VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Course Code : **CSE2005**

Course Name : **Operating Systems Lab**

Degree : **B.Tech.**

Date : **29.05.2021**

Faculty ID : **51656**

Faculty Name : **Dr.A.Menaka Pushpa**

Name :**Pratyush Motha**

Reg No :**19BLC1046**

DIGITAL ASSIGNMENT 2

MULTICORE OPTIMIZATION

Multi-core processors continue this trend and are capable of sharing work and executing tasks on independent execution cores concurrently.In many cases, taking full advantage of the performance benefits of these processors will require developers to thread their applications.

Multi-core processors are comprised of multiple processor cores in the same package and offer increased performance, but at a cost to the embedded developer. In many cases, taking advantage of the performance benefits requires developers to thread their applications. Effectively threading an application is a nontrivial task that requires domain knowledge in multi-core architecture, parallelism fundamentals, and a threading methodology.

This article is focused on three main topics:

(1) an overview of multi-coreprocessor architecture and the benefit that multi-core processors offer;
(2) a primer on parallelism and discusses key topics such as scalability, parallelism& threading techniques, and,

(3) a threading methodology which can be employed to effectively thread, optimize, and debug an application.

Multi-core Processors
There are several definitions of multi-core processor. In this paper,multi-core processor implies two or more homogenous processor cores inthe same physical package. An example of such a processor is the **Intel Core Duo processo**r which is comprised of two similar processor cores in the same die. Other more general definitions of multi-core may include heterogeneous processor cores and multiple processors in a system; however this paper limits discussion to the previous.

Multi-core Architecture
The move to multi-core processor architectures in the embedded processor industry has been motivated by several trends occurring over the past few years. These trends are summarized as follows:

1) More transistors available per die ” As transistor process technology shrinks, more transistors are available for use in a processor design. It is now feasible to place multiple processor cores in the same package.

2) Clock scaling reaching limits due to power constraints ” An empirical study suggests that a 1% clock increase results in a 3% power increase. Space, power,and fan noise of cooling is a key customer constraint in many market segments.

Figure 1 below provides insight into why multi-core is particularly attractive today. The power and performance of three different configurations running a set of applications are compared. These configurations from left-to-right are:

1. Standard processor over-clocked 20%
2. Standard processor
3. Two standard processors each under-clocked 20%

The observation for ***configuration1*** compared to ***configuration 2*** is that performance is increased 1.13x with a power increase of 1.73x.This compares **where performance is increased 1.73x with a power increase of1.02x. On the right class of applications, multi-core processors offer higher performance with smaller increase in power compared to straight frequency scaling. Two caveats from this observation is that not all applications will be able to take equal advantage of multi-core processors and that more work is required from the developer to realize these performance gains.**

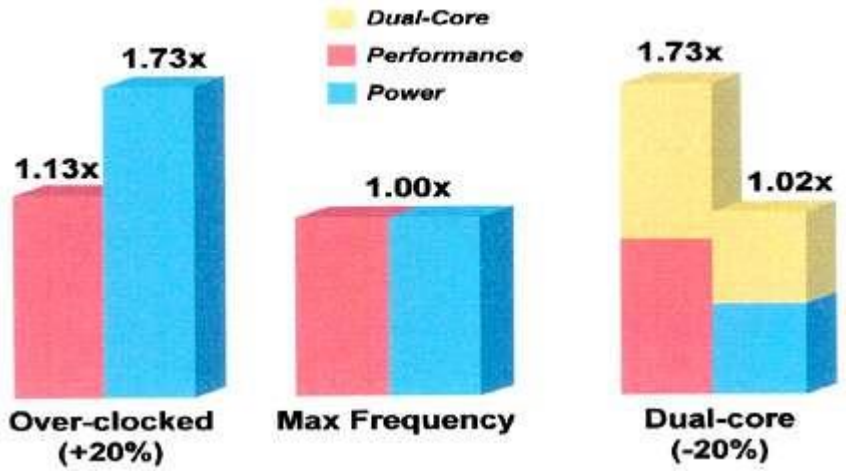

Figure1. Multi-core Performance & Power Scenario<

Figure 2, below shows two typical multi-core processor architectures. A somewhat simple manner of envisioning a multi-core processor is as two processors connected together inside the chip packaging as opposed to a multiprocessor where multiple processors may be in a system, but connected via the system bus. The two multi-core architectures in **Figure 2** differ in the location of connection between the two processor cores.

