Computer Science 335
Parallel Processing and HPC
Siena College
Fall 2024

Programming Project 6: Pthreads Programming

Due: 4:00 PM, Friday, November 15, 2024

In this programming project, you will use POSIX threads to parallelize two programs: one is a program to find palindromic words in a large dictionary and the other is the very familiar Jacobi iteration.

You may work alone or with a partner or two on this programming project. However, in order to make sure you learn the material and are well-prepared for the exams, those who work in a group should either collaborate closely while completing the programs or work through the them individually then discuss them within your group to agree on a solution. In particular, the "you do these and I'll do these" approach is sure to leave you unprepared for upcoming tasks and the exams.

There is a significant amount of work to be done here. It will be difficult if not impossible to complete the assignment if you wait until the last minute. Expect to ask a lot of questions. A slow and steady approach will be much more effective.

Learning goals:

1. To gain experience working with POSIX threads

Getting Set Up

In Canvas, you will find a link to follow to set up your GitHub repository, which will be named `pthreadprogs-proj-yourgitname`, for this programming project. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

All GitHub repositories must be created with all group members having write access and all group member names specified in the `README.md` file by 4:00 PM, Wednesday, November 6, 2024. This applies to those who choose to work alone as well!

You may choose to answer the lab questions in the `README.md` file in the top-level directory of your repository, or upload a document with your responses to your repository, or add a link to a shared document containing your responses to the `README.md` file.

Palindromic Word Finder

Please complete this part in the `pal` directory of your repository.

There is a dictionary file `/usr/share/dict/words` on noreaster that can be used by programs that need to do spell checking. You know that a *palindrome* is a word or phrase that reads the same in either direction, *i.e.*, if you reverse all the letters you get the same word or phrase. A word is *palindromic* if its reverse is also in the dictionary. All palindromes are trivially palindromic, like "noon" for example, because it is own reverse. A word like "draw" is palindromic because "ward" is also in the dictionary.

Your task is write a C program to find all palindromic words in the dictionary. Include a `Makefile` that builds your program. You should first write a sequential program and then parallelize it, using the bag-of-tasks paradigm. However, keep your plan for parallelization in mind when designing and implementing the sequential version. You might want to do some things in the sequential program that you will need in the parallel program, such as finding where each letter begins in the dictionary (see below for more on this).

Your sequential program should have the following phases:

- Read the file `/usr/share/dict/words` into an array of strings. You can determine the size of this array ahead of time by running `wc` on the dictionary to see how many words it contains. You may assume that each word is at most 28 characters long when running on FreeBSD and 45 characters long when running on Linux.
  Aside: I figured this out with the command

  ```
  awk ' { if (length($1) > max) max = length($0) } \
    END { print max } ' words
  ```

  and the longest word (on Linux) is "pneumonoultramicroscopicsilicovolcanoconiosis".
- For each word, compute its reverse and do a linear search of the dictionary to see if the reverse of the word is in the dictionary. If so, mark the word and increment your counter of the total number of palindromic words. To improve efficiency, keep track of the positions in your array that store the first word beginning with each letter of the alphabet. Then you can search only within that letter's range when searching for a word's reverse.
- Print the total number of palindromic words to the screen (`stdout`).

Words in the dictionary could start with numbers or punctuation; you can either skip over them or process them, as you wish. None are of these are palindromic in FreeBSD's `words` file, so this choice will not affect your total count. (**This is no longer true with `linux.words`, which now contains "2", but you may still ignore it or not, as you see fit.** Some words start with capital letters (and hence the dictionary is not sorted in ASCII order). To keep your program simple, leave the capitals alone and do case-sensitive comparisons.

The sequential program should be fairly straightforward, but it will likely take longer than you expect to implement. Please ask if you run into trouble with the sequential version! You will need to get it done soon enough to leave enough time left to work on the parallelization – which is the whole point, of course.

Question 1: Give the GitHub link from the last commit that completed your sequential implementation.

After you have a working sequential program, modify it to use the bag-of-tasks paradigm, implemented using the *pthreads* library. Your parallel program should use W worker threads, where W

is a command-line argument. Use the workers just for the compute phase; do the input and output phases sequentially. Each worker should count the number of palindromic words that it finds. Sum these W values during the output phase. This avoids a critical section during the compute phase – you'll need to deal with others, though. Use 26 tasks in your program, one for each letter of the alphabet. In particular, the first task is to examine all words that begin with "a" (and numbers), the second task is to examine all words that begin with "b", and so on. During the input phase you should build an efficient representation for the bag of tasks; I suggest using an array, where the value in `task[0]` is the index of the first "a" word, `task[1]` is the index of the first "b" word, and so on. You can also use this array during the search phase to limit the scope of your linear searches. Your parallel program should also time the compute phase. You may use the `timer.c` and `timer.h` code from our class examples. Start the clock just before you create the workers; read it again as soon as they have finished. Write the elapsed time for the compute phase to `stdout`.

