OPTIMIZING APPLICATION PERFORMANCE ON MULTICORE PROCESSORS USING CILK ++

Sponsored By: Persistent Systems Ltd,
Pune, India
Contents

CHAPTER 1. INTRODUCTION .................................................................7
  1.1 Serial Programming Execution ..................................................7
  1.2 Hardware level parallelism ......................................................7
    1.2.1. Bit level parallelism ....................................................7
    1.2.2. Instruction level parallelism ...........................................8
    1.2.3. Thread level parallelism ...............................................8
  1.3. Evolution of multi core processors .......................................10
  1.4. Comparison of Single core and multi core processors .............10
  1.5. Software level parallelism ..................................................12
    1.5.1. Data parallelism ......................................................13
    1.5.2. Task parallelism .....................................................14
  1.6. Algorithmic complexity measures and Parallelism .................15

CHAPTER 2. TECHNICAL RESEARCH ..................................................18
  2.1. Technology used ....................................................................18
    2.1.1. Research At Cilk Arts ................................................18
    2.1.2. Cilk++ .......................................................................18
    2.1.3. Components of Cilk++ ................................................19
    2.1.3.4. Limitations of Other Schedulers ...............................21
    2.1.4. The Cilk++ Race Detector .............................................23
    2.1.5. Cilk++ Reducers and Hyperobjects ................................24
    2.1.6. Features of Cilk++ .....................................................25
    Miser Limitations .....................................................................25
    2.1.7. Benefits of Cilk++ Technology .......................................26
CHAPTER 3. PROJECT PLAN ................................................................. 38
3.1. Purpose of the Project ............................................................... 38
3.2 Project Scope ........................................................................... 38
3.3 Project schedule ...................................................................... 39
3.4 Documentation ......................................................................... 40

CHAPTER 4. SOFTWARE REQUIREMENT SPECIFICATION ............ 41
4.1 Project Scope ........................................................................................................... 41
4.2 Overall Description ................................................................................................. 41
  4.2.1 Project Perspective ............................................................................................ 41
  4.2.2 Operating Environment .................................................................................... 42
  4.2.3 Implementation Constraints ............................................................................ 42
4.3 Functional Requirements ......................................................................................... 43
4.4 Other Nonfunctional Requirements ......................................................................... 44
  4.4.1 Performance Requirements ............................................................................ 44
4.5 Deployment scenario ............................................................................................... 45

CHAPTER 5. DESIGN SPECIFICATION ................................................................. 46
  5.1 Activity Diagram ................................................................................................... 46
  5.2 State Diagram ....................................................................................................... 46
  5.3 Block Diagram of Cilk++ Environment ................................................................. 48

CHAPTER 6. IMPLEMENTATION DETAILS ......................................................... 49
  6.1 Cilkification of Mergesort ...................................................................................... 49
  6.2 Cilkification of MikeNet ......................................................................................... 57
    6. 2.1 Compile serial version of Mikenet and Regression testing ......................... 58
    6.2.2 Identifying performance bottlenecks ............................................................. 58
    6. 2.3 Adding Cilk++ keywords to expose parallelism ............................................ 60
    6. 2.4. Compile and run regression tests to verify serial Correctness .................... 65
    6. 2.5 Run Cilkscreen to verify parallel correctness and locate races ................. 66
    6. 2.6 Resolving races using reducers ................................................................. 66
  6.3 Parallel correctness .............................................................................................. 67
  6.4 Improvement in performance .............................................................................. 67
CHAPTER 7. TEST CASES ..................................................................................71

Test Case 1: Output of cilkified merge sort and its performance. .................... 71
Test Case 2: Performance of cilkified N queens program ............................... 71
Test Case 3: Performance of Mikenet after cilk_for keyword addition ............ 72
Test Case 4: Output of Mikenet after interchanging the loop ......................... 72
Test case 5: Performance of Mikenet application after using reducer .............. 73
Test case 6: Performance of Mikenet application after appropriate use of reducer
............................................................................................................................. 73

APPENDIX A. Intel VTune performance Analyzer ........................................ 74
GLOSSARY ........................................................................................................ 76
CONCLUSION .................................................................................................... 79
REFERENCES ................................................................................................. 80
List of Figures

Figure 1.1. Pipelining................................................................. 9
Figure 1.2. Thread level parallelism ............................................. 9
Figure 1.3. Instruction Level Parallelism........................................ 9
Figure 1.4. Processing data set on single CPU core ......................... 13
Figure 1.5. Processing data set on multiple CPU cores ...................... 14
Figure 2.1. Components of Cilk++ .............................................. 19
Figure 2.2. Running a Program under cilkscreen race detector ........... 24
Figure 2.3. Reported races ......................................................... 24
Figure 5.1. Activity diagram ....................................................... 46
Figure 5.3. State Diagram ........................................................... 47
Figure 5.3. Block Diagram of Cilk++ environment ............................ 48
Figure 7.1. Typical Workflow of cilkification ................................. 57
Figure 6.2. VTune Snapshot 1 ..................................................... 59
Figure 6.3. VTune Snapshot 2 ..................................................... 60

List of Tables

Table 1.1. Comparison of Single Core and Multi core Processors ........... 12
Table 3.1. Project Schedule ....................................................... 39
Table 6.1. Comparison of Serial and cilkified codes ............................ 70
CHAPTER 1. INTRODUCTION

1.1 Serial Programming Execution

Programs were written earlier for a serial execution on a single core processor, that is, a program consisted of a sequence of instructions, where each instruction executed one after the other. It ran from start to finish on a single core processor. For example, a quick sort algorithm began and executed each instruction after the other until it finished.\[11\]

However this serial programming has some drawbacks:-

1. **More idle time**

   The processor has to wait for an instruction to get over before it can proceed to next. If these instructions are I/O instructions the processor remains idle for a considerable amount of time which is wastage of processor time.\[11\]

2. **Processing speed is dependent on how fast data can move through hardware.**

   To overcome these drawbacks parallel programming came into picture. Parallelism can be achieved at hardware level and software level.\[11\]

1.2 Hardware level parallelism.

   The underlying hardware structure is capable of supporting certain tasks/instructions to run in parallel. Parallelism has been achieved in different forms at the hardware level.

1.2.1. **Bit level parallelism:** - It is a form of parallel computing based on increasing the word size of processor. Increasing the word size reduces the number of instructions the
processor must execute in order to perform an operation on variables whose sizes are greater than the length of the word. 4-bit microprocessors were replaced with 8-bit, then 16-bit, then 32-bit microprocessors. The advent of x86-64 architectures, have made 64-bit processors a commonplace.\cite{12}

1.2.2. Instruction level parallelism: - It is parallelism at machine-instruction level. ILP is a measure of how many, of the operations in a computer program can be performed simultaneously. The processor can reorder, pipeline instructions, split them into micro instructions and do aggressive branch predictions.\cite{12}

1.2.3. Thread level parallelism: - A thread is a unit of executable code. It shares all global variables and file descriptors of the parent process which allows the programmer to separate multiple tasks easily within a process. Each thread has its own stack, program counter and registers thereby having an individual copy of local variables. Thread based multitasking deals with concurrent execution of pieces of same program. It is parallelism at coarser scale. But even with thread level parallelism and instruction level parallelism the basic processing was still serial that is only one thread could be executed at a time.\cite{12}

The solution to this problem came in the form of simultaneous multi threading. Consider a Pentium processor that had two computing units, one for integer and the other for floating point calculations. A simultaneous multi threading provided for simultaneous execution of two threads, one of which had an integer computation and the other had floating point computation.

