PERFORMANCE TUNING OF AN EMBEDDED TEMPLATE ANALYSIS TOOL

by

ASHWIN ARIKERE

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

MAY 2011

To my parents and Teju!

ACKNOWLEDGEMENTS

I would like to thank my research advisor, Dr. Roger Walker for guiding me and motivating me through this research effort. He has been instrumental in getting me through a number of challenges I faced along the way with his constant support. I am really thankful for his patience, his cheerful disposition and positive feedback.

I would like to thank my supervising committee that includes Dr. Gergely Zaruba and Dr. Carter Tiernan for their time and suggestions.

Without the love and support of my mother and father, I would not be the man I am today. I thank you both for all the love you have showered up on me, and can only hope I have made you proud.

Finally, I would like to thank all my friends for being my family in a land far away from home.

April 18, 2011

ABSTRACT

PERFORMANCE TUNING OF AN EMBEDDED

TEMPLATE ANALYSIS TOOL

Ashwin Arikere, M.S.

The University of Texas at Arlington, 2011

Supervising Professor: Roger Walker

As multi core processors become ubiquitous, programs must be designed to utilize the additional power at their disposal. Extracting maximum performance out of these processors requires utilizing latest parallel programming techniques as well as using the latest tools available to increase performance. This thesis delves into enhancing performance of existing programs by applying parallel techniques, utilizing performance analyzers and various other tools developed specially for multi-core system programming. We discuss these techniques and tools and also demonstrate how a Template Analysis program was modified to take advantage of the additional power a multi-core machine has over single core machines to enable its use in an embedded application. The overall efficiency of the application was increased using these techniques, and then tested using several data sets as well as various core machines. More information was gathered using this data and the program was fine tuned to extract maximum performance. Additional considerations were made towards developing a real time Template Analysis program.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Introduction

Developing an embedded application is a challenge unto itself. Limited resources in such systems force a developer to come up with new ways to solve common programming problems. However, with the advent of increased computing performance available to an embedded application developer, conventional methods of programming fail to utilize this extra power. Multi-core processors allow a developer to utilize parallel programming paradigms to increase the maximum possible performance of an application. Existing applications also need to be updated so they can utilize the increased power available. This research effort tries to address these issues by presenting some of the tools and interfaces available to an embedded developer. These are used to modify an existing application, namely the Template Analysis Tool.

The Template Analysis Tool is an application developed to process road profile data. After an initial pavement profile has been obtained using the road profiler, the profile needs to be analyzed so as to determine the conditions of the road as well as to determine the locations of various defects in the road. Because this is currently a post processing application, it is not bound by real time processing constraints as embedded applications usually are. However, the need for results on the field emphasizes the embedded nature of the application, thus increasing the need for rapid processing of large data sets with the available resources at hand.

1.2 Embedded Systems

An embedded system is a computer designed to execute specific tasks under certain restrictions such as limited resources, real-time computing constraints. They may be coupled with various sensors to accept data from the environment. Computation on this input data is performed and results are then presented. They differ from conventional computers in that they are designed for dedicated functions. By and large embedded systems are designed to use function specific processors. Microcontrollers fall closely into this category. They provide a good mix of computing power at low energy consumption and low cost.

Parallel computing has traditionally been used in the domain of high performance computing. In the simplest sense, utilizing multiple computing resources to solve a computational problem is called parallel computing. However with the advent of modern day multi-core processors, parallel programming is no longer limited to a specific set of computing. Parallel computing is now becoming more main stream as multi-core processors, simultaneous multithreading support is provided on the latest processors. With advanced power techniques being employed to reduce power consumption in these processors, we can now consider using these processors in embedded systems. With real time considerations in embedded systems, making optimal use of increased resources is a challenge.

1.3 Contributions of this thesis

This effort details the methods and tools employed to convert an existing embedded serial application to one which makes use of parallel programming concepts. As a specific example, the Template Analysis Procedure (TAP) program is considered for conversion. Various parallel programming models are investigated and compared. Tools are employed to identify various performance bottlenecks which exist in the serial application. New techniques used to improve performance by removal of bottlenecks are examined. As parallel programming models are introduced, new issues unique to parallel systems arise. Methods to overcome these are then discussed. The existing system is detailed and comparisons with the new and improved method are then presented.

1.4 Overview

The rest of the thesis is organized as follows. The chapter 2 describes all related work done thus far in relation to the Template Analysis Procedure. A brief introduction to road profiling systems is followed by the overview of the TAP. The chapter 3 introduces the various tools which were used to convert the existing serial TAP to an application capable of extracting maximum performance out of the latest embedded processors. An overview of various multi-threading schemes is also provided. The chapter 4 details the results obtained using the new system and complete analyses of these results are made by comparing them to the performance of the serial application. Conclusions and future work are discussed in chapter 5.

CHAPTER 2

RELATED WORK

2.1 Road Profiling

Road profile measurements are used to provide information about roads and pavements. These measurements provide information regarding road surface roughness. A profile is a two-dimensional slice of the road surface, taken along an imaginary line and this is obtained by combining three parameters:

1. A reference elevation,
2. A height relative to the reference
3. Longitudinal distance.

To achieve this we use a wide variety of instruments such as Dipstick, Rod and Level, and Inertial Profilers. Current technologies utilize inertial profilers to gather profile data typically at highway speeds. An accelerometer provides the inertial reference. Using various algorithms we can thus determine the instant height of the ground from the accelerometer. We measure this instantaneous distance using a non-contacting sensor such as a laser transducer. The longitudinal distance of the instruments is obtained using some distance sensor. Combining the outputs from these 3 devices, we can now compute the profile. Once such a profile has been obtained, various statistics can be inferred from this, by processing this profile. Depending on the nature of the statistic desired, various techniques may be applied to the profile data to obtain pavement condition. For example, International Roughness Index (IRI) is one such statistic which can be obtained by analyzing the profile data.

2.2 High Speed Inertial Profiler Systems

As described in the earlier section, high speed inertial profilers work by combining reference elevation, a height relative to the reference and the longitudinal distance to produce true road profile. Usually the profile is computed for the traveled wheel path. For each path an accelerometer is used to determine the inertial reference which is the height of the accelerometer at that moment. This is evaluated by double integrating the acceleration measurements. The laser sensor is used to find the height of the road surface with respect to the reference. A distance encoder yields information regarding the longitudinal distance covered. The schematic of this is as shown below.

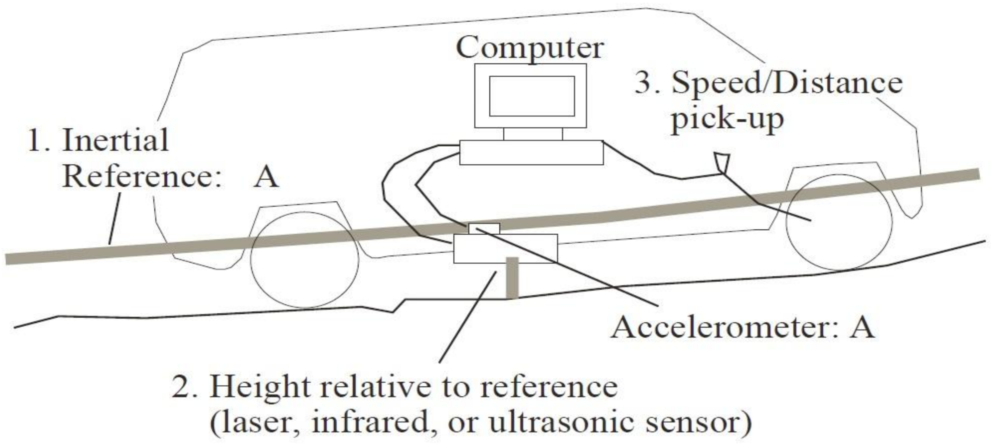

Computer
3. Speed/Distance pick-up
1. Inertial Reference: A
Accelerometer: A
2. Height relative to reference (laser, infrared, or ultrasonic sensor)

Figure 2.1 Road Profiler Systems

$$p(t) = \iint a(t)dtdt - H(t)$$

Where

a(t) is the acceleration,

H(t) is the height measured by the laser sensor.

Figure 2.2 Equation for computing Road Profile

The profile itself is computed using the above equation. A high pass filter is used to remove the effects of long wavelengths on the profile. These wavelengths represent the overall road curvature and underlying grade and are thus attenuated using filters.

As described above, accelerometers are used to compute the inertial reference. As profile is computed for multiple wheel paths, each wheel path uses an accelerometer which is mounted in the measurement vehicle near the wheel path to be measured. Because such setup can require extensive modification and can be time consuming, a move towards developing a profiler module which can be easily mounted or removed from typical vehicles was made. Towards this goal, a portable profiler module was developed as part of TxDOT Research Project 0-6004. Profiles generated by this device were compatible with existing formats as well. The portable profiler computes road surface profile using the same technique as an inertial profiler. The module has all sensors, i.e. accelerometer, laser, power and signal conditioning on board. The distance encoder is attached to the vehicle wheel. Data from these sensors is converted to digital values using a data acquisition board and sent to a notebook PC, which computes the profile. The software is a modified version of the UTA RideConsole program.

The program interfaces with the data acquisition board and retrieves data from it. Data from all sensors are acquired by this process. The program then calls upon a DLL file which contains the necessary functions which actually compute profile. This profile however is phase shifted because of the use of an IIR filter in the profile computation process. A reverse filter is then applied on the profile to correct for the non-linear phase characteristics present in the profile. Figures 2.3 and 2.4 depict the program flow of the program developed at UTA which is used for computing profile and performing this reverse filtering operation.

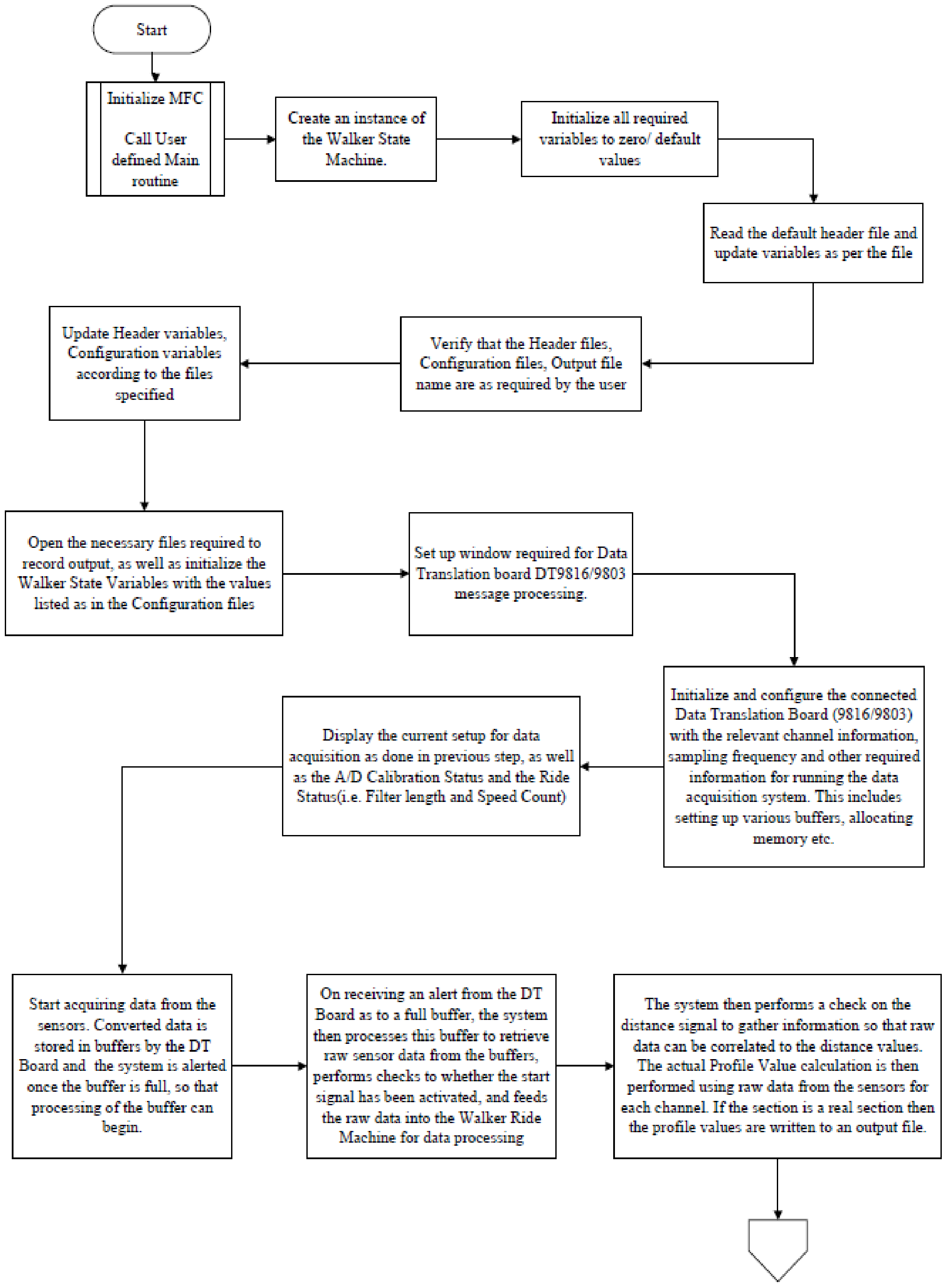
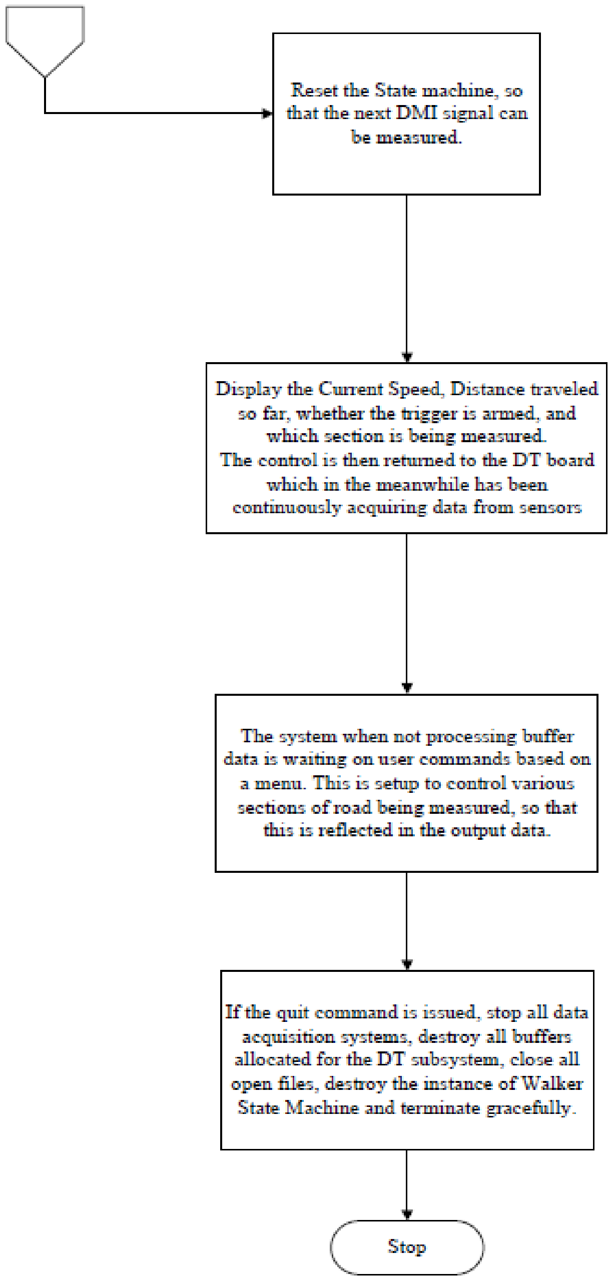

Start
Initialize MFC
Call User defined Main routine
Create an instance of the Walker State Machine.
Initialize all required variables to zero/ default values
Read the default header file and update variables as per the file
Verify that the Header files, Configuration files, Output file name are as required by the user
Update Header variables, Configuration variables according to the files specified
Open the necessary files required to record output, as well as initialize the Walker State Variables with the values listed as in the Configuration files
Set up window required for Data Translation board DT9816/9803 message processing.
Initialize and configure the connected Data Translation Board (9816/9803) with the relevant channel information, sampling frequency and other required information for running the data acquisition system. This includes setting up various buffers, allocating memory etc.
Display the current setup for data acquisition as done in previous step, as well as the A/D Calibration Status and the Ride Status(i.e. Filter length and Speed Count)
Start acquiring data from the sensors. Converted data is stored in buffers by the DT Board and the system is alerted once the buffer is full, so that processing of the buffer can begin.
On receiving an alert from the DT Board as to a full buffer, the system then processes this buffer to retrieve raw sensor data from the buffers, performs checks to whether the start signal has been activated, and feeds the raw data into the Walker Ride Machine for data processing
The system then performs a check on the distance signal to gather information so that raw data can be correlated to the distance values. The actual Profile Value calculation is then performed using raw data from the sensors for each channel. If the section is a real section then the profile values are written to an output file.

