Introduction to Cilk++ Programming

Tutorial at PADTAD

Laboratory Exercises

Pablo Halpern

20 July 2009

1 Choice of Development Environment

To do the exercises in this tutorial, you have a choice of three of development environments:

(a) Windows (with Microsoft Visual Studio) on your own laptop. If you have not already installed Visual Studio on your machine, we ask that you find a partner that did. Visual Studio can take a long time to install and patch, and we are very time constrained today. Please download Cilk++ from http://www.cilk.com/home/download-cilk/.

(b) Linux on your own laptop. Please download Cilk++ from http://www.cilk.com/home/download-cilk/.

(c) Linux on a PADTAD server via an ssh connection. Cilk++ is already installed. Everybody is sharing the same user account (Username: padtad, Password: qwer4321. Please create a subdirectory under the home directory for your own use.

2 Introduction to Cilk++

The Cilk++ platform provides a compiler for the Cilk++ parallel-programming language (a straightforward extension to C++), the *cilkscreen* race detector, and the *cilkview* scalability analyzer. In this laboratory, you will install Cilk++ on your Windows or Linux computer (if you haven't already done so), compile and run an example program, run cilkscreen on the program to check for data races, and run cilkview on the program to analyze theoretical and actual multicore speedup.

(a) Many computers, especially laptops, are configured to conserve power by slowing down the CPU clock during periods of relative inactivity. While this may be a good idea in general, it interferes with getting accurate timing and speedup measurements. For the duration of this class, please disable this power-saving feature as follows:

- On (most) Linuxes, type:

```
# mkdir  /oldcpusettings
# cd /sys/devices/system/cpu
# for cpu in cpu?; do
>  cat $cpu/cpufreq/scaling_governor >  /oldcpusettings/$cpu
> done
```

If you wish to restore the previous settings later, type:

```
# cd  /oldcpusettings
# for cpu in cpu?; do
>  cat $cpu > /sys/devices/system/cpu/$cpu/cpufreq/scaling_governor
> done
```

- On Windows XP:
  1. Select "Power Options" from the control panel.
  2. Select "Home/Office Desk" from the Power schemes drop-down.
  3. Click "OK"
  4. run "`powercfg /q`" from a command shell. The "Processor Throttle (AC)" setting should be "NONE".
- On Windows Vista:
  1. Select "Power Options" from the control panel.
  2. Click on the radio button for "High Performance."
  3. Click the "Change plan settings" link.
  4. Click the "Change advanced power settings" link.
  5. Scroll down to "Processor power management". Click the "+" to expand the options.
  6. Click the "+"s for "Minimum processor state" and "Maximum processor state" to verify that the values for "Plugged in" is "100

(b) Shut down any background programs that may consume significant cpu resources such as email programs and web sites that run Flash or Javascript (e.g., gmail). If any of these programs chooses to grab some resources while you are running a timing test, your numbers will be skewed.

(c) In the materials that were distributed to you electronically, you will find a directory named `cilk`+1.1.0+, which contains the Cilk++ Programmer's Guide in PDF format along with separate installation files for 32-bit Linux, 64-bit Linux and 32-bit Windows. Read and follow the instructions in the Programmer's Guide for installing Cilk++ for your operating system.

(d) Linux only: Use your favorite package manager to install the `gnuplot` program. `gnuplot` is used by the cilkview scalability analyzer.

(e) Windows only: The Cilk++ examples for Windows are packaged in a separate installer file in the Cilk++ installation directory. Install the examples after you install Cilk++ itself.

(f) In the "Getting Started" section of the manual, follow the instructions to "Build and Run a Cilk++ Example". (On the PADTAD machines, you will need to copy the examples into your own subdirectory.) Note that, if you use Microsoft Visual Studio, you currently need to open a command window and invoke cilkview from the command

line. (cilkview is not yet integrated with the Visual Studio IDE.) NOTE: When running cilkview, specify the "`-trails all`" option to generate actual run data in addition to the predicted speedup data. (If you have n processors, your program will run $n + 1$ times.)

(g) If you are using a PADTAD machine, unpack `exercises.tgz` into your private subdirectory. If you have your own laptop, copy `exercises.tgz` or `exercises.zip` from the PADTAD servers using scp:

```
$ scp padtad@aml.cs.byu.edu:exercises.tgz .
```

Unpack the exercises archive.