But simultaneous multi threading could also not provide true parallelism. As apart from the two calculating units the major part of any code still executed serially. With this in mind multi core processors were developed.
Figure 1.1. Pipelining \[18\]

Figure 1.2. Thread level parallelism \[18\]

Figure 1.3. Instruction Level Parallelism \[18\]
1.3. Evolution of multi core processors

Around 2003 the clock speed of single core processors reached its limit and could not be increased further for reasons that were related to fundamental physics. It was observed that clock speed beyond 5 GHz would melt the chips. In an effort to increase performance and to reduce power through multiprocessing, production of multiple processing cores per chip started. [2]

A multi-core processor is an integrated circuit (IC) to which two or more processors has been attached and the memory may be shared by the processors.

By enabling enhanced performance, reduced power consumption and more efficient simultaneous processing of multiple tasks, multi-core processors promise to improve the user experience in home and business environments. [2]

1.4. Comparison of Single core and multi core processors

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Single core processor</th>
<th>Multicore processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>One</td>
<td>Two, four, eight</td>
</tr>
<tr>
<td>Characteristics</td>
<td>Single core processor</td>
<td>Multicore processor</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Power consumption</td>
<td>▪ Smaller gates</td>
<td>▪ Much low power consumption per core</td>
</tr>
<tr>
<td></td>
<td>▪ Higher clock rates</td>
<td></td>
</tr>
<tr>
<td>Heat problems</td>
<td>▪ Extensive heat rejection of multiple kilowatts is undesirable from a thermal perspective</td>
<td>▪ Lower heat dissipation through decreased frequency.</td>
</tr>
<tr>
<td>Performance</td>
<td>Clock rate cannot be increased beyond a certain limit</td>
<td>▪ Low power processors are usually more efficient.</td>
</tr>
<tr>
<td>Parallelism</td>
<td>▪ Limited parallelism</td>
<td>▪ Good at ILP</td>
</tr>
<tr>
<td></td>
<td>▪ Interdependent instructions</td>
<td>▪ Great at thread level parallelism</td>
</tr>
<tr>
<td>Design time complexity</td>
<td>▪ Hard to design</td>
<td>▪ Reduced design time and complexity</td>
</tr>
<tr>
<td></td>
<td>▪ Large number of transistors</td>
<td>More efficient designs at no expense in backward compatibility.</td>
</tr>
<tr>
<td></td>
<td>▪ Complex methods to increase ILP</td>
<td></td>
</tr>
</tbody>
</table>
With the advent of multi core architectures, a fundamental change in the thinking of software developers is needed. Programmers should no longer create single-threaded applications that execute faster with each successive processor generation. In order to utilize the potential of multi core processors parallel programs i.e. programs that express units of work that can execute in parallel have to be developed. Most programmers today simply don’t think of their programs in terms of parallel tasks. [2]

Parallel programming brings along with it bugs like livelocks, deadlocks, data races, and lost wakeups. These bugs are mostly related to timing related issues and poses challenges to developers.

The multi core development places new demands on the software development industry:

- In order to exploit full capabilities of multi core architectures, developers need to adopt multi core software platforms.
- The developers need to train themselves in multithreading programming practices.

In the past, the challenge for software developers was to run software applications on high clock speeds but now the priority is given to drive more parallelism into software itself. [2]

### 1.5. Software level parallelism

Software has to keep pace with the hardware. Thus in order to exploit the capabilities of multi-core processors fully techniques to parallelize applications at software level arose.
1.5.1. **Data parallelism:** - It is a programming technique wherein large data set is split into smaller chunks that can be operated on in parallel. After the data has been processed, it is combined back into a single data set. With this technique, a process that typically would not be capable of utilizing multi-core processing power could be modified so that it uses all the processing power available.

In traditional programming methods, a large data set is processed on a single CPU core, while the other CPU cores remain idle. This is demonstrated in the figure below.\[18\]

![Figure 1.4. Processing data set on single CPU core \[12\]](image)

By using the programming technique of data parallelism, a large data set can be processed in parallel on multiple CPU cores.
1.5.2. Task parallelism: - The concurrent execution of independent tasks in software is known as task parallelism. For example, we have a Web browser and a word-processing program running on a single core processor at the same time. Although these applications run on separate threads, they ultimately share the same processor. In case these programs are run on a dual-core machine, these two applications essentially can run independently of one another. Although they may share some resources that prevent them from running completely independently, the dual-core machine can handle the two parallel tasks more efficiently.\textsuperscript{12}

These techniques of making programs run in parallel can have different synchronization requirements and are useful under different circumstances.

Three main paradigms used for the purpose are:-

Master-Slave: - In this, the main master thread launches several slave threads and allocates to each slave a portion of the work to be done. Once the work of all the slaves is done, a new batch of them can be started.\textsuperscript{12}

Work pile: - In this case each thread fetches a portion of work from some form of a queue, called the work pile, until there are no more entries in the work pile. \textsuperscript{12}
**Pipeline paradigm:** - It is based on the simple producer/consumer relation: one pipeline stage produces data for the next stage, which digests this data and on its turn passes it on to the next stage.\(^{[12]}\)

ILP and TLP as mentioned above is limited in providing parallelism because there is only one thread present for computation purposes. Thus there was a need for a computational model with multiple flows of control. Automatic parallelization has been successful on numeric FORTRAN programs where the data dependencies can be analyzed statically.

But parallelizing general purpose non-numeric C programs has not been successful because it requires pointer-disambiguation to statically analyze the dependencies. For this to be possible we need hardware and software mechanisms that eliminate the need for data dependence analysis to ensure correct program execution.\(^{[12]}\)

In addition to this, the execution threads should be mapped to the multiple cores of the processor by some technique.

To address such issues the software developers are developing programming languages that are meant to bring explicit parallelism into software sectors such as loops, graphics, networking, debuggers and libraries.\(^{[12]}\)

Ct, mpC and Cilk++ are some of these languages. We have used Cilk++ for achieving parallelism at software level.

### 1.6. Algorithmic complexity measures and Parallelism

Let 
- \( T_1 \): Time taken for one unit of work.
- \( T_p \): Execution time on \( p \) processors
- \( T_\infty \): span (Also called critical path length)

Project_BE_Report 15 Created in March 2009
Work is the total amount of time spent in all the instructions. Work law states that
\[ T_p \geq T_1 / T_p \]

Span is the longest path of dependencies in the direct acyclic graph.
\[ T_p \geq T_\infty \]

Speed up on \( P \) processors will be \( T_1 / T_p \)

If \( T_1 / T_p = \Theta(P) \cdot P \), we have linear speedup;
= \( P \), we have perfect linear speedup;
> \( P \), we have superlinear speedup,

Which is not possible in our model, because of the lower bound \( T_p \geq T_1 / P \). \[^{17}\]

Because we have the lower bound \( T_p \geq T_\infty \), the maximum possible speedup given \( T_1 \) and \( T_\infty \) is

\[ T_1 / T_\infty = \text{parallelism} \]

= the average amount of work per step along the span.

Parallelism can be denoted as
\[ T_1 / T_\infty \]
The parallelism $T_1/T_\infty$ is the maximum possible speedup that can be achieved by any number of processors greater than the parallelism $T_1/T_\infty$. The more processors you have greater than the parallelism the less perfect the speedup. [17]
CHAPTER 2. TECHNICAL RESEARCH

2.1. Technology used

A language called Cilk was developed in 1994 at MIT laboratory for computer science. Cilk language is an extension of C, developed for parallel computing. Cilk++ is a software platform inspired by the award winning Cilk multithreading language.