Figure 2.3 Flow Diagram of RideConsole program



Reset the State machine, so that the next DMI signal can be measured.
Display the Current Speed, Distance traveled so far, whether the trigger is armed, and which section is being measured.
The control is then returned to the DT board which in the meanwhile has been continuously acquiring data from sensors
The system when not processing buffer data is waiting on user commands based on a menu. This is setup to control various sections of road being measured, so that this is reflected in the output data.
If the quit command is issued, stop all data acquisition systems, destroy all buffers allocated for the DT subsystem, close all open files, destroy the instance of Walker State Machine and terminate gracefully.
Stop

Figure 2.4 Flow Diagram Continued

2.3 Template Analysis Procedure (TAP)

One of the possible applications for profiler data is to use it to detect bumps and dips on the road surface. When developing a new ride equation for TxDOT as part of research project (0-4901), it was determined that the new equation was more sensitive to the occurrence of localized roughness than IRI or existing ride equation. During the evaluation of the newly developed ride equation, comparisons between the current and new equations were made. As part of repeat verification of the equation, it was found that one section was significantly different from previous readings. This was determined to be because of the presence of a pothole in the wheel path of the measured section. Investigations were made to specifically locate the pothole and its characteristics, from profile data. This was accomplished by performing correlation of a bump template with the road profile.

To perform this cross correlation, we need to establish a bump template. The cross correlation between the profile and an appropriate bump signature should be larger when the bump signature is moved over the pothole than when it was not covering the pothole. Therefore a typical bump signature is needed which could be used to cross correlate profile with the signature and thus determine the location of the pothole.

In previous research efforts unrelated to this, to line up two profile measurements to compare for repeatability statistics, a small board was placed under the wheel paths. By use of cross correlation of repeat runs, the position of the board could easily be determined. Fabricated bumps were used to investigate the effects of different bump characteristics on ride. These bumps were investigated to determine if they could be used for cross correlation. The bumps were profiled and inserted into smooth profiles. A particular bump i.e. bump5 was chosen as it exhibited better and smoother transitions when inserted. It was then tested as to how this affected serviceability index (SI) and IRI values. This was also tested using bumps of different lengths and amplitudes. It was established that the new SI was more affected by the wavelength variations than IRI and was better at distinguishing length variations than IRI.

From the results obtained above, a more robust procedure, the template analysis procedure (TAP) was developed for detecting localized roughness. For this first, a specific SI or IRI level is selected. All sections having a SI less than this value is investigated to determine locations needing correction. To achieve this, the first derivative of the newly generated bumps is cross-correlated with the first derivatives of the sections under investigation. If the absolute values of the correlation are greater than a threshold value, then the location is marked. With this it becomes critical that the threshold value be chosen carefully such that it is neither too low nor too high. The former would generate a large set of bumps whereas the latter would result in actual bumps not being detected.

It was determined through experimentation with TAP that the number of bumps detected were almost three times as much as the old system. One of the reasons for this would be the threshold value selected. Another possible cause for this was that even a bump within a bump was counted as a distinct and unique bump. Thus some method to eliminate overlapping bumps needed to be developed. As TAP lists the start and end point of each bump, a program was written which would look for overlaps and thus eliminate such bumps and dips.
Given how TAP uses correlation to identify local roughness, it presents as an application where parallelism could be introduced.

The next chapter introduces the concept of threading, the need for it and the various APIs that support this. Also discussed are some of the tools available to analyze and tune existing applications.

CHAPTER 3

PARALLEL PROGRAMMING AND ANALYSIS TOOLS

3.1 Introduction

Traditionally, computing software algorithms work in a serial manner. That is, the set of instructions required to solve that problem are executed one after the other, and only one instruction may be executed at a time. However, this is a slow approach to solving problems. In order to solve this problem, one of the first suggested solutions was to execute the instructions at a very fast rate. This led to increase in clock speed of the central processing unit (CPU), and therefore a faster instruction execution rate. Other micro-architectural solutions such as instruction pipelining, Superscalar execution, out of order execution enhanced and exploited instruction level parallelism (ILP). But such hardware solutions have reached a barrier where further speedup through ILP is not feasible. Thus to speed up program execution one method is to improve the algorithm itself by breaking it into independent parts so that each processing element can execute parts simultaneously.

The next section introduces and explains how an algorithm can be split into separate parts to allow for simultaneous execution, thus allowing for more work to be done faster. The program units employed for this purpose are called threads, and enabling the use of multiple threads concurrently is known as multi-threading. The sections following this discuss threading packages and libraries, and the last section introduces the use of analytical tools used to check performance of applications to identify potential regions for parallelization in serial applications as well as improve performance of multi-threaded applications.

3.2 Designing for threads

In the previous section, the idea of splitting an algorithm into separate parts was introduced. To achieve this, the algorithm must be scanned for activities that can be executed in parallel. This must also include identification of all dependencies between activities as well. Breaking down the algorithm into these individual tasks and identifying dependencies is known as decomposition. There are three major forms of decomposition. They are:

1. Task Decomposition
2. Data Decomposition
3. Data Flow Decomposition

Task decomposition involves decomposing a program based on the functions it performs. If two tasks can run concurrently, then the developer can schedule them to do so. Only minor modification may be needed to avoid conflicts.

Data decomposition (also called data-level parallelism) works by dividing data among a number of threads all performing the same task. Thus computation on large datasets can be achieved easily using multiple threads, all performing the same calculation, but on different data items.

Data flow decomposition can be done when we have the output of one thread serving as input to another. Here there exists a dependency among threads known as the producer/consumer problem which can be solved using pipeline and wave-front programming patterns in a scalable manner.

Table 3.1 Summary of the Major forms of Decomposition

Decomposition	**Design**	**Comments**
Task	Different activities assigned to different threads	Common in GUI apps
Data	Multiple threads performing the same operation but on different blocks of data	Common in audio processing, imaging and in scientific programming
Data Flow	One thread's output is the input to a second thread	Special care is needed to eliminate startup and shutdown latencies

Although the use of threads does improve performance significantly, this additional benefit gained is not without its pitfalls. Managing simultaneous activities and their interactions can lead to certain issues which may not be present in a serial program. Some of the following issues which arise through parallel programming are listed below.

1. Synchronization - an example of this is the producer-consumer problem. Proper synchronization is required so that the workings of the threads are coordinated.
2. Load Balancing - Distribution of work across threads must be roughly equal. If this is not done, issues could arise with synchronization.
3. Communication - When threads need to share information, proper methods must be established to share data, else results obtained might be at conflict with one another.
4. Scalability - When a program is multithreaded, it might be designed with a specific system in mind. However when it is run on a different machine, it needs to scale properly to make use of all resources available.

In order to solve these issues, a few common parallel programming patterns have been developed. A few of these patterns and their relationships to different decompositions is shown in the following table.

Table 3.2 Common Parallel Programming Patterns

Pattern	**Decomposition**
Task-level parallelism	Task
Divide and Conquer	Task/Data
Geometric Decomposition	Data
Pipeline	Data Flow
Wave front	Data Flow

3.3 Threading Tools

Developing parallel programs especially for embedded systems is complex. To aid program development, modern programming languages provide support through use of special libraries and compiler based directives. Compiler directives such as OpenMP require minimalistic changes to serial code, to enable parallelism. Other methods to parallelize include invocation of threading libraries such as Intel TBB, Win32 threading APIs etc. which are linked to code during compilation. The next few subsections introduce these different threading tools, with special focus on the OpenMP and Intel TBB tools.

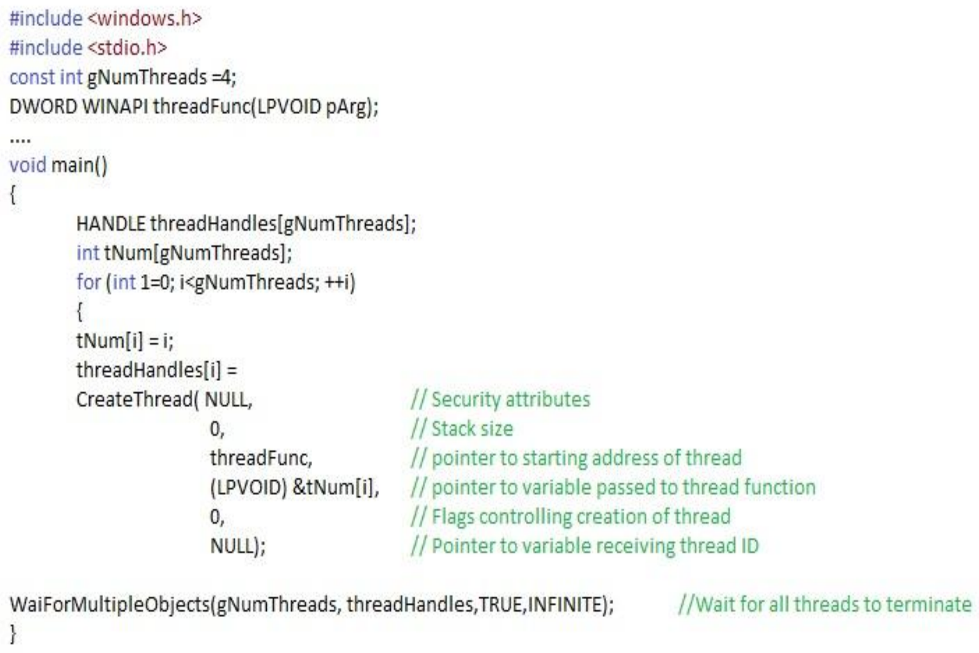
3.3.1 Windows Threads (Win32 API)

Windows threads or Win32 API is a thread library for C and C++. Initially each program gets a single thread also known as the parent thread. This is created by Windows Object Manager. This thread can spawn more threads known as child threads. Win32 uses a CreateThread call, which returns a handle of the thread. The call takes multiple parameters, which allow the programmer to pass access privileges, set the stack size, set whether the thread should start in a suspended mode. It also has a pointer to the function the thread executes and another pointer to store parameters to this function.

Additional API calls such as SetThreadPriority, SuspendThread, and GetThreadPriority enable the programmer to raise the base priority of the thread, suspend execution of a thread and get the current thread's base priority level respectively. After a thread completes execution, the thread can be terminated by calling ExitThread. Windows calls the ExitThread function when a thread finishes execution; however the handle to the thread persists. This can be closed using the CloseHandle call.

Consider a word processing program which uses two threads; thread 1 reads a file, thread 2 writes to it. But this scenario can cause issues if the read thread accesses the file before the write thread finishes. This can cause the program to fail, and such a situation is known as a race condition. The race condition occurs as there is improper synchronization between the two threads. Another problem with multithreading is situations such as deadlocks. A deadlock occurs when two are waiting on each other to finish before completing their execution. Win32 provides multiple objects such as mutexes, semaphores, critical sections, events etc. to allow for thread synchronization. All synchronization primitives work roughly the same but have differing contextual scopes. For e.g. a critical section and a mutex are very similar, but while a mutex can be shared between processes, a critical section cannot. Usually only one thread can gain access to any synchronization primitive.

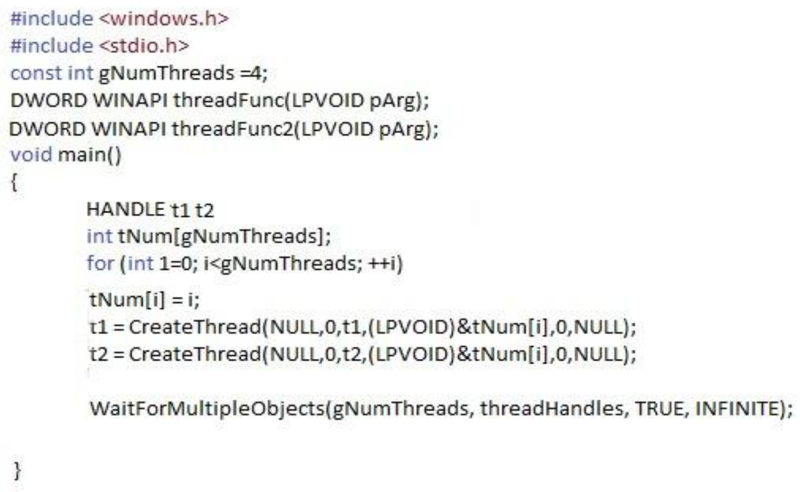
Because of the way threads can be created using Win32, it is easy to decompose a problem into either data based or task based. The following code snippets illustrate how Win 32 threads can be used to achieve both task and data decomposition.



```
#include <windows.h>
#include <stdio.h>
const int gNumThreads =4;
DWORD WINAPI threadFunc(LPVOID pArg);
....
void main()
{
        HANDLE threadHandles[gNumThreads];
        int tNum[gNumThreads];
        for (int 1=0; i<gNumThreads; ++i)
        {
        tNum[i] = i;
        threadHandles[i] =
        CreateThread( NULL,                    // Security attributes
                        0,                     // Stack size
                        threadFunc,            // pointer to starting address of thread
                        (LPVOID) &tNum[i],     // pointer to variable passed to thread function
                        0,                     // Flags controlling creation of thread
                        NULL);                 // Pointer to variable receiving thread ID

WaiForMultipleObjects(gNumThreads, threadHandles,TRUE,INFINITE);        //Wait for all threads to terminate
}
```

Figure 3.1 Data Decomposition example using Win32 API

In the first example we see that an array of threads is created with all threads executing the same function. However as the parameters to the function are different, i.e. data passed to the function is different; each thread tackles a different data set. This is Data Decomposition.



```
#include <windows.h>
#include <stdio.h>
const int gNumThreads =4;
DWORD WINAPI threadFunc(LPVOID pArg);
DWORD WINAPI threadFunc2(LPVOID pArg);
void main()
{
        HANDLE t1 t2
        int tNum[gNumThreads];
        for (int 1=0; i<gNumThreads; ++i)
        tNum[i] = i;
        t1 = CreateThread(NULL,0,t1,(LPVOID)&tNum[i],0,NULL);
        t2 = CreateThread(NULL,0,t2,(LPVOID)&tNum[i],0,NULL);

        WaitForMultipleObjects(gNumThreads, threadHandles, TRUE, INFINITE);

}
```

Figure 3.2 Task Decomposition example using Win32 API

In the above example, multiple threads are created but each thread has a different function to execute. This is a clear case of Task Decomposition.

