Java and Erlang.

The problem with C++ is that there is no built-in support for concurrency. A platform-
dependent library such as POSIX Threads is generally required.

However, this may not be true for long. There is a new, platform-independent API supported
by several recent C, C++ and Fortran compilers known as OpenMP. This interface provides a
simple programming model for multithreading on symmetric multiprocessor (SMP) as well
as multicore computers. The idea is to increase concurrency while reducing errors for shared-
state concurrency by providing automatic, implicit parallelism and synchronization.

Java has excellent support for concurrency, especially shared state concurrency. (Goetz,
2006) provides a practical guide to using concurrency in Java. In particular, Java versions
from Java 5 onwards have a sound memory model, built-in language support and powerful,
extensive concurrency libraries.

Erlang has excellent built-in support for concurrency as well as distributed parallelism using
“processes” which communicate using the message-passing model (Armstrong, 1996).

Erlang processes are independent of operating system processes. As such, they work
consistently on all platforms.
Processes can send messages to and receive messages from other processes. This is done in essentially the same manner whether the process is running locally on the same computer or remotely on another system on the network.

**POSIX Threads**

POSIX Threads is a standard API for developing multi-threaded applications. Most Unix variants and Linux distributions support the POSIX Thread standard. This allows for portable, multi-threaded programs to be developed in C or C++.

The interface specifies macros and functions for creating threads, managing threads, sharing state (memory), locking, signalling and more. This is a prototypical shared state concurrency API.

Each thread is given a *start routine*. This is the address of a function which takes a single argument (pointer to void) and returns a pointer to void. For example, the following function can be used as a start routine:

```c
/*
 * Simple function that counts up to a limit.
 * The input parameter is converted to a size_t, then the functions counts up to that value, and finally returns how high it actually counted to.
 */
void *f(void *arg) {
    size_t count = (size_t)arg;
    size_t mycount = 0;
    while (mycount < count) {
        mycount++;
    }
    return (void *)mycount;
}
```

In order to run this function in a thread, a call must be made to *pthread_create*. The function *pthread_join* waits for the thread to complete, and retrieves the return value. The
sample program below demonstrates how this works.

```c
#include <pthread.h>
#include <time.h>
#include <sys/time.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/*
   Defined below: used to count up to some value and return.
*/
void *f(void *arg);

/*
   first arg - number of threads to run (default 1)
   second arg - how high to count in thousands (default 1000).
*/
int main(int argc, char *argv[]) {
    size_t nthreads = 1;
    size_t count = 1000 * 1000;
    if (argc > 1) {
        nthreads = atoi(argv[1]);
    }
    if (argc > 2) {
        count = atoi(argv[2]) * 1000;
    }
    printf("doing %u operations using %u threads\n", count, nthreads);

    // create an array of threads
    pthread_t *threads = (pthread_t *)calloc(nthreads, sizeof(pthread_t));
    if (!threads) {
        perror("calloc (threads) failed!");
        exit(1);
    }
    memset(threads, 0, nthreads * sizeof(pthread_t));

    // used for timing the operation (total time)
    struct timeval begin, end;

    // set begin time
    gettimeofday(&begin, NULL);

    //
    
```
// start the threads, each to run f() with its share of the work.
size_t i;
for (i=0; i<nthreads; ++i) {
    pthread_t t;
    memcpy(&threads[i], &t, sizeof(pthread_t));

    // here is the call to pthread_create()
    if (pthread_create(&threads[i], NULL, f, (void *)(count/nthreads))) {
        perror("pthread_create failed!");
        exit(1);
    }
}

// wait for all the threads to finish
for (i=0; i<nthreads; ++i) {
    void *tcount;

    // here is the call to pthread_create()
    pthread_join(threads[i], &tcount);
    printf("thread %u counted %u\n", i, (size_t)tcount);
}

// get the end time
gettimeofday(&end, NULL);

    int diff = (end.tv_sec - begin.tv_sec) * 1000000 + (end.tv_usec -
begin.tv_usec);
    printf("%u operations using %u threads took %d us\n", count, nthreads,
    diff);

    free(threads);
    return 0;
}

For synchronization of shared state and other resources, the POSIX Thread API provides several services including mutexes, condition variables, and read-write locks. The example below demonstrates the use of mutexes.

/*
 Scoped Lock idiom from (Schmidt, 2000).
 When object is created it locks a mutex. The mutex is unlocked when
the object goes out of scope (any return statement, exception, etc)

/*
class ConcurrentGuard {
private:
  pthread_mutex_t *mutex;
public:
  ConcurrentGuard(pthread_mutex_t &mutex) : mutex(&mutex) {
    pthread_mutex_lock(this->mutex);
  }
  ~ConcurrentGuard(void) {
    pthread_mutex_unlock(this->mutex);
  }
};