2.1.1. Research At Cilk Arts

A private company Cilk Arts founded in 2006 developed Cilk++ in order to help ease the transition to multi core computing for software developers and users. Cilk++ is inspired by 15 years of research at MIT’s Supercomputing Technologies research group, led by its founder Prof. Charles E. Leiserson. [13]

2.1.2. Cilk++

Cilk++ is an extension of the C++ language that makes it easy to write applications that efficiently take advantage of multi-core processors. Cilk++ allows programmers to easily identify parallelism in an application so that the runtime environment can efficiently execute parallel tasks. In most cases, the parallel program will have the same semantics regardless of the number of processor cores on which it is run. [14]

Cilk++ uses a work-stealing scheduler to distribute the executing strands among processors. This scheduler is built upon an operating-system software mechanism, called threads. Each operating-system thread participating in work stealing is called a worker. The default for the Cilk++ runtime is to run one worker thread per physical processor.
Cilk++ solves the two large problems facing the software industry as a result of the multicore revolution:

1. Enabling today's mainstream programmers to develop multithreaded (or parallel) applications; and

2. Providing a smooth path to multi core for legacy applications that otherwise cannot easily utilize the performance capabilities of multi core processors. [14]

2.1.3. Components of Cilk++

2.1.3.1. The three Cilk++ keywords

cilk_spawn:

The cilk_spawn keyword modifies a function-call statement to tell the Cilk runtime system that the function may run in parallel with the caller. [4]
cilk_sync:

The cilk_sync statement causes the current function to wait for all of its spawned children to complete. [4]

cilk_for:

A cilk_for loop permits iterations of the loop to run in parallel. The Cilk++ compiler converts a cilk_for loop into an efficient divide-and-conquer recursive traversal over the loop iterations.

The above keywords, when applied to a serial application, produce a Cilk++ source file which is a multicore-enabled version of the serial application. Other than the added keywords, the Cilk++ source is nearly identical to the serial original. Moreover, the Cilk++ source retains the serial semantics of the original C++ code. [4]

2.1.3.2. The Cilk++ compiler

The Cilk++ compiler recognizes the three Cilk++ keywords and generates a parallel application. It is an extension of an industry standard compiler (Microsoft Visual C++ and GCC). The resulting Cilk++ program runs with minimal, often immeasurable overhead on a single processor. [4, 14]

2.1.3.3. The Cilk++ Runtime System:
The Cilk++ Runtime System (RTS) enables a Cilk++ program to dynamically and automatically exploit an arbitrary number of available processor cores. It is a component delivered with the binary executable of the Cilkified program.

The run-time library resides in the customer's application above the operating system. It is not dependent on operating system specifics, and is not tightly integrated with the operating system. [4]

2.1.3.4. Limitations of Other Schedulers

Most schedulers use a central scheduler which coordinates with their worker threads. The scheduler parcels out chunks of work and communicates with each worker to determine which processors are busy and which are available for new tasks. These approaches have following limitations:

1. Scheduling work in this fashion requires that the scheduler knows how many processors are available to do work in advance. This introduces a dependency on the number of processors.

2. They introduce communication overhead between the central coordinating scheduler and the processor/workers. This overhead increases dramatically with the number of processors, making it difficult for these solutions to scale. [4]

2.1.3.5. Work stealing algorithm used by Cilk++ RTS

The Cilk++ RTS takes a very different approach. Firstly, there is no constraint on knowing the available resources: the "Cilk-ified" program runs on the resources allocated to it by the
operating system - resources that might change dynamically. **The numbers of processor cores are thus determined by the intelligent runtime system.**

Each worker (processor) maintains a work deque of ready threads and it manipulates the bottom of the deque like a stack. When a processor executing a thread runs out of work, it "steals" work from the top of the work queue maintained by another processor. It uses an algorithm to decide which work to steal. Thus the processor which would have otherwise remained idle does almost all the work.

The only communication required is if the processor happens to steal a piece of work that is being worked on. Thus, Cilk++ imposes very low overhead.

If an application has sufficient parallelism, stealing is very infrequent. This helps RTS to achieve linear speed up. [4, 14]

### 2.1.3.6. Setting the number of workers

Cilk++ runtime scheduler is built upon the operating-system software mechanism, threads. We call each operating-system thread participating in work stealing a "**worker**". The default for the Cilk++ runtime is to run one worker thread per physical processor. For example, on a system with 2 4-core processors the default for the Cilk++ runtime is to create 8 workers.

In earlier version of Cilk++ a “boilerplate” code could be added. It was possible to set the worker count through this code as shown below

```c
if (nWorkers)
    ctx->set_worker_count(nWorkers);
```

However, the current Cilk++ version does not require the boilerplate code.
We can use cilk_main() as the entry point for both GCC and Windows Cilk++ programs. This makes it much easier to convert an existing C++ program and to specify the number of worker threads on the command line. The worker count can also be set in through the properties of the Visual studio project.\textsuperscript{[4]}

2.1.3.7. Dynamic Load Balancing

The Cilk++ RTS enables the dynamic load-balancing capability demanded by real-world application environments. If a processor gets de-scheduled by the operating system, the work of that processor will automatically be stolen away by the RTS - automatic adaptation. Similarly, if processors become available to the Cilk++ job, they will automatically steal work and make themselves busy.\textsuperscript{[4]}

2.1.4. The Cilk++ Race Detector:

The Cilkscreen race detector is used to test a Cilk++ binary program to ensure its parallel correctness. The race detector is mathematically guaranteed to find all race conditions in a Cilkified code. It identifies all cases where the result of the parallel execution could differ from the serial version and thus ensures parallel correctness.\textsuperscript{[4]}

The following snapshot shows how the Cilkscreen race detector tool shows the race conditions:
2.1.5. Cilk++ Reducers and Hyperobjects:

Global and other non-local variables can inhibit parallelism by inducing data races. Cilk hyperobjects are an innovative construct designed to solve data race problems created by parallel accesses to global variables.\[4\]

Reducers are a special form of hyperobjects. Cilk++ ensures that each worker has access to a private copy of the variable, eliminating the possibility of races without requiring locks and
when the strands synchronize, the copies of the reducer are merged into a single variable. This version of Cilk++ provides the following reducers:

- reducer_list_append
- reducer_list_prepend
- reducer_max
- reducer_max_index
- reducer_min
- reducer_min_index
- reducer_opadd
- reducer_ostream
- reducer_basic_string

2.1.6. Features of Cilk++

2.1.6.1. The Miser Memory Manager

Cilk++ strands that use memory management might not achieve their performance potential, or may slow down when using more than one worker. Therefore, Cilk++ provides an additional memory manager, "Miser", as an alternative to conventional memory management with C/C++ runtime functions (new, delete, malloc, calloc, realloc, and free). [4]

Miser Limitations:
• Miser cannot be disabled. Once you enable it, all C/C++ memory management will be handled by Miser.
• You cannot use the Windows FreeLibrary() function to free the Miser.dll module.
• The project must use the "Multithreaded DLL" "Runtime Library" "Code Generation" compiler option, /MD. Miser cannot intercept calls to the static C/C++ runtime library.[4]

2.1.6.2. cilk_grainsize

The cilk_for statement creates multiple spawn operations, with overhead associated with each spawn. Cilk_for divides the iterations of the loop into chunks to be executed serially. A chunk is a sequential collection of one or more loop iterations. The maximum size of each chunk is called the “grain size”. If the grain size is too small, then the cilk_spawn overhead will reduce performance an if the grain size is too large, then there will not be enough logical parallelism to keep the processors busy, and multicore performance will suffer. [4]

2.1.6.3. cilk::strand_mutex

Strand mutexes ensure that only one Cilk++ strand can enter the strand_mutex at any time. The key difference between strand_mutex and CRITICAL_SECTION is that strand mutexes are only defined if the code is compiled as Cilk++. If the code is compiled as C or C++, all of the strand_mutex methods become noops. [4]

2.1.7. Benefits of Cilk++ Technology

Cilk++ offers the easiest, quickest, and most reliable way to maximize application performance on multicore processors. Cilk++ provides a simple set of extensions for C++,
coupled with a powerful runtime platform for multicore-enabled applications. Cilk++ enables rapid development, testing, and deployment of multicore applications by combining unmatched programmer productivity, world-class scalable performance, and outstanding testability and reliability.