Although Win32 has good synchronization primitives and threading support, the task of thread resource management is left to the developer. This can lead to longer development times and more complex issues may arise.

3.3.2 OpenMP (Open Multi Processing)

OpenMP is an API that supports multi-platform shared memory multiprocessing programming in C, C+ and FORTRAN. It consists of compiler pragmas, function calls and environment variables that explicitly instruct the compiler how and where to use parallelism in the application. As the details of the actual threading mechanism is left to the compiler, the developer can spend more time determining where parallelism can be introduced, and how best to restructure algorithms for better performance on a multi-core platform.

OpenMP employs a shared memory model, which means that all threads have access to the same globally shared memory. Data can be shared or private, and accordingly private data can be accessed only by the owning thread.

OpenMP implements multi-threading by using a master thread which spawns a number of slave threads and divides a task among these slave threads. These slave threads can work concurrently with the runtime environment performing the thread allocation to specific cores of the processor. After execution of the slave threads, they revert control to the master thread which continues with further program execution.

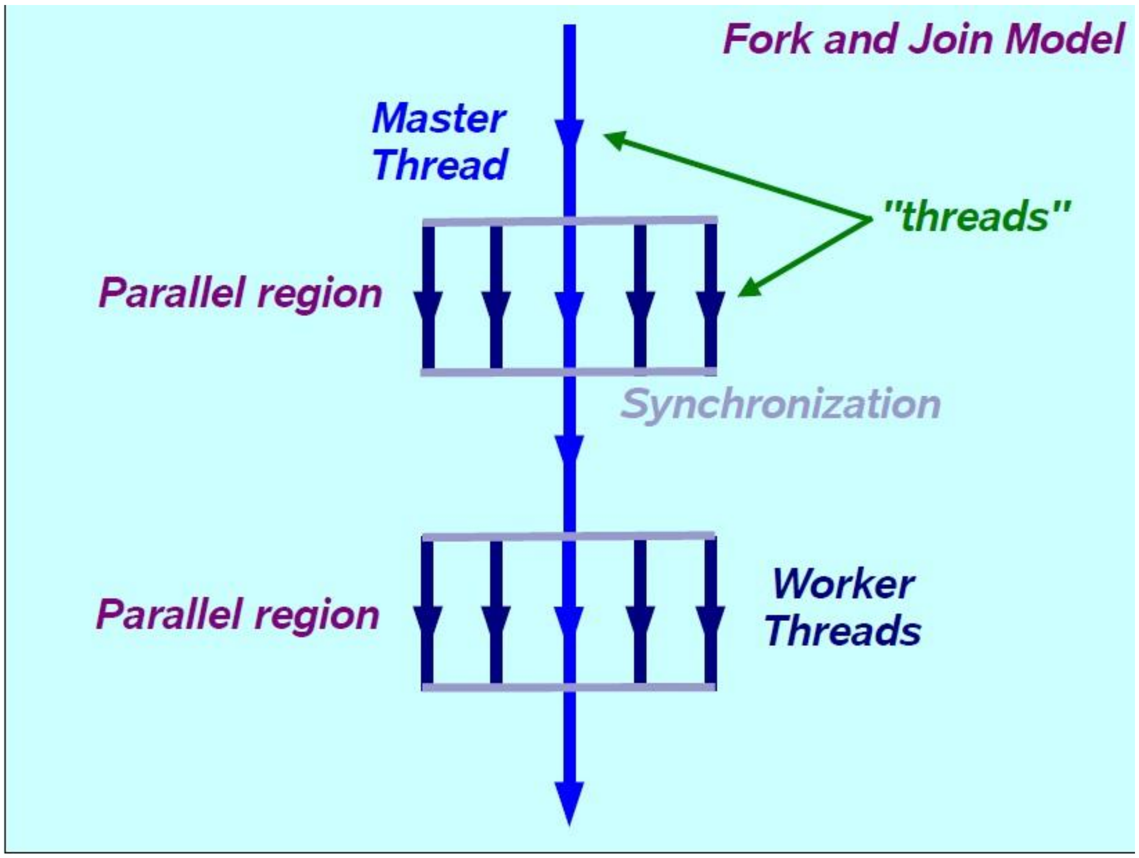


Figure 3.3 OpenMP execution model

Programs usually spend most of their time executing loops. OpenMP works very well to process loops in parallel, i.e. OpenMP utilizes work-sharing and hence can split iterations of the loop across multiple threads. OpenMP compiler and runtime libraries take care of details like creating, initializing, managing and killing threads.

As stated earlier, all OpenMP directives usually consist of pragmas. Usually this is indicated with a **#pragma omp** followed by parameters ending in a newline. The pragma usually applies only to the statement immediately following it. Because OpenMP utilizes compiler directives, the serial nature of the code is untouched, and thus on an unsupported compiler, the program will compile and execute serially as intended. This gives developers an advantage when converting serial applications to parallel as the application need not be rewritten from scratch. However it must be remembered as the actual parallelization is left to the compiler; it may not parallelize a program in the same way as the developer intended it to. OpenMP also allows a program to be both serial and parallel, i.e. different sections of the program can execute either in serial or parallel.

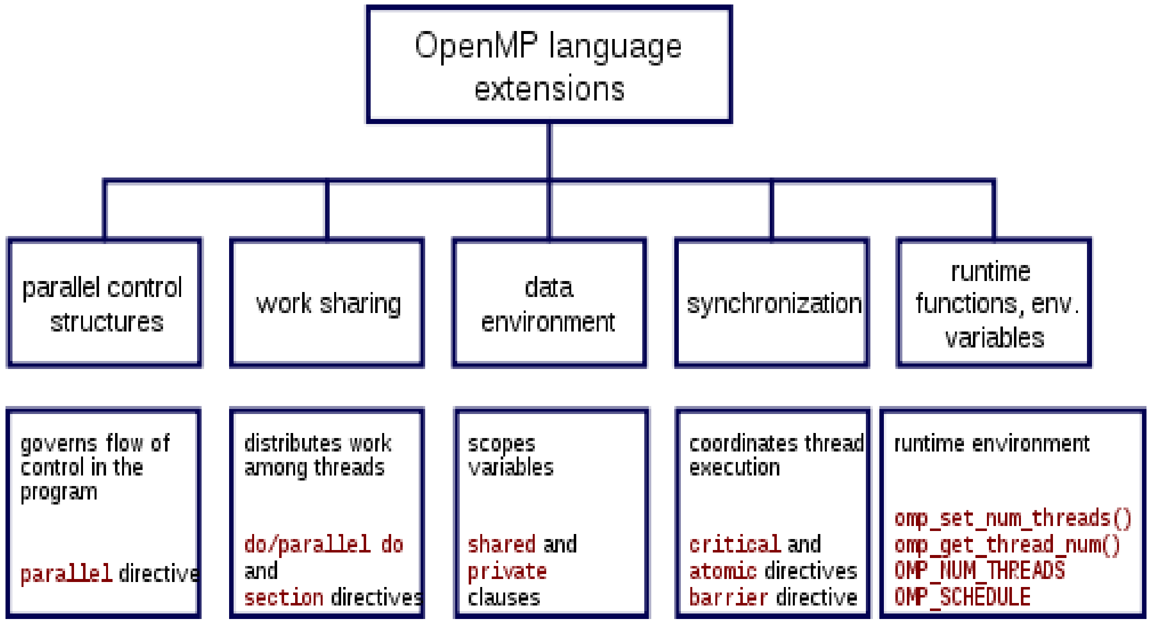
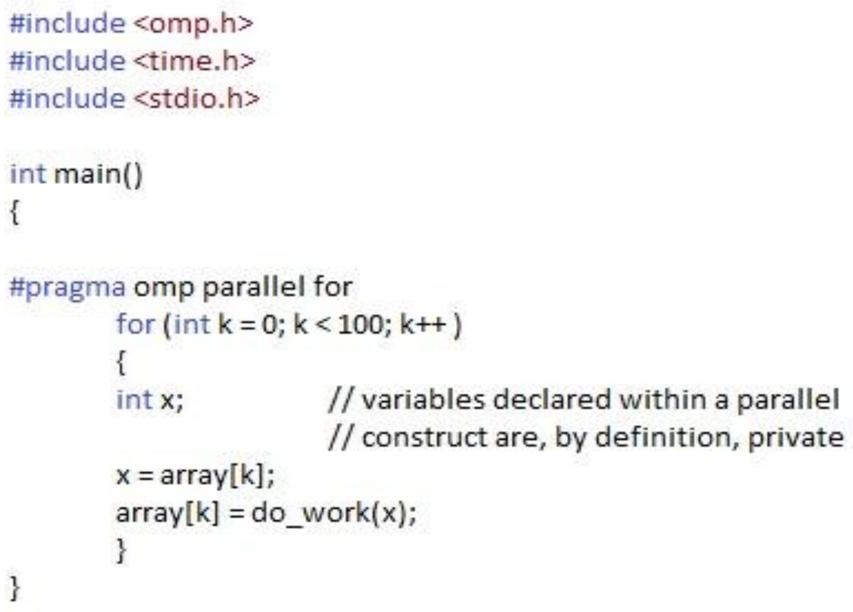

OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives
barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

Figure 3.4 OpenMP Core elements

As seen from the chart, there exist multiple directives to achieve and manage parallelization. Directives for thread creation, workload distribution, data environment managing, synchronization etc. are available.

The parallel pragma starts a parallel block. It creates a team of N threads (usually the number of CPU cores), all of which execute the following statement. The threads merge back into one after execution. Loop directives such as the ‘for’ splits a for-loop in such a way that each thread handles a different portion of the loop. The following example illustrates how a ‘for’ loop can be parallelized using OpenMP. It also demonstrates data decomposition.



```
#include <omp.h>
#include <time.h>
#include <stdio.h>

int main()
{

#pragma omp parallel for
          for (int k = 0; k < 100; k++ )
          {
          int x;              // variables declared within a parallel
                              // construct are, by definition, private
          x = array[k];
          array[k] = do_work(x);
          }
}
```

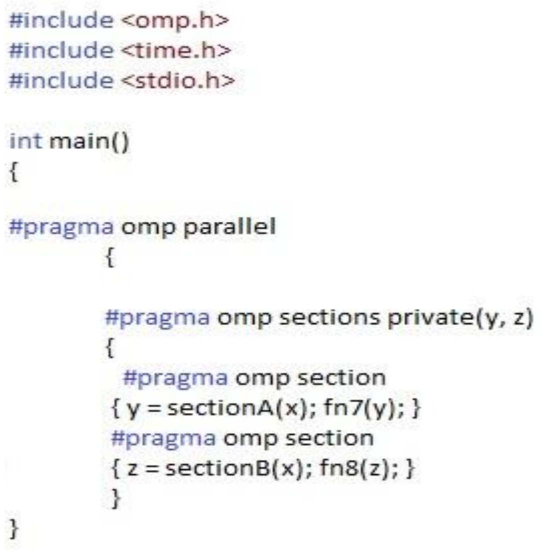
Figure 3.5 Data decomposition example using OpenMP

Threading a loop is to convert independent loop iterations to threads and run them in parallel. As long as there is no dependency between loops, statements in two different iterations may be executed simultaneously. Therefore the developer must first identify and if needed restructure code to ensure there is no loop-carried dependency.

The shared and private clauses and variations on this such as the firstprivate clause enable the programmer to specify the scope of a variable between threads. With OpenMP utilizing a shared memory model, all variables in a parallel region are shared. The exceptions to this are loop indices, variables local to the parallel block and any variables listed in the clauses.

OpenMP also provides other directives to ensure thread-safety and synchronization. Mutual exclusion can be achieved by the use of directives such as the atomic, critical and reduction clause.

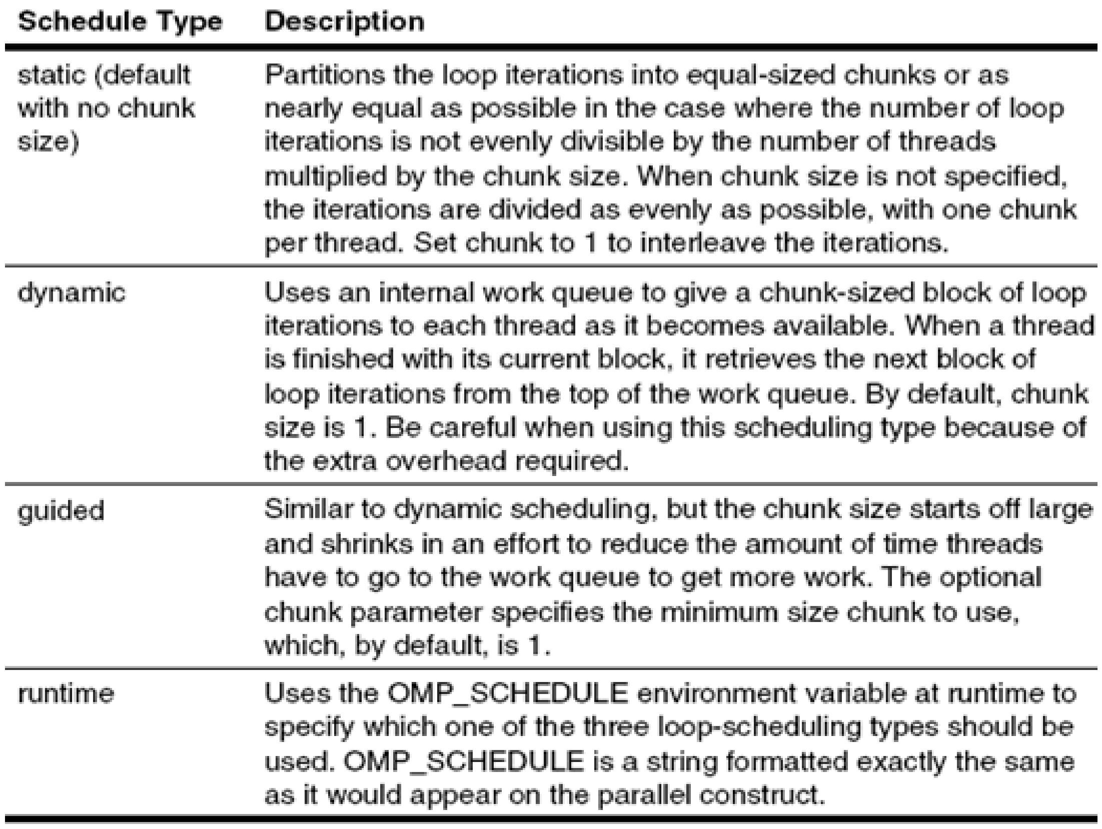
Although OpenMP is extremely proficient at threading loops, it can also be used to distribute sections of an application among multiple threads. The example below shows how this is achieved using the work-sharing construct ‘section’



```
#include <omp.h>
#include <time.h>
#include <stdio.h>

int main()
{

#pragma omp parallel
          {

          #pragma omp sections private(y, z)
          {
            #pragma omp section
           { y = sectionA(x); fn7(y); }
           #pragma omp section
           { z = sectionB(x); fn8(z); }
           }
}
```

Figure 3.6 Task Decomposition example using OpenMP

OpenMP also allows for adjusting workload among cores by the use of scheduling.