/*
  A queue that supports concurrency via multithreading and internal locking.
*/
template <class T>
class ConcurrentQueue {
public:
  /*
   * push an item onto the queue
   */
  void push(T item) {
    ConcurrentGuard guard(mutex);
    queue.push_back(item);
  }
  /*
   * pop the front of the queue. if queue is empty, a C string exception is thrown
   */
  T pop(void) {
    ConcurrentGuard guard(mutex);
    if (queue.empty())
      throw "Empty Queue!";
    T result = queue.front();
    queue.pop_front();
    return result;
  }
  bool empty(void) {
    return queue.empty();
  }
private:
  std::list<T> queue;
  pthread_mutex_t mutex;
};
```cpp
int main(int argc, char *argv[]) {
    ConcurrentQueue<int> q;
    q.push(4);
    q.push(5);
    q.push(1);
    while (!q.empty()) {
        std::cout << "popped off " << q.pop() << std::endl;
    }
    return 0;
}
```

POSIX Threads provide all of the basics required for concurrent programming. However, there are some limitations. While supported on most Unix and Unix-like operating systems, POSIX Threads are not supported on other popular platforms such as Microsoft Windows. Another disadvantage of POSIX Threads is that it is a low-level API. This forces programmers to manage thread creation and communication manually which can be tedious, repetitive and error-prone.

**OpenMP**

The OpenMP specification defines a high-level API for shared-state concurrency. It is designed with multiprocessor or multi core computers in mind. The idea is to direct the compiler to divide algorithms into multiple streams that run in parallel. The number of threads may be determined at run time based on the number of CPU cores present.

OpenMP is a language extension to C, C++ or Fortran. In C or C++, the program contains `#pragma omp` directives. These directives specify such things as parallel control structures, variable sharing, synchronization and runtime parameters.

The example below demonstrates a simple for loop directed to run in parallel. On each pass
through the loop, the identifier of the current thread is printed along with the value of the
loop counter.

```c
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h>

int main(int argc, char *argv[]) {
    size_t count = 1000 * 1000;
    int nthreads = 1;

    if (argc > 1)
        count = (size_t)atoi(argv[1]) * 1000;

    int mycount = 0;

    // used for timing the operation (total time)
    struct timeval begin, end;

    gettimeofday(&begin, NULL);

    // a section to run in multiple threads
    #pragma omp parallel
    {
        if (omp_get_thread_num() == 0)
            nthreads = omp_get_num_threads();
        #pragma omp for
        // run this for loop in parallel
        for (mycount=0; mycount<count; mycount++)
            ;
    }

    gettimeofday(&end, NULL);

    int diff = (end.tv_sec - begin.tv_sec) * 1000000 + (end.tv_usec -
        begin.tv_usec);
    printf("%u operations using %u threads took %d us\n", count, nthreads, diff);

    return 0;
}
```
This code performs the exact same operations as the first POSIX Threads example. However, the code is considerably simpler and easier to understand.

The above program was compiled using GCC 4.2 on Ubuntu 7.10, and run with 1,000,000 operations using 1, 2, and 4 threads. The results, summarized in Table 1, demonstrate the immediate effect of additional threads to the OpenMP library on the dual-core computer. The result from 2 threads shows an 87% efficiency. The number of threads in a run was controlled by the environment variable OMP_NUM_THREADS.

<table>
<thead>
<tr>
<th>OMP_NUM_THREADS</th>
<th>Number of Threads</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>62882</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>35875</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>33516</td>
</tr>
</tbody>
</table>

*Table 1: OpenMP For Loop Results on Dual-Core*

OpenMP contains directives to split a sequential task in two different ways (OpenMP, 2005). The FOR directive divides a loop among threads by splitting the *data*, while the SECTIONS directive splits according to *function*. There are also directives for critical sections, atomic operations, and more. See (OpenMP, 2005) for the full description of the API.

As demonstrated above, the number of threads of a parallel section can be set at runtime. The default is the number of CPU cores.

OpenMP is a language *extension*, and as such requires support from the C, C++, or Fortran compiler. This limits the number of platforms that can utilize OpenMP, and makes portability more difficult. At present, Microsoft Visual C++ 2005 and 2008, GCC 4.2 and ICC 9.0 and 10.0 all support OpenMP.
Concurrency in Java

The Java programming language and the JVM have excellent support for concurrent programming. In Java, Thread objects exist and have the same semantics on all platforms.

The primary concurrency model for Java programs is shared state with locking. Java has had synchronization and object wait/notify since its beginning. This is essentially the Monitor Object Design Pattern of (Schmidt, 2000).

Java versions from 1.5 (Java 5) have a well-defined memory model that defines how shared state concurrency must behave in order to create reliable, highly concurrent programs. Such things as object synchronization, thread safety, volatile variables have been rigourously analysed and standardized.

Also since Java 5, the Java libraries have much richer support for concurrency. There are many new container classes that support concurrency in a variety of ways. In the past containers had to be synchronized, but now there are concurrent collections, copy-on-write collections, and blocking queues.

The libraries have been enriched with new thread executors. Several classes of thread pool allow separation of concurrency policy from task definition.

Finer grained support for locks allow for much better concurrent performance. Additional classes for semaphores, latches, and barriers enhance task synchronization and flow control.