Cilk++ solves the two large problems facing the software industry as a result of the multicore revolution:

1. Enabling today's mainstream programmers to develop multithreaded (or parallel) applications; and

2. Providing a smooth path to multicore for legacy applications that otherwise cannot easily leverage the performance capabilities of multicore processors.

With Cilk++, you can retain the serial semantics of your existing applications, use existing serial methodologies for programming, tooling, debugging, and regression testing. \[2, 15\]

The other benefits of Cilk++ are as follows:

2.1.7.1. Compress Development Time

To multithread an application without Cilk++, software developers must drastically restructure their code. With Cilk++, existing applications need not be rewritten or restructured to take advantage of multiple processors. Cilk++ programmers continue to write serial code in familiar C++, and then simply add a handful of keywords to identify parallelism in an application. Cilk++ can be learned easily, and maximizes programmer productivity.

Cilk++ functions work seamlessly with non-Cilk++ code. Cilk is a no-risk solution: simply remove the Cilk keywords to return your application to its original form. \[2, 15\]
2.1.7.2. Maximize Application Performance

One needs best-in-class performance — but not just on the current generation of 4-core machines. One doesn’t want to rewrite the code each time the number of cores increases.

Moreover, one would like to use a single binary for all the customers, including those who still run 1- or 2-core machines. As a result, a need of linear scaling up and down and minimal overhead on a single processor is generated.

The Cilk++ Runtime Platform (RTP) enables a Cilk++ program to dynamically and automatically exploit an arbitrary number of available processor cores, and, with sufficient parallelism and memory bandwidth, the RTP delivers near-perfect linear speed-up as the number of cores increases. On a single core, typical programs run with negligible overhead (less than 2%). [2, 4, 15]

2.1.7.3. Ensure Software Reliability

In a move to multithreading, the challenging problem of race conditions must be dealt with. These bugs are nondeterministic, making them difficult to avoid, find, and resolve. A race bug may not show up after hundreds of hours of testing, yet take down an application in the field with regularity.

Cilk++ programs are easy to write and test, because they have serial semantics. Any Cilk++ program can be debugged on a single processor using a standard serial debugger, assuring serial correctness. The Cilkscreen race detector monitors a running Cilk++ application during regression testing and identifies any data-race condition that could cause the behavior to differ from a serial execution, identifying the filenames, lines, and variables involved in the offending races. By ensuring that the multithreaded version always operates identical to the original, serial version, Cilkscreen enables a regression-testing methodology that fits into the existing release processes, leading to reliable applications and confident releases. [2]
2.1.8. Limitations:

The current version of Cilk++ has the following limitations and restrictions:

2.1.8.1. General and OS-specific language issues:

- There is no mechanism to abort parallel computation.[4]

- The cilk_for construct supports limited forms of loops. The general cilk_for syntax is:[4]

  cilk_for (declaration; conditional expression; incrementexpression)

  {
    BODY
  }

  The limitations on cilk_for loop are:

  - The declaration must declare and initialize a single variable, called the "control variable". The constructor's syntactic form does not matter. If the variable type has a default constructor, no explicit initial value is needed.[4]

  - The conditional expression must compare the control variable to a "termination expression" using one of the following comparison operators:[4]

    <, <=, !=, >=, >

  The termination expression and control variable can appear on either side of the comparison operator, but the control variable cannot occur in the termination expression. The termination expression value must not change from one iteration to the next.[4]

  - The increment expression must add to or subtract from the control variable using one of the following supported operations:[4]
The value added to (or subtracted from) the control variable, like the loop termination expression, must not change from one iteration to the next. [4]

- Cilk++ cannot execute an infinite cilk_for loop such as:
  
cilk_for (unsigned int i = 0; i != 1; i += 0);

2.1.8.2. Linux Specific Issue:

- cilk_for may not be used in a constructor or destructor. It may be used in a function called from a constructor or destructor. [4]

2.1.8.3. Windows Specific Issues:

- Pointers to members of classes with virtual base classes (data or functions) cannot be used as Cilk++ function arguments.
- Methods returning methods returning pointers to data members will not work properly.
The Microsoft VC++ compiler will accept some constructs that are not part of the C++ standard. In some rare cases, the Cilk++ compiler will not accept the nonstandard construct.\textsuperscript{[4]}

2.2. Parallel Software

Different concurrent programming languages, libraries, APIs, and parallel programming models have been created for programming parallel computers. These can generally be divided into classes based on the assumptions they make about the underlying memory architecture—shared memory, distributed memory, or shared distributed memory.

2.2.1. Ct

The language developed at Intel laboratory is an extension of C/C++, which is familiar to software developers getting. Whereas most languages require developers to manually partition code to run on specific cores, Ct does it automatically. The compiler takes the task of parallelizing the code runtime.

The Ct compiler developed by Intel chops up the code to run on separate cores based on the type of data and the operation being performed on the data.\textsuperscript{[5, 9]}

2.2.2. C*

C* is an object-oriented, data-parallel superset of ANSI C with synchronous semantics. C* adds a domain data type and a selection statement for parallel execution in domains.\textsuperscript{[10]}
2.2.3. mpC

mpC is a high-level parallel language (an extension of ANSI C), designed specially to develop portable adaptable applications for heterogeneous networks of computers. The main idea underlying mpC is that an mpC application explicitly defines an abstract network and distributes data, computations and communications over the network. The mpC programming system uses this information to map the abstract network to any real executing network in such a way that ensures efficient running of the application on this real network. This mapping is performed in run time and based on information about performances of processors and links of the real network, dynamically adapting the program to the executing network. [8]

The mpC programming system includes a compiler, a run-time support system (RTSS), a library, and a command-line user interface. The compiler translates a source mpC program into the ANSI C program with calls to functions of RTSS. RTSS manages processes, constituting the parallel program, and provides communications. It encapsulates a particular communication platform (currently, a subset of MPI) ensuring platform-independence of the rest of system components. [8]

2.2.4. NESL

NESL integrates various ideas from the theory community (parallel algorithms), the languages community (functional languages) and the system's community (many of the implementation techniques). The most important new ideas behind NESL are

1. Nested data parallelism: This feature offers the benefits of data parallelism, concise code that is easy to understand and debug, while being well suited for irregular algorithms, such as algorithms on trees, graphs or sparse matrices.
2. A language based performance model: This gives a formal way to calculate the work and depth of a program. These measures can be related to running time on parallel machines.
The main emphasis in the design of NESL was to make parallel programming easy and portable. Algorithms are typically significantly more concise in NESL than in most other parallel programming languages. Furthermore the code closely resembles high-level pseudo code. [7]

2.3. Some applications that can benefit from parallel programming

In general an application with Thread-level parallelism, benefit from parallel programming. Some of them are:

2.3.1. Parallelizing imaging applications

In the case of imaging applications, it's pretty easy to find computation-intensive tasks to parallelize. Image processing algorithms consist of pixel operations that are often independent. Hence their computations can be done in parallel. [22]

2.3.2. Divide and Conquer Algorithms

The idea of the divide-and-conquer paradigm is to fragment a problem into sub problems of the same kind and solve the sub problems recursively. These sub problems can be solved independently on different cores and, finally, the solutions of the sub problems can be combined into a solution of the original problem. Thus algorithms with a divide-and-conquer structure such as Traveling Salesman Problem, Quick sort, etc are suitable candidates for parallelization. [22]
2.3.3. **Exploration of Minmax trees**