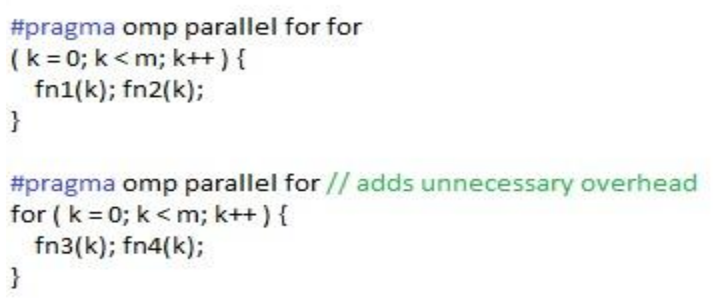


Schedule Type	Description
static (default with no chunk size)	Partitions the loop iterations into equal-sized chunks or as nearly equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. When chunk size is not specified, the iterations are divided as evenly as possible, with one chunk per thread. Set chunk to 1 to interleave the iterations.
dynamic	Uses an internal work queue to give a chunk-sized block of loop iterations to each thread as it becomes available. When a thread is finished with its current block, it retrieves the next block of loop iterations from the top of the work queue. By default, chunk size is 1. Be careful when using this scheduling type because of the extra overhead required.
guided	Similar to dynamic scheduling, but the chunk size starts off large and shrinks in an effort to reduce the amount of time threads have to go to the work queue to get more work. The optional chunk parameter specifies the minimum size chunk to use, which, by default, is 1.
runtime	Uses the OMP_SCHEDULE environment variable at runtime to specify which one of the three loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as it would appear on the parallel construct.

Figure 3.7 OpenMP Scheduling schemes

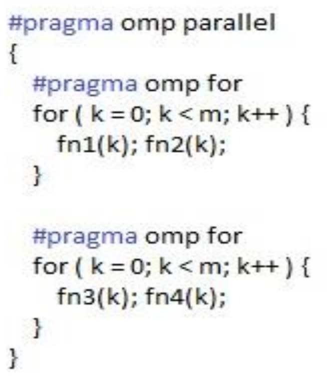
OpenMP also provides certain functions which are available through the header file. These functions allow for operations such as setting the number of threads, set scheduling patterns, querying the number of processors, general purpose locking routines (semaphores).

Because OpenMP is a compiler dependent method of parallelizing programs, there exists some level of overhead caused by the fork-join execution model. The compiler and run time libraries usually have built in optimizations which enable faster performance and lesser threading overhead. However application performance can be improved further by the programmer by threading intelligently.



```
#pragma omp parallel for for
( k = 0; k < m; k++ ) {
  fn1(k); fn2(k);
}

#pragma omp parallel for // adds unnecessary overhead
for ( k = 0; k < m; k++ ) {
  fn3(k); fn4(k);
}
```

Figure 3.8 Example code showing extra overhead



```
#pragma omp parallel
{
  #pragma omp for
  for ( k = 0; k < m; k++ ) {
    fn1(k); fn2(k);
  }

  #pragma omp for
  for ( k = 0; k < m; k++ ) {
    fn3(k); fn4(k);
  }
}
```

Figure 3.9 Example code with improved performance due to lesser overhead

In the examples listed above, the first has two parallel for pragmas; one for each for loop. At execution, each pragma causes the application to use the fork-join mechanism twice; once for each loop. The second example has only a single parallel pragma, with two for loop directives being used within the parallel block. This causes the block to employ only a single fork-join, therefore reducing overhead incurred in thread suspension and resumption. The second example is functionally the same but runs faster than the first.

Thus, OpenMP is a small yet powerful API which allows a programmer to add parallelism into existing source code without significantly having to rewrite it.

3.3.3 Intel Thread Building Blocks (Intel TBB)

Intel Threading Building Blocks is a runtime-based parallel programming model for C++ code that uses threads. It consists of a template-based runtime library which can harness the latent performance of multicore processors. It does not require special languages or compilers. It is designed to promote scalable data parallel programming.

Intel TBB library consists of data structures and algorithms that allow a programmer to avoid issues arising from thread creation, synchronization etc. It provides a high level abstraction for parallelism by shifting focus from threads to the actual work that the threads perform. This is similar to OpenMP in that the actual threading mechanism is hidden from the programmer. This enables the programmer to focus more on parallelizing than on constructs used for the parallelizing process.

3.4 Performance Analysis and Tuning of an Application

Developing an application is generally easy. Developing an application which is optimized to run at maximum performance, making use of resources in the right way is a challenge. The application must also be able to scale correctly. To achieve this first the area of the system which is critical for performance improvement must be identified. This is called the bottleneck. The application must then be modified such that the application functionality is not affected yet the bottleneck is mitigated or removed altogether.

Performance analysis of the program, known as profiling, is the investigation of a program's behavior using information gathered as the program executes. Tuning is the process by which a program is profiled and then optimized for better execution.

Performance itself can then be increased through techniques such as code optimization where portions of a program may be rewritten to run faster. This could also involve utilizing different algorithms to perform a task. An example of this would be to utilize a better sorting algorithm such as the quicksort.

There are multiple performance analysis tools available. All of them provide some basic form of profiling, a memory analysis and error detection tool. Intel developed the Parallel Studio and the VTune Amplifier packages, which can tune both serial and parallel programs. It provides support to multithreaded applications as it contains tools to detect common parallel programming issues such as races, deadlocks and memory errors. These packages analyze an application by profiling it during execution, and collecting various performance related and statistical data. In the following subsections, these tools will be introduced and discussed briefly.

3.4.1 Intel VTune Amplifier XE

The Intel VTune Amplifier XE is part of the Parallel Studio XE tool and provides information on code performance for users developing serial and multithreaded applications on Windows and Linux. The VTune Amplifier package enables a developer to identify the hot-spots in a program, i.e. locate the most time-consuming regions of the application. It also can identify regions of code which do not make effective use of resources available. Impact of different synchronization methods, the number of threads used, etc. can be measured using this tool.
Some of the key features of the VTune Amplifier are listed below.

- Powerful profile analysis - VTune includes aspects such as timeline, filtering and frame analysis which help turn profiled data into actionable information
- Low Overhead - VTune keeps the overhead incurred in collecting profiled data to a minimum, allowing for faster data collection.
- Hotspot analysis - Determine where the application is spending most of the CPU time.
- Concurrency - Identify which cores are underutilized.

- Event Based Sampling - Hardware Event Based Sampling allows the programmer to find tuning opportunities like cache misses, increased CPI (clocks per instruction), retire stalls etc.
- Source code Analysis - Pinpoint bottlenecks down to the exact line of code causing the issue. This can also be utilized to view the source in assembly and pinpoint a specific assembly instruction which is the cause of the bottleneck. Thus issues caused by micro-architecture can be mitigated by rewriting code.
- Locks and Waits - Allows for identifying common causes of slow performance in parallel programs, for e.g. waiting for a long time on a lock, while cores are underutilized.

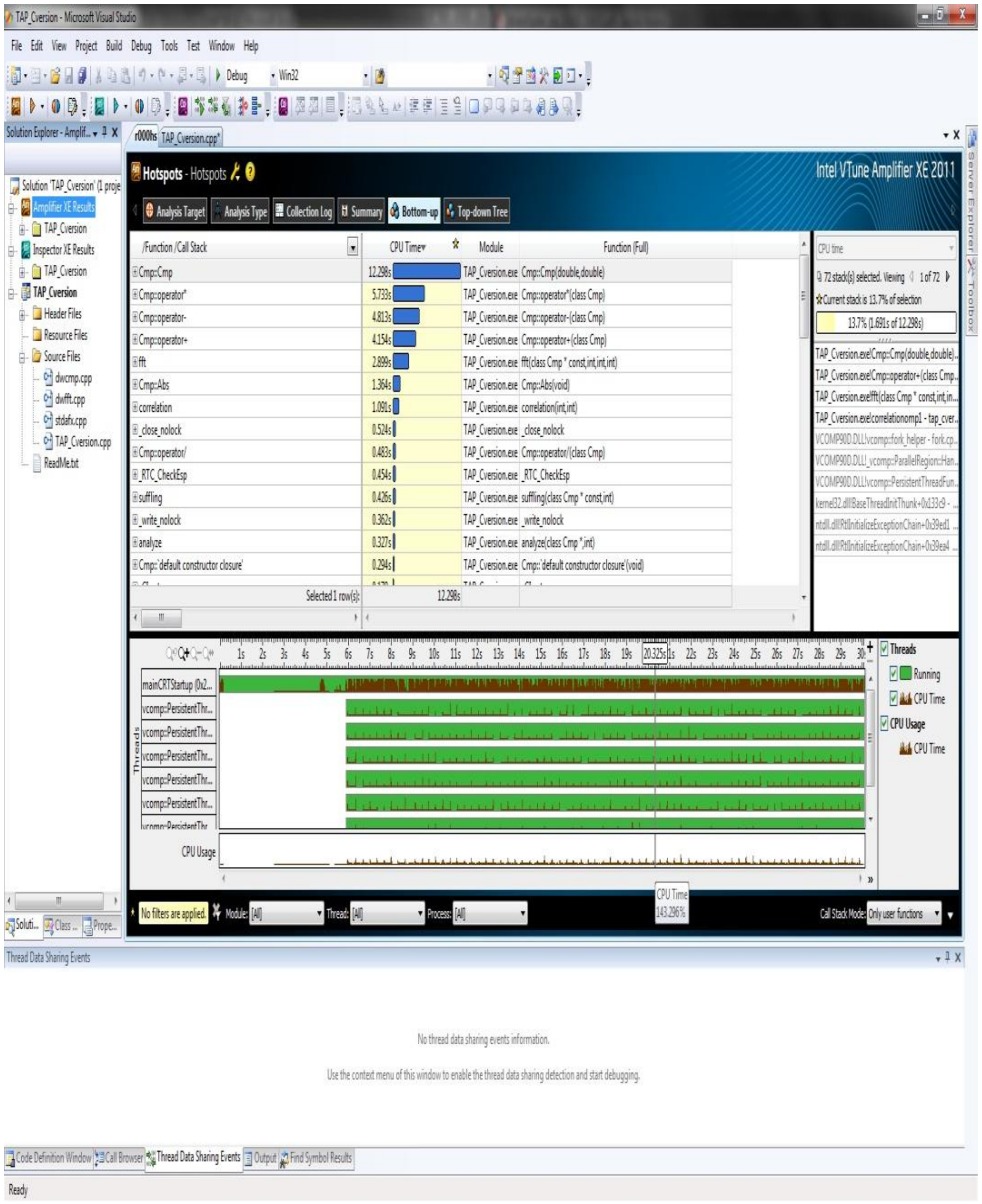
3.4.2 Intel Inspector XE

The Intel Inspector XE is part of the Parallel Studio XE tool which facilitates application reliability, by effectively finding crucial memory and threading defects. It gives detailed insights into application memory and threading behavior to improve application reliability. Although it is relatively easy to thread a program, locating threading errors and memory issues takes a long time. This tool helps in quick pinpointing of issues caused by such mistakes.

Some of the key features of Inspector XE include:

- Memory Checking Analysis for serial and parallel applications - It enables easy detection of memory leaks and memory corruption along with inconsistent memory API usage.
- Thread Checking Analysis- Allows the programmer to easily detect data races, deadlocks, and track memory accesses between threads.
- Error Map feature - It directly maps errors to source-code line and displays the call stack. This enables the programmer to speed up the process of detecting and fixing errors.

The following figures illustrate how the above mentioned tools integrate into the Microsoft Visual Studio IDE. Also listed are a few workflow charts which describe the process to be followed while using tools from the Parallel Studio XE.



/Function /Call Stack	CPU Time	Module	Function (Full)
Cmp::Cmp	12.298s	TAP_Cversion.exe	Cmp::Cmp(double,double)
Cmp::operator*	5.733s	TAP_Cversion.exe	Cmp::operator*(class Cmp)
Cmp::operator-	4.813s	TAP_Cversion.exe	Cmp::operator-(class Cmp)
Cmp::operator+	4.154s	TAP_Cversion.exe	Cmp::operator+(class Cmp)
fft	2.899s	TAP_Cversion.exe	fft(class Cmp * const,int,int,int)
Cmp::Abs	1.364s	TAP_Cversion.exe	Cmp::Abs(void)
correlation	1.091s	TAP_Cversion.exe	correlation(int,int)
_close_nolock	0.524s	TAP_Cversion.exe	_close_nolock
Cmp::operator/	0.483s	TAP_Cversion.exe	Cmp::operator/(class Cmp)
_RTC_CheckEsp	0.454s	TAP_Cversion.exe	_RTC_CheckEsp
suffling	0.426s	TAP_Cversion.exe	suffling(class Cmp * const,int)
_write_nolock	0.362s	TAP_Cversion.exe	_write_nolock
analyze	0.327s	TAP_Cversion.exe	analyze(class Cmp *,int)
Cmp::`default constructor closure'	0.294s	TAP_Cversion.exe	Cmp::`default constructor closure'(void)
Selected 1 row(s):	12.298s		

72 stack(s) selected. Viewing 1 of 72

Current stack is 13.7% of selection

13.7% (1.691s of 12.298s)

```
TAP_Cversion.exe!Cmp::Cmp(double,double)...
TAP_Cversion.exe!Cmp::operator+(class Cmp...
TAP_Cversion.exe!fft(class Cmp * const,int,in...
TAP_Cversion.exe!correlationomp1 - tap_cver...
VCOMP90D.DLL!vcomp::fork_helper - fork.cp...
VCOMP90D.DLL!_vcomp::ParallelRegion::Han...
VCOMP90D.DLL!vcomp::PersistentThreadFun...
kernel32.dll!BaseThreadInitThunk+0x133c9 - ...
ntdll.dll!RtlInitializeExceptionChain+0x39ed1 ...
ntdll.dll!RtlInitializeExceptionChain+0x39ea4 ...
```

No filters are applied. Module: [All] Thread: [All] Process: [All] Call Stack Mode: Only user functions

No thread data sharing events information.

Use the context menu of this window to enable the thread data sharing detection and start debugging.

