Refining the Sieve of Eratosthenes

Demo Student

Objective

The purpose of the objective is to define the objective of the work being accomplished.

Example: The objective of this assignment was to have the students learn how to apply timing improvements to a program by identifying key code in the program to be refined and modified based on the computer architecture and techniques for parallelism.

Methodology

This section defines the methodology being used as a part of the lab. It should contain pertinent facts such as languages, machines, etc.

Example: The program being implemented and refined was the Sieve of Eratosthenes – a method for identifying prime numbers invented by the Greek mathematician Eratosthenes. This was accomplished by marking all the multiples of the primes up to √N in each processors address space - anything unmarked when finished is considered a prime number. Once the initial program is implemented as a parallel program, three techniques are applied. The first is to have each of the processors implement the discovery of the primes less than √N by themselves. The second is to identify all the prime multiples within a block-size before moving on the next block. The last is to only mark the odd primes since all the even numbers greater than 2 are multiples of 2.

The MPI tool chain was used for parallelization and communication between the processes. The initial implementation and each of the improvements was run 10 times on the CoGrid machine with the results being averaged and the outliers removed. These results are then plotted using gnuplot and the corresponding functions identified using the fit function.

Hypothesis

What hypothesis did you approach the problem with?

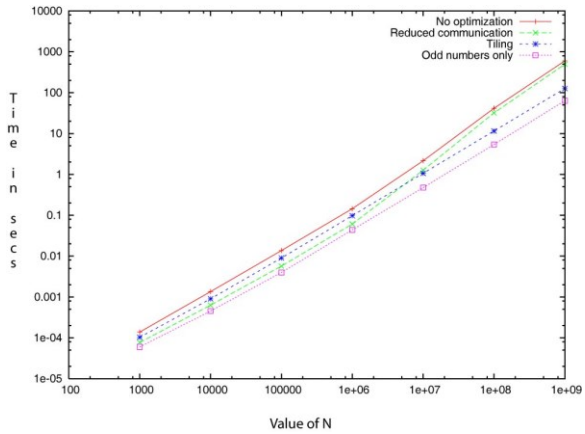
Example: The hypothesis is that the computation time necessary to perform the calculation of the primes will increase linearly as N increases. In addition, that applying each of the changes to the computation defined in the methodology will provide a quantifiable increase to the timing of the program based on the mapping of the computations to the architecture and to the parallelization as the number of processors increases.

Summary of Results

The results showed the hypothesis that the execution time necessary increased on a linear basis as N increased was correct and having the computation done in a parallel manner decreased the computational time based on a definable equation. Each of the improvements provides a quantifiable decrease in execution time resulting in a computational time using the final version that is 13.3% of the time of the first version.

Discussion

Here you put the facts behind your conclusions.



Results of each of the improvements as N increases using logscale

Programming the model

The first program implemented the basic algorithm by increasing N to a multiple of P and assigning N/P memory locations to each of the processors. This was done instead of using a balanced assignment – I've added a discussion of this in a later section. Processor 0 was assigned the task of identifying the seed primes and communicating them to the other processors for marking. The processors then marked each of the multiples of the prime within their address space. Once all the primes below √N had their multiples marked, the sums were reduced to processor 0 to provide the total. Running the timing measurements with P = 1, graphing them, and using the fit function within gnuplot provided the function:

$$f(x) = 6.0\ e^{-7} * x$$

The first improvement assumes that the price of communicating each of the initial set of primes from processor 0 to the other processors is higher than the price of having each of the processors compute those primes themselves. As you can see from the above graph, this indeed is the case – eliminating the communication of each of the initial primes improved the timing with P=1 to:

$$f(x) = 4.9\ e^{-7} * x$$

The second improvement was to mark all the primes within a specific block size before moving on to the next block. The assumption is that keeping the block within cache allows the marking to happen much quicker. A separate action was taken to identify the block size on the CoGrid machine, this is discussed in the next section. The timing runs using 1 processor showed an initial price – but as N increased, the improvements were dramatic. Using the fit function, the resulting equation was

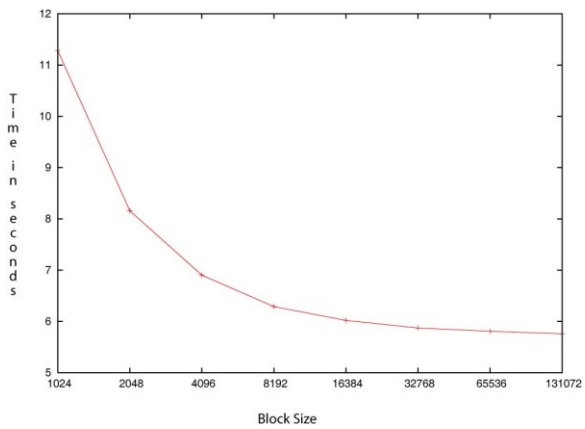
$$f(x) – 1.26e^{-7} * x$$

The last required improvement was to reduce the amount of memory each processor by half by only computing the odd primes. The assumption is that the bulk of the time of the algorithm is doing this marking, reducing this work by half should result in a reduction in time by a similar percentage. The resulting algorithm bears this out:

$$f(x) = .63e^{-7} * x$$

Computing the optimal block size

To find the optimal block size used in the third program, a separate action was taken by keeping P=2 and N = 100M. The block size was then alternated from 1K to 128K.