The minmax procedure is a depth-first, depth limited search procedure. It can be parallelized by exploring the sons of the root node, and their sons and so on up to the point where one has many son nodes waiting to be explored. At this point, each processor can explore the sub tree rooted at one of these nodes using sequential minimax algorithm. [1]

2.3.4. **Alpha Beta Pruning Algorithm**

It is one of the most efficient methods to traverse a minmax game tree. This algorithm effectively prunes portions of the search tree whose leaf node scores are known to be inferior to the scores of leaf nodes already visited. At each node, when the first branch is explored and a bound is obtained, then all the branches can be executed in parallel. Thus it is possible to search multiple branches of the game tree at the same time and achieve a reasonable speedup. [1]

2.4. **Mikenet Neural Network Simulator Library**

MikeNet is a neural network simulator library

Mike Net is implemented as a 'c' library. A simulation can be built by writing a short program in 'c', including the header files and linking to the library. The library itself consists of routines to create and run the network. [21]

2.4.1 **General Design**

Mike Net has five main types of objects. They are-

- a network (containing a set of groups and connection blocks and parameters),
- a group (a set of units with the same layout),
a **layer** in normal PDP parlance),

a **connection block** (array of weights connecting two groups),

an **example**, 

an **example set.**\(^{[21]}\)

The general format of a Mike Net simulation has these parts in a 'c' file:

- Define the network objects (net and groups)
- Add groups to the net object
- Connect groups, and add connection objects to the net
- Randomize connections
- Train the network.

After picking an example the network can be trained by;

- Forward propagate
- Backward Propagate error
- Adjust weights
- Repeat until done
- Then, optionally save weights and/or print out status.\(^{[21]}\)

**2.4.2 Features**

The design of MikeNet allows for some things that are harder to do in other simulators.
The sharing of connection between networks and instantiating multiple networks is possible in MikeNet. Training (forward or backward propagation) works over network objects, so only the groups in the network object are affected. Because of this training up subnetworks is easy, as parts of larger networks.

MikeNet is designed for large networks, and is most fast for such large projects. It is very memory-efficient and can hold very large example files without taking up too much memory.

2.4.3 Building the Network

2.4.3.1 Net Object

A net object is basically a structure that holds pointers to arrays of group objects and connection objects. As stated above, network operations such as forward and backward propagation take place over net objects. To instantiate a net object, first define the variable for it somewhere: [21]

Net * mynet;

2.4.3.2 The Group Object

After creating a network object, groups are created and added to the network. [21]

Group variables are declared as,
Group *input,*hidden,*output,*bias;

2.4.3.3 The Connections Object

Connection objects are declared in the beginning. For every set of connection between layers one connection object is required. [21]
Connections *c1,*c2,*c3,*c4;

Then, these groups are connected. [21]

2.5 Outcome of Literature Survey

As part of our literature survey, we studied the various techniques of parallelism. With emergence of multicore processors, there was a need to design new languages which would exploit the capabilities of these processors.

There are several such multicore programming languages. One such language is Cilk++. Therefore in our project we aim to prove that with the use of ‘Cilk++’, the performance of an application can be enhanced. To get acquainted with the cilk++ environment we cilkified some smaller applications like: Minmax algorithm, prime number, merge sort, matrix transpose, quicksort, nqueen, backpropagation algorithm, etc. The major application that we parallelized Mikenet neural network simulator code. Mikenet is an appropriate candidate for cilkification as it is implemented in C, and it is a computation intensive application.
CHAPTER 3. PROJECT PLAN

3.1. Purpose of the Project

The purpose of the project is to provide a substantial proof so as to show that using Cilk++ language, the execution time of an application can be reduced multifold.

3.2 Project Scope

The scope of our project covers the cilkification of Mikenet. Mikenet is an open source neural network application. It is written using serial programming. In this project we would identify potential functions or parts which can be parallelized in the code and cilkify those using Cilk++ keywords. Modified code shall be compiled using Cilk++ compiler and tested on a multi-core machine. Performance gains of new program shall be measured and compared with the non-cilkified version of the code.
### 3.3 Project schedule

<table>
<thead>
<tr>
<th>No.</th>
<th>Tasks</th>
<th>Aug</th>
<th>Sep</th>
<th>Up to Oct 15&lt;sup&gt;th&lt;/sup&gt;</th>
<th>From 15&lt;sup&gt;th&lt;/sup&gt; Dec</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Study of Cilk++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cilkification of smaller applications</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Study of neural networks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Study of mikenet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Cilkification of mikenet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Testing Cilkified Mikenet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Working on Linux version of cilk++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1. Project Schedule
3.4 Documentation

The reference documentation includes:

1. Information gathering document (Literature survey)
2. Project schedule
3. Software requirement specification
4. High level design
5. Testing document
4.1 Project Scope

The purpose of our project is to optimize performance of applications on multi-core processors.

The scope of our project covers the cilkification of Mikenet. MikeNet is an open source neural network application. It is written using serial programming. In this project we would identify potential functions or parts which can be parallelized in the code and cilkify those using Cilk++ keywords. Modified code shall be compiled using Cilk++ compiler and tested on a multi-core machine. Performance gains of new program shall be measured and compared with the non-cilkified version of the code.

Our project aims at proving that Cilk++ helps programmers to extract parallelism in applications easily and efficiently. This would give us a linear speed up and would improve performance on multi-core processors effectively.

4.2 Overall Description

4.2.1 Project Perspective

CilkArts developed Cilk++ which is a powerful runtime platform for multi-core applications. Our project aim is to identify such an application and cilkify it for the better performance results. [13]
4.2.2 Operating Environment

The system used by the programmers should meet the following software and hardware requirements [4]

Software Requirements

- Windows XP Service Pack 2 or Windows Vista
- Visual Studio 2005 IDE
- Visual Studio Service Pack 1
- Cilk++ (1.0.3.7007)

Hardware Requirements

- Intel 3.0GHz Core 2 Duo Xeon Based Machine or an equivalent or any Intel multi-core processor.

4.2.3. Implementation Constraints

The serial code should be a C/C++ application and it should implement the basic functions or algorithms that can be parallelized. That is, the serial code should be computational intensive and should have portions of code that can run in parallel.
### 4.3 Functional Requirements

<table>
<thead>
<tr>
<th>REQ 1</th>
<th>Identify candidates for cilkification</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Programmer</td>
</tr>
<tr>
<td>Pre Condition</td>
<td>The serial code should be flawless and computational intensive.</td>
</tr>
<tr>
<td>Flow</td>
<td>Execute serial code in Vtune Performance analyzer</td>
</tr>
<tr>
<td></td>
<td>Identify the hotspots.</td>
</tr>
<tr>
<td></td>
<td>Check if these hotspots can be cilkified</td>
</tr>
<tr>
<td>Post condition</td>
<td>Programmer works on these hotspots</td>
</tr>
</tbody>
</table>
### REQ 2

<table>
<thead>
<tr>
<th>User</th>
<th>Programmer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pre condition</strong></td>
<td>Hotspots are identified.</td>
</tr>
<tr>
<td><strong>Flow</strong></td>
<td>Place Cilk++ keywords at appropriate places.</td>
</tr>
<tr>
<td></td>
<td>Run under Cilkscreen race conditions to locate race conditions.</td>
</tr>
<tr>
<td></td>
<td>Remove the race conditions by adding appropriate reducer hyper objects.</td>
</tr>
<tr>
<td><strong>Post condition</strong></td>
<td>The output of the cilkified code should not differ from that of the serial code</td>
</tr>
</tbody>
</table>

### 4.4 Other Nonfunctional Requirements

#### 4.4.1 Performance Requirements

The multi-core application should have the same semantics as the serial application. It should give the exact same output as the serial code.
4.5 Deployment scenario

To check the application performance we will run the serial and cilkified code on same multicore machine.
CHAPTER 5. DESIGN SPECIFICATION

5.1 Activity Diagram

Figure 5.1. Activity diagram

5.2 State Diagram
Figure 5.3. State Diagram
5.3 Block Diagram of Cilk++ Environment

![Block Diagram of Cilk++ Environment](image)

Figure 5.3. Block Diagram of Cilk++ environment
CHAPTER 6. IMPLEMENTATION DETAILS

6.1 Cilkification of Mergesort

Before cilkifying Mikenet, we cilkified some smaller applications like quicksort, Minmax algorithm, backpropagation algorithm, N queens etc.