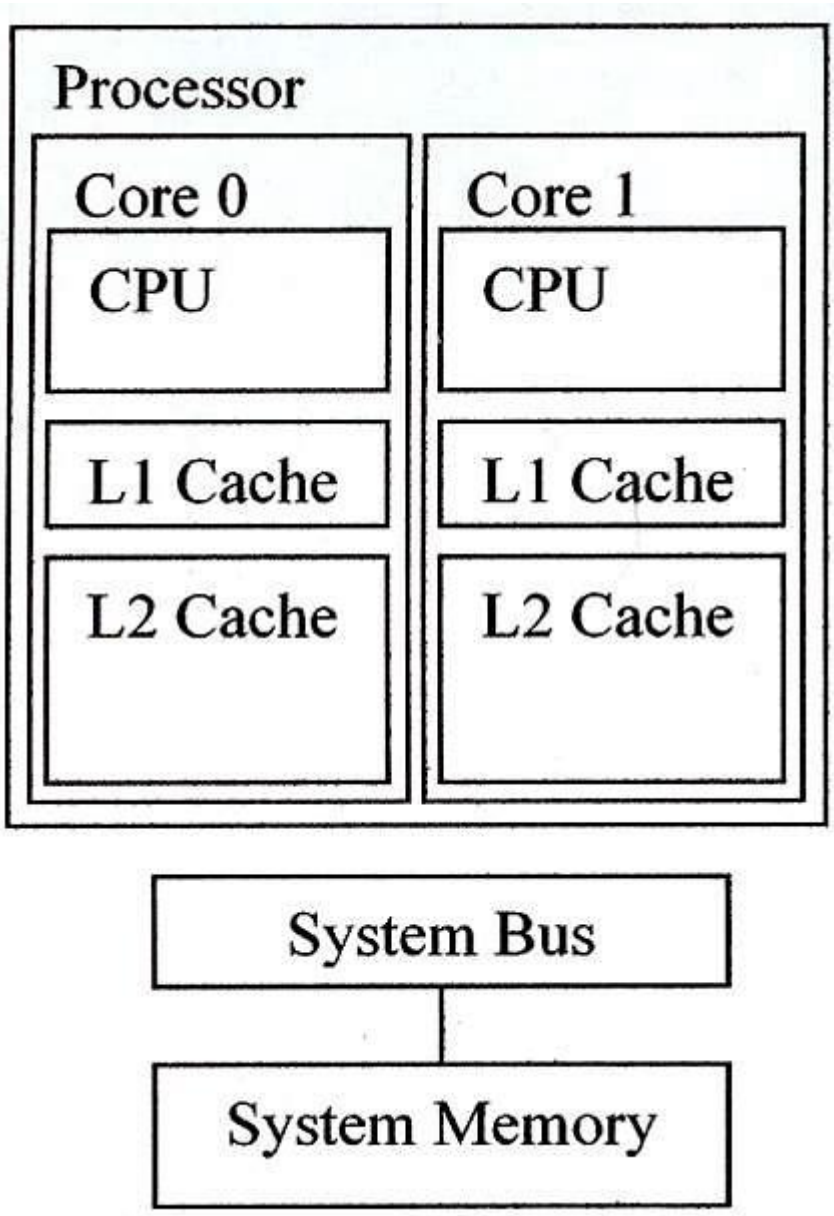

Figure2a. Multi-core processor separate L2

Figure 2a above shows two independent processor cores, each with a level 2 cache, and sharing system bus and connection to memory. **Figure 2b** below displays two independent processor cores sharing the level 2cache, system bus, and connection to memory.

