

Computer Science 335
Parallel Processing and HPC
Siena University
Fall 2026

Programming Project 6: Pthreads Programming

Due: 4:00 PM, Wednesday, November 18, 2026

In this programming project, you will use POSIX threads to parallelize two programs: one is a program to find palindromic strings 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 the starter GitHub repository for this programming project, and you should fork that repository into your own GitHub account and make sure it is shared with your instructor. Only one member of the group should fork 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, Friday, November 6, 2026. 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 word list file `/usr/share/dict/words` on noreaster that can be used by programs that need to perform lookups or spell checking. You know that a *palindrome* is a string that reads the same in either direction, *i.e.*, if you reverse all the characters you get the exact same sequence. A string is *palindromic* if its reverse is also present in the list. All palindromes are trivially palindromic, like "noon" or "1001", because each is its own reverse. A string like "draw" is palindromic because "ward" is also in the file.

Your task is to write a C program to find all palindromic strings in a given set of strings. 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 include structures in the sequential program that you will need in the parallel version, such as locating where each starting character group begins in the dataset array.

Your sequential program should have the following phases:

1. Read the file (name provided in the first command-line parameter, such as `/usr/share/dict/words`) line-by-line into an array of dynamically allocated string pointers (`char **words`). Valid input files must be sorted ASCIIbetically. You may determine the total number of lines ahead of time (e.g., using `wc -l` and pass it as an additional command-line parameter, or by a preliminary scanning pass within your program. This allows you to dynamically allocate the pointer array using `malloc`. Each string must be allocated dynamically using functions such as `getline()` or `strdup()`.
2. For each string, compute its reverse and perform a search within the array to determine if its reverse exists. Treat all strings as general sequences of ASCII characters (which may include lowercase letters, uppercase letters, digits, and punctuation marks). Comparisons must remain strictly case-sensitive and exact. To improve search efficiency, build an index tracking the starting positions in your array for every distinct leading ASCII character present in the file. Use this index to constrain your search strictly within the range of elements sharing that starting character.
3. Print the total number of palindromic strings found to standard output (`stdout`).

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 the second command-line parameter. Use the workers only for the compute phase; perform input and output phases sequentially. Each worker should count the number of palindromic strings that it finds. Sum these W values during the output phase to avoid critical sections during calculation.

The task bag must be partitioned by starting ASCII character. Each distinct starting ASCII character present in the dataset (or every valid printable ASCII character range) represents one discrete task. For instance, one task consists of processing all strings starting with `'a'`, another for strings

starting with `'b'`, another for digits starting with `'0'`, and so forth. Note that there is no guarantee that there will be a string that startes with every printable ASCII character, so your program must be able to handle this.

During the input phase, build an efficient representation for your bag of tasks (such as an array mapping character values to their starting index and element count in the string array). Your parallel program must also time the compute phase. Start the clock immediately before creating worker threads and read it again as soon as all threads have been joined. Write the elapsed time for the compute phase to `stdout`.

To summarize, your program should produce the following output on `stdout`:

1. Total number of palindromic strings found
2. The count of palindromic strings found by each individual worker
3. The total elapsed time for the compute phase

Question 2: Summarize how you parallelized your program, noting the pthreads functions you used and why (you may omit create and join), how you managed the bag of tasks, and how your code ensures that there are no race conditions. (5 points)

Question 3: Execute your parallel program on noreaster using 1, 2, 4, 8, 16, and 32 (or more) workers. Run each test 3 times. Create a table of results containing all output values for all runs, alongside a brief analysis of the speedup and parallel efficiency achieved. (10 points)

Question 4: Repeat your experiment on a Stampede3 node. (5 points)

Your `pal` directory should include your `Makefile`, your C source code, and a `README.md` file explaining how to build and execute your program. Please do *not* include object files or binaries in your repository.

Bonus Opportunity

Note: this bonus opportunity can only be achieved once a working version of the program satisfying all base requirements has been completed. Once completed, place your optimized implementation inside the `bonus` directory of your repository.

A bonus of up to 10 points will be awarded for submissions that solve the problem significantly faster by applying advanced algorithmic or high-performance optimization techniques (such as replacing range linear searches with lock-free hash indexes or dynamic chunk-based work stealing).

Include your benchmark timing results and a short technical summary in `bonus/README.md` detailing the optimizations applied and their measured performance impact.

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 5: Summarize how you parallelized your program, noting the pthreads functions you used and why (you may omit create and join), how you managed partitioning of the work, and how your code ensures that there are no race conditions. (5 points)

Question 6: 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 110 points.

Feature	Value	Score
Palindromic `Makefile`	1	
Palindromic sequential correctness	12	
Palindromic Pthread correctness	15	
Palindromic code style/documentation	5	
Question 2: Palindromic parallelism summary	5	
Question 3: Palindromic timing studies	10	
Question 4: 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 5: Jacobi parallelism summary	5	
Question 6: Jacobi timing studies	10	
Total	110	