Figure 3.10 VTune Amplifier Tool Integration into Visual Studio

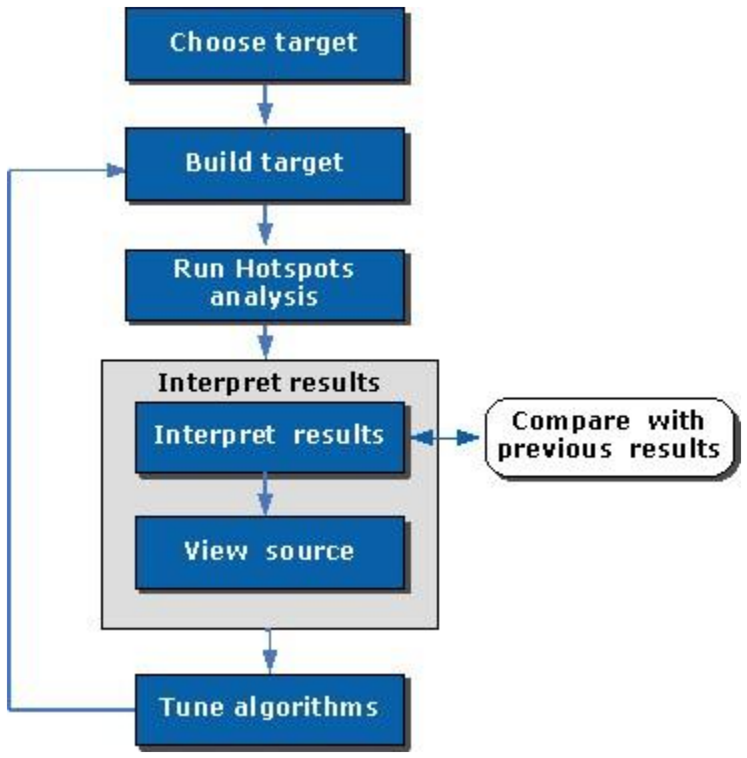


Figure 3.11 Workflow steps to identify and analyze Hotspots

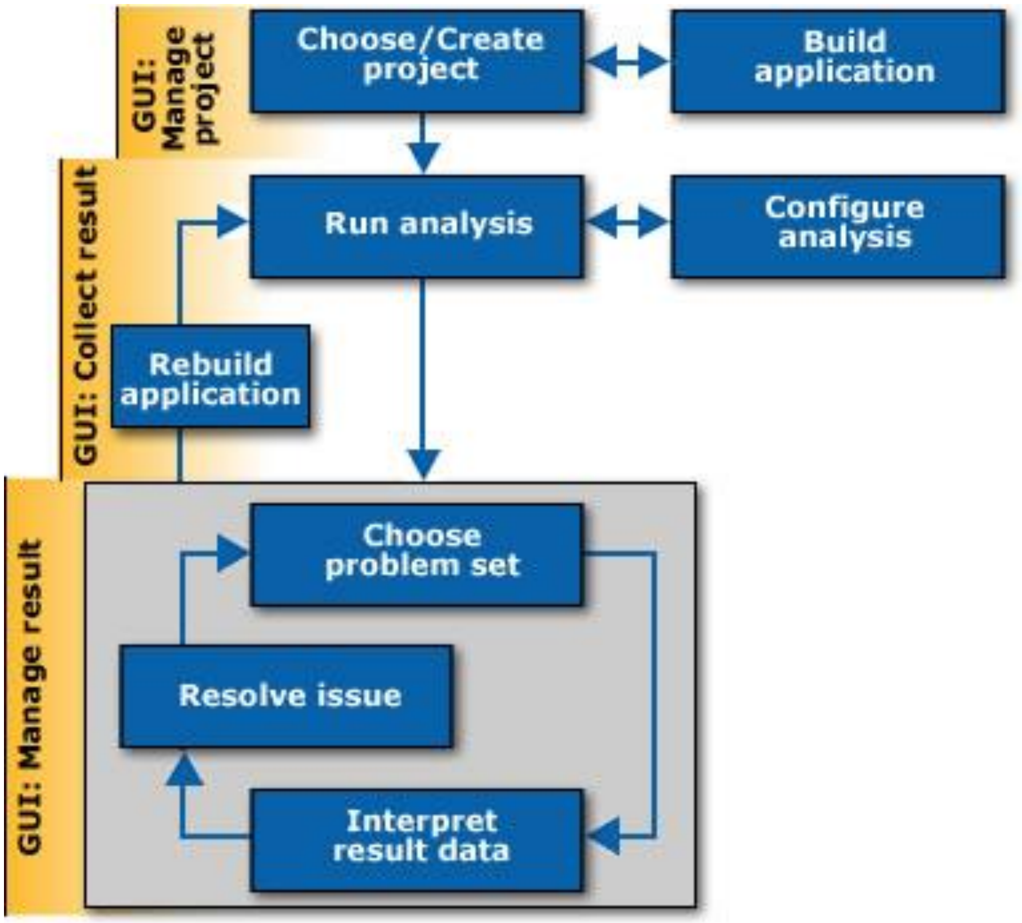


Figure 3.12 Workflow steps to identify, analyze, and resolve memory errors

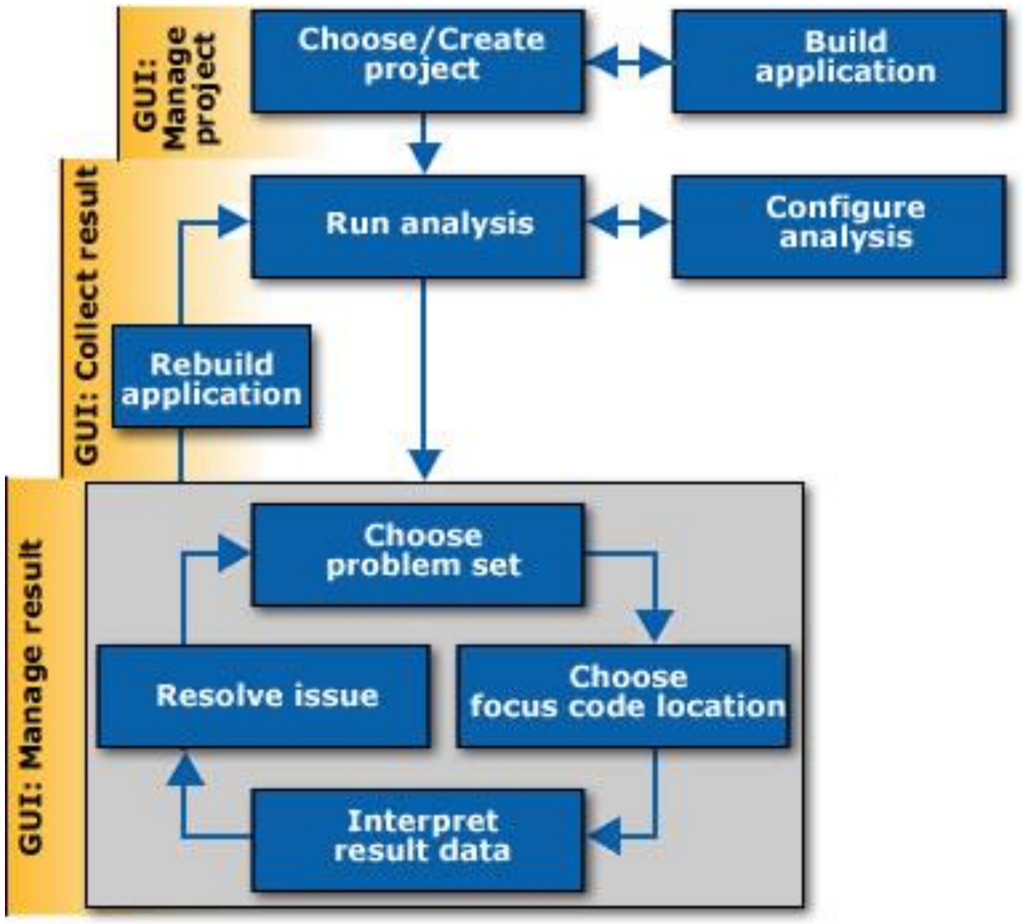


Figure 3.13 Workflow steps to identify, analyze, and resolve threading errors.

The following chapter discusses how the some of the APIs and tools introduced in this chapter can be used to improve the performance of TAP.

CHAPTER 4

PERFORMANCE TUNING OF THE TEMPLATE ANALYSIS TOOL

4.1 Introduction

In the second chapter, the Template Analysis Procedure was introduced. This is a program which uses profile data as input and detects bumps and dips in a section of road. The third chapter introduced concepts of parallel programming as well as discussed some of the available threading packages. Analytical software used to identify issues with program execution, tools to improve performance and further tune existing applications were also presented.

The following chapter will examine the various methods applied to the Template Analysis tool to enhance and extract better performance as compared to the initial serial version. Section 2 describes the baseline performance of TAP. This is the serial version of the program and different issues identified during the course of code evaluation are discussed. Section 3 shows how some of the issues identified in section 1 can be tackled. Performance measurements of TAP with these improvements are then made. Section 4 discusses program correctness of TAP with the addition of parallelism.

4.2 Baseline Performance Analysis of TAP

Before an application is considered for any sort of performance tuning, a baseline profile must first be established. This allows for a comparison between the performance of a modified version of the application and the original application. This baseline must be measurable and reproducible.

4.2.1 Run Times and Resource Utilization

One of the most common baseline parameter is run time or execution time of a program. Under analogous system conditions, a program is expected to produce highly similar execution times. However, there will always be some amount of extraneous behavior which causes run time to differ minutely. To account for this, each run was performed multiple times, and the average run time was computed.

Another important factor to consider is the size of the dataset on which a program works. If the dataset happens to differ each time, then the run times obtained are invalid. To compare performance, it is necessary that the data is the same each time. To ensure this, all performance metrics were measured using the same road profile data set.

The following table shows the different run times taken determine the average run time performance of the application. The table also shows the actual CPU time taken during these runs. This is important as any delay caused by the user is removed from analysis.

Table 4.1 Run times and actual CPU times of the serial version of TAP

Run Number	**Total Time (s)**	**Actual CPU Time (s)**
1	26.055	20.021
2	26.459	19.975
3	26.343	19.924
4	26.672	19.758

Another parameter to consider is the effective use of resources available to the program. Since the program is being run on multi-core systems, each core can be considered to be a resource. Effective core utilization is very important on a multi-core system. However, if a program works on a dual core machine, utilizing both cores, it may not scale well on a system with more cores. Thus scalability is also important when parallelizing a program.

However, because we are now analyzing the serial version of the program, we expect the program to utilize a single core on the system. This is exhibited in the core usage histogram as shown below in Figure 4.1. Similarly Figure 4.2 shows the number of threads being currently executed by the program. This also shows that only a single thread is created and used.

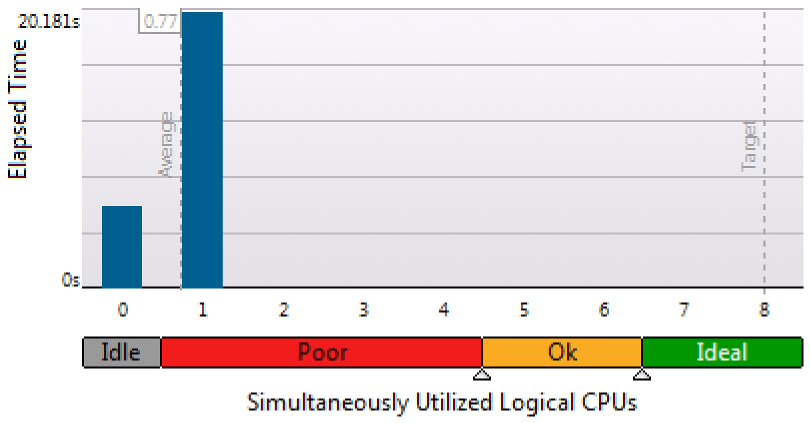


Figure 4.1 Core Utilization in serial TAP

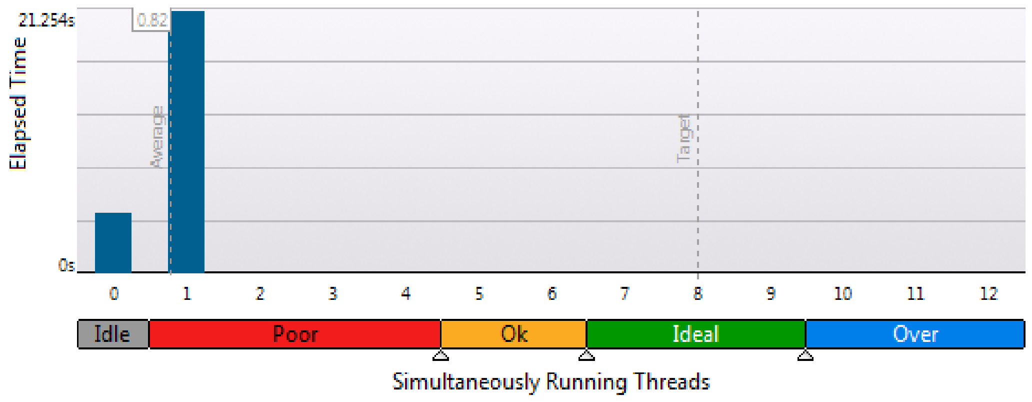


Figure 4.2 Thread Concurrency in serial TAP

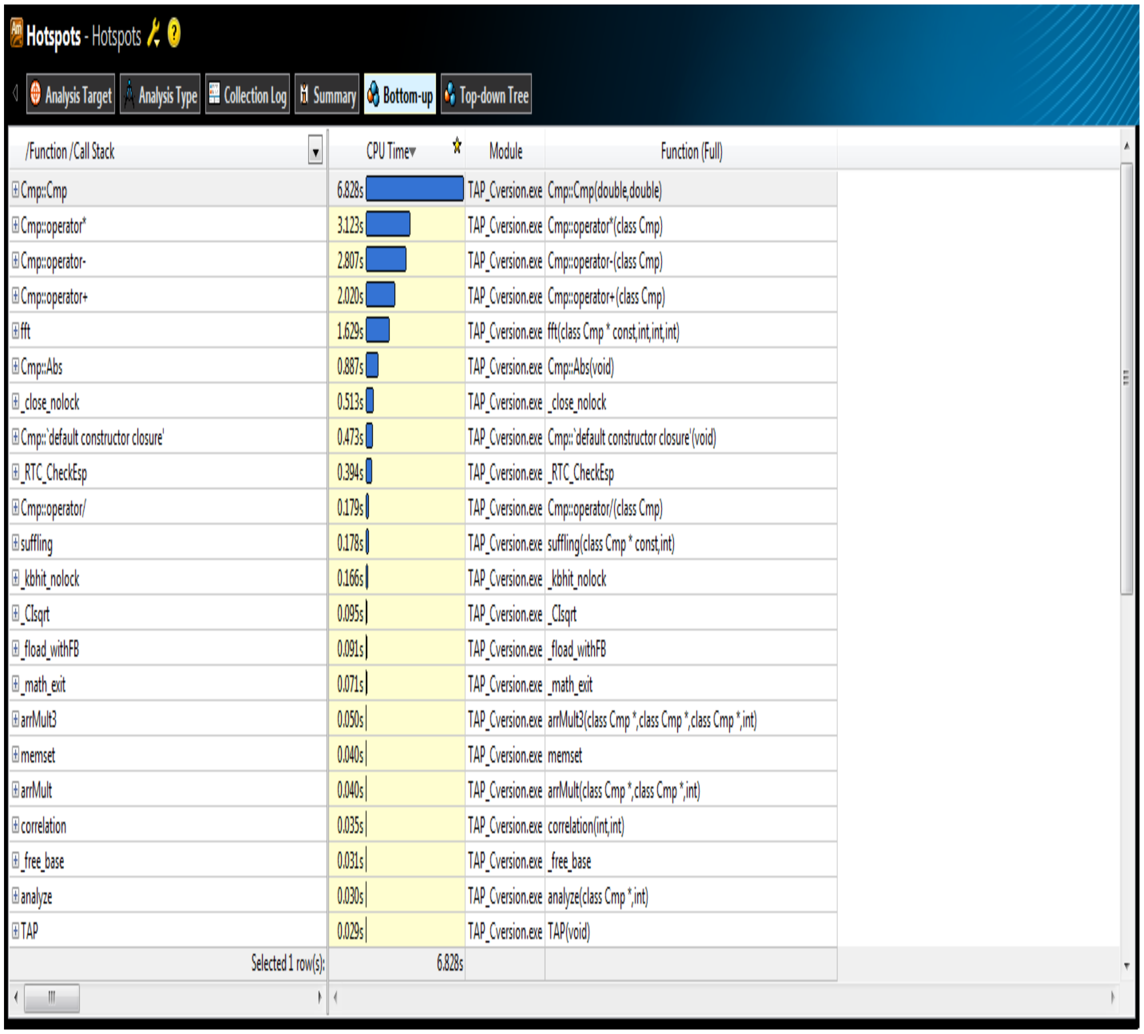
4.2.2 Using the VTune Amplifier to detect hotspots

The Intel VTune Amplifier package has different methods to analyze an application. When an application needs to be analyzed for locations where performance can be improved, these methods assist in identifying these regions. The tool provides both user-mode sampled and hardware event-based sampled data for analysis.

One of the most important methods is of type Hotspot Analysis. This method helps identify which region of code takes the most amount of time to execute. It also collects stack and call tree information.

TAP is analyzed using the Hotspots method. The tool is run multiple times to average out any possible external influences. Figure 4.3 shows one instance of the hotspot window.

