Sep. 25, Assignment #1, due Oct. 6 in class, and Oct. 14 @ 11:59 pm (10%)

Gardner Home | Course Home

Marking: For this assignment, the software is worth 60% and the documentation 40%. Software that does not compile, link, and execute on mako will receive 0%. The report does not have to be beautifully bound, but should be presentable and not be hand written.

Detecting paliodromes

A palindrome is a string of characters that reads the same both forwards and backwards. For example, "able was I ere I saw elba" is a palindrome (with a Napoleonic reference). They can be complete phrases (in which punctuation and spaces, inserted for readability, are typically disregarded) or just single words (like noon, civic, rotor, etc.). You can easily find a multitude of examples on the Web.

For this problem, you will be given a file that contains a series of strings of no more than 200 ASCII bytes each (i.e., maximum strlen=199), containing only lower case alphabetic characters, one string per line. Your program will check each string and report the palindromes. Since this file is auto-generated, do not expect the strings to form recognizable English words. It is enough to analyze whether they read the same forwards and backwards. This problem becomes worthy of parallel processing when the size of the input file becomes quite large.

Data specification

Data will be furnished in one file of any length as described above.

Output must be produced in a file named **pal.out** . Output the line number (starting from 1) of each discovered palindrome using a format that gives 10 digits with leading zeroes, e.g., 0000001234 is the way to print line no. 1234. Your output file does not need to be in any particular order. The TA will sort and diff your output against the correct answers.

To help with testing, you can find on mako in /home/wgardner/a1 files with such names:

- **strings-n:** *n* lines of sample data (ranging from 5% to 50% palindromes)
- **pals-n:** sorted answers for the corresponding strings file

Your program will be marked against these files, and others that will not be disclosed in advance. Of course, you should also create your own small test cases with known good results. The output has to be the same regardless of the number of processors used. This doesn't mean that the *order* has to be the same, as long as all the correct line numbers are there in the right format. Your program must handle the case where the number of processors exceeds the nuber of lines in the file. You will not be given a zero length file.

Program specification

Your program will utilize Pilot processes in a master/worker pattern. It is up to you how you distribute the data to the workers, but you should aim to balance the load among them.

NOTE: Your program must be capable of running in serial fashion when it detects that only one Pilot process is available. In serial mode, it does not send messages to workers, but instead does all the work in PI_MAIN. 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. We expect that the 1-process serial version will run faster than the 2-process master+1 worker version because the latter will start to involve parallelization overhead.

The palindrome checking algorithm is completely up to you. It can be as "brute force" or as clever as you want. Note that the algorithm itself is serial; parallelism is obtained here by executing the same algorithm on multiple CPUs in classic "data parallel" fashion.

The program must be called **pal** , and take 1 command line argument:

sqsub ... **./pal** *datafile* [*Pilot options*]

- datafile is the path (absolute or relative) to the file of strings to be checked

Your program must accept a pathname, not just a bare filename, because the TA will have input files in another directory. For example, if "pal ../myfiles/mystrings" doesn't work because of the relative path and the TA is forced to work around the defect, that will result in a deduction of marks.

Note that when you call PI_Configure, the Pilot options (if any) get removed from argc/argv, after which you can check for the ones you expect to by typed. Print an error message and abort (call PI_Abort) if the filename is missing or cannot be opened for reading.

The findings of pal are printed by PI_MAIN on the file **pal.out** (in the current working directory, *not* in whatever directory the datafile came from). Be sure to follow the correct format so that your output can be easily verified against the correct answers. Programs that print to a different filename, different directory, or output in the wrong format will be penalized.

A rule is that only PI_MAIN is allowed to read/write files; the worker functions must strictly rely on messages and do no file I/O.

Finally, pal must report its total wallclock execution time by printing on stdout (which will end up in the output file designated with "sqsub -o"). Start timing by calling PI_StartTime right after PI_Configure. Obtain the wallclock time by calling PI_StopTime and print the elapsed time as seconds to 2 decimal places (e.g., "12.45 sec").

Suggestions for using Pilot

- Use the return value of PI_Configure to calculate how many worker processes can be created. Don't forget that one process is reserved for PI_MAIN. If the return value is one, you must run in serial mode. A two means that you're creating one worker. (Note that the return value of PI_Configure will be correct whether or not the Pilot deadlock detector is enabled; that is, your program does not have to know whether the deadlock detector is running.)
- Since you don't know in advance how many lines are in the input file, how are you going to divide it among the workers? Do you want to start farming it out while PI_MAIN is reading the file? If so, you need to organize a method of determining which processes are finished with their current allotment of data, and sending them more, after collecting their results.
- Do you want to read the whole file into memory initially? (the 10 million string file is around a gigabyte) Then you could give the workers all the data and simply wait for their results. But in this design, PI_MAIN may be sitting idle and wasting its processor, thus dragging down your performance graph. Can you give it something useful to do?
- Look for opportunities to use collective functions such as: PI_Broadcast, PI_Scatter, PI_Gather, PI_Select.
- Also remember the non-blocking calls: PI_ChannelHasData, PI_TrySelect.
- To make sure all the processes finish cleanly, send every worker a special value (known as a "sentinel") that essentially signals EOF. The workers can then return from their functions.
- Don't forget to use maximum Pilot error checking (-picheck=3) and deadlock detection (-pisvc=d) initially until you are confident of your program's logical correctness.