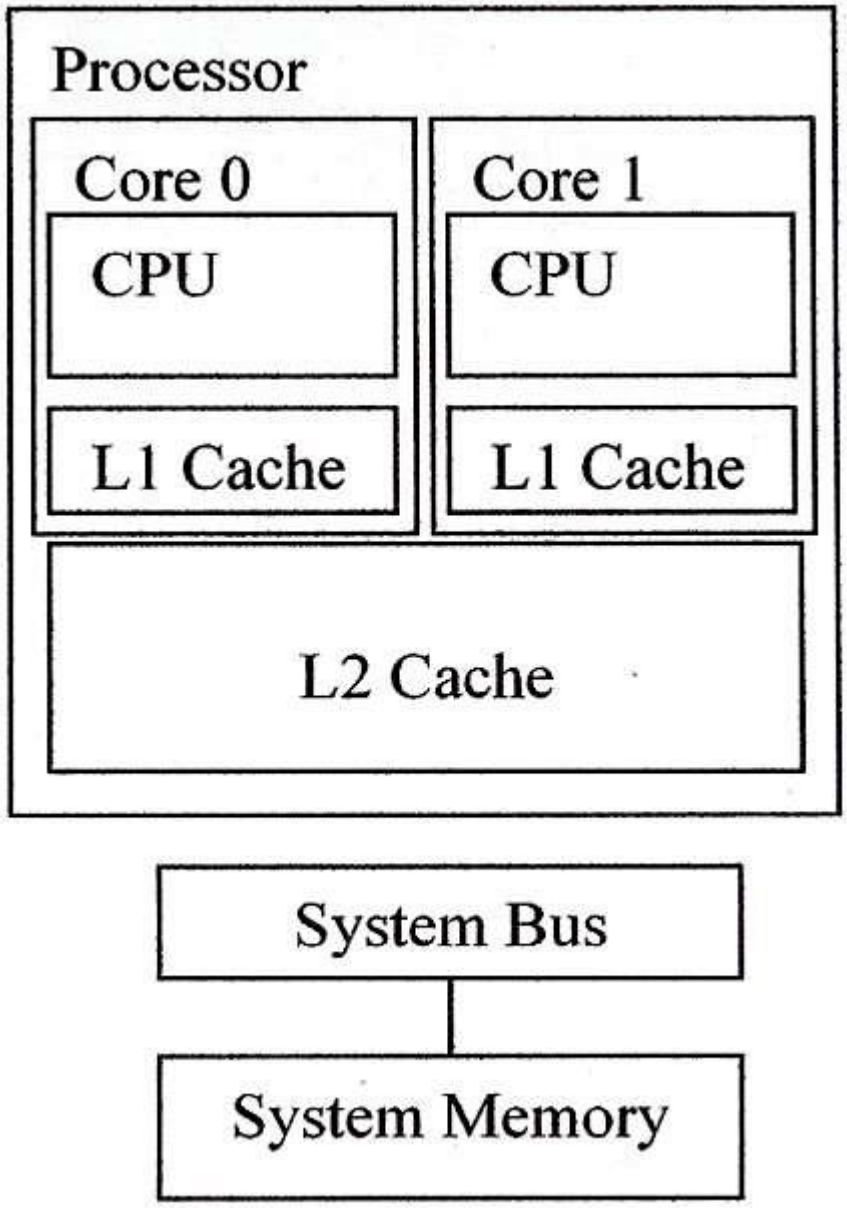

Figure2b. Multi-core processor shared L2

There are advantages and disadvantages of the two layouts; however the current trend is toward shared level 2 cache which offers the following advantages:

1) Size allocated per core can be dynamically adjusted with the potential of the total level 2 cache being available to applications that require it.
2) Sharing of items between cores. Threads on separate cores can synchronize through the faster level 2 cache as opposed to main memory or the next level of cache.