The tool shows that CPU time is spent mostly in the Cmp::Cmp. Expanding the call stack helps determine the different paths in code calling on this one function. The next three regions where time is spent are again based on the Cmp function. VTune also provides for a method to identify specific lines in code which are responsible for the hotspot. It can be determined from this that the hotspot is mainly caused in dereferencing the Cmp object. VTune even allows for an assembly code view, which pinpoints the hotspot down to a single instruction as shown in Figure 4.4.

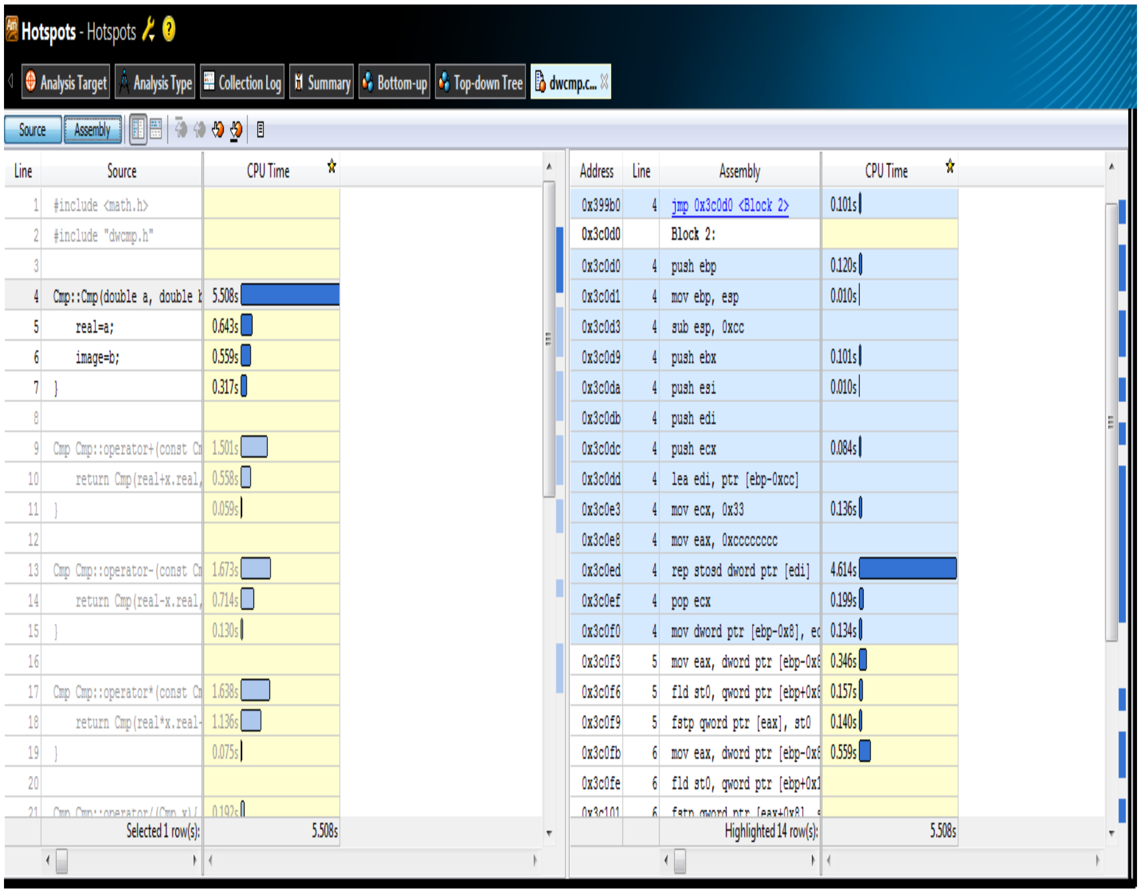


Hotspots - Hotspots

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree

/Function /Call Stack	CPU Time	Module	Function (Full)
Cmp::Cmp	6.828s	TAP_Cversion.exe	Cmp::Cmp(double,double)
Cmp::operator*	3.123s	TAP_Cversion.exe	Cmp::operator*(class Cmp)
Cmp::operator-	2.807s	TAP_Cversion.exe	Cmp::operator-(class Cmp)
Cmp::operator+	2.020s	TAP_Cversion.exe	Cmp::operator+(class Cmp)
fft	1.629s	TAP_Cversion.exe	fft(class Cmp * const,int,int,int)
Cmp::Abs	0.887s	TAP_Cversion.exe	Cmp::Abs(void)
_close_nolock	0.513s	TAP_Cversion.exe	_close_nolock
Cmp::`default constructor closure'	0.473s	TAP_Cversion.exe	Cmp::`default constructor closure'(void)
_RTC_CheckEsp	0.394s	TAP_Cversion.exe	_RTC_CheckEsp
Cmp::operator/	0.179s	TAP_Cversion.exe	Cmp::operator/(class Cmp)
suffling	0.178s	TAP_Cversion.exe	suffling(class Cmp * const,int)
_kbhit_nolock	0.166s	TAP_Cversion.exe	_kbhit_nolock
_CIsqrt	0.095s	TAP_Cversion.exe	_CIsqrt
_fload_withFB	0.091s	TAP_Cversion.exe	_fload_withFB
_math_exit	0.071s	TAP_Cversion.exe	_math_exit
arrMult3	0.050s	TAP_Cversion.exe	arrMult3(class Cmp *,class Cmp *,class Cmp *,int)
memset	0.040s	TAP_Cversion.exe	memset
arrMult	0.040s	TAP_Cversion.exe	arrMult(class Cmp *,class Cmp *,int)
correlation	0.035s	TAP_Cversion.exe	correlation(int,int)
_free_base	0.031s	TAP_Cversion.exe	_free_base
analyze	0.030s	TAP_Cversion.exe	analyze(class Cmp *,int)
TAP	0.029s	TAP_Cversion.exe	TAP(void)
Selected 1 row(s):	6.828s		

Figure 4.3 Hotspot Detection in TAP



Hotspots - Hotspots

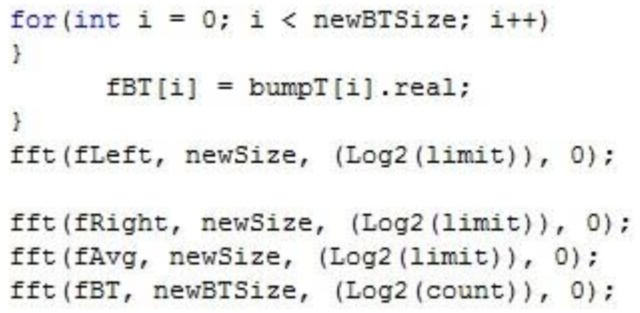
Line	Source	CPU Time
1	#include <math.h>	
2	#include "dwcmp.h"	
3		
4	Cmp::Cmp(double a, double b	5.508s
5	real=a;	0.643s
6	image=b;	0.559s
7	}	0.317s
8		
9	Cmp Cmp::operator+(const Cm	1.501s
10	return Cmp(real+x.real,	0.558s
11	}	0.059s
12		
13	Cmp Cmp::operator-(const Cm	1.673s
14	return Cmp(real-x.real,	0.714s
15	}	0.130s
16		
17	Cmp Cmp::operator*(const Cm	1.638s
18	return Cmp(real*x.real-	1.136s
19	}	0.075s
20		
21	Cmp Cmp::operator/(Cmp x){	0.192s
	Selected 1 row(s):	5.508s

Address	Line	Assembly	CPU Time
0x399b0	4	jmp 0x3c0d0 <Block 2>	0.101s
0x3c0d0		Block 2:	
0x3c0d0	4	push ebp	0.120s
0x3c0d1	4	mov ebp, esp	0.010s
0x3c0d3	4	sub esp, 0xcc	
0x3c0d9	4	push ebx	0.101s
0x3c0da	4	push esi	0.010s
0x3c0db	4	push edi	
0x3c0dc	4	push ecx	0.084s
0x3c0dd	4	lea edi, ptr [ebp-0xcc]	
0x3c0e3	4	mov ecx, 0x33	0.136s
0x3c0e8	4	mov eax, 0xcccccccc	
0x3c0ed	4	rep stosd dword ptr [edi]	4.614s
0x3c0ef	4	pop ecx	0.199s
0x3c0f0	4	mov dword ptr [ebp-0x8], ec	0.134s
0x3c0f3	5	mov eax, dword ptr [ebp-0x8	0.346s
0x3c0f6	5	fld st0, qword ptr [ebp+0x8	0.157s
0x3c0f9	5	fstp qword ptr [eax], st0	0.140s
0x3c0fb	6	mov eax, dword ptr [ebp-0x8	0.559s
0x3c0fe	6	fld st0, qword ptr [ebp+0x1	
0x3c101	6	fstp qword ptr [eax+0x8], s	
		Highlighted 14 row(s):	5.508s

Figure 4.4 Source Code view of Hotspot

The first four hotspots detected by the tool are not candidates for parallelization as they are primarily constructors and overloaded functions. One possible solution to mitigate this is to use a better data structure. However this requires considerable code rewriting to remove such a hotspot and is not in the scope of this effort. The locations of these hotspots are also unusual considering they are basic data retrieval based operations.

The fifth hotspot detected is the ‘fft’ function. On performing a review of this region of code, it is determined that there exist multiple calls to this function and they all involve different data sets. For e.g. the code snippet in Figure 4.5 shows that there are four instances of the ‘fft’ function being called, with each call working on a different data set. This means that the individual function calls are independent of each other and thus provide an ideal location for parallelization.



```
for(int i = 0; i < newBTSize; i++)
}
      fBT[i] = bumpT[i].real;
}
fft(fLeft, newSize, (Log2(limit)), 0);

fft(fRight, newSize, (Log2(limit)), 0);
fft(fAvg, newSize, (Log2(limit)), 0);
fft(fBT, newBTSize, (Log2(count)), 0);
```

Figure 4.5 Code Snippets from TAP

4.2.3 Other locations for parallelization

The above code snippet in Figure 4.5 also shows a ‘for’ loop. Examining this loop, it was determined that there were no loop carried dependencies, and thus a good candidate for parallelizing. This would be an example of data decomposition.

A complete review of the code was made, and multiple sites were then identified for parallelization. To minimize overhead caused by the threading package, only sites where appreciable performance benefit might be observed were actually parallelized.

4.3 Parallelizing TAP using OpenMP

In the previous section we located several potential areas for parallelization. This section describes the results of the parallelization process. This is done using OpenMP. Some of the issues caused by this process are identified.

4.3.1 Parallelizing loops using OpenMP

TAP utilizes eight different bump templates in performing the template analysis. Each of the bump templates are cross-correlated with the profile to detect bumps and dips in profile. TAP performs the cross-correlation on the profile data set, one bump template at a time. From this it appears that there is an opportunity to execute the cross-correlation in parallel. Thus by performing this in parallel, we can potentially increase program performance.

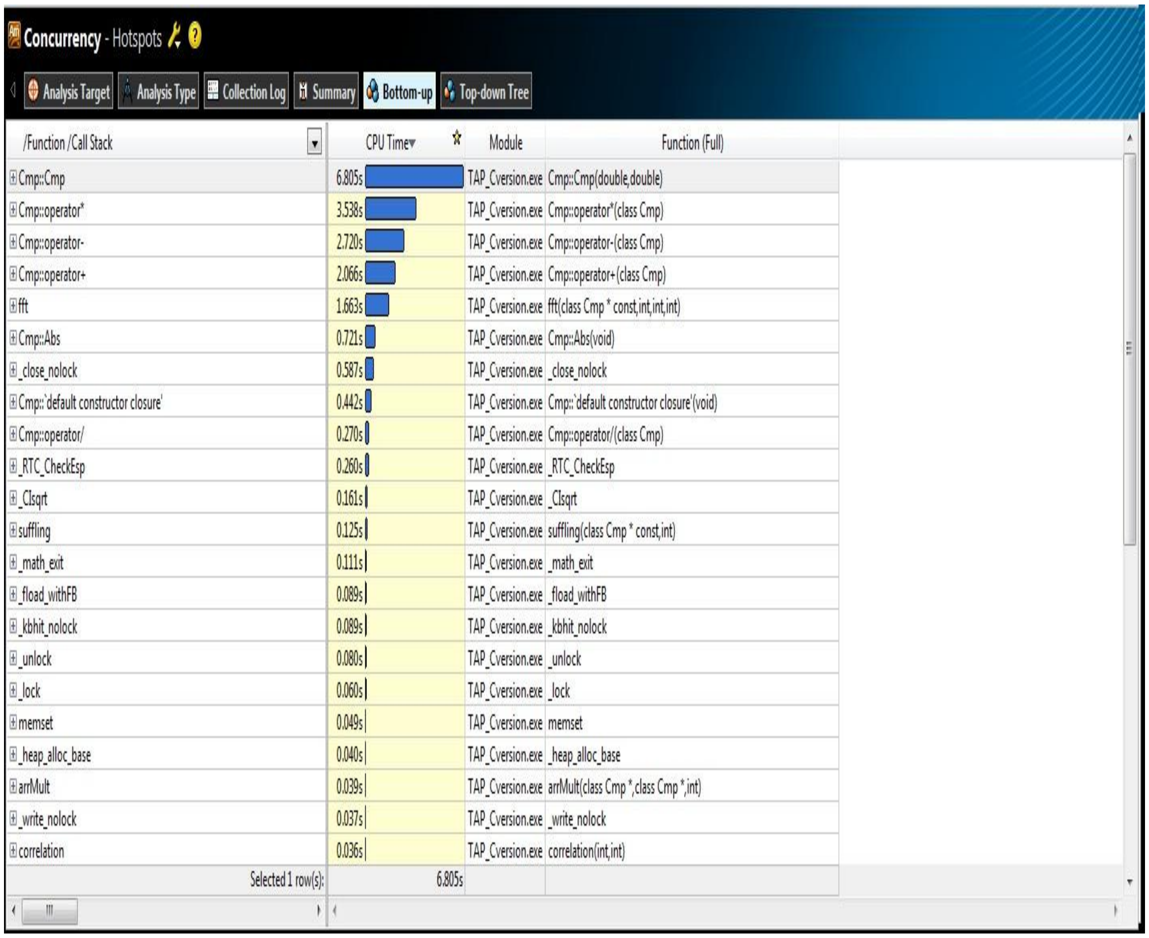
However, on closer review, there exist too many data dependencies throughout the entire loop for this to be parallelized readily using OpenMP. There are more suitable regions to parallelize further in the program. Many of the loops involve initialization or other execution where there exist no data dependencies. After ensuring that there are no dependencies in all the loops considered for parallelization, the necessary pragmas were inserted and recompiled. The program was run with different thread counts to view the performance variation. The following tables show the run time performance benefits obtained.

Average run time for the non-threaded version is 19.919s

Table 4.2 Run time measurements with different thread sizes (loop parallelized)

Run Number	**With two threads**	**With four threads**	**With eight threads**
1	20.182	20.217	19.964
2	20.055	19.955	19.873
3	19.866	20.427	20.089
4	19.951	20.014	19.942

As seen from the table there is no appreciable performance benefit through this type of parallelization. The overhead incurred in the creation of threads is greater than any performance advantage gained through this parallelism. Viewing the hotspots for each of this suggests the same. Figure 4.6 shows that there were very minor performance increases but the overhead incurred destroys all benefits.