New interfaces like Callable and Future that support both synchronous and asynchronous task execution. Exceptions from asynchronous operations are passed back to the calling thread when using Futures.
Concurrency in Erlang

Erlang is a functional programming language designed with concurrency in mind. It is used to develop “concurrent, real-time, distributed fault-tolerant systems.” (Armstrong 1996). Some of the design goals of Erlang include: concurrency, distributed programming, real-time, high-availability, and garbage-collection.

In particular interest to this paper, is Erlang's support for concurrency. Recent versions of Erlang (since OTP 5.5 R11B) have support for SMP and multi-core CPUs.

Erlang is a functional programming language. Some of the characteristics of a functional language, including Erlang, are:

- Lack of side effects: Programs are divided into functions which have no effect, other than input/output, on anything outside of their scope. The result is that functions are automatically thread-safe. In particular, there is no global program state, only local variables.

- Recursion: Repetitive tasks are accomplished with recursion rather than iteration. In Erlang, there are no loop constructs. Iteration must be implemented with recursion. See the example below.

- Referential Transparency: The value of a function does not depend upon the context in which it is called. (Armstrong, 1996). This implies that the order of evaluation of functions does not affect their results, and so expressions such as f(x) + g(y) do not depend on the order of evaluation.

Processes are the backbone of Erlang's concurrency system. An Erlang program can easily and cheaply create processes. These processes can communicate by passing messages. There
is no other inter-process communication mechanism in Erlang. This greatly simplifies the
program because there are no chances of the all-too-common concurrency problems of
deadlock, livelock and race conditions.

The example below shows how to create (spawn) and end a process, as well as send and
receive messages from the spawned process.

%%% p_reverse - demonstrate process spawning and message passing.
%%% creates a process that receives an atom and returns
%%% the atom reversed, by using the reverse/0 function.
p_reverse() ->
  %% first we create some data to work with
  Out_msg = abcdefg,
  %% create the process, running examples:reverse()
  Pid2 = spawn(examples, reverse, []),
  %% send the message to the spawned process, giving self() as
  %% return address
  Pid2 ! {self(), Out_msg},
  %% receive the response.
  receive
    {Pid2, In_msg} ->
      io:format("~w reversed is ~w.~n", [Out_msg, In_msg])
  end,
  %% stop the other process.
  Pid2 ! stop.

%%% reverse - receive atom in message, reverse it and continue.
%%% stop when atom 'stop' is received.
reverse() ->
  receive
    {From, Msg} ->
      In_str = atom_to_list(Msg),
      Rev_str = lists:reverse(In_str),
      Rev = list_to_atom(Rev_str),
      From ! {self(), Rev},
      reverse();
  stop ->
    true
  end.
CHAPTER V

CASE STUDY: CONCURRENCY IN SORTING ALGORITHMS

Parallel Quicksort

In order to verify the principle of parallelism in sort, a parallel version of quicksort should be developed. The results can be compared with those found in (Garcia, 2005).

The quicksort algorithm uses a “divide and conquer” strategy. In the “divide” phase, the input data set is split into 3 parts: a single element known as the pivot, the subset of elements less than the pivot, and the subset of elements greater than or equal to the pivot.

In the “conquer” phase, the 2 subsets are each sorted by recursively calling quicksort on them. The resulting sorted subsets are simply joined together. After this join, the result contains the same elements as the input, but in sorted order.

The test implementation of quicksort in Erlang looks like this:

```erlang
%% sort: sort a list of comparable items.
%%
%% param: List of items to sort
%%
%% handle the empty list
sort([]) ->
    [];
%% at least one item in the list
sort([Pivot|Rest]) ->
    %% divide: split the list into those less than the Pivot and
    %% those greater than (or equal to) the Pivot
    Lessthan = [Item || Item <- Rest, Item < Pivot],
    Morethan = [Item || Item <- Rest, Item >= Pivot],
    %% conquer: recursively call sort on the split parts, and join.
    sort(Lessthan) ++ [Pivot|sort(Morethan)].
```
From (Goodrich, 2001) we examine the running time of quicksort, and find that it is $O(n \log n)$ in the average case, but $O(n^2)$ for the worst case. In fact, the worst case running time can be observed by providing an already sorted input. The worst-case running time is a result of $n^2$ comparison operations required by the “divide” portion of the algorithm.

By examining the quicksort algorithm from the top down, we find two initial candidates for running in parallel: the “divide” operation and the “conquer” operation. First, we consider the “conquer” operation. In our Erlang version of quicksort, we perform two sort operations, one on each subset `LessThan` and `GreaterThan`. The first step to parallel quicksort then, will be to execute these sort operations in parallel.

In Erlang, parallelism is accomplished by executing a function in another Erlang process. Communication between processes is accomplished by sending and receiving messages. So, we must first define the function, `proc_sort` that can be called in a new process, sort a list and send back the result as a message.

```erlang
%% Proc_sort: receive a list as a message, return the sorted list
%%            as response.
proc_sort() ->
    receive
        {From, L} ->
            From ! {self(), sort(L)}
    end.
```

Next, we must define the `parallel_sort` function. The steps are:

- Divide the input into `Pivot`, `LessThan`, `MoreThan`,
- Spawn a new process which will sort `LessThan`
- Sort `MoreThan` in the current process
- Wait for the spawned process to send back its result
- Join the three parts together and return the result.