Benefits of Multi-core Processors
Multi-core processors offer developers the ability to apply more compute resources at a particular problem. These additional resources can be employed to offer two types of advantages, improved turnaround time or solving larger problem domains. An improved turnaround time example is the processing of a transaction in a **Point-Of-Sale** system.

The time it takes to process a transaction may be improved by taking advantage of multicore processors. While one processor core is updating the terminal display, another processor core could be tasked with processing the user input. A larger problem domain example would be the servers that handle backend processing of the transactions arriving from the various Point-Of-Sale terminals. By taking advantage of multi-core processor, any one server could handle a greater number of transactions in the desired response time.

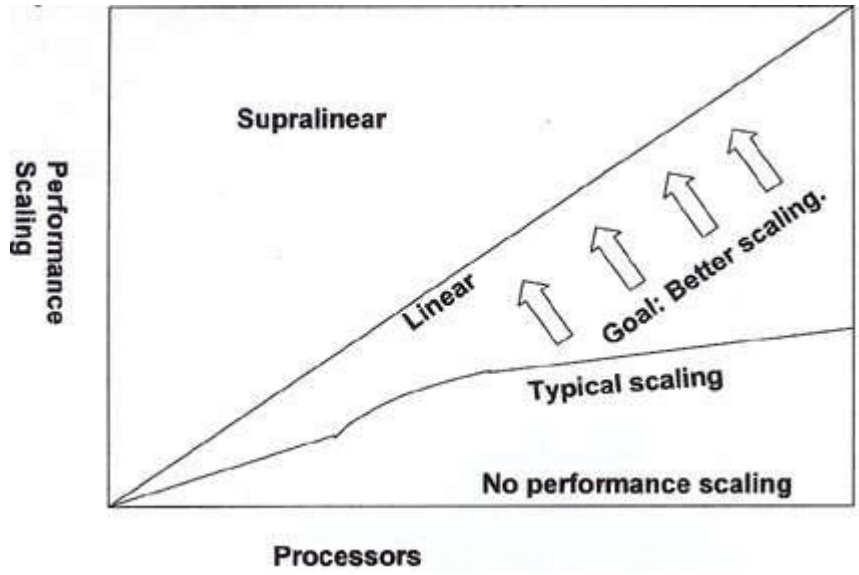

Figure3. Multi-core Performance Scaling Goal

Figure 3, above shows possible performance scaling improvements derived from multi-core processors. The area labeled *"Supralinear"* indicates a performance improvement that scales greater than the number of processors. Supralinear speedup is rare, but possible in cases where the application benefits from the collective increased cache size and layout.

The first line shows the theoretical case where the performance scales with the number of processors. The *"Typical scaling"* curve shows what a sample application may return performance-wise as the number of processors used to run the application increases. This curve displays where there is diminishing performance as the number of processors employed is increased. Less than linear scaling is indicative of either insufficient parallelism or communication overhead of the parallelism. The goal of your effort to employ parallelism is to increase the amount of scaling possible in your application.

Parallelism Primer

Understanding the process to effectively thread an application requires a general understanding of key parallelism concepts. This section provides an overview on parallelism including such topics as estimating the benefits of parallelism, different methods of employing parallelism, and the advantages and disadvantages of various threading techniques.

The implementation of parallelism in a system can take many forms ;one commonly used type is shared memory parallelism which implies the following:

1) Multiple threads execute concurrently.
2) The threads share the same address space. This is compared to multiple processes which can execute in parallel but each with a different address space.

3) Threads coordinate their work.
4) Threads are scheduled by the underlying operating system and require OS support.

To illustrate the keys to effective parallelism, consider an example in the physical world of multiple workers mowing a lawn. The first consideration is how to divide the work evenly. This even division of labour has the effect of keeping each worker as active as possible. Second, the workers should each have their own lawn mower; not doing so would significantly reduce the effectiveness of the multiple workers. Finally, access to items such as the fuel can and clipping container needs to be coordinated. The keys to parallelism illustrated through this example are generalized as follows:

1) Identify the concurrent work.
2) Divide the work evenly.
3) Create private copies of commonly used resources.
4) Synchronize access to unique shared resources.