/Function /Call Stack	CPU Time	Module	Function (Full)
⊞Cmp::Cmp	6.805s	TAP_Cversion.exe	Cmp::Cmp(double,double)
⊞Cmp::operator*	3.538s	TAP_Cversion.exe	Cmp::operator*(class Cmp)
⊞Cmp::operator-	2.720s	TAP_Cversion.exe	Cmp::operator-(class Cmp)
⊞Cmp::operator+	2.066s	TAP_Cversion.exe	Cmp::operator+(class Cmp)
⊞fft	1.663s	TAP_Cversion.exe	fft(class Cmp * const,int,int,int)
⊞Cmp::Abs	0.721s	TAP_Cversion.exe	Cmp::Abs(void)
⊞_close_nolock	0.587s	TAP_Cversion.exe	_close_nolock
⊞Cmp::`default constructor closure'	0.442s	TAP_Cversion.exe	Cmp::`default constructor closure'(void)
⊞Cmp::operator/	0.270s	TAP_Cversion.exe	Cmp::operator/(class Cmp)
⊞_RTC_CheckEsp	0.260s	TAP_Cversion.exe	_RTC_CheckEsp
⊞_CIsqrt	0.161s	TAP_Cversion.exe	_CIsqrt
⊞suffling	0.125s	TAP_Cversion.exe	suffling(class Cmp * const,int)
⊞_math_exit	0.111s	TAP_Cversion.exe	_math_exit
⊞_fload_withFB	0.089s	TAP_Cversion.exe	_fload_withFB
⊞_kbhit_nolock	0.089s	TAP_Cversion.exe	_kbhit_nolock
⊞_unlock	0.080s	TAP_Cversion.exe	_unlock
⊞_lock	0.060s	TAP_Cversion.exe	_lock
⊞memset	0.049s	TAP_Cversion.exe	memset
⊞_heap_alloc_base	0.040s	TAP_Cversion.exe	_heap_alloc_base
⊞arrMult	0.039s	TAP_Cversion.exe	arrMult(class Cmp *,class Cmp *,int)
⊞_write_nolock	0.037s	TAP_Cversion.exe	_write_nolock
⊞correlation	0.036s	TAP_Cversion.exe	correlation(int,int)
Selected 1 row(s):	6.805s		

Figure 4.6 Hotspots detected with use of parallelized for loops.

4.3.2 Data Decomposition using OpenMP

The previous section demonstrated that although minor performance benefits were obtained using parallel for loops, the magnitude of increase in performance was not justified by parallelizing TAP. To further improve parallel performance, other measures such as decomposing areas of code by data were applied.

The TAP program has to compute the FFT of the profile as well as the bump template. Fast Fourier Transform is a method used to compute the discrete Fourier transform. As this needs to be computed for three wheel paths, i.e. left, right and average wheel path, this section can be decomposed into different tasks.

The following table help describe the increase in performance observed after this parallelization.

Table 4.3 Run time measurements with different thread sizes for data based decomposition

Run Number	**With two threads**	**With four threads**	**With eight threads**
1	19.14	19.474	19.431
2	19.235	19.511	19.615
3	19.181	19.582	19.586
4	19.298	19.456	19.422

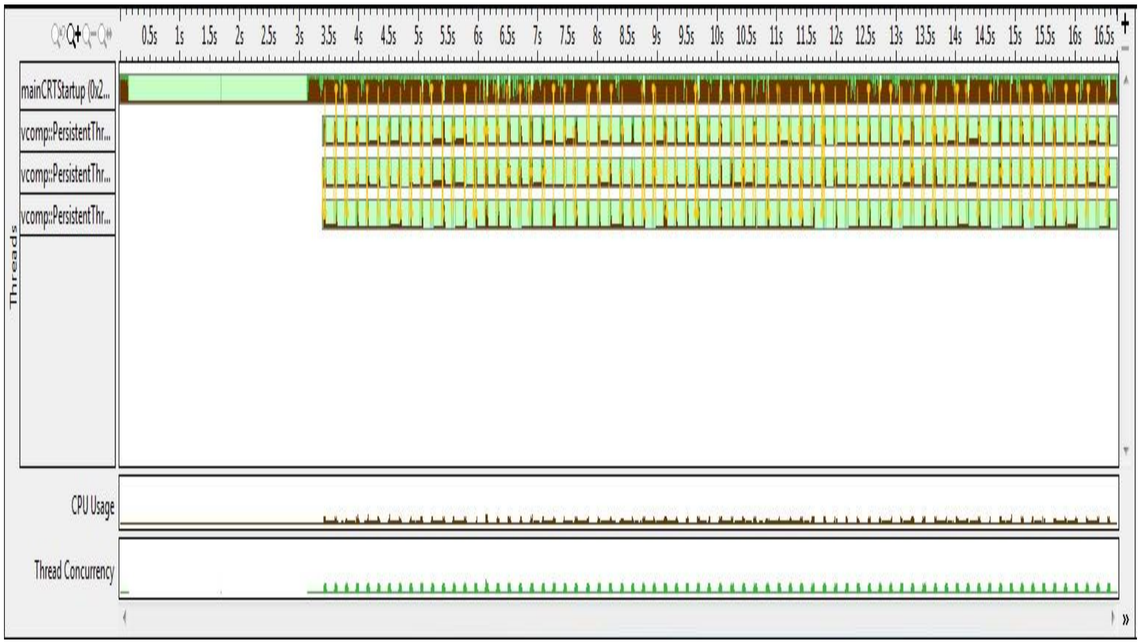
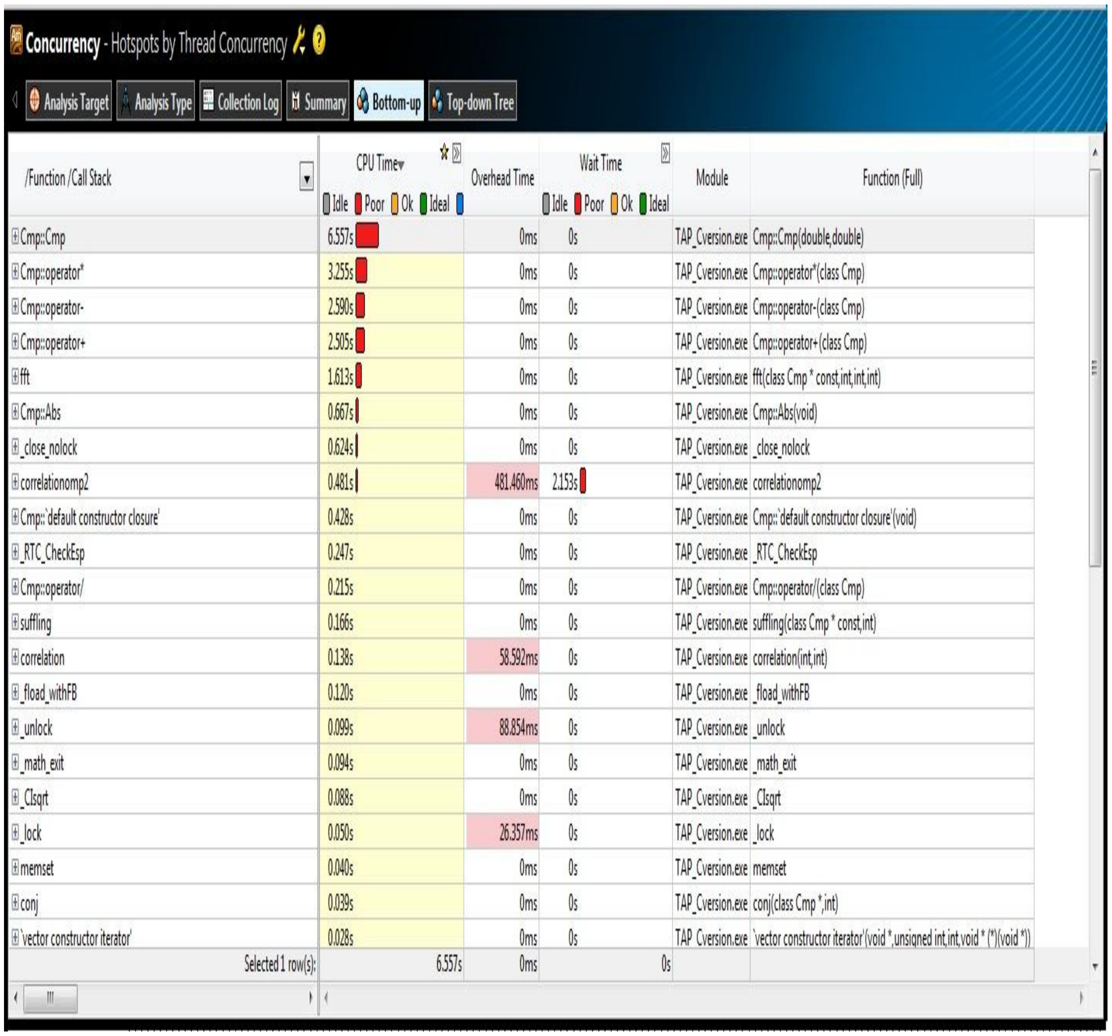


Figure 4.7 Threading Concurrency for TAP with four threads

The above figure shows the system utilizing four threads to perform all parallelized work. Because the program is only parallelized in parts, CPU usage for three of the threads increases only when a parallel section is encountered. The yellow lines between threads indicate the context switches needed for thread synchronization.



Concurrency - Hotspots by Thread Concurrency

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree

/Function /Call Stack	CPU Time (Idle, Poor, Ok, Ideal)	Overhead Time	Wait Time (Idle, Poor, Ok, Ideal)	Module	Function (Full)
Cmp::Cmp	6.557s	0ms	0s	TAP_Cversion.exe	Cmp::Cmp(double,double)
Cmp::operator*	3.255s	0ms	0s	TAP_Cversion.exe	Cmp::operator*(class Cmp)
Cmp::operator-	2.590s	0ms	0s	TAP_Cversion.exe	Cmp::operator-(class Cmp)
Cmp::operator+	2.505s	0ms	0s	TAP_Cversion.exe	Cmp::operator+(class Cmp)
fft	1.613s	0ms	0s	TAP_Cversion.exe	fft(class Cmp * const,int,int,int)
Cmp::Abs	0.667s	0ms	0s	TAP_Cversion.exe	Cmp::Abs(void)
_close_nolock	0.624s	0ms	0s	TAP_Cversion.exe	_close_nolock
correlationomp2	0.481s	481.460ms	2.153s	TAP_Cversion.exe	correlationomp2
Cmp::`default constructor closure'	0.428s	0ms	0s	TAP_Cversion.exe	Cmp::`default constructor closure'(void)
_RTC_CheckEsp	0.247s	0ms	0s	TAP_Cversion.exe	_RTC_CheckEsp
Cmp::operator/	0.215s	0ms	0s	TAP_Cversion.exe	Cmp::operator/(class Cmp)
suffling	0.166s	0ms	0s	TAP_Cversion.exe	suffling(class Cmp * const,int)
correlation	0.138s	58.592ms	0s	TAP_Cversion.exe	correlation(int,int)
_fload_withFB	0.120s	0ms	0s	TAP_Cversion.exe	_fload_withFB
_unlock	0.099s	88.854ms	0s	TAP_Cversion.exe	_unlock
_math_exit	0.094s	0ms	0s	TAP_Cversion.exe	_math_exit
_CIsqrt	0.088s	0ms	0s	TAP_Cversion.exe	_CIsqrt
_lock	0.050s	26.357ms	0s	TAP_Cversion.exe	_lock
memset	0.040s	0ms	0s	TAP_Cversion.exe	memset
conj	0.039s	0ms	0s	TAP_Cversion.exe	conj(class Cmp *,int)
`vector constructor iterator'	0.028s	0ms	0s	TAP_Cversion.exe	`vector constructor iterator'(void *,unsigned int,int,void * (*)(void *))
Selected 1 row(s):	6.557s	0ms	0s		

Figure 4.8 Improved performances of Hotspots due to data based threading

Figure 4.9 shows the hotspots detected post parallelization. A much improved performance of the different function calls shows that threading has improved performance across the board. The next figure shows the comparison between the performances of the threaded application, i.e. Only loop parallelization vs. data decomposition.

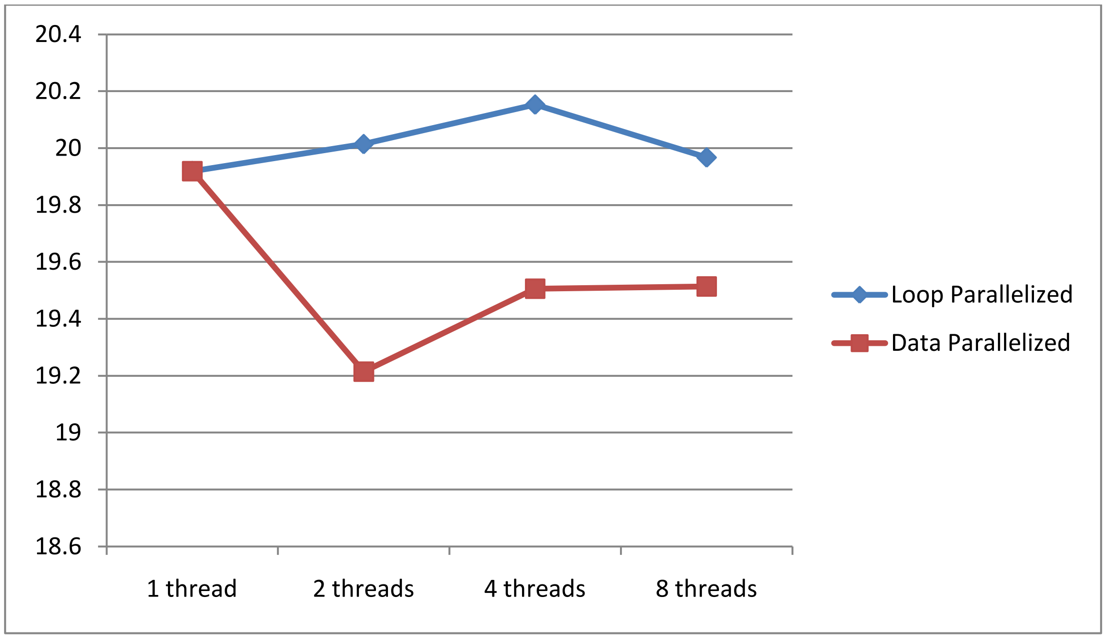


Figure 4.9 Average runtimes of TAP with different number of threads.

The above figure shows that for the loop parallelized version of TAP actually has a performance decrease. This can be attributed to the fact that the overhead costs more than the benefit of threading.

The figure also shows that the data decomposed version of TAP has a better runtime for the version running two threads. The performance of the application overall is much better than the single threaded version.

Figures 4.10 and 4.11 show us the thread concurrency and CPU core usage histograms for the parallelized TAP.