The process followed for cilkification of mergesort is as follows:

Serial code:

/*
 * msort.cpp
 * An implementation of mergesort.
 */
#include <iostream>
#include <algorithm>
#include <iterator>
#include <functional>
#include <cstdlib>
#include <windows.h>

#define MAX 10*1000*1000
int *a = new int[MAX];

int *b = new int[MAX];

void merge(int low, int m, int high)
{
    int i, j, k;

    // copy both halves of a to auxiliary array b
    for (i=low; i<=high; i++)
        b[i]=a[i];

    i=low; j=m+1; k=low;
    // copy back element in descending order
    while (i<=m && j<=high)
        if (b[i]<=b[j])
            a[k++]=b[i++];
        else
            a[k++]=b[j++];

    // copy back remaining elements of first half
    while (i<=m)
        a[k++]=b[i++];

    while (j<=high)
        a[k++]=b[j++];
}
void mergesort(int low, int high)
{
    int m;
    if (low<high)
    {
        m=(low+high)/2;
        mergesort(low, m);
        mergesort(m+1, high);
        merge(low, m, high);
    }
}

int merge_main()
{
    for (int i = 0; i < MAX; i++)
    {
        a[i]=i;
    }
}
std::random_shuffle(a, a + MAX);

std::cout << "Sorting " << MAX << " integers...." << std::endl;

mergesort(0,MAX-1);

std::cout << "Confirming that the array is sorted...." << std::endl;
// Confirm that a is sorted and that each element contains the index.
for (int i = 0; i < MAX-1; ++i) {
    if ( a[i] >= a[i+1] || a[i] != i ) {
        std::cout << "Sort failed at location i=" << i << " a[i] = " << a[i] << " a[i+1] = " << a[i+1] << std::endl;
        delete[] a;
        return 1;
    }
}

std::cout << "Sort succeeded." << std::endl;

delete[] a;

return 0;
}

int main(int argc, char *argv[])
{
    return merge_main();
}

Project_BE_Report 52 Created in March 2009
Mergesort algorithm uses a divide and conquer strategy. The above program implements the Mergesort algorithm recursively. Moreover the recursive calls:

```c
mergesort(low, m);
mergesort(m+1, high);
```

are independent of each other and hence can be executed in parallel. [15]

Thus by introducing cilk_spawn keyword we can execute the caller and the called function in parallel as:

```c
cilk_spawn mergesort(low, m);
mergesort(m+1, high);
```

```c
cilk_sync;
```

Thus Mergesort is cilkified as:

```c
#include <iostream>
#include <algorithm>
#include <iterator>
#include <functional>
#include <cstdlib>
#include <windows.h>
```
#include <cilk.h>
#define MAX 10*1000*1000
int *a = new int[MAX];
int *b = new int[MAX];

void merge(int low, int m, int high)
{
    int i, j, k;

    // copy both halves of a to auxiliary array b
    for (i=low; i<=high; i++)
        b[i]=a[i];

    i=low; j=m+1; k=low;
    // copy back element in descending order
    while (i<=m && j<=high)
    {
        if (b[i]<=b[j])
            a[k++]=b[i++];
        else
            a[k++]=b[j++];
    }

    // copy back remaining elements of first half
    while (i<=m)
    {
        a[k++]=b[i++];
    }
}
while (j<=high)
    a[k++]=b[j++];
}

void mergesort(int low, int high)
{
    int m;
    if (low<high)
    {
        m=(low+high)/2;
        cilk_spawn mergesort(low, m);
        mergesort(m+1, high);
        cilk_sync;
        merge(low, m, high);
    }
}

// A simple test harness
int merge_main()
{
    for (int i = 0; i < MAX; i++)
    {
    
}
a[i]=i;
}
std::random_shuffle(a, a + MAX);
  std::cout << "Sorting " << MAX << " integers...." << std::endl;

mergesort(0,MAX-1);

// Confirm that a is sorted and that each element contains the index.
std::cout<"Confirmation that the array is sorted...."<std::endl;
for (int i = 0; i < MAX-1; ++i) {
  if ( a[i] >= a[i+1] || a[i] != i ) {
    std::cout << "Sort failed at location i=" << i << " a[i] = " << a[i] << ", a[i+1] = " << a[i+1] << std::endl;
    delete[] a;
    return 1;
  }
}
std::cout << "Sort succeeded." << std::endl;
delete[] a;
return 0;

int cilk_main(int argc, char *argv[])
{
  return merge_main();
}

Project_BE_Report  56  Created in March 2009
To make the code computation intensive we the array size was taken as $10^7$

**Observations:**

The serial version of mergesort program took **10.93s**

The cilkified version of mergesort program took **7.26s**

---

6.2 Cilkification of MikeNet

![Typical Workflow of cilkification](image)

**Figure 6.1. Typical Workflow of cilkification**
6.2.1 Compile serial version of Mikenet and Regression testing

We compiled the serial version of Mikenet in Visual C++ 2005. In Mikenet application ±0.04 error is permitted. In order to check the serial correctness, we had to check the value of this error. On every execution of Mikenet a progress.txt file is created. This file contains the error value. We assured serial correctness comparing this value to 0.04. [21]

6.2.2 Identifying performance bottlenecks

Choosing any portion of mikenet application and cilkifying it did not make any sense. Ideally, the portion of the program which takes maximum amount of execution time should be cilkified. Such parts are called bottlenecks. They can be cilkified if they are candidates for cilkification the program. For identifying performance bottlenecks, we used a performance analyzer tool called Vtune. [23]

This tool displays a list of functions along with the execution time. In case of Mikenet, Vtune gave a list of such functions as follows:
From this list we identified the following three functions that were suitable candidates for cilkification:
6.2.3 Adding Cilk++ keywords to expose parallelism

The first three functions took the maximum execution time. In these functions nested for loops were implemented.

```c
void mikenet_matrix_vec_mult_p(Real * outvec,int nout,Real *invec,
                               int nin,Real **mat)
{
    int i,j;

    for(i=0;i<nout;i++)
    {
        for(j=0;j<nin;j++)
        {
            outvec[i] += mat[i][j] * invec[j];
        }
    }
```
void mikenet_matrix_vec_mult_t_p(Real * outvec, int nout, Real *invec,
    int nin, Real **mat)
{
    int i,j;

    for(i=0;i<nout;i++)
    {
        for(j=0;j<nin;j++)
        {
            outvec[i] += mat[j][i] * invec[j];
        }
    }
}

void mikenet_matrix_outer_product(Real ** matrix,
    Real * v1,
    int n1, /* rows */
    Real * v2,
    int n2) /* cols */
{
    int i,j;

if (default_useBlas)
{
    #ifdef USE_BLAS
        cblas_sger(CblasRowMajor,n1,n2,1.0,v1,1,v2,1,(float *)&matrix[0][0],
                   n2);
    #endif
    return;
}
else
{
    for(i=0;i<n1;i++)
    {
        for(j=0;j<n2;j++)
        {
            matrix[i][j] += v1[i] * v2[j];
        }
    }
}