Scalability
Scalability is critical to any parallelism effort because the number of processor cores available in a single package is projected to increase substantially over the next 10 years from two cores to perhaps 8 &16 cores.

Scalability with regards to parallelism is a measure of how the performance of an application increases as the number of processor cores in the system increases.

Amdahl's law provides the key insight into determining the scalability of an application. In brief, Amdahl's law states that the performance benefit of parallelism is limited by the amount of the application that must run serially,e.g. is not or cannot be parallelized. For example, take an image processing application where 300 images are to be processed with the following characteristics:

1) 10minutes of initialization that must run serially
2) 1 minute to process one image(processing of different images can be accomplished in parallel)
3) 10 minutes of post-processing that must run serially

Table 1 below shows the calculations of scalability and efficiency for a number of different processing cores. Efficiency is a measure of how effectively the total number of processors is being employed in running the application inparallel; the goal is a 100% measure of efficiency. In other words, to increase the benefits of parallelism and take advantage of the available processor cores as much of the application should be parallelized as possible.

Processor cores	Initialization	Image	Post-processing	Total Time	Scalability	Efficiency
1	10	300	10	320	1.00	100%
2	10	150	10	170	1.88	94.1%
4	10	75	10	95	3.37	84.2%
8	10	37.5	10	57.5	5.57	69.6%
16	10	18.75	10	38.75	8.26	51.6%
32	10	9.375	10	29.375	10.89	34.0%
300	10	1	10	21	15.24	5.1%

Table1. Image Processing Scalability

Parallelism Techniques
The benefits of future multi-core architectures scale with the effort employed by the embedded system developer. The different levels of effort are summarized as follows:

1. No effort
2 . Take advantage of multiple processes executing on a symmetric multiprocessor operating system kernel
3. Take advantage ofmulti threading on a symmetric multiprocessor operating system kernel

The first level specifies no explicit effort to take advantage of future multi-core processors. Some benefit can be reaped in this case from the ever increasing single thread processor performance being offered in successive generations of processor cores.

Second, parallelism can be derived in many cases just by employing asymmetric multiprocessor and multi-core aware operating system such as **Linux**. System processes have the ability to run concurrently with your application process in this case. Scalability may be limited as the number of cores made available increases; there are only a limited number of tasks an operating system needs to accomplish and can accomplish in parallel.

The third level is true multithreading on a symmetric multiprocessor& multi-core aware operating system. This level can take advantage of the parallelism made available at the process level (level 2) as well as parallelism that is expressed in the multiple threads in an application. In order to derive this benefit, it will be required to thread the application.

Decomposition
One consideration in parallelism is how to decompose the work in an application to enable dividing the task between the different processor cores. There are two categories of decomposition summarized as follows:

***** Functional decomposition – division based upon the type of work
***** Data decomposition- division based upon the data needing processing

Functional decomposition attempts to find parallelism between independent steps in your application. For example, consider an intrusion detection system that performs the following checks on a packet stream:

1. Check for scanning attacks.
2. Check for Denial of Service attacks**.**
3. Check for penetration attacks.

Parallelism Limiters
All problems do not lend themselves to parallelization and understanding some common parallelism limiters is essential. Also, some problems may be parallelized only after special handling of these potential limiters. The key limiters to parallelism are summarized as:

1. Data dependency – data values that have dependencies between them.
2. State in routines -routines with values live across invocations

The **code shown below** (*removing the **OpenMP** pragmas*) has a data dependency between iterations on the sum variable and would typically limit running the code in parallel; however there are methods such as using the OpenMP reduction clause that allow parallelism.

```
#include <stdio.h>
#include <omp.h>
static int num_steps = 100000;
double step;
#define NUM_THREADS 2
int main ()
{       int i;
        double x, pi, sum = 0.0;
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
        for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
        printf("%lf\n", pi);
}
```

An example of state in routines is a routine that maintains data between invocations of the routine using statically defined variables .Examples of routine with state include typical memory allocation routines, random number generators, and I/O routines.

