Oct. 27, Assignment #2, due Nov. 3 & Nov. 11 @ 11:59 pm (15%)

Gardner Home | Course Home

Marking for this assignment (breaking down the 15%):

- Part 1 pseudocode & report: 5%
- Part 2 software: 6%
- Part 2 report: 4%

Software that does not compile, link, and execute on lockhart will receive 0%. The report does not have to be beautifully bound, but should be presentable and not be hand written.

Note: Special thanks to Dave McCaughan for the basis of this assignment, and the parallaldo generator program.

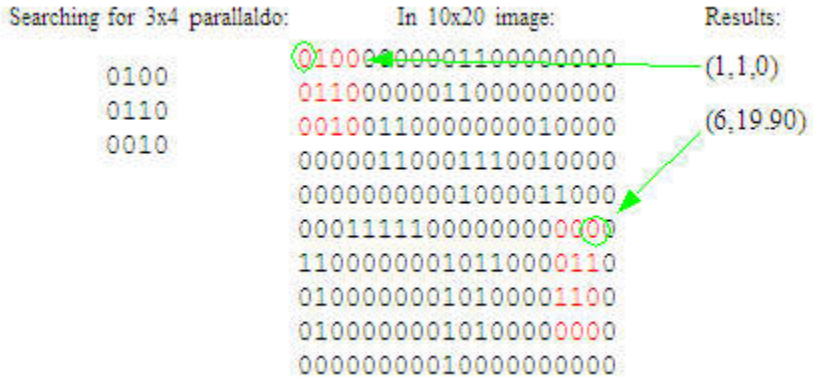
Where's Parallaldo? parallel pattern searching

"Where's Waldo?" is a popular series of books (ostensibly for children) in which they are challenged to find a particular character in a very complex picture. From a computational point of view, this can be seen as a form of template matching, in which we attempt to locate a region of a larger space which matches some small template example. This is easiest to visualize with images where the problem becomes one of finding a region of a larger image which matches a small exemplar pattern.

For this assignment, you will implement a parallel version of a template matching algorithm in which a large number of images are scanned in order to locate a smaller number of template images (called **parallaldos** to make our abuse of the real name of this book complete). To simplify the problem, we will use basic black-and-white images that can be represented in plain text as a two-dimensional array of 1-or-0 "pixels."

Example

In the example below, the parallaldo is found in two places (red) in the image, the second being a 90-degree clockwise rotation of the template.



Note how the findings are reported: (y,x,r) where (y,x) denotes the location in the image of the (unrotated) template's upper left corner, and r being the degrees of *clockwise* rotation. The convention of the first coordinate being the row (here numbered starting from 1) is normal for graphics and matrix data. Of course, you can store the data in memory however you want.

Also note the opportunity for ambiguity: If the parallaldo is symmetrical--which is true for the example--it could be validly matched in any of 2 or even 4 rotations. That is, (1,1,0) is the same finding as (3,4,180), and (6,19,90) is the same as (9,17,270).

For the purpose of further simplifying the program, we'll make the following rules:

1. There won't (intentionally) be more than *one* parallaldo in a given image. (If someone finds a test case with more than one, it will be replaced.)

2. Symmetrical parallaldos will be avoided, so that the rotation ambiguity does not arise.

These rules will insure determinism: that there is only one correct result for any pair of parallaldo and image files, regardless of how you write your search algorithm. Furthermore, if your program finds a match, it can terminate its search immediately without looking for additional matches or other rotations of the same parallaldo in the current image.

Data specification

Data will be furnished in two directories, one containing candidate parallaldos for which to search, the other images in which to search. Your program will search for every combination of parallaldo and image pairs.

The directories will be populated with text files consisting of a header line giving the dimensions of the rectangular image in R rows x C columns, followed by R rows of C 1s and 0s. Your program may assume that all files in the specified directories are intended as input.

To help with testing, you can find in Public Documents on lockhart the following files:

- **parallaldos:** sample templates to search for (*.txt)
- **targets:** sample images to be search in (*.img)
- **results.txt:** correct output for above sample files
- **parallaldos.tgz** (19 MB): tarball of the above in case you want to test on another computer
- **parallaldo_gen.c** : program to generate random test cases (read the comments for how to run)

Program specification

NOTE: Your program should be capable of running in serial fashion when it is started with only one thread. This is not difficult to arrange if you plan it from the start. We need to have timing from a serial version in order to calculate speedup.

