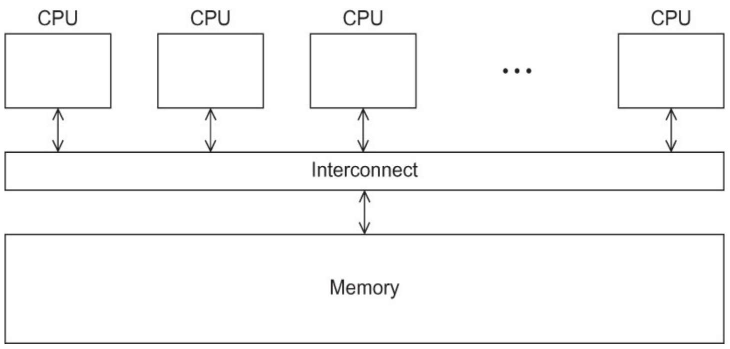
DT081 Year 4 COMPUTER ARCHITECTURE 3

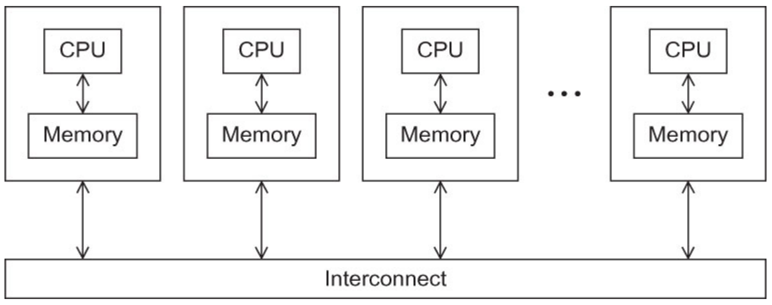
Running Threads Efficiently on Multicore Processors

Introduction

Multicore processor systems can be designed using the shared memory model or the distributed memory model.



Shared Memory Model



Distributed memory model

When the amount of cores is relatively low, the shared memory model is the most commonly used. When a system consists of a large amount of cores the shared memory model becomes a bottleneck and the distributed model is more appropriate. The distributed model uses message passing for communications and there is library of functions named MPI (Message Passing Interface) that can be called from C or C++ to achieve this.

The PCs in this lab use dual core processors and the shared memory model. To benefit from the power of these multicore processors we can use either of 2 programming approaches

1. POSIX threads, i.e. pthreads.
2. OpenMP which is an API for shared- memory parallel programming

You should already be familiar with option1 as we have used these in our real time operating systems course. The idea here is that you implement programs with 2 or more threads and the operating system will schedule these on the available cores. This gives you fine control over when and where you use threads. You can also pin a thread to a core if this is required e.g. due to cache performance issues.
With option 2 you let the compiler decide when to create threads. You just identify where parallel operation is possible. In this lab we will be concentrating on OpenMP.

Intel Parallel Studio Software

To fully utilize the power of Intel multicore processors and achieve maximum application performance on multicore architectures, you must effectively use threads to partition software workloads. When adding threads to your code to create an efficient parallel application, you will typically encounter the following questions:

- Which parts of your application are most appropriate to parallelize to obtain the best performance gains and avoid memory conflicts?
- What programming model and specific threading techniques are most appropriate for your application?
- How do you detect and fix threading and memory errors, which are hard to reproduce because the threaded software runs in a non-deterministic manner, where the execution sequence depends on the run?
- How can you actually boost performance of your threaded application on multicore processors and make the performance scale with additional cores?

Intel Parallel Studio is a software addition to Microsoft Visual Studio that address the issues listed above. It is composed of 3 tools:

Intel Parallel Composer

which helps you to develop effective applications with a C/C++ compiler and advanced threaded libraries.

- Build executables with Intel C++ Compilers for 32-bit processors, a cross compiler to create 64-bit applications on 32-bit systems, and a native 64-bit compiler.
- Write code with Intel Integrated Performance Primitives (Intel IPP), a foundation-level set of building blocks for threaded applications in engineering, financial, digital media, data processing, and mathematics. Intel IPP can also be used with the Microsoft Visual C++ compiler.
- Debug with Intel Parallel Debugger Extension, which integrates with the Microsoft Visual Studio debugger.

- Write code with Intel Threading Building Blocks (Intel TBB), a C++ template library that abstracts threads to tasks to create reliable, portable, and scalable parallel applications. Intel TBB can also be used with the Microsoft Visual C++ compiler.