There are two general techniques for creating routines and loops that can be parallelized. Routines that are made re entrant are capable of being run in parallel. These routines would not have dependencies between other invocations of the same routines. A method of testing if a particular loop can be parallelized is to run the loop backwards interms of its loop indexes.

Threading Techniques

There are several techniques in use today to thread applications. Two broad categories are library-based and compiler-based threading. Examples of library-based threading include **Windows Thread API** and **POSIX** threads. Two examples of compiler-based threading are OpenMP and auto-parallelization. The two examples of library-based threading technologies are explicit in that the developer is responsible for explicitly creating, controlling, and destroying the threads via function calls.

Table 2 below shows a comparison between explicit threading technologies and compiler-based technologies. Explicit threading tends to be more flexible than compiler-based threading in adapting to the particular needs required to parallelize an application. Explicit threading can parallelize all of the same applications that compiler-based threading can handle and more. Explicit threading tends to be more difficult to use than compiler-based threading.

In compiler-based threading, the compiler handles threading of the application with input from the user coming from command line arguments and/or language directives. Explicit threading can perform both functional and data decomposition whereas compiler-based threading is more suited for data decomposition. Finally, explicit threading models such as POSIX have been ported to several operating systems. The availability of compiler-based threading is dependent on compiler support and the underlying threading support.

Category	Explicit threading	Compiler-based threading
Flexibility	High	Moderate
Ease-of-use	Moderate	High
Decomposition	Functional and Data	Data, some functional
Portability	Moderate (POSIX)	Moderate

Table2. Comparison of explicit threading and compiler-based threading

OpenMP

OpenMP is an open, portable, shared memory multiprocessing application program interface supported by multiple vendors on several operating systems and under the following programming languages: **Fortran77,Fortran 90**, C, and C++. OpenMP simplifies parallel application development by hiding many of the details of thread management and thread communication behind a simplified programming interface.

Developers specify parallel regions of code by adding pragmas to the source code. In addition, these pragmas communicate other information such as properties of variables and simple synchronization. The sample code in the appendix is an OpenMP program that calculates the value of pi by summing the area under a curve. The program is very similar to the original serial version of the code except for the addition of a few lines of code. The key line is the following pragma,

#pragmaomp parallel for reduction (+:sum) private (x)

which specifies the for loop should be executed by a team of threads, temporary partial results represented by the sum variable should be aggregated or reduced at the end of the parallel region by addition, and finally the variable x is private, meaning each threadgets its own private copy. The keys in parallelizing the code aresummed up as follows:

* **Identify the concurrent work.** The concurrent work is the area calculation encompassing different parts of the curve
* **Divide the work evenly.** The number of rectangle areas to compute is 100000 and is equally allocated between the threads.
* **Create private copies** of commonly used resources. The variable x needs to be private, as each thread's copy will be different.
* **Synchronize access** to unique shared resources. The only shared resource, ***step*** does not require synchronization in this example because it is only read by the threads, not written.

Automatic Parallelization

Automatic Parallelization, which is also called auto-parallelization, analyzes loops and creates threaded code for the loops determined to be beneficial to parallelize. Automatic parallelization is a good first technique to try in parallelizing code, as the effort to do so is fairly low. The compiler will only parallelize loops that can be determined to be safe to parallelize. The following tips may improve the likelihood of successful parallelization:

1) Expose the trip count of loops whenever possible. The compiler has a greater chance of parallelizing loops whose trip counts are statically determinable.

2) Avoid placing function calls inside loop bodies. Function calls may have effects on the loop that cannot be determined at compile time and may prevent parallelization.

3) If available, use optimization reporting options. This option provides a summary of the compiler's analysis of every loop and in cases where a loop cannot be parallelized, a reason as to why not. This is useful in that even if the compiler cannot parallelize the loop, the developer can use the information gained in the report to identify regions for manual threading.