In Erlang, it looks like this:

```
%% parallel_sort: sort a list of comparable items in 2 parallel processes.
%%
%% params: List of unsorted items.
%%
%% handle the empty list
parallel_sort([]) -> [];
%% at least one element
parallel_sort([Pivot|Rest]) ->
    %% divide: same as in sort
    Lessthan = [Item || Item <- Rest, Item < Pivot],
    Morethan = [Item || Item <- Rest, Item >= Pivot],
    %% spawn a process to handle the "Lessthan" part
    Pid0 = spawn(quick_sort, proc_sort, []),
    Pid0 ! {self(), Lessthan},
    %% handle the "Morethan" part in the current process
    SortedMore = parallel_sort(Morethan),
    %% get the result of the other process (the sorted Lessthan)
    receive
        {Pid0, SortedLess} ->
            %% Returned the joined parts
            SortedLess ++ [Pivot|SortedMore]
    end.
```

It is significant that each recursive call to `parallel_sort` (other than case of the empty list) causes a new process to be spawned. Thus there is one process created for each level of the search. This means that on average there will be $O(\log n)$ processes, with $O(n)$ in the worst case.

In the Erlang environment, spawning processes is cheap. However, other operations that might be cheap in an imperative language may be expensive. For example, in Erlang adding an element to the end of a list is an $O(n)$ operation because the entire list has to be searched
for the end. On the other hand, adding an element to the beginning of a list can be done in constant time.

Erlang also supports a feature known as “tail recursion optimization.” This means that a recursive function call that is the last statement in a function will not be called in the usual manner using the stack. Rather, the stack space from the previous call is overwritten since it is no longer needed. It is important to emphasize that this recursive call must be the last statement executed.

These idiosyncrasies affect the implementation in Erlang of the quicksort and the other algorithms. Several minor adjustments were made along the way to accommodate them.

There are several common enhancements to the quicksort algorithm. For example, there is a version that sorts in-place, and often the pivot is chosen as the median of the first, middle and last element of the input list to prevent the algorithm from taking quadratic time for inputs that are already sorted. These enhancements were not implemented.

The next step is to extend the parallelism of quicksort by splitting the “divide” operation, and running half in the spawned process. The “divide” operation is implemented as two list comprehensions, producing the LessThan and MoreThan subsets. Thus, the entire “divide and conquer” is done in parallel, with the final join done in the original process.

```erlang
%% Parallel sort a list using quicksort, by recursively calling parallel_sort/1 on sublists using quicksort algorithm.
%%
%% Case of empty list
parallel_sort([]) -> [];
%% Case of single element
parallel_sort([_] = L) -> L;
%% Case of 2 element, first is less or equal
```
parallel_sort([X, Y] = L) when X =< Y ->
    L;
%%% Case of 2 entities, first is greater
parallel_sort([X, Y]) ->
    [Y, X];
%%% Remaining cases (more than 2 elements)
parallel_sort([Pivot|Rest]) ->
    Me = self(),
    spawn(fun() -> Me ! sort([X || X <- Rest, X < Pivot]) end),
    SortedMore = parallel_sort([X || X <- Rest, X > Pivot]),
    receive
        SortedLess ->
            SortedLess ++ [Pivot|SortedMore]
    end.

Introsort

The name *introsort* is a contraction of *introspective sort*. The algorithm is introspective, in
that it monitors its own progress, and changes course in case it detects a long running time.

In particular, introsort starts out using quicksort, and monitors the level of the recursion.
Should the recursion reach the pre-defined limit, the original introsort algorithm aborts the
quicksort operation, and sorts the original input list using heapsort. The limit for recursion is
set at $2\log n$ where $n$ is the size of the data input.

The nature of the heapsort algorithm makes it inherently difficult to split into independent,
parallel sections. Since mergesort is similar in performance to heapsort, and can be easily
split into parallel execution paths, the implementation of introsort was modified to use
mergesort instead of heapsort for pathological data sets.

The Erlang code looks like this:

```erlang
sort(List) ->
    case catch sort(List, max_depth(List), 0) of
        timeout ->
            merge_sort:sort(List);
```
Result ->
  Result
end.

%%% Empty list
sort([], _, _) ->
  [];
%%% Single element
sort([_ = L, _, _]) ->
  L;
%%% Two elements
sort([X, Y], _, _) ->
  if
    X < Y ->
      [X, Y];
    true ->
      [Y, X]
  end;
%%% Reached the max depth - abort
sort(_, Max_depth, Current_depth) when Current_depth > Max_depth ->
  throw(timeout);
%%% General case
sort([Pivot|Rest], Max_depth, Current_depth) ->
  sort([X || X <- Rest, X < Pivot], Max_depth, Current_depth+1) ++ [Pivot|
  sort([X || X <- Rest, X >= Pivot], Max_depth, Current_depth+1)].