To summarize, your program should have the following output:

- total number of palindromic words (to *stdout*)
- the number found by each worker (to *stdout*)
- the elapsed time for the compute phase (to *stdout*)

Question 2: Execute your parallel program on noreaster using 1, 2, 4, 8, 16, and 26 workers. Run each test 3 times. Create a table of results; it should contain all the values written to standard output (but not the words themselves) for all 18 test runs, and a brief analysis of the speedup and parallel efficiency you have achieved. (10 points)

Question 3: Repeat your experiment on a Stampede3 node. Please see the notes below about this. (5 points)

Note that the dictionary file for Linux is different, and not installed by default on Stampede3 nodes. You should transfer a copy of `/usr/share/dict/linux.words` from another Linux machine such as `olsen.cs.siena.edu` (use `sftp` from a Stampede3 login node back here to olsen to keep this simple). Place it right in your directory with your code and executable, but since the file is large, *please do not have it tracked by git.* I will obtain my own copy for testing when I grade your submissions. This file has more words and includes longer words, so be sure to make any needed code changes before running there. You must follow the guidelines about where to compile (login nodes) and where you can run your tests (an interactive node allocated with `idev` or a batch node allocated through Slurm).

Your `pal` directory should include your `Makefile`, your C source code, and a note in `README.md` file expaining how to run your program. Please do *not* include object files or your executable in the repository.

Bonus Opportunity

Note: this bonus opportunity can only be achieved once a working version of the program satisfying the requirements has been completed. Once you have done that, create a new version in the `bonus` directory of your repository for your bonus version.

Some have expressed concerns that the requirements above do not allow for the most efficient possible solutions. This is a good point – high-performance computing is not just about parallelism, but about getting the solution as quickly as possible taking all possible mechanisms to do so into account. So let's see what we can come up with. A bonus of up to 10 points will be awarded for bonus submissions that solve the problem significantly faster. The fastest submission will earn the full 10 points, second-fastest 7 points, and all others judged to achieve a significant speedup over the original version will earn 5 points.

In addition to your program, you must include timing results and a some text describing what enhancements you made and how they improve the overall efficiency. Include these in the `README.md` file in the `bonus` directory

---

Multithreaded Jacobi Iteration

It's back again! In the `jacobi` directory of your repository, write a version of the Jacobi iteration program that uses POSIX threads for parallelization. You may use your own serial implementation as a starting point, or use my sample solution (available on request).

The functionality should be the same as our previous versions, and your solution output should be identical to the serial version and to your MPI version.

Requirements and suggestions:

- Your program should take an extra first command-line parameter, which specifies the number of threads to create. The usage message from my version now looks like this:

  ```
  Usage: ./jacobi num_threads size max_iter tolerance [outfile]
  ```

- As with previous versions, the optional "`outfile`" parameter determinies how the solution data is output: only the summary line if the parameter is omitted, a human-readable matrix of solution values in the parameter is specified as `-`, and for any other string a gnuplot-plottable output file with that string as the filename.
- You may create a new set of computational threads for each iteration pair and wait for them to finish, or create your threads once and have them perform the entire computational loop.
- It might not be a great practice in general to have a lot of global variables, but it is a very convenient way to share data among all of your threads. Take advantage of this to keep your code as simple as possible, especially in sharing the grids and other simulation parameters that will be needed within the threads.
- You will need to be careful about using barriers and mutexes where necessary, but avoid extraneous use of these constructs, which can lead to performance degradation.
- Take advantage of the fact that each row should cost about the same to compute and divide up the rows among threads like we did among the processes for the MPI version. Good news: no messages need to be exchanged among threads, but make sure you don't start using values computed by other threads before they've been computed!

- You are not required to parallelize the grid initialization step but you may do so if you wish.
- Don't forget to destroy any barriers or mutexes your code uses.
- All solution output can be done by the original thread that executes `main` after the computational threads have completed their work.

Question 4: Find a problem size and a number of iterations that runs for between 2 and 5 minutes using a single thread on noreaster. Run that same problem instance with 2, 4, 8, 16, and 32 threads and time each. Present a table of run times, speedups, and efficiencies. (10 points)

Submission

Commit and push!

Grading

This assignment will be graded out of 100 points.

Feature	Value	Score
Palindromic `Makefile`	1	
Palindromic sequential correctness	12	
Palindromic Pthread correctness	15	
Palindromic code style/documentation	5	
Question 2: Palindromic timing studies	10	
Question 3: Palindromic timing studies	5	
Bonus Version (up to 10 pts)	0	
Jacobi `Makefile`	1	
Jacobi command-line parameters	2	
Jacobi grid allocation	5	
Multithreaded Jacobi iteration	15	
Jacobi correct and efficient iteration limit stop	4	
Jacobi correct and efficient error tolerance stop	4	
Jacobi print simulation stats	2	
Jacobi print/write solutions	4	
Jacobi style/documentation	5	
Question 4: Jacobi timing studies	10	
Total	100	