3 Parallelizing Matrix Multiplication

In this laboratory, you will write a multithreaded program in Cilk++ to implement matrix multiplication. One of the goals of this assignment is for you to get a feeling of how ***work***, ***span***, and ***parallelism*** affect performance. First, you will parallelize a program that performs matrix multiplication using three nested loops. Then, you will write a serial program to perform matrix multiplication by divide-and-conquer and parallelize it by inserting Cilk keywords. If you have time, you can implement a parallel version of Strassen's algorithm. Finally, you may optimize your program however you wish and possibly win a prize!

For those of you who have not looked at matrix multiplication in a little while, the problem is to compute the matrix product

$$C = AB \ ,$$

where C, A, and B are $n \times n$ matrices. Each element c_{ij} of the product C can be computed by multiplying each element a_{ik} of row i in A by the corresponding element b_{kj} in column j in B, and then summing the results, that is,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \ .$$

For more information on matrix multiplication, please see `http://en.wikipedia.org/wiki/Matrix_multiplication#Ordinary_matrix_product`. A good description of the algorithms used in this lab can be found in [1].

The nested-loop and divide-and-conquer versions of these programs can be adapted to work with arbitrary rectangular matrices. To simplify the interface, however, we limit ourselves to $n \times n$ square matrices. Strassen's algorithm is further limited to $n \times n$ square matrices where n is an exact power of 2.

3.1 Matrix multiplication using loop parallelism

The sources for this exercise can be found in the `mm` subdirectory of the `exercises` directory. Use the `Makefile` to build for Linux, and `mm_loops.sln` to start Visual Studio for Windows.

The file `mm_loops.cilk` contains two copies of a $\Theta(n^3)$-work matrix multiplication algorithm using a triply nested loop. The first copy (`mm_loop_serial`) is the control for verifying your results — leave it unchanged. The second copy (`mm_loop_parallel`) is the one that you will parallelize.

This file also contains a test program that verifies the results of your parallel implementation and also provides infrastructure for timing and measuring parallelism.

(a) Compile `mm_loops` with optimization, and verify that it operates correctly. Supply the `--verify` command-line option to force running all tests. (Microsoft Visual Studio users should also supply the `--pause` option.)

(b) Parallelize the `mm_loop_parallel` function by changing the outermost `for` loop into a `cilk_for` loop. Verify correct results with the `--verify` option. Run cilkview on your program to determine theoretical and actual speedup:

- On Linux, type:

```
$ cilkview ./mm_loops
```

- In Microsoft Visual Studio, select:

  `Tools|Visual Studio 2005 Command Prompt` to open a command shell. Change directory to your `mm` directory and type:

```
mm> cilkview Release\mm_loops.exe
```

Do not use the `--verify` or `--pause` options when running cilkview.

(c) Change the outermost `cilk_for` back into a serial `for` loop and change the middle `for` loop into a `cilk_for` loop. Repeat the test with the `--verify` option, and then repeat the cilkview test. Did any results change? Try making both loops parallel. Which of the three combinations produces the best results?

(d) (*Optional.*) What happens if you change the innermost `for` loop into a `cilk_for`?

3.2 Matrix multiplication by divide-and-conquer

Divide-and-conquer algorithms often run faster than looping algorithms, because they exploit the microprocessor cache hierarchy more effectively. This section asks you to write a divide-and-conquer implementation of matrix multiplication. You will find the source code for the incomplete program in `mm_recursive.cilk`. The program contains two implementations of matrix multiplication. The `mm_loop_serial` function is the same as in Section 3.1 and is provided for verification and timing comparisons. The `mm_recursive_parallel` function is the skeleton of a divide-and-conquer implementation.

Your recursive implementation will be based on the identity

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix},$$

where A_{11}, A_{12}, etc., are submatrices of A. In other words, matrix multiplication can be performed by subdividing each matrix into four parts, then treating each part as a single element and (recursively) performing matrix multiplication on these partitioned matrices. (The number of columns in A_{11} must match the number of rows in B_{11}, and so forth.) Although the algorithm operates recursively, its work is still $\Theta(n^3)$, the same as the straightforward algorithm that employs triply nested loops.

(e) Compile `mm_recursive`, and verify that it compiles but fails to run successfully when using the `--verify` command-line argument. The failure is caused by the fact that the `mm_recursive_parallel` has not been fully implemented yet. You may also get numerous warnings about unused variables. All those warnings will go away when you write your recursive code.

(f) In the file `mm_recursive.cilk`, fill in code in the `mm_recursive_internal` function to implement the divide-and-conquer algorithm. The error-prone task of subdividing the matrices into four parts has been done for you. All you need to do is to fill in the recursive calls (eight in total – one for each of the eight matrix-multiplications in the algorithm). Compile and run your new `mm_recursive` program and verify that it runs successfully.