Note that each call to sort passes along the maximum and current depths of recursion (Max_depth, Current_depth). The maximum depth allowed is 2log₂n, where n is the size of the input.

The parallel implementation of introsort uses the same model as parallel quicksort. The difference is, a second process is created for the input subset that is less than the pivot. If either of the processes reach the maximum depth, it will abort and that part of the input will be sorted using mergesort. The final join will be done in the main process.

parallel_sort(List) ->
  parallel_sort(List, max_depth(List), 0).

safe_sort(List, Max_depth, Current_depth) ->
  case catch sort(List, Max_depth, Current_depth) of
timeout ->
    merge_sort:parallel_sort(List);
Result ->
    Result
end.

%% Empty list
parallel_sort([], _, _) ->
    [];
%% Single element
parallel_sort([_] = L, _, _) ->
    L;
%% Two elements
parallel_sort([X, Y], _, _) ->
    if
        X < Y ->
        [X, Y];
    true ->
        [Y, X]
    end;
%% Reached the max depth - abort
parallel_sort(_, Max_depth, Current_depth) when Current_depth > Max_depth ->
    throw(timeout);
%% General case
parallel_sort([Pivot|Rest], Max_depth, Current_depth) ->
    Me = self(),
    spawn(fun() -> Me ! safe_sort([X || X <- Rest, X < Pivot], Max_depth, Current_depth+1) end),
    SortedMore = safe_sort([X || X <- Rest, X >= Pivot], Max_depth, Current_depth+1),
    receive
        SortedLess ->
            SortedLess ++ [Pivot|SortedMore]
    end.

Thus, as long as the sort is well-behaved, the level of parallelism for introsort is the same as parallel quicksort. However, for pathological cases, parallel introsort behaves like parallel mergesort.

The parallel mergesort implementation in Erlang has the first “divide” operation run in parallel but lower level divides and the final merge are run linearly. Since the merge is run
at each recursive step, in fact only the final merge is run in a single process. The implementation of parallel mergesort looks like this:

```erlang
parallel_sort([]) -> [];
parallel_sort([] = L) -> L;
parallel_sort([X, Y]) ->
  if X =< Y ->
    [X, Y];
  true ->
    [Y, X]
  end;
parallel_sort(L) ->
  N = length(L),
  Me = self(),
  spawn(fun() -> Me ! sort(lists:sublist(L, 1, N div 2)) end),
  L2 = sort(lists:sublist(L, N div 2 + 1, N)),
  receive
    L1 ->
      lists:reverse(merge(L1, L2, []))
  end.
merge(L1, L2) ->
  lists:reverse(merge(L1, L2, [])).
merge([], [], Result) -> Result;
merge([], [H2|R2], Result) ->
  merge([], R2, [H2|Result]);
merge([H1|R1], [], Result) ->
  merge(R1, [], [H1|Result]);
merge([H1|R1]=L1, [H2|R2]=L2, Result) ->
  if H1 =< H2 ->
    merge(R1, L2, [H1|Result]);
  true ->
    merge(L1, R2, [H2|Result])
  end.
```

The mergesort algorithm can be further parallelized with some effort, but this algorithm
provides a significant level of concurrency when run on a dual-core processor.

Summary of Results

The introsort algorithm has been extended to run in two separate threads concurrently in Erlang. For the main sort algorithm, quicksort, this entails running part of the “divide and conquer” operations in a separate Erlang process. The algorithm was modified to use mergesort instead of heapsort since a concurrent version of mergesort is easy to implement.

The parallel version of introsort runs about 32% faster than the sequential version for random input on a dual core AMD Athlon 64 X2 CPU, while showing a 0% improvement on a single core PowerPC G5. For inputs that are known to be pathological for quicksort, the net improvement of parallel introsort is about 18% on the dual core, 0% on the single core. Detailed results can be found below.
CHAPTER VI

CASE STUDY RESULTS

Testing Methodology

A test driver program was developed in Erlang and executed in the Erlang/OTP R12B environment, without High Performance Erlang (HiPE). The program allows the tester to test one of a set of sort algorithms on an input file.

The Erlang program can run one of the following sort algorithms, selected by an input parameter:

- quicksort,
- parallel quicksort,
- heapsort,
- introsort,
- parallel introsort,
- mergesort and
- parallel mergesort

The input files contain one integer per line. The files were generated in advance, and contain between 10,000 and 160,000 integers in multiples of 10,000. There is one input file of each size containing random data, and one containing already sorted data since this is known to cause pathological $O(n^2)$ behaviour to the quicksort algorithm.
An additional shell script was created to run the program in batch, meaning all of the dataset files of one type, from 10,000 to 160,000 integers. Either the random or pathological datasets can be selected, as well as one or more sort algorithms.

After the raw sorting times were recorded, the value of $C$ in $Cn \log n$ was calculated for each value. Since all of the algorithms have average run times of $O(n \log n)$, the algorithms can be compared using this value of $C$. When comparing the sequential version against the parallel version, a percent difference for each data set is calculated as follows:

$$\Delta_n \% = \frac{100 \times (\text{SEQUENTIAL}_n - \text{PARALLEL}_n)}{\text{SEQUENTIAL}_n}$$

This value of $\Delta_n$ measures the percent improvement of parallelism.