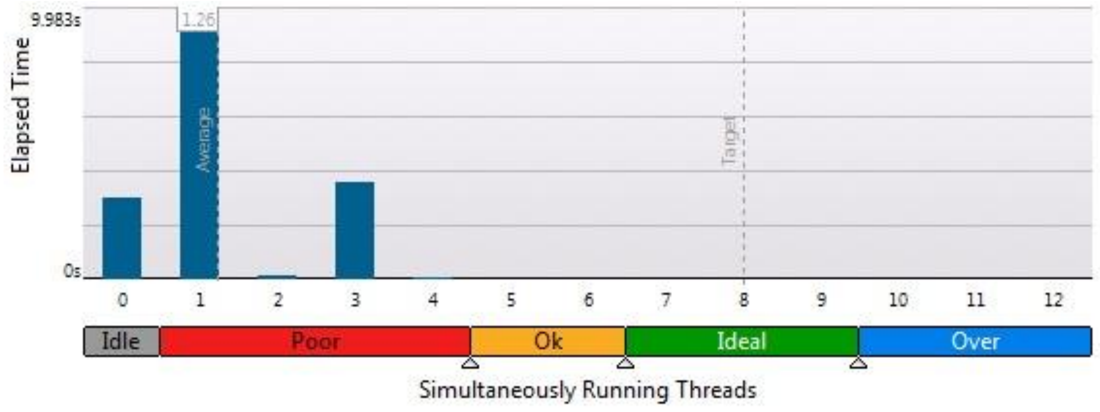


Figure 4.10 Thread Concurrency Histogram (4 threads)

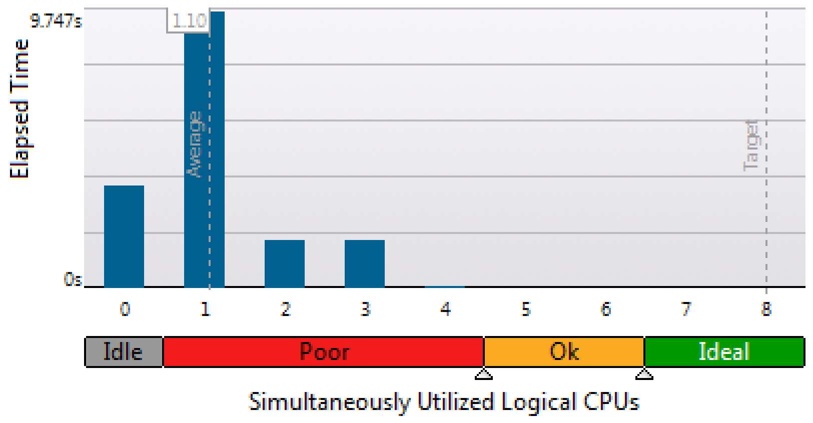


Figure 4.11 CPU Core Usages for TAP (4 Threads)

The above two figures indicate that the ideal thread usage or core usage is not being met. A large part of this can be attributed to Amdahl's law. Amdahl's law states that the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program [ref wiki]. Therefore even though the performance of TAP has increased, it is limited by the parts which have to occur in a sequential manner. This results in the majority of execution time occurring in a single thread or core. Further performance benefits are limited when using OpenMP to parallelize TAP. Better performance in this regard can be achieved by rewriting TAP as a wholly parallel application.

4.3.3 Compiler benefits

TAP was developed using the Microsoft Visual Studio IDE. This uses the Microsoft compiler to compile and build the executable. As TAP is mainly run on Intel platforms, the application was compiled using the Intel Composer XE.

From Table 4.2, we see that the actual CPU times when compared to table 4.1 are quite different. There is a marked improvement in performance when compiled with the Intel Composer. Table 4.3 shows the percentage improvement obtained by this method.

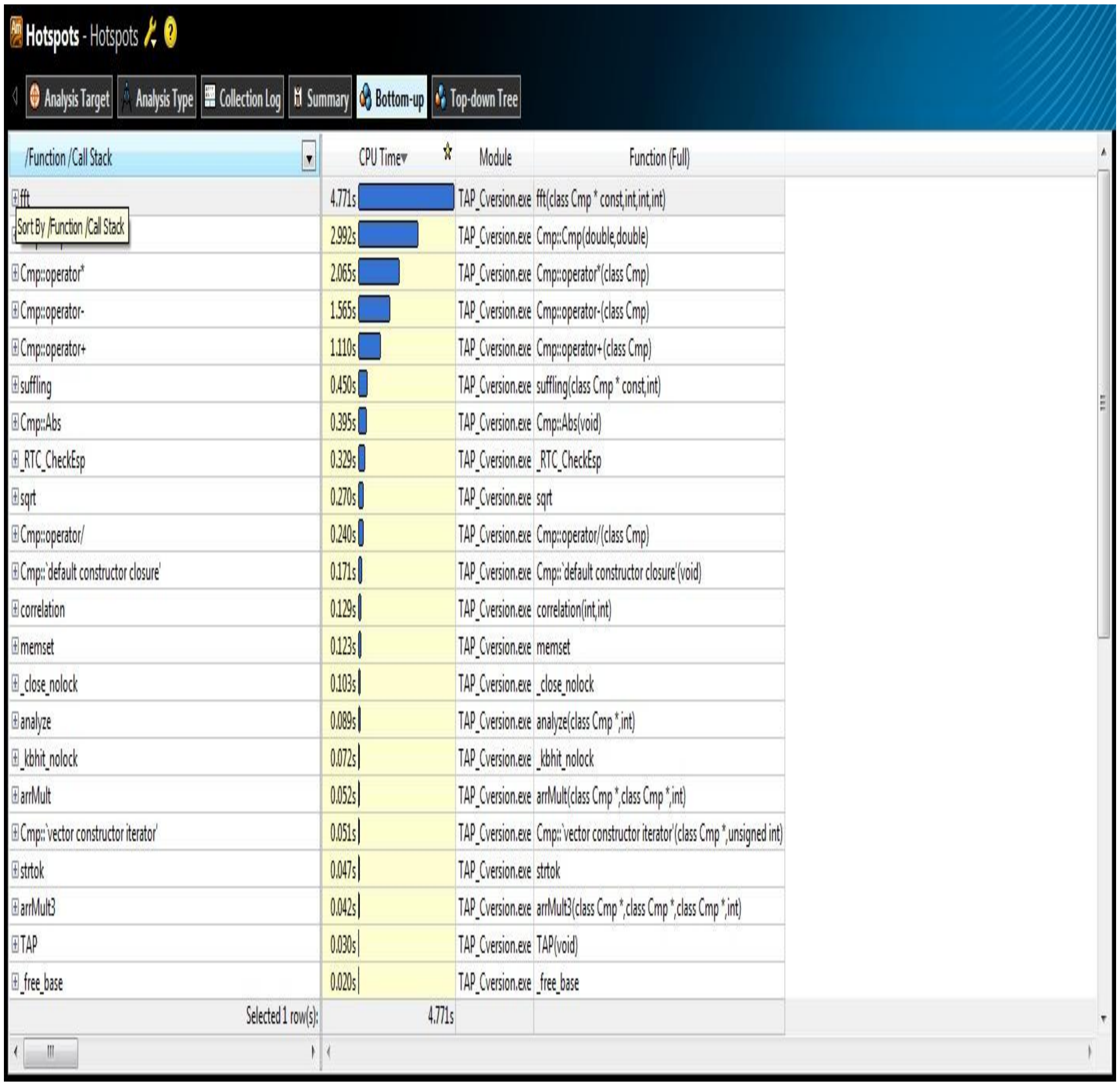
Table 4.4 Run times and actual CPU times of TAP built using Intel Composer XE

Run Number	**Total Time (s)**	**Actual CPU time (s)**
1	20.994	15.581
2	20.579	15.621
3	21.168	15.962
4	20.883	15.438

Table 4.5 Performance improvement of TAP using Intel Composer XE

Run Number	**CPU time using Microsoft compiler**	**CPU time using Intel compiler**	**Percentage difference (%)**
1	19.14	15.581	18.59
2	19.235	15.621	18.78
3	19.181	15.962	16.78
4	19.298	15.438	20.007

Analyzing this hotspot view, we see that the 'fft' function is the hottest region of code. This makes more sense as most of the program executes in this region. The other hotspots in the program can be attributed to the constructor and overloaded functions. Thus a simple change in compiler can result in much faster execution, due to compiler optimizations.



/Function /Call Stack	CPU Time	Module	Function (Full)
fft	4.771s	TAP_Cversion.exe	fft(class Cmp * const,int,int,int)
[illegible]	2.992s	TAP_Cversion.exe	Cmp::Cmp(double,double)
Cmp::operator*	2.065s	TAP_Cversion.exe	Cmp::operator*(class Cmp)
Cmp::operator-	1.565s	TAP_Cversion.exe	Cmp::operator-(class Cmp)
Cmp::operator+	1.110s	TAP_Cversion.exe	Cmp::operator+(class Cmp)
suffling	0.450s	TAP_Cversion.exe	suffling(class Cmp * const,int)
Cmp::Abs	0.395s	TAP_Cversion.exe	Cmp::Abs(void)
_RTC_CheckEsp	0.329s	TAP_Cversion.exe	_RTC_CheckEsp
sqrt	0.270s	TAP_Cversion.exe	sqrt
Cmp::operator/	0.240s	TAP_Cversion.exe	Cmp::operator/(class Cmp)
Cmp::`default constructor closure'	0.171s	TAP_Cversion.exe	Cmp::`default constructor closure'(void)
correlation	0.129s	TAP_Cversion.exe	correlation(int,int)
memset	0.123s	TAP_Cversion.exe	memset
_close_nolock	0.103s	TAP_Cversion.exe	_close_nolock
analyze	0.089s	TAP_Cversion.exe	analyze(class Cmp *,int)
_kbhit_nolock	0.072s	TAP_Cversion.exe	_kbhit_nolock
arrMult	0.052s	TAP_Cversion.exe	arrMult(class Cmp *,class Cmp *,int)
Cmp::`vector constructor iterator'	0.051s	TAP_Cversion.exe	Cmp::`vector constructor iterator'(class Cmp *,unsigned int)
strtok	0.047s	TAP_Cversion.exe	strtok
arrMult3	0.042s	TAP_Cversion.exe	arrMult3(class Cmp *,class Cmp *,class Cmp *,int)
TAP	0.030s	TAP_Cversion.exe	TAP(void)
_free_base	0.020s	TAP_Cversion.exe	_free_base
Selected 1 row(s):	4.771s		

Figure 4.12 Hotspot Detection in TAP built using Intel compiler

4.4 Program Correctness

After a program has been parallelized, it is very important to check that the program still performs its tasks correctly. A parallelized program could compile and even execute without any runtime errors, but only after comparison of outputs between the serial and parallel version can we ascertain if a program is correct. To check for this TAP_parallel was run on multiple data sets and the outputs were compared to the outputs from TAP_serial.

The data sets used for comparison are primarily profile runs done on US290 roads. It was determined after comparison that parallelizing did not affect program correctness. Also on examination of the output files, the locations of the bumps were also determined to be the same. Table 4.6 - 4.8 show that the number of bumps detected between the serial and parallel version remain the same.

Table 4.6 Number of rough sections detected by two versions of TAP (Left Wheel Path)

Road Section	**Serial Version**	**Parallel Version**
US290 L2A	13	13
US290 R1A	20	20
US290 R2A	6	6

Table 4.7 Number of rough sections detected by two versions of TAP (Right Wheel Path)

Road Section	**Serial Version**	**Parallel Version**
US290 L2A	15	15
US290 R1A	15	15
US290 R2A	10	10

Table 4.8 Number of rough sections detected by two versions of TAP (Average Wheel Path)

Road Section	Serial Version	Parallel Version
US290 L2A	19	19
US290 R1A	25	25
US290 R2A	10	10

The NSI level was fixed at 3.75 for the data above. The number of bumps listed for each road is the number found along the left, right and average wheel paths in the 3rd mile section of the road. The bump template used was the 7th template.

After comparisons, it can be said that performance of TAP has indeed been improved by the use of OpenMP.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusion

This research effort addresses some of the complexities in adapting an existing program for use in a real-time multi-core system. With the advent of multi-core systems, there exists an opportunity for increased performance. However, newer programming paradigms are necessary to fully utilize the additional computing power. Parallel computing has become the dominant paradigm in computer architecture, in the form of multicore processors. Also with the move towards smaller CPU die area to achieve better power performance; sufficient computing horsepower is available for an embedded system. This means that performance no longer needs to be sacrificed in lieu of power. Therefore, multicore embedded processors require parallel algorithms to extract maximum performance.

This research investigates different programming models and APIs to convert an existing sequential program into one which is able to utilize the parallel processing power of an embedded multicore processor. The program considered for this effort is the Template Analysis Procedure or TAP. TAP is a program used to analyze road profile data collected with inertial profilers and identify any unacceptable roughness that may be present. The method involves cross-correlating different bump templates with the profile data. Whenever the correlation values exceed a set threshold, that location is identified as a defect.

The method of using cross-correlation in TAP means that we need to first compute the FFT of the profile and bump template data. FFT is an algorithm which can be highly parallelized and can make efficient use of multi-core computing resources. TAP also performs this correlation on the profile along with 8 different bump templates. This means that there exists another region where parallel processing could be utilized. However the inherent parallelism present in FFT and the TAP program structure cannot be extracted without a considerable effort towards modification and/or developing new code. This research addresses this concern by the use of APIs designed for use in shared memory systems. OpenMP is one such API which allows for a developer to introduce parallelism in existing program. A complete analysis of OpenMP and the performance benefits to TAP through the use of it were made.

To improve and tune the performance of an application, it is necessary to record and characterize its efficiency. Introducing parallelism in small steps and observing the performance of an application allows a developer to test and identify any issues which arise. Intel's Parallel Studio assists a developer in profiling, optimizing a program. Through proper analysis, multi-threading programming issues such as race conditions, deadlocks, concurrency errors etc. can be detected in the development cycle, allowing for a more robust program.

5.2 Future work

From the investigations made, TAP has multiple sites which are good for parallel processing. This research focused its investigations on OpenMP as the API to achieve the goal. Further efforts can be made towards investigating other APIs such as Intel TBB and native Win32 API. This could potentially require a complete overhaul of the TAP, thus allowing for true parallelism to be achieved. Another major improvement that can be made is to convert TAP from a post processing tool to a real time add-on to the embedded road profiling system.

REFERENCES

[1] R.S. Walker, E.G. Fernando. *Pilot Implementation of Bump Detection Profiler*. Research Report. 5-4385-01-1. University of Texas, Transportation Instrumentation Laboratory, Arlington, TX. December 2007

[2] R. Walker, E. Fernando, and Y. Sho, "Develop a Methodology For Establishing Bump Detection Using Inertial Profile Measurements For Implementation With The Ride Specification", Texas Department of Transportation (TxDOT) Research Report 4479, 2004 .

[3] M. W. Sayers and S. M. Karamihas, the Little Book of Profiling, University of Michigan Transportation Research Institute, September 1998.

[4] R.S. Walker, E.G. Fernando, "A Portable Profiler for Pavement Profile Measurements - Interim Report," Technical Report. 0-6004-1. Texas Transportation Institute, College Station, TX. May 2009.

[5] http://en.wikipedia.org/wiki/OpenMP

[6] http://faq.programmerworld.net/programming/win32-multithreading-and-synchronization.html

[7] http://en.wikipedia.org/wiki/OpenMP#The_core_elements

[8] http://drdobbs.com/high-performance-computing/225702895

[9] Multi-Core Programming Increasing Performance through Software Multi-threading, Shameem Akhter, Jason Roberts, Intel Press, 2006

[10] http://openmp.org/wp/

[11] http://openmp.org/mp-documents/ntu-vanderpas.pdf

[12] http://software.intel.com/en-us/articles/choosing-between-openmp-and-explicit-threading-methods/

[13] http://www.compunity.org/training/tutorials/3%20Overview_OpenMP.pdf

[14] Asanovic, Krste et al. (December 18, 2006). "The Landscape of Parallel Computing Research: A View from Berkeley" (PDF). University of California, Berkeley. Technical Report No. UCB/EECS-2006-183

[15] http://software.intel.com/en-us/articles/intel-parallel-studio-xe/

[16] http://software.intel.com/en-us/articles/intel-composer-xe/

[17] R. Walker, and E. Fernando, EVALUATION OF RIDE EQUATION USING CURRENT PROFILER SYSTEMS AND NEW SENSOR TECHNOLOGY, January 2002

BIOGRAPHICAL INFORMATION

Ashwin Arikere was born in Bangalore, India in 1985. He completed his Bachelor’s in Instrumentation Technology from Visvesvaraya Technological University, India in 2007. He then joined The University of Texas at Arlington, Texas to pursue his Masters in Computer Science Engineering. Ashwin was working as a graduate teaching assistant for the Embedded Systems and Real Time Embedded Systems courses. His research interests lies in various aspects of parallel processing, multi-core systems and system architecture. He plans to pursue a doctorate degree in embedded systems.