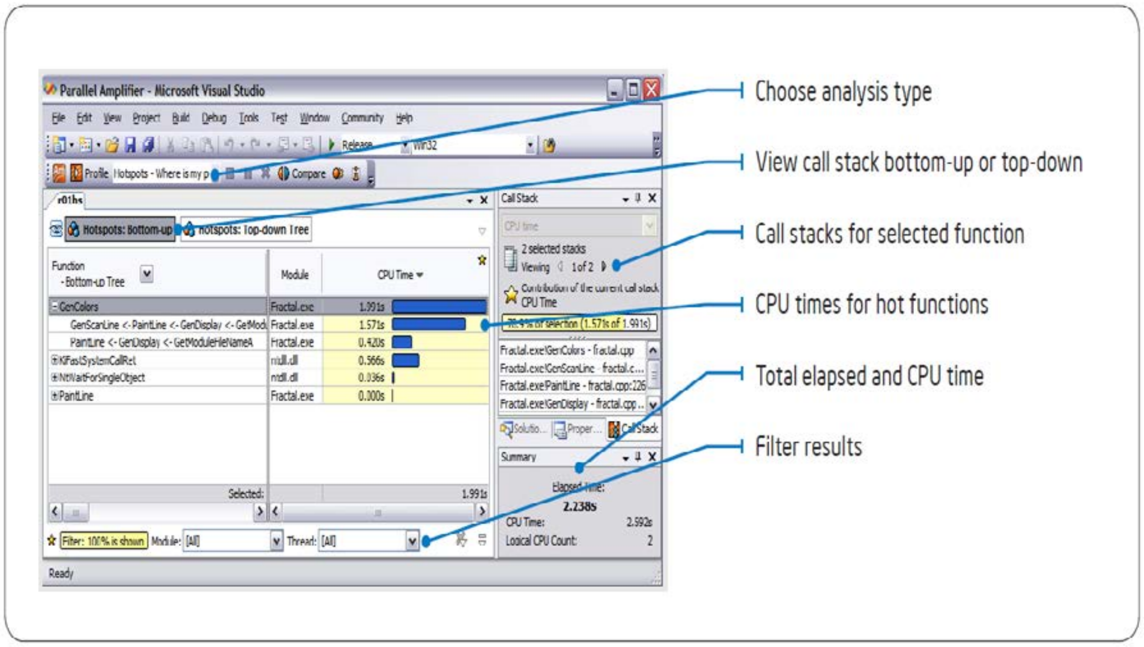
Intel Parallel Amplifier
which helps you to quickly find bottlenecks and tune parallel applications for scalable multicore performance.

- Find application hotspots and drill down to the source code
- Tune parallel applications for scalable performance using
- concurrency analysis
- Use locks & waits analysis to find critical waits that limit parallel
- performance
- Compare results to quickly see what changed, or find regressions

Intel Parallel Inspector
which helps to ensure application reliability with proactive parallel memory and threading error checking.

- Find threading-related errors such as deadlocks and data races
- Find memory errors such as memory leaks and corruption

Intel Parallel Amplifier Hot Spot analysis:
Hot Spot analysis tells you where your application is spending most of its time. Use this to find the functions in your application that consume most of the time. This is where to tune or add parallelism to make your program faster. Intel Parallel Amplifier also shows the stack so you know how the function is being called. For functions with multiple calling sequences, you can see if one of the call stacks is hotter than the others.


Choose analysis type
View call stack bottom-up or top-down
Call stacks for selected function
CPU times for hot functions
Total elapsed and CPU time
Filter results
Parallel Amplifier - Microsoft Visual Studio
File Edit View Project Build Debug Tools Test Window Community Help
Release Win32
Profile Hotspots - Where is my p Compare
r01hs
Hotspots: Bottom-up Hotspots: Top-down Tree
Call Stack
CPU time
2 selected stacks
Viewing 1 of 2
Contribution of the current call stack CPU Time
Solutio... Proper... Call Stack
Summary
Ready

Function - Bottom-up Tree	Module	CPU Time
GenColors	Fractal.exe	1.991s
GenScanLine <- PaintLine <- GenDisplay <- GetMod	Fractal.exe	1.571s
PaintLine <- GenDisplay <- GetModuleFileNameA	Fractal.exe	0.420s
KiFastSystemCallRet	ntdll.dll	0.566s
NtWaitForSingleObject	ntdll.dll	0.036s
PaintLine	Fractal.exe	0.000s
Selected:		1.991s