**Single Core Platform**

The single core platform consists of an Apple iMac with a single core 2.1 GHz PowerPC G5, 1.5 GB RAM. The CPU has 512 kB of L2 cache and a 700 MHz bus speed. The operating system is Mac OS X 10.4.10.

**Dual Core Platform**

The dual core platform consists of a generic dual core AMD Athlon 64 X2 based computer, with 1 GB RAM. The CPU has 512 kB of L2 cache and a 2100 MHz bus speed. The operating system is Ubuntu 7.10 Gutsy Gibbon for AMD64, with Linux kernel 2.6.22.

Each core of the dual core AMD is roughly equivalent to the PowerPC G5 in computing power.
**Quicksort Results**

For the randomized data sets, on the single core computer there was little difference between the sequential and the parallel versions of quicksort, as can be seen in the table below.

<table>
<thead>
<tr>
<th>N</th>
<th>C – Sequential Quicksort</th>
<th>C – Parallel Quicksort</th>
<th>% improvement of Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>2.4</td>
<td>2.6</td>
<td>-8.33</td>
</tr>
<tr>
<td>20000</td>
<td>2.04</td>
<td>2</td>
<td>1.89</td>
</tr>
<tr>
<td>30000</td>
<td>1.76</td>
<td>1.9</td>
<td>-7.69</td>
</tr>
<tr>
<td>40000</td>
<td>1.73</td>
<td>1.58</td>
<td>9.01</td>
</tr>
<tr>
<td>50000</td>
<td>1.73</td>
<td>1.73</td>
<td>0</td>
</tr>
<tr>
<td>60000</td>
<td>1.66</td>
<td>1.7</td>
<td>-2.26</td>
</tr>
<tr>
<td>70000</td>
<td>1.8</td>
<td>1.85</td>
<td>-3.02</td>
</tr>
<tr>
<td>80000</td>
<td>2.24</td>
<td>2.41</td>
<td>-7.62</td>
</tr>
<tr>
<td>90000</td>
<td>1.78</td>
<td>1.84</td>
<td>-3.19</td>
</tr>
<tr>
<td>100000</td>
<td>1.66</td>
<td>1.76</td>
<td>-6.04</td>
</tr>
<tr>
<td>110000</td>
<td>1.66</td>
<td>1.78</td>
<td>-7.26</td>
</tr>
<tr>
<td>120000</td>
<td>1.62</td>
<td>1.59</td>
<td>1.74</td>
</tr>
<tr>
<td>130000</td>
<td>1.61</td>
<td>1.55</td>
<td>3.62</td>
</tr>
<tr>
<td>140000</td>
<td>1.62</td>
<td>1.62</td>
<td>0</td>
</tr>
<tr>
<td>150000</td>
<td>1.56</td>
<td>1.62</td>
<td>-3.93</td>
</tr>
<tr>
<td>160000</td>
<td>1.81</td>
<td>1.7</td>
<td>5.95</td>
</tr>
</tbody>
</table>

*Table 2: Value of C for Quicksort on Single Core*

On the dual core computer, the results show some improvement of the parallel algorithm.

The average decrease in the value of C is about 24%.
<table>
<thead>
<tr>
<th>N</th>
<th>C – Sequential Quicksort</th>
<th>C – Parallel Quicksort</th>
<th>% improvement. of Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>3.5</td>
<td>2.4</td>
<td>31.43</td>
</tr>
<tr>
<td>20000</td>
<td>3.11</td>
<td>2</td>
<td>35.8</td>
</tr>
<tr>
<td>30000</td>
<td>2.98</td>
<td>1.81</td>
<td>39.39</td>
</tr>
<tr>
<td>40000</td>
<td>2.67</td>
<td>1.48</td>
<td>44.44</td>
</tr>
<tr>
<td>50000</td>
<td>2.67</td>
<td>1.97</td>
<td>26.43</td>
</tr>
<tr>
<td>60000</td>
<td>2.55</td>
<td>1.85</td>
<td>27.57</td>
</tr>
<tr>
<td>70000</td>
<td>2.74</td>
<td>2.75</td>
<td>-0.28</td>
</tr>
<tr>
<td>80000</td>
<td>1.31</td>
<td>1.15</td>
<td>12.5</td>
</tr>
<tr>
<td>90000</td>
<td>1.26</td>
<td>0.85</td>
<td>32.58</td>
</tr>
<tr>
<td>100000</td>
<td>1.28</td>
<td>1.09</td>
<td>14.51</td>
</tr>
<tr>
<td>110000</td>
<td>1.21</td>
<td>1.04</td>
<td>14.02</td>
</tr>
<tr>
<td>120000</td>
<td>1.19</td>
<td>0.99</td>
<td>16.55</td>
</tr>
<tr>
<td>130000</td>
<td>1.25</td>
<td>0.85</td>
<td>31.98</td>
</tr>
<tr>
<td>140000</td>
<td>1.17</td>
<td>0.99</td>
<td>15.58</td>
</tr>
<tr>
<td>150000</td>
<td>1.15</td>
<td>0.92</td>
<td>19.95</td>
</tr>
<tr>
<td>160000</td>
<td>1.1</td>
<td>0.87</td>
<td>20.67</td>
</tr>
</tbody>
</table>

