Objective: Speedup

Prime Number Sieve

In this project you write a parallel version of the classic *Sieve of Eratosthenes* for computing prime numbers. The program is highly efficient and can achieve very large speedups.

The first step is to consider a sequential program for the Sieve of Eratosthenes. To compute the prime numbers from 1 to \( n \), the basic data structure is a simple Boolean array, named \( \text{Prime} \), containing \( n \) elements. The program initializes all the array elements to true. The remainder of the program gradually changes array elements from true to false. At the end, all remaining array elements that are still true will indicate prime numbers. The program is shown below:

```c
int main (void)
{ int Prime[n+1];
  int i, num, loc;

  for (i = 1; i <= n; i++)
    Prime[i] = TRUE;
  for (num = 2; num <= (int)sqrt((double)n); num++)
    if (Prime[num])
    { loc = num + num;
      for (loc = num + num; loc < n; loc += num)
        Prime[loc] = FALSE;
    } /* if */
  return 0;
} /* main */
```

The program has a for loop that scans the array from 2 up to the square root of \( n \). Each element that still has value true when the scan reaches it must be a prime. The first such prime identified is the number 2. For each such prime identified (named \( \text{num} \) in the program), an inner for loop changes all multiples of that prime to false in the array. The variable \( \text{loc} \) is used to step through the array in increments of size \( \text{num} \). This process eliminates all nonprime numbers in the array. Any remaining array elements that are true at the end of the program are prime numbers.
Your job is to write a parallel version of this Sieve of Eratosthenes program. The program is parallelized by partitioning the array Prime into equal-sized portions, and creating a parallel process to work on each portion. Make the partitions large enough so that all the elements up to the square root of $n$ are contained in the first portion. The process assigned to the first portion is almost identical to the sequential program given above. The main difference is that the for loop stops when it reaches the end of the first portion of the array. The other portions are handled by their own processes.

All processes begin by initializing their own portion to true. This can all be done in parallel. Then the first process begins to search for the first true value, which, of course, is 2. This number 2 is then broadcast to all the other processes. As each process receives this number, it begins to step through its own portion in jumps of size 2, thereby changing all the even numbers to false. Then the first process loops around again to search for the next true value, and again broadcasts this to the other processes, where it is used to step and change true to false. In this parallel version, there is a separate Broadcasting each “step” number to all the processes can be made more efficient by having the first process send it to the second process only. The second process sends it to the third, and the third to the fourth, and so on. By passing the “step” number through a process pipeline in this way, the overall execution time is reduced.

Another important issue is how each process determines the starting point for stepping after it receives a given “step” number $num$. It takes steps of size $num$, but where should it start? The answer is at the first number in its portion of the array that is evenly divisible by $num$. Assuming that the first element of the array in this portion is $Prime[first]$, then the following technique locates the starting point:

```plaintext
remainder = first % num;
if (remainder == 0)
    starting point is first;
else
    starting point is (first / num + 1) * num;
```

Write this parallel version of the Sieve program and run it, using the Rex system. Test the program for correctness, and then experiment with different size arrays to see how much speedup can be achieved. You should see that very large speedups can be achieved for big arrays. You can get larger speedups and use larger arrays if you use an array of bits instead of an array of integers.

**Notes:**
- Make sure your program conforms to the Programming Style document on the course home page.
• Write a report that shows your results in a way that will impress the grader. Your report must be in one of the following formats: .doc, .docx, .pdf, .odt.
• Submit your finished program to the Eagle system, including your makefile, as a compressed tar file.