The search algorithm is completely up to you. It can be as "brute force" or as clever as you want. There are several places where parallelism can be exploited (suggestions below).

The program should be called **aldo** , and take 3 command line arguments:

aldo *aldodir imagedir cores*

- aldodir is the directory populated with parallaldo templates
- imagedir is the directory populated with images to search in
- cores is the total number of cores to utilize (see below)

Print an error message and abort if the arguments are invalid, or if either directory is empty.

The *cores* argument is intended to reflect the number of cores in use for computation. Depending on your work allocation methodology, your main thread may be doing some non-CPU intensive I/O and supervision and then just waiting for the worker threads to join, in which case it does not need to be counted as a core. This means that with an argument of 8, for example, you could call pthread_create() 8 times and with your main/master thread actually be using 9 threads. But since you're only fully utilizing 8 cores, this still counts as 8 for this exercise. And since, like for A1, you are not required to produce a separate purely serial version, it is understood that with 1 core you will essentially be running your parallel version in serial mode. With 2 cores, you can already have 2 workers. Thus the case we had with Pilot--where 2 processes meant PI_MAIN + one worker--does not apply here, since in Pilot

PI_MAIN has to consume a process just by itself. Anyway, this *cores* argument will later be used as the X axis on your graphs, so you should be clear what it represents.

What about *cores* > 16? This can happen if you are intentionally "oversubscribing" the physical cores. A legitimate use for this in a scalable program could be appointing threads to read files (which assumes they will mostly be blocked on I/O and hardly occupy a core). If you do all your file reading in the main thread, this comment does not apply, and you should only oversubscribe to see for yourself if/how it affects the performance.

The output of aldo is printed on stdout, where it can be captured in a log file using the command redirection operator ">". Start each line of result output with the character "$" so that meaningful output can be readily grepped out from among any other lines printed (e.g., in case you inserted printf's for debugging).

Each match is printed on a separate line using the following format **exactly** (to facilitate automatic verification):

$parfile imfile (y,x,r)

where parfile and imfile are the bare filenames (no directory path) of the parallaldo and its matching image, respectively, and (y,x,r) are bracketed integers printed without extra spaces or leading zeroes. Thus, the result line will have **only two spaces.** Nothing is printed for image files that have no parallaldo match. The order of output lines is not significant; it will be sorted for verification purposes.

End with a report of total elapsed time in seconds as follows:

- Use the StartTime() and EndTime() functions supplied in Public Documents; they work like the Pilot functions: **C:\Public\Users\Documents\pa-demo\PrimeSingle\wallclock.h**
- Start timing as the very first thing in main().
- Stop timing after all output has been printed, then print the value of EndTime() using "%lf" format, for example, "5.2356 seconds". This should be the final line printed on stdout.

The sample file **results.txt** is in the required format (except for missing the elapsed time report).

Suggestions for parallelizing

You're under no obligation to follow any of these suggestions. If you can think of a faster way to do something, go for it.

One way to structure the program is to start by reading all the parallaldos into memory (since there aren't a lot of them and they are small). The program can then have a main loop that reads in each target image (some of which are quite large), and an inner loop that searches for each parallaldo in turn. Note that when you find a match, you can break out of the inner loop and go on to the next image.

Opportunities for exploiting data parallelism include:

- The main image file loop (these iterations are independent of one another).
- The inner parallaldo search loop (these are independent, *but* once one match is found, further searching in that image can be aborted; this problem characteristic might have the potential to yield superlinear speedup if less work is done than for serial search).
- The loops of the pixel search algorithm (going through the rows and columns of each image).

Consider carefully how you can overlap (slow) disk I/O operations with (fast) computation. One popular technique is known as "double buffering." This is where you are reading one image file into a buffer while the processor(s) are searching through another image already read in. There are a couple ways of approaching this. One is to use an asynchronous read (you start the read, then check the status later to see if it finished); another is to devote a separate thread to do a normal synchronous read (while it blocks, other threads are carrying on so no CPU time is wasted).

Keep load balancing in mind, because the image sizes vary greatly. Furthermore, if you find a match, the search

could terminate quickly; but if you don't, you'll have to run right through the entire image.

Anyway, the trick is to find useful work to keep all 16 cores busy, thus reducing idle time. Generally speaking, the higher/coarser levels of parallelism will be easier to program, so if you can occupy all the cores that way it's good enough, and there may be nothing to be gained by parallelizing at the lowest/fine-grained level.