*Table 3: Value of C for Quicksort on Dual Core*

On average, the parallel quicksort is about 24% faster on the dual core. The improvement can be seen clearly in the graphs below, which also indicate an irregularity in the data set of 70,000 integers.
For the basic quicksort algorithm, it is possible to produce pathological $O(n^2)$ behaviour by providing an already sorted input. The following tables show how the parallel implementation of quicksort actually produces significantly worse results than the sequential version. There is no difference between the single and dual core CPUs in this regard.
Pathological Quicksort (ms)  |  Parallel Pathological Quicksort (ms)
--- | ---
10000 | 5207 | 6969
20000 | 18811 | 30327
30000 | 41404 | 58393
40000 | 100218 | 100970
50000 | 201720 | 281528
60000 | 166394 | 248535
70000 | 251494 | 379994
80000 | 307913 | 432230
90000 | 608989 | 511519
100000 | 595631 | 655303

*Table 4: Quicksort Sort Times, Single Core, Pathological Data*

Pathological Quicksort (ms)  |  Parallel Pathological Quicksort (ms)
--- | ---
10000 | 3795 | 5747
20000 | 16817 | 23868
30000 | 47967 | 55829
40000 | 76448 | 97937
50000 | 107693 | 150654
60000 | 165001 | 218384
70000 | 244157 | 299884
80000 | 285307 | 389644
90000 | 369245 | 495515
100000 | 474194 | 611369

*Table 5: Quicksort Sort Times, Dual Core, Pathological Data*

**Introsort Results**

For the randomized data, the introsort results are similar to quicksort. For pathological data however, where parallel quicksort exhibited *worse* running times than sequential quicksort, parallel introsort shows *the same* running time as the sequential version. Results are
summarized in Table 5 and Figure 3.

<table>
<thead>
<tr>
<th>N</th>
<th>C - Introsort</th>
<th>C - Parallel Introsort</th>
<th>C - Pathological Introsort</th>
<th>C - Parallel Pathological Introsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>2.9</td>
<td>2.7</td>
<td>8.1</td>
<td>8.3</td>
</tr>
<tr>
<td>20000</td>
<td>2.57</td>
<td>2.54</td>
<td>7.11</td>
<td>6.96</td>
</tr>
<tr>
<td>30000</td>
<td>2.03</td>
<td>1.76</td>
<td>11.89</td>
<td>11.35</td>
</tr>
<tr>
<td>40000</td>
<td>1.92</td>
<td>1.94</td>
<td>5.93</td>
<td>6.16</td>
</tr>
<tr>
<td>50000</td>
<td>2.12</td>
<td>1.95</td>
<td>6.03</td>
<td>5.9</td>
</tr>
<tr>
<td>60000</td>
<td>1.86</td>
<td>2.25</td>
<td>5.51</td>
<td>5.5</td>
</tr>
<tr>
<td>70000</td>
<td>1.94</td>
<td>2.25</td>
<td>5.56</td>
<td>5.61</td>
</tr>
<tr>
<td>80000</td>
<td>2.31</td>
<td>2.33</td>
<td>5.64</td>
<td>5.63</td>
</tr>
<tr>
<td>90000</td>
<td>1.85</td>
<td>1.84</td>
<td>6.21</td>
<td>5.98</td>
</tr>
<tr>
<td>100000</td>
<td>1.85</td>
<td>1.92</td>
<td>5.27</td>
<td>5.38</td>
</tr>
<tr>
<td>110000</td>
<td>1.7</td>
<td>1.72</td>
<td>6.56</td>
<td>6.53</td>
</tr>
<tr>
<td>120000</td>
<td>1.99</td>
<td>1.97</td>
<td>5.41</td>
<td>5.27</td>
</tr>
<tr>
<td>130000</td>
<td>1.9</td>
<td>1.84</td>
<td>5.13</td>
<td>5.46</td>
</tr>
<tr>
<td>140000</td>
<td>1.86</td>
<td>2.12</td>
<td>5.86</td>
<td>5.42</td>
</tr>
<tr>
<td>150000</td>
<td>2.12</td>
<td>1.93</td>
<td>5.48</td>
<td>6.1</td>
</tr>
<tr>
<td>160000</td>
<td>2.08</td>
<td>1.91</td>
<td>5.5</td>
<td>5.44</td>
</tr>
</tbody>
</table>

Table 6: Value of C for Introsort on Single Core Computer

Figure 3: Introsort on Single Core
On the dual core computer, parallel introsort demonstrates about 32% improvement for the randomized data and about 18% improvement for the pathological data. Results are summarized in Table 6 and Figure 4.