We cilkified these functions by adding the Cilk++ keyword cilk_for. [15]

void mikenet_matrix_vec_mult_p(Real * outvec,int nout,Real *invec,
                               int nin,Real **mat)
{

int i,j;

cilk_for(i=0;i<nout;i++)
{
    for(j=0;j<nin;j++)
    {
        outvec[i] += mat[i][j] * invec[j];
    }
}

void mikenet_matrix_vec_mult_t_p(Real * outvec,int nout,Real *invec,
    int nin,Real **mat)
{
    int i,j;

cilk_for(i=0;i<nout;i++)
{
    for(j=0;j<nin;j++)
    {
        outvec[i] += mat[j][i] * invec[j];
    }
}
}
void mikenet_matrix_outer_product(Real ** matrix,

    Real * v1,
    int n1, /* rows */
    Real * v2,
    int n2) /* cols */
{
    int i,j;

    if (default_useBlas)
    {
        #ifdef USE_BLAS
            cblas_sger(CblasRowMajor,n1,n2,1.0,
                        v1,1,v2,1,(float *)(&matrix[0][0]),
                        n2);
        #endif
        return;
    }
    else
    {
        cilk_for(i=0;i<n1;i++)
        {
            for(j=0;j<n2;j++)
            {
                matrix[i][j] += v1[i] * v2[j];
            }
        }
    }
}
6. 2.4. Compile and run regression tests to verify serial Correctness

We executed the Cilk++ code under the C++ compiler to ensure that the serial semantics did not change. We checked the execution time of each function separately. We observed that there was not a remarkable performance improvement in the second function. After studying the function, we found that we could make it more cache friendly, by interchanging the for loops.

```c
void mikenet_matrix_vec_mult_t_p(Real * outvec, int nout, Real *invec,
                                 int nin, Real **mat)
{
    int i,j;

    cilk_for(j=0;j<nin;j++)
    {
        for(i=0;i<nout;i++)
        {
            outvec[i] += mat[j][i] * invec[j];
        }
    }
}
```

Project_BE_Report Created in March 2009
6. 2.5 Run Cilkscreen to verify parallel correctness and locate races

We executed the code under the Cilkscreen race detector. The Cilkscreen located race conditions. These races were introduced due to the interchanging of the for loops. \[13\]

6. 2.6 Resolving races using reducers.

There was a need to use a reducer to eliminate race condition. Cilk++ provides a standard set of reducer. However we required a reducer which handled arrays. Cilk++ did not provide such an array reducer. So there was a need to write an array reducer to resolve the races. We had to write various such array reducers for this purpose. \[4\]

The way in which we used the final array reducer is as follows.

```c
void mikenet_matrix_vec_mult_t_p(Real * outvec,int nout,Real *invec,
    int nin,Real **mat)
{

cilk::hyperobject<array_reducer> r_outvec(nout, outvec);

cilk_for(int j=0;j<nin;j++)
{
    Real * tmpvec = r_outvec().vec;
    for(int i=0;i<nout;i++)
    {
        tmpvec[i] += mat[j][i] * invec[j];
    }
}
```

Project_BE_Report 66 Created in March 2009
6.3. Parallel correctness.

To ensure the correctness of the cilkified program we followed the steps given below:

6.3.1 We compared the output of the cilkified program with that of its serial counterpart to ensure that the parallel code is semantically correct.

6.3.2 We executed the cilkified program under Cilkscreen race detector to ensure the parallel correctness of the code.

6.4. Improvement in performance

The execution time of the two programs i.e serial and parallel programs was noted by using the QueryPerformanceCounter API \cite{24} and a remarkable performance improvement was observed.

Our observations are summarized in the following table. It gives a comparison between the execution time of serial application (i.e. before cilkification) and the execution time of parallel application(i.e. after cilkification)when run on a dual core machine.
<table>
<thead>
<tr>
<th>Sr. no.</th>
<th>Program Name</th>
<th>Cilk feature used</th>
<th>Time taken for serial execution</th>
<th>Time taken for parallel execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><strong>Minmax algorithm</strong></td>
<td>cilk_spawn and cilk_sync</td>
<td>4.47s</td>
<td>0.14s</td>
</tr>
<tr>
<td></td>
<td>This program finds the minimum and maximum element from an array of numbers using divide and conquer strategy. Size of array: $10^7$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td><strong>Prime numbers</strong></td>
<td>cilk_for</td>
<td>0.47s</td>
<td>0.32s</td>
</tr>
<tr>
<td></td>
<td>This program determines the series of prime numbers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td><strong>Merge sort</strong></td>
<td>cilk_spawn and cilk_sync</td>
<td>10.93s</td>
<td>7.26s</td>
</tr>
<tr>
<td></td>
<td>This program implements merge sort using divide and conquer strategy. Size of array: $10^7$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sr. no.</td>
<td>Program Name</td>
<td>Cilk feature used</td>
<td>Time taken for serial execution</td>
<td>Time taken for parallel execution</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------</td>
<td>------------------------------------</td>
<td>---------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>4.</td>
<td>Matrix Transpose</td>
<td>cilk_for</td>
<td>0.15s</td>
<td>0.05s</td>
</tr>
<tr>
<td></td>
<td>This program computes the transpose of a matrix. Size of matrix: 100*100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Quick Sort</td>
<td>cilk_spawncilk_sync</td>
<td>49.58s</td>
<td>7.16s</td>
</tr>
<tr>
<td></td>
<td>This program uses divide and conquer to sort arrays. Size of array: $10^7$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Reducer</td>
<td>cilk::hyper_ptr<a href="">cilk::reducer_opadd</a></td>
<td>0.23s</td>
<td>0.12s</td>
</tr>
<tr>
<td></td>
<td>This program uses reducer_opadd. Size of matrix: 100*100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>N queens problem</td>
<td>cilk::hyper_ptr&lt;cilk::reducer_opadd&lt;int&gt; &gt;, cilk_spawn, cilk_sync</td>
<td>1.37s</td>
<td>0.14s</td>
</tr>
<tr>
<td></td>
<td>This program determines the solutions to N queens problem. Number of queens: 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sr. no.</td>
<td>Program Name</td>
<td>Cilk feature used</td>
<td>Time taken for serial execution</td>
<td>Time taken for parallel execution</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------</td>
<td>-------------------</td>
<td>----------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>8</td>
<td><strong>Backpropagation</strong></td>
<td>cilk_for</td>
<td>1.89s</td>
<td>1.01s</td>
</tr>
<tr>
<td></td>
<td>This program implements backpropagation algorithm for a neural network. Net with four layers having 3,3,3 and 1 neuron respectively.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><strong>Mikenet</strong></td>
<td>cilk_for</td>
<td>33.16min</td>
<td>17.25 min</td>
</tr>
<tr>
<td></td>
<td>A Neural Network simulator library</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1. Comparison of Serial and cilkified codes
CHAPTER 7. TEST CASES

Following are the test cases which support the concept of performance improvement using Cilk++.

Test Case 1: Output of cilkified merge sort and its performance.

Purpose : To verify that the addition of cilk++ keywords shows performance improvement in terms of execution time. The output should also be equal to the serial output of the program.

Input Given: A shuffled integer array of size $1 \times 10^7$. Keywords cilk_spawn and cilk_sync were added before the recursive merge sort call.

Output Expected: The array should be sorted and the time taken should be less than the serial merge sort.

Test Result: Succeeded.

Test Case 2: Performance of cilkified N queens program

Input Given: 16 number of queens. Cilk_for keyword was added before the loop that decided the position of the queen.