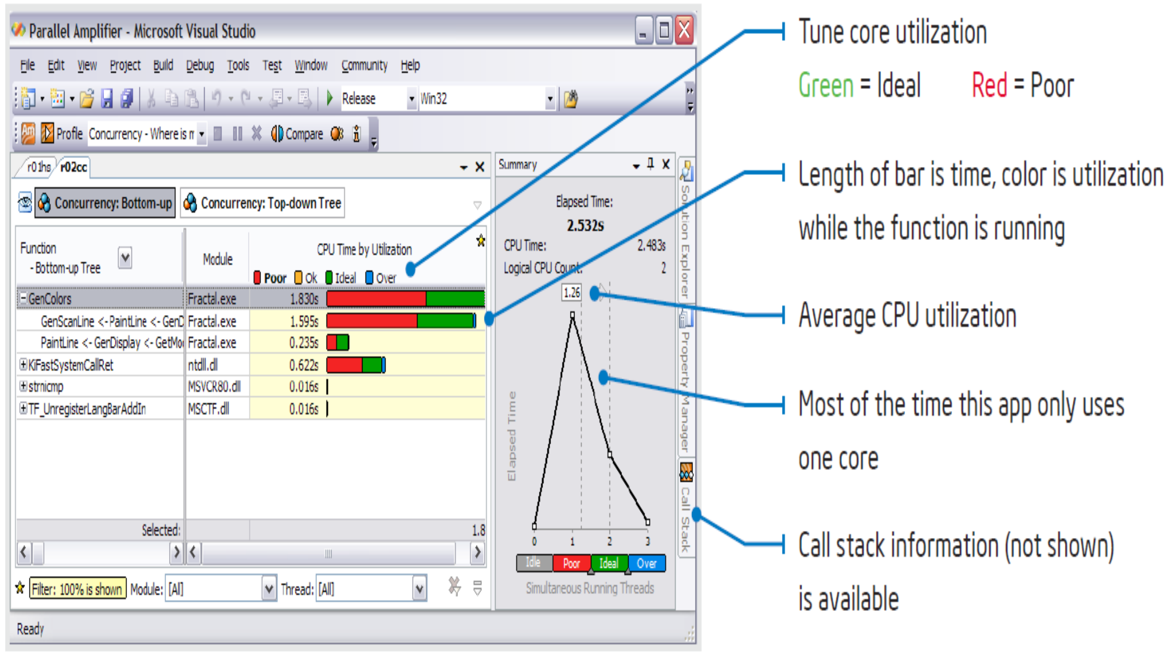
78.9% of selection (1.571s of 1.991s)

Fractal.exe!GenColors - fractal.cpp
Fractal.exe!GenScanLine - fractal.c...
Fractal.exe!PaintLine - fractal.cpp:226
Fractal.exe!GenDisplay - fractal.cpp...

Elapsed Time: 2.238s
CPU Time: 2.592s
Logical CPU Count: 2

Filter: 100% is shown Module: [All] Thread: [All]

<u>Intel Parallel Amplifier Concurrency analysis:</u>
Concurrency analysis finds the functions where you are spending the most time. But it also shows you how well you are utilizing multiple cores. Colour indicates the core utilization while the function is running. A green bar means all the cores are working. A red bar means cores are underutilized. When there is red, you should add parallelism and get all the cores working for you. This helps you ensure application performance scales as more cores are added.



Function - Bottom-up Tree	Module	CPU Time by Utilization
GenColors	Fractal.exe	1.830s
GenScanLine <- PaintLine <- GenD	Fractal.exe	1.595s
PaintLine <- GerDisplay <- GetMo	Fractal.exe	0.235s
KiFastSystemCallRet	ntdll.dll	0.622s
strnicmp	MSVCR80.dll	0.016s
TF_UnregisterLangBarAddIn	MSCTF.dll	0.016s

OpenMP 3.0

OpenMP is a set of specifications for parallelizing programs in a shared memory environment. OpenMP provides a set of pragmas, runtime routines, and environment variables that programmers can use to specify shared-memory parallelism in Fortran, C, and C++ programs.

When OpenMP pragmas are used in a program, they direct an OpenMP-aware compiler to generate an executable that will run in parallel using multiple threads. Little source code modifications are necessary (other than fine tuning to get the maximum performance). OpenMP pragmas enable you to use an elegant, uniform, and portable interface to parallelize programs on various architectures and systems. OpenMP is a widely accepted specification, and vendors like Sun, Intel, IBM, and SGI support it.

OpenMP takes parallel programming to the next level by creating and managing threads for you. All you need to do is insert appropriate pragmas in the source program, and then compile the program with a compiler supporting OpenMP and with the appropriate compiler option. The compiler interprets the pragmas and parallelizes the code. When using compilers that are not OpenMP-aware, the OpenMP pragmas are silently ignored.

