MA/CSSE Homework 5
Due 4/21

Directions

- **Each problem must be self-contained in a file**, named properly, and **must compile using the command** `mpicc filename.c`
- Turn in each `.c` file to the dropbox. **Do not create a `.zip` file**
- On this homework you may not use any of the standard default MPI collective operations.

Problems

Level: Easy

1. *Goldbach's Conjecture* is one of the most famous unsolved problems in mathematics. First stated in 1742 in a letter from Christian Goldbach to Leonhard Euler, Goldbach's conjecture claims that every even integer greater than two may be written as the sum of two prime numbers. For example, 6 may be written as $3 + 3$, 8 as $5 + 3$, 10 as $5 + 5$, and so on. No one has been able to prove that *every* even integer greater than two is indeed expressible as a sum of two primes, however, no one has ever found a counterexample either.

   Two prime numbers, p and q that sum to a given integer n form a *Goldbach partition* of n. So, 3 and 5 form a Goldbach partition of 8, and 11 and 13 form a Goldbach partition of 24. Goldbach partitions are not unique, for example 3 and 7 form a Goldbach partition of 10, as do 5 and 5.

   Write a program in a file `goldbach.c` that calculates Goldbach partitions for lists of numbers and prints the partitions. Use a worker pool (dynamic work allocation) strategy to do so. The standard code is supplied in `objs/goldbach_standard.o` and may be compiled with
   `mpicc objs/golbach_standard.o -o goldbach_standard`
   You need to have a working copy of `goldbach_standard` in order for the testing code to run.

   The file `goldbach_start.c` contains some starting code to get you going. In particular, reading command line options and determining which numbers need to be checked is taken care of for you. Once you call `parse_options`, the list of numbers that need to be checked is stored in the `to_partition` member of the `options` struct. Run the standard code with `-h` to see a full list of options for specifying which numbers to partition. Just by using `parse_options` you will automatically support most of the options provided.

   Here are the requirements that must be met for this problem:

   - Partitions must be formed using the `goldbach_partition` function that is supplied.
   - Every single number in the `to_partition` list must be passed to `goldbach_partition`, even if it is a negative or odd number.
   - Partitions must be displayed using the format `n:  a/b` where `n` is the integer being partitioned, `a` is the smallest number in the partition, and `b` is the largest number in the partition. For example, the line for $n = 8$ should read `8:3/5`. Partitions **may be printed by the workers** – there is no need to send data back to the master processor.
   - `goldbach_partition` will return 2 if n is odd or ≤ 2. Do not print result line if the result of `goldbach_partition` is 2. However, if the result is 3, print out a result line and notify me immediately!

- Follow a worker-pool strategy for balancing the work. The root processor should never call `goldbach_partition`. The worker processes should all spend a similar amount of time running (within 10%).
- Respect the `--chunksize` option. The `chunksize` is the number of integers to assign to a processor as part of a given assignment.
- Respect the `--print_results` option.
- In order to check to see if the load balancing is good, your code is run using 2,3,5,9,17,33 processors. Your code should be nearly perfectly parallel when the number of integers to check is large (average efficiency of $\geq 60\%$). Notice that you'll need to run the tests on `grendel` to get good efficiency with many processes.

  In order to facilitate your testing, you can have the testing script not do a full efficiency test for you. The testing script takes an option `--max-procs=n` that tells the script not to test efficiency using more than n processes. For example,
  `./test_goldbach.py --max-procs=4` tells the testing script not to test efficiency using more than 4 processes. Setting `--max-procs=0` skips these tests altogther.
- We might want to partition some big numbers, so make sure you can support partitioning `long long int` types.

Level: Difficult

Complete the problem above. In addition:

1. Many numbers have many different Goldbach partitions. For example, 500 can be partitioned as:

$$13 + 487$$
$$37 + 463$$
$$43 + 457$$
$$61 + 439$$
$$67 + 433$$
$$79 + 421$$
$$103 + 397$$
$$127 + 373$$
$$151 + 349$$
$$163 + 337$$
$$193 + 307$$
$$223 + 277$$
$$229 + 271.$$

For an even integer greater than 2, n, let's call the smallest number appearing in any partition of n the *Goldbach minimum* (that is not a standard name) of n, and denote it as $\mathcal{M}(n)$. We can see from above that $\mathcal{M}(500) = 13$. The function `goldbach_partition` has the special property that it returns $\mathcal{M}(n)$ in the variable `a`.

It is interesting that $\mathcal{M}(n)$ is usually a pretty small number. Indeed, from our first program we can tell that $\mathcal{M}(4) = 2$, $\mathcal{M}(6) = 3$, $\ldots$, $\mathcal{M}(98) = 19$, $\mathcal{M}(100) = 3$.

Define $\mathcal{N}(p)$ to be the smallest even integer n such that $\mathcal{M}(n) \geq p$. For instance, $\mathcal{N}(2) = 4$, $\mathcal{N}(3) = 6$, $\mathcal{N}(5) = 18$,$\mathcal{N}(7) = 30$ and $\mathcal{N}(19) = 98$.

Write a program using a worker-pool approach that prints out $\mathcal{N}(p)$ and the corresponding Goldbach partition of $\mathcal{N}(p)$ for a given value of p.

For example, when $p = 2$ your program should print
`N(2)=4:  2/2`
and when $p = 7$ your program should print
`N(7)=30:  7/23`.

In addition to submitting `goldbach_plus.c`, also turn in a file `big_number.txt` that includes the output of your program when run with `--p=2000`.

I've included a file `goldbach_plus_helpers.c` to that reads the command line options for you. You need not respect `--chunksize` though it might be useful for you to do so.

I've included `goldbach_plus_start.c` to start from; leave `goldbach_partition` there and use it in your code.

I've included the standard code in `objs`, you can compile as you did in the first problem. Again, you a working copy of the standard in order for the testing code to work.

The important option to respect is `--p`, which is the value of p for which we are calculating $\mathcal{N}(p)$.

Here are the grading criteria for this problem:

- Your program must always return the correct result, your format must match mine exactly.
- Every calculation of $\mathcal{M}(n)$ must be done using `goldbach_partition`, which is provided.
- Follow a worker-pool strategy for balancing the work. The root processor should never call `goldbach_partition`. The worker processes should all spend a similar amount of time running (within 10%).

- In order to check to see if the load balancing is good, your code is run using 2,3,5,9,17,33 processors. Your code should be nearly perfectly parallel when the number of integers to check is large (average efficiency of $\geq 60\%$). Notice that you'll need to run the tests on `grendel` to get good efficiency with 3,5,9 processes. As in the last problem, you can use a `--max-procs` option to avoid this test, or to reduce the number of processors it uses.
- There is no ambiguity about what list you need to partition here: it is just $1, 2, 3, 4, \ldots$. So your work assignments absolutely should not be lists of numbers to check, your work assignments should just be a range (top and bottom) of numbers to check.
- You must respect the `--p` option.