Output Expected: The program should give the correct number of ways for positioning the queens. It should take less execution time than the serial N Queens program.

Test Result: Succeeded.
Test Case 3: Performance of Mikenet after cilk_for keyword addition.

Purpose: Before addition of Cilk++ keywords, mikenet application took 33 minutes for execution. The purpose of this test case is to ensure that cilkified mikenet application takes less than 33 sec.

Input Given: cilk_for keyword added before functions which involved matrix calculation

Output Expected: Ideally the execution time should be half of that taken by the serial version of mikenet.

Test Result: Failed. The performance improvement was not remarkable.

Reason: In the second function the for loop was not cache friendly.

Solution: By interchanging the loop the code can be made more cache friendly.

Test Case 4: Output of Mikenet after interchanging the loop

Purpose: This test case ensures that interchanging of for loops makes the code more cache friendly. To check this dry running of code was done.

Input Given: The cilkified loops are interchanged.

Output Expected: The output of the parallel mikenet application should be same as that of the serial counterpart. The back propagation error should be ±0.04.

Test Result: Failed. Unhandled memory exception occurred.

Reason: Possibility of a race condition on the array used by each iteration of the for loop.
Solution: Use of a reducer that handles array.

Test case 5: Performance of Mikenet application after using reducer

Purpose: This test case checks if the execution time is reduced after the addition of reducer or not. Also the output of the cilkified code should be verified, that is, it should be correct.

Input given: A cilkified mikenet library that uses an array reducer.

Output Expected: The execution time taken by cilkified mikenet should be at least half the time taken by serial version.

Test Result: Failed

Reason: Improper use of array reducer. A copy of array was being created for each strand even if it was not necessary.

Solution: a pointer to the array can be maintained and a copy of array should be created only when it is necessary for a particular iteration.

Test case 6: Performance of Mikenet application after appropriate use of reducer

Purpose: This checks the mikenet code after implementing the above solution.

Input given: A cilkified mikenet library that uses an array reducer.

Output Expected: The execution time taken by cilkified mikenet should be at least half the time taken by serial version.
APPENDIX A. Intel VTune performance Analyzer

Introduction

The Intel VTune Performance Analyzer provides performance analysis environment with graphical user interface that helps in analyzing code performance.

Features

The Intel VTune Performance Analyzer can be used to speedup applications and produce faster, more efficient code. Some of its features can be listed as follows:

Source and disassembly views:

It displays the source and disassembly view which shows us exactly which lines of code are taking the most time.

Compatibility

Programming Language and Compiler Independent

VTune analyzer supports all compilers that follow industry standards including Microsoft and Intel compilers for C, C++ and Fortran.
**Processor Support**

It supports the latest Intel processors, including Intel quad-core processors.

**Call Graph Profiling**

Call Graph determines calling sequences and graphically displays the critical path. It also displays the context of the bottleneck.

**Microsoft Visual Studio Support**

The VTune analyzer plugs into the following versions of Microsoft Visual Studio:

- Microsoft Visual Studio .NET 2002
- Microsoft Visual Studio .NET 2003
- Microsoft Visual Studio 2005
- Microsoft Visual Studio 2008

Data collection can be directly launched from the Microsoft Visual Studio IDE. The VTune analyzer can also be launched as an independent tool from the IDE.
GLOSSARY

Core
One of the processors of a multicore chip

Deadlock
A situation when two or more strands are each waiting for another to release a resource, and the “waiting-for” relation forms a cycle so that none can ever proceed.

Race condition
A source of nondeterminism whereby the result of a concurrent computation depends on the timing or relative order of the execution of instructions in individual strands

Serial semantics
The meaning (or behavior) of the program when run on one processor.

Inline function
An inline function is a programming language construct used to suggest to a compiler that a particular function be subjected to in-line expansion; that is, it suggests that the compiler insert the complete body of the function in every context where that function is used.

**Speedup:**

How many times faster a program is when run in parallel than when run on one processor.

\[ \frac{T_1}{T_p} = \text{Linear speed up on processors} \]

If we have

\[ \frac{T_1}{T_p} = \Theta(p) \quad \text{Linear speed up} \]

\[ \frac{T_1}{T_p} = P \quad \text{Perfect linear speed up} \]

\[ \frac{T_1}{T_p} \geq P \quad \text{Super linear speed up} \]

**Strand**

A concurrent agent consisting of a serial chain of instructions without any parallel control (such as a spawn, sync, return from a spawn, etc.).

**Thread**

A concurrent agent consisting of a serial chain of instructions. Threads in the same job share memory. Scheduling of threads is typically managed by the operating system.
**Work stealing**

A scheduling strategy where processors post parallel work locally and, when a processor runs out of local work, it steals work from another processor. Work-stealing schedulers are notable for their efficiency, because they incur no communication or synchronization overheads when there is ample parallelism. Cilk++ employs a work-stealing scheduler.
CONCLUSION

Our project aimed at proving the claim of Cilk Arts that with the use of cilk++, programmers can exploit full capabilities of underlying processing cores.

Cilk++ is suitable and beneficial for applications that are computation bound rather than I/O bound. The Cilk++ platform dynamically determines the available number of cores in your system. So the cilk++ code need not be changed when the number of cores in the system changes. It shows a linear speed up as the number of cores increases.

We used the alpha and beta versions of Cilk++ while working on our project. However in December 2008, Cilk++ platform was commercialized. Cilk++ is now available in academic, professional and open source editions.

In future single core processors will be extinct and multi core processors will be all pervasive. Parallelizing programs using Cilk++ can be used on a large scale then.

We used Mikenet library as a candidate for cilkification. It showed a remarkable performance on addition of Cilk++ keywords. Any neural network application that makes use of Mikenet library will automatically benefit from this cilkification.

We have demonstrated use of cilk++ keywords and written an array reducer which improves application performance on multi core processors.

Thus in years to come, Cilk++ will have an edge over the other multi core programming languages.
REFERENCES

**E-Books**


http://books.google.co.in/books?hl=en&id=t2cMNX81U4oC&dq=Minimax+and+Applications+By+Dingzhu+Du,+Panos+M.+Pardalos&printsec=frontcover&source=web


**Papers**


**URL’s**


[7] NESL language

http://www.cs.cmu.edu/~scandal/nesl.html

Project_BE_Report 80  Created in March 2009
[8] mpC language
http://www.ispras.ru/~mpc/

[9] Ct language

[10] C* language
http://en.wikipedia.org/wiki/C*

http://www.cs.cmu.edu/~fp/courses/15213-s06/lectures/27-multicore.pdf

[12] Levels of Parallelism
http://www.cs.bu.edu/~best/courses/cs551/lectures/lecture-03.html

http://cilk.com

[14] Cilk++ Solution Overview
http://www.cilk.com/multicore-products/cilk-solution-overview

[15] Cilk++ Training
http://www.cilk.com/Home/cilk-customer-training/

[16] Cilk++ Technical Specifications (Cilk++ Data Sheet)
http://www.cilk.com/Home/technical-specs/

[17] Performance Measures for Parallel programs

www.supertech.csail.mit.edu%2Fcilk%2Flecture-1.ppt

[18] Ways of Parallelism

http://ilab.usc.edu/packages/cell-processor/course/6.189-lecture3-architecture.pdf


http://www.cilk.com/multicore-blog/bid/5365/What-the-is-Parallelism-Anyhow

[20] Addressing the Multicore Programming Challenge


[21] Mikenet Neural Network Simulator

http://www.cnbc.cmu.edu/~mharms/research/tools/mikenet/

[22] Applications that benefit from parallelization

http://www.developers.net/intelmcs/showcase/datablock

[23] Intel VTune Performance Analyzer


[24] Query Performance Counter