OpenMP Pragmas

The OpenMP specification defines a set of pragmas. A pragma is a compiler directive on how to process the block of code that follows the pragma. The most basic pragma is the **#pragma omp parallel** to denote a ***parallel region***.

OpenMP uses the fork-join model of parallel execution. An OpenMP program begins as a single thread of execution, called the initial thread. When a thread encounters a parallel construct, it creates a new team of threads composed of itself and zero or more additional threads, and becomes the master of the new team. All members of the new team (including the master) execute the code inside the parallel construct. There is an implicit barrier at the end of the parallel construct. Only the master thread continues execution of user code beyond the end of the parallel construct.

The number of threads in the team executing a parallel region can be controlled in several ways. One way is to use the environment variable **OMP_NUM_THREADS**. Another way is to call the runtime routine **omp_set_num_threads()**. Yet another way is to use the **num_threads** clause in conjunction with the **parallel** pragma.

OpenMP supports two basic kinds of work-sharing constructs to specify that work in a parallel region is to be divided among the threads in the team. These work-sharing constructs are loops and sections. The **#pragma omp for** is used for loops, and **#pragma omp sections** is used for ***sections*** -- blocks of code that can be executed in parallel.

The **#pragma omp barrier** instructs all threads in the team to wait for each other before they continue execution beyond the barrier. There is an implicit barrier at the end of a parallel region. The **#pragma omp master** instructs the compiler that the following block of code is to be executed by the master thread only. The **#pragma omp single** indicates that only one thread in the team should execute the following block of code; this thread may not necessarily be the master thread. You can use the **#pragma omp critical** pragma to protect a block of code that should be executed by a single thread at a time. Of course, all of these pragmas make sense only in the context of a **parallel** pragma (parallel region).

Open MPRuntime Routines

OpenMP provides a number of runtime routines can be used to obtain information about threads in the program. These include **omp_get_num_threads(), omp_set_num_threads(), omp_get_max_threads(), omp_in_parallel()**, and others. In addition, OpenMP provides a number of **lock** routines that can be used for thread synchronization.

OpenMP Examples

Using a simple matrix multiplication program you can see how OpenMP can be used to parallelize the program. Consider the following small code fragment that multiplies 2 matrices. This is a very simple example and, if you really want a good matrix multiply routine, you will have to consider cache effects, or use a better algorithm (Strassen's, or Coppersmith and Winograd's, and so on).

```
for (ii = 0; ii < nrows; ii++) {
        for (jj = 0; jj < ncols; jj++) {
                for (kk = 0; kk < nrows; kk++) {
                        array[ii][jj] += array[ii][kk] * array[kk][jj];
                }
        }
}
```

Note that at each level in the loop nest above, the loop iterations can be executed independently of each other. So parallelizing the above code segment is straightforward: Insert the **`#pragma omp parallel for`** pragma before the outermost loop (**`ii`** loop). It is beneficial to insert the pragma at the outermost loop, since this gives the most performance gain. In the parallelized loop, variables **`array`**, **`ncols`** and **`nrows`** are shared among the threads, while variables **`ii, jj`**, and **`kk`** are private to each thread. The preceding code now becomes:

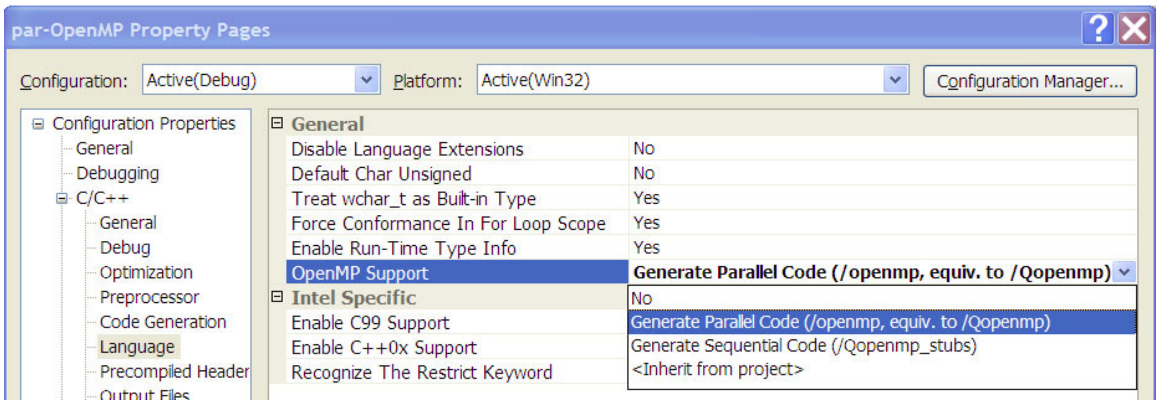
```
#pragma omp parallel for shared(array, ncols, nrows) private(ii, jj, kk)
for (ii = 0; ii < nrows; ii++) {
        for (jj = 0; jj < ncols; jj++) {
                for (kk = 0; kk < nrows; kk++) {
                        array[ii][jj] = array[ii][kk] * array[kk][jj];
                }
        }
}
```