(g) Make your recursive function parallel by adding `cilk_spawn` in front of six of the eight recursive calls. You will first need to reorder the calls into two groups of four, with a `cilk_sync` after each group, so that no submatrix of `C` is updated twice within the same group of parallel calls. Alternatively, you can create a helper function that calls `mm_recursive_internal` twice in series, then change `mm_recursive_internal` to call the helper function four times in parallel. Compile and run your new `mm_recursive` program. Verify that it is correct and run cilkview. For large matrices, how does the performance of the recursive algorithm compare with the nested-loops algorithm on a single processor?

(h) Uncomment the line in the program that reads "`#define USE_LOOPS_FOR_SMALL_MATRICES`". This change causes the algorithm to change from divide-and-conquer recursion to a triply nested loop for small matrices. How does this change impact performance? How does the performance of this new version compare to the loop-only version?

3.3 Matrix multiplication by Strassen's algorithm (*optional*)

For large matrices, Strassen's algorithm can outperform the other two methods we've seen, because it entails $\Theta(n^{\lg 7}) = O(n^{2.81})$-work, rather than $\Theta(n^3)$. Section 3.3.1 describes Strassen's algorithm in detail (although it doesn't provide insight as to how Volker Strassen discovered this bizarre method). Section 3.3.2 describes the exercise itself.

3.3.1 Description of Strassen's Algorithm

Strassen's algorithm can multiply two $n \times n$ matrices in $\Theta(n^{\lg 7}) = O(n^{2.81})$ work, which is asymptotically better than more straightforward methods that require $\Theta(n^3)$ work. Let A and B be two $n \times n$ matrices. For the rest of the assignment, assume that n is an exact power of 2. Recall that the product of A and B is defined to be $C = AB$, where

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \ .$$

Although this definition leads to a straightforward $\Theta(n^3)$-work algorithm to compute the product, a remarkable identity can be exploited which leads to a divide-and-conquer algorithm with work

$\Theta(n^{\lg 7}) = O(n^{2.81})$. Partition C, A, and B as follows:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}, \quad A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

Then, we have

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

Instead of doing the usual divide-and-conquer, perform the following calculations:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\ M_2 &= (A_{21} + A_{22})B_{11}, \\ M_3 &= A_{11}(B_{12} - B_{22}), \\ M_4 &= A_{22}(B_{21} - B_{11}), \\ M_5 &= (A_{11} + A_{12})B_{22}, \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}). \end{aligned}$$

We can now express C in terms of the M's as follows:

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7, \\ C_{12} &= M_3 + M_5, \\ C_{21} &= M_2 + M_4, \\ C_{22} &= M_1 - M_2 + M_3 + M_6. \end{aligned}$$

If all intermediate results are appropriately saved, a multiplication of size n can be reduced to 7 multiplications of size $n/2$, plus 18 matrix additions.

3.3.2 Exercise (matrix-multiplication using Strassen's algorithm)

(i) Modify your divide-and-conquer solution implement Strassen's algorithm. Verify its correctness, run cilkscreen and run cilkview.

To make a practical implementation, you should probably switch to the ordinary serial nested-loop algorithm for small enough submatrices. For some applications, Strassen's algorithm produces lower-quality answers than the standard algorithm does, due to loss of precision from round-off. You may not actually see round-off error using our test code, since the matrices are initialized with random integers.

3.4 Maximize Speed (*optional*)

The last part of the lab is a no-holds-barred effort to speed up your matrix multiplication.

(j) Revise your matrix-multiplication program to enhance its performance.

Here are some ideas, which may or may not work, for speeding up your code:

- Use the `-S` option to `cilk++` (Linux) or the `/cilkp keep` option to `cilkpp` (Windows), and look at the assembly language produced by the compiler. (*Note:* On Windows you should look at the `.ysm` rather than the `.asm` file.) See whether rearranging the source code produces a faster executable. Remember that memory references are expensive, and register operations are cheap.
- Exploit the processor's registers and pipeline by coarsening the base case to use a loop for sufficiently small matrices. Experiment with different thresholds for coarsening.
- In addition to coarsening the base case to use a loop, open-code small 2×2 matrix multiplications within the loop.
- Reorder how subproblems are spawned to get better locality.
- Change the representation of matrices from row-major to some other order. Remember to change it back before you output the result.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.