<table>
<thead>
<tr>
<th>N</th>
<th>Introsort</th>
<th>Parallel Introsort (Path. Data)</th>
<th>Parallel Introsort (Path. Data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>77</td>
<td>30</td>
<td>97</td>
</tr>
<tr>
<td>20000</td>
<td>128</td>
<td>82</td>
<td>199</td>
</tr>
<tr>
<td>30000</td>
<td>247</td>
<td>110</td>
<td>334</td>
</tr>
<tr>
<td>40000</td>
<td>289</td>
<td>150</td>
<td>436</td>
</tr>
<tr>
<td>50000</td>
<td>450</td>
<td>111</td>
<td>588</td>
</tr>
<tr>
<td>60000</td>
<td>217</td>
<td>169</td>
<td>335</td>
</tr>
<tr>
<td>70000</td>
<td>277</td>
<td>246</td>
<td>401</td>
</tr>
<tr>
<td>80000</td>
<td>355</td>
<td>234</td>
<td>486</td>
</tr>
<tr>
<td>90000</td>
<td>330</td>
<td>279</td>
<td>523</td>
</tr>
<tr>
<td>100000</td>
<td>392</td>
<td>320</td>
<td>610</td>
</tr>
<tr>
<td>110000</td>
<td>441</td>
<td>294</td>
<td>674</td>
</tr>
<tr>
<td>120000</td>
<td>485</td>
<td>385</td>
<td>791</td>
</tr>
<tr>
<td>130000</td>
<td>445</td>
<td>333</td>
<td>829</td>
</tr>
<tr>
<td>140000</td>
<td>515</td>
<td>488</td>
<td>844</td>
</tr>
<tr>
<td>150000</td>
<td>620</td>
<td>476</td>
<td>916</td>
</tr>
<tr>
<td>160000</td>
<td>617</td>
<td>409</td>
<td>998</td>
</tr>
</tbody>
</table>

*Table 7: Value of C for Introsort on Dual Core Computer*
Summary

The expected results of the case study have been demonstrated with an Erlang program. This program makes use of Message Passing Concurrency to speed up sorting integers on a dual core computer system.

A simple, high abstraction level application of concurrency has been applied to the quicksort algorithm in Erlang. It has resulted in a modest but significant improvement of 24% in sorting time for 2 cores. This confirms the results of (Garcia, 2005). However, in the worst case, the parallel quicksort performs even worse than the sequential.

This is not so for introsort where a qualitative improvement has been observed. The parallel version of introsort utilizes parallel mergesort as well as parallel quicksort. The result for 2 cores is faster than the sequential by around 18% for the already sorted input, which causes worst case behaviour in quicksort. The results for random input on 2 cores were about 32%
improvement of the parallel version. On the single core CPU, the parallel version showed no improvement but also did not show degradation.
CHAPTER VII

CONCLUSION

Discussion

The high level approach used in the case study has resulted in a significant improvement of the introsort algorithm on a dual core CPU. This is true for both random input as well as for input that is known to be pathological for quicksort. However, only 32% improvement was observed for double the computing power. This represents about 66% efficiency.

Further analysis of the algorithms may reveal other opportunities to increase this efficiency. Perhaps greater control over the number of currently running processes, together with a finer breakdown of functions would allow greater utilization of both cores. A particular example of lost opportunity from the mergesort implementation is the merge operation on the sorted subsets. In the current implementation this is performed sequentially. Another example from mergesort, is that the data-set is split into 2 equally sized segments, and each is sorted in a separate process. It is quite possible for one of these sorts to finish significantly ahead of the other, thus wasting one core's power.

Quicksort is also a “divide and conquer” sort. Therefore, it may be possible to find a strategy to overcome these limitations that can be applied to both mergesort and quicksort.

Taking a sequential algorithm and writing in parallel may be made less efficient due to CPU operations which take longer in the parallel version. There are many causes for this. Hardware limitations and compiler optimizations are two of them.
Recommendations

It would be interesting to compare the Message Passing approach of the case study to Shared State Concurrency and to Software Transactional Memory. How each of them scale over large numbers of cores, how easy they are to conceptualize, how easy they are to apply, how error-prone they are, are obvious questions.

Recommended for further research is to improve the efficiency of the parallel sorting algorithms developed in the case study. There are two types of problems that may need to be solved. The first is that parts of the algorithms have not yet been made to run in parallel. For example, the final merge of the mergesort might be implemented in parallel.

The second type of problem with the efficiency of the sorting algorithms is CPU operations which run more slowly in the parallel implementation. For example, simultaneous memory access is often not possible for multiple cores or CPU's. A possible solution might be the optimization of CPU cache access. However, CPU caches can vary in size, architecture, and cache algorithm, making this difficult to do in a general way.

The case study is customized to a dual-core CPU. This was done to limit thread management and thread resources. In the future it is expected that computers will contain far more than two cores. Therefore, it is recommended that the solutions presented in the case study be generalized for multi-core CPU's.

The case study has demonstrated tangible benefit to algorithm adaptation for dual core CPU's. It is recommended therefore that programming libraries be updated with parallel algorithms for sorting.
CHAPTER VIII

REFERENCES


Goetz, B. et al. (2006). *Java Concurrency in Practice.* Addison Wesley. USA.