4) If available, adjust the command line option threshold needed for autoparallelization. The compiler estimates how much computation is occurring not occur. This can be overridden by the threshold option. Reduce use of global and static variables. These types of variables make it more difficult for the compiler to identify and parallelize code.

Explicit Threading
Explicit threading includes library-based methods such as Win32* API and POSIX threads. Spawning threads, thread synchronization, and thread destruction are controlled by the developer by placing calls to threading library routines in the application code.

Typically, a fair amount of modification is required to thread an existing application using explicit threading. The process forthreading involves identifying regions to thread, encapsulating the code into functions, and spawning threads based upon these functions. This process makes explicit threading very amenable to functional decomposition.

Other advantages over compiler-based threading include the ability to create thread pools that make it possible to queue independent and dynamic tasks without explicit thread creation. In addition, explicit threading allows threads to have priority and affinity which provides a finer level of control for performance and tuning.

For example, the supra-linear speedup observed in the threading of an intrusion detection system was partially due to cache benefits made possible by processor affinity. Explicit threading is very general and can express many types of concurrency over compiler-based techniques, but does require some knowledge to know when it is best to apply explicit threading.

Threading Methodology
Embedded application development is aided by two components, the right development tools and the right development process. This section details a threading development process that can aid developers employing threading in their application. This process is iterative and concludes when the application requirements have been met. The first step before beginning this process is to tune the application for serial performance.

Serial tuning is less labour intensive and can lead to dramatic performance improvements that can enhance the benefits of parallelizing if done properly. There is a caveat; some serial optimization can limit the ability to parallelize. The best way to consider if overoptimization may inhibit parallelism is to determine if your optimizations are adding the elements that inhibit parallelism defined in the previous Parallelism Limiters section.

A classic example of loop optimization that limits parallelism via threading is strip mining which adds dependencies across iterations of a loop. **Figure 4 below** details the threading development process which is summarized by these four steps:

*** Analysis** ” Determining which portionof the application to thread.
*** Design** ” Deciding how to thread andimplementing.
*** Debug** ” Findingand fixing threading issues in the code.
*** Tune** ” Optimizing for threadperformance issues.

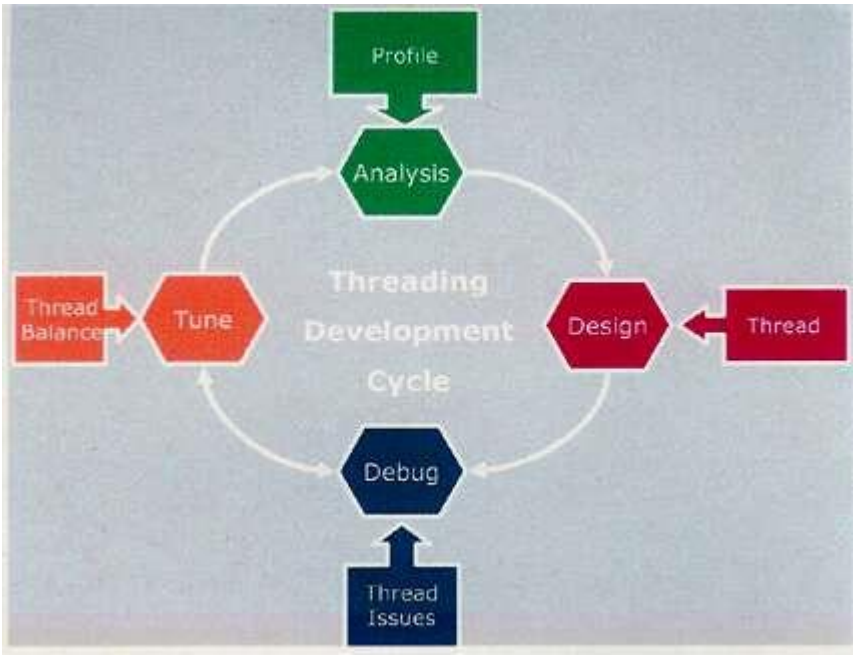

Figure4. Threading Development Cycle