Performance measurement

We are interested in how the execution time varies as the number of processors increases. We hope it's going to decrease in a fairly linear fashion, since this is an "embarrassingly parallel" computation after all. Nonetheless, it is still subject to Amdahl's Law, and won't decrease indefinitely.

We're also interested in how much influence **compiler optimization** has. Sometimes it can have quite a startling effect, particularly if it can vectorize loops.

To present the results, draw three graphs based on the **strings-10M** data, being careful to label your axes. On each graph, draw 2 curves, one with -O0 (=no) optimization and the other with -O2. The graphs:

- Total execution time vs. number of Pilot processes.
- Speedup (=serial time/parallel time) vs. number of Pilot processes. Serial time is for one process. **Also draw the linear speedup line on this graph** to help you compare with the speedup you obtained.
- Efficiency (=speedup/no. of processes) vs. no. of Pilot processes. Ideal efficiency would stay at 1.0 as processes increase, but this is not likely.

The 2nd and 3rd graphs are just different ways of presenting the 1st graph's numbers. The calculations are easy in a spreadsheet, but be sure to use the right formulas.

For the above graphs' X axis, do timings for 1-8 processors, and then 12, 16, and 20. You may go higher if you wish. This means that you are plotting a minimum of 11 points, where the first point is serial execution, and the second point (master + one worker) will be even worse than serial.

To make sure the timings are fairly consistent, and aren't being unduly disturbed by other mako users, do some of the runs at least a few times and verify that the timing variance is small.

> **NOTE:** The Intel C/C++ compiler, unlike gcc, defaults to high optimization -O2. (You can also try other, more complex optimization options.) To compare the impact of optimization, you have to turn it off explicitly with -O0.

CAUTION: Mako may have non-SOCS users who are directly running programs on some processors without going through the job scheduler (sqsub). This is the nature of an experimental cluster, and it could impact your run times if you are unlucky enough to get scheduled onto such nodes. This is one reason you want to take multiple timings and check their variance.

Don't underestimate how long it will take to do these timing runs, record the measurements, and draw the graphs! Your assignment isn't finished just because your program compiles and runs correctly.

Part 1 Deliverables (due Oct. 6 in class)

The purpose of Part 1 is to get people working with Pilot and the mako environment, which is a big part of the learning curve. You simply need to code a bare skeleton of your pal application containing Pilot configuration code, and some printf's in worker functions to prove that they can successfully receive and send channel messages from and to PI_MAIN. Your output for Part 1 is a paper submission consisting of:

1. A process/channel diagram showing the architecture of your Pilot program (can be hand drawn). There are some crude process/channel diagrams in the Pilot tutorial. There are no strict conventions for these: just use circles for processes, arrows for channels, and you can draw a "lasso" around a group of channels to show a bundle.

2. A printed log file of a *short* sample run, demonstrating that your skeleton program runs, and that the channels shown in your diagram have been exercised. **Mark up the printout** so that the TA can easily understand what it is showing. Don't expect the TA to decode it for you.

Do not submit source code for Part 1. *Not* handing in Part 1 on time, or a lame submission, will result in loss of functional marks (see below).

Part 2 Deliverables (due Oct. 14 source code & 15 report)

- Source code via electronic submission using Subversion:

    1. Pilot program "pal" in .c source file(s). 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. Makefile that compiles pal on mako using mpicc when make is executed. You should use the Makefile for the Pilot labs as a model. You can introduce compiler flags by adding a line like **CFLAGS += -O0 -Wall** . Have a **clean** target in your makefile!

- Documentation, printed and delivered to class the next day:

  1. Write your impressions of working with the Pilot library. We're looking for feedback! What did you like/dislike? Was there anything you found hard to understand or to use properly? In your opinion, would Pilot be easy for a novice scientific programmer to learn? Why or why not?

  2. Describe your application's parallel architecture and work distributions scheme. See the part above "Suggestions for using Pilot." Either explain how you implemented these suggestions, or explain the approach you adopted.

  3. Print your three graphs and write some observations about them. **Given the speedup and efficiency results, how scalable was your parallel architecture?**

  4. Did compiler optimization make a difference? Did it affect both execution time and speedup, or just execution time? (That is, did it especially help the parallel version, or did it equally help the serial version?)

Marks for Assignment #1

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, makefile problematic, program name wrong, argument 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.

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