Part 1 Deliverables (due Nov. 3 in class)

Your output for Part 1 is a paper submission consisting of:

1. **Pseudocode** using the Peril-L conventions. It should cover the entire program logic (not necessarily in detail; it is pseudocode, after all), showing where you are planning to use threads. Use a word processor so that you can **underline global variables.** Once you get down to the level of the loop that is parallelized, it is not necessary to give deeper detail. For example, if you are parallelizing the pixel search algorithm, you need to show how you're dividing up the image's pixels. But if you're parallelizing a higher level, you needn't expose details about rows and columns. You also needn't show details of how you're listing the contents of the Windows directories and opening files (which is not trivial code, BTW).

2. **Report** explaining *why* you organized your solution the way you did. Mention as many factors as you can, and explain clearly how they were reflected in design decisions. It is not sufficient to throw around terms like "granularity" and "false-sharing" recklessly. Ad hoc designs that have no thoughtful rationale will not be highly rated.

In this part, the pseudocode, including Peril-L conventions, and the report will carry equal weight.

Pthreads implementation, Part 2

Code your solution to run on lockhart using pthreads in Intel Parallel Studio XE 2016 according to the program specification above.

You can assume that the maximum no. of cores is 16, for the purpose of defining static data structures. As mentioned above, you may choose to spawn additional threads that are not for computation.

Part 2 Deliverables

- Source code via electronic submission using Subversion under tag **cis3090-A2** :

  1. "aldo" program in a Visual Studio "solution" folder. Each source file that you create must include a **file header** containing your name and student number, the filename, and a brief description of the purpose of the file. Submissions lacking proper file headers will be penalized.

  2. Accompanying Visual Studio files in the same directory, so that the solution can be readily opened and built.

- Documentation, printed and handed in the following day in class:

  1. Write your impressions of working with Parallel Studio and pthreads.

  2. Describe any changes you made to your application's parallel architecture as you implemented the pseudocode and tuned the program. Justify the changes based on the concepts we've studied (e.g., reducing false sharing, load balancing, etc.).

  3. Make a timing graph of total execution time (Y axis) vs. number of cores (1-16 with at least 8 data points) running the Release (optimized O2) version using the sample directories provided in Public Documents. Also make speedup and efficiency graphs based on the same data. Draw the linear speedup line on the speedup graph. Write some observations about the graphs. **Given the results, how scalable was your parallel design's performance?**

  4. For this assignment, you don't need to rerun and graph every trial with the unoptimized Debug

version, but do enough testing with O0 so you can report whether compiler optimization made a difference and to what extent.

Part 2 Bonus marks

Bonuses of 15% (=entire assignment mark * 1.15), 10%, and 5% will be given for the 3 fastest programs. A particular set of files **to announced on CourseLink** will be applied to all submissions, and the programs that compute the correct output in the shortest times on an unloaded system win. Timing will be determined by the average of 3 runs.

NOTE: Rematch provision! The 2nd and/or 3rd place runners-up *may* request a rematch. In that event, block time (exclusive use of machine) will be available for further profiling and tuning prior to the runoff match using a different set of files. The bonuses will be redistributed among the 3 contenders according to the new times.

Marks for Assignment #2 Part 2

For the software, mark is 0-6 based chiefly on functionality:

6 = Parallel program that produces correct results for all data files, reports timings, and runs in serial mode when started with one process

1.5-5.5 = Same as 6, but with deductions according to deficiencies

1 = Parallel program that runs cleanly, and is coded to address the problem, but results are all incorrect or missing

0 = Fails to compile and link, or crashes without producing any results

In addition, 1/2 mark penalties will be deducted for specific "sins": numerous compiler remarks/warnings, file headers missing/incomplete, Visual Studio "solution" folder incomplete, program name wrong, arguments mishandled, output filename incorrect, etc.

[Concerning Part 1, 2 functional marks will be deducted for no submission, or 1 mark if the submission does not meet the requirements.] --*This was the rubric for A1 Part 1 copied here by mistake.*

The documentation starts with 4 marks, then deduct 1 mark for any item missing, or 1/2 mark for any lame or poorly-done item. If you didn't have any working software to graph, you'll lose at least 1 more mark to be fair to those who conducted timing and produced their graphs.

Gardner Home | Course Home