<u>Confirm that Visual Studio will use the Intel C++ compiler</u>

1. From the Microsoft Visual Studio main menu choose **Project** > **Intel Parallel Composer**. If you see Use Visual C++ then everything is o.k.
2. Otherwise from the Microsoft Visual Studio main menu choose **Project** > **Intel Parallel Composer** > **Use Intel C++**.
3. Click **Yes** in the **Confirmation** dialog box. This configures the solution to use the Intel® C++ Compiler.

<u>Set up OpenMP recognition in Visual Studio.</u>

1. From the main menu choose **Project** > **Properties**.
2. In the **Property Pages** window, locate and select **C/C++** > **Language** > **OpenMP Support**.
3. From the drop-down menu, select **Generate Parallel Code (/openmp, equiv. to /Qopenmp)**.


par-OpenMP Property Pages

Configuration: Active(Debug)   Platform: Active(Win32)   Configuration Manager...

- Configuration Properties
  - General
  - Debugging
  - C/C++
    - General
    - Debug
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Header
    - Output Files

General	
Disable Language Extensions	No
Default Char Unsigned	No
Treat wchar_t as Built-in Type	Yes
Force Conformance In For Loop Scope	Yes
Enable Run-Time Type Info	Yes
OpenMP Support	Generate Parallel Code (/openmp, equiv. to /Qopenmp)
Intel Specific	
Enable C99 Support	
Enable C++0x Support	
Recognize The Restrict Keyword	

No
Generate Parallel Code (/openmp, equiv. to /Qopenmp)
Generate Sequential Code (/Qopenmp_stubs)
<Inherit from project>

4. Click **Apply**.

Enable the OpenMP diagnostics

5. While still in Property Pages, add /Qdiag-enable:openmp under Command Line
   ➢ **Additional Options**.

Laboratory Task

The code below intialises 2 large matrix arrays and then multiplies them to produce a product matrix.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

#define SIZE 1000

void MatrixMult(int size, float m1[][SIZE], float m2[][SIZE], float
result[][SIZE]);

int _tmain(int argc, _TCHAR* argv[])
{
      int size = SIZE;
      static float m1[SIZE][SIZE]; // must be static if we want to use
                                   // large arrays. This way matrix is
                                   //stored on the heap rather than
      static float  m2[SIZE][SIZE]; // on the stack. If a large array is
                                    // stored on the stack it would
      static float result[SIZE][SIZE]; // cause a stack overflow.
      int i,j,k=0;

      for(i =0; i<SIZE; i++){
            for(j=0;j<SIZE;j++){
                  m1[i][j] = i*j;
            }
      }
      for(i =SIZE; i>0; i--){
            for(j=SIZE;j>0;j--){
                  m2[i-1][j-1] = k++;
```

```
			}
		}

		MatrixMult(size, m1, m2, result);
		cout<<"\n result[0][0] = "<<result[0][0]<<"\n";
		return 0;
}


void MatrixMult(int size, float m1[][SIZE], float m2[][SIZE], float 
result[][SIZE])
{
		int a,b, i, j, k;
		int nthreads;

		for(i=0; i<size; i++){
			for (j=0; j< size; j++){
				result[i][j] = 0;

				for(k=0; k<size; k++){
					result[i][j] += m1[i][k] * m2[k][j];
				}
			}
		}
}
```

1. Create a new project in Visual Studio and copy the code into it.
2. Build and run the code.
3. Use the Parallel Amplifier profiling tool to identify where the the hot spots are. Use the top down tree view. Double click on the hottest spot (in blue) to see the code inside the function. There you can see the actual lines that are consumming the most time.
4. Find the total elapsed time and the total CPU time consumed by your program and the unused CPU time. Ideally the unused CPU time should be close to zero.
5. Use the Parallel Amplifier profiling tool to identify where the concurrency is poor.
6. Identify the parts of the code that could be parallelised.
7. Add OMP pragma directives to allow the compiler to produce threaded code at the appropriate points in the code.
8. Rerun the Parallel Amplifier profiling tool and confirm that the concurrency is now maximised.
9. How does the total elapsed time relate to the total CPU time?

Dr. R. Lynch