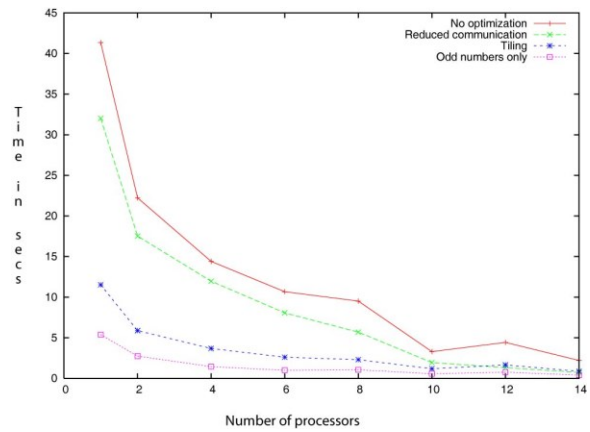


Finding the optimal block size (shown with logscale y 2)

I plotted the data using logscale y 2, because I chose this progression when I performed the testing. By examining the above graph, I chose a block size of 32K to use for the rest of the assignment.

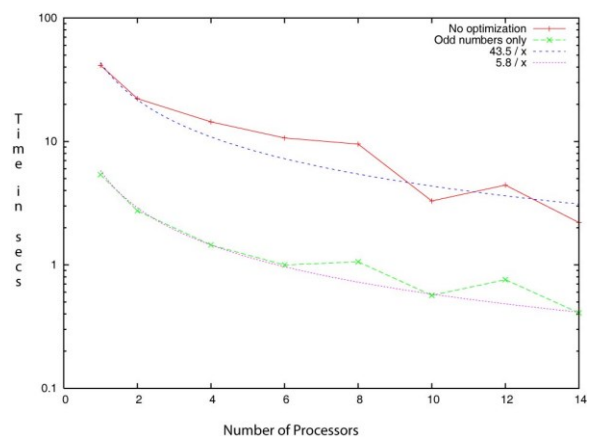
Increasing the number of processors

Once the algorithms had been implemented and tuned – a series of run was then made keeping a block size of 32K and N = 100M varying the number of processors. Once again – each run included each of the improvements and was made a total of 10 times. The resulting runs had their average taken, the outliers removed, and the average recomputed.



Varying number of processors from 1 to 14

This graph matches the assumptions identified initially – each improvement decreases the time required to complete the run – and the resulting graph shows the typical a/x+b equation. I broke out the initial version and the final version to perform the fit function; these are shown in the below graph.



Version 1 and 4 with N=100M

The two resulting equations showed:

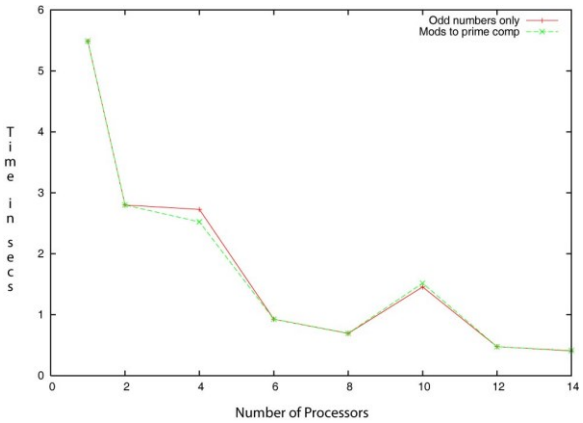
Version 1 – f(x) = 43.5 / x
Version 4 – f(x) = 5.8 / x

This represents an improvement of 7.5 times after applying the improvements.

Additional tweak

I made two additional tweaks to the program to try to improve the times – the first was to provide better load balancing by only allocating memory for the memory from √N to N – since the processors are computing the first √N primes as a part of the algorithm. The second tweak was to apply the same improvement to the computation of the primes that was applied in the fourth version – only compute the odd primes.

The assumption I made before making them was there should be a slight increase in the performance – but it wouldn't be much in the overall scheme of things. Both of these improvements only affect the computation of √N primes – a very small percentage of the overall computation.



Little change after final tweak

Using the comparison between version 4 and the additional improvements, I ran each of the computations from 1 to 14 processors keeping N at 100M. As you can see from the above graph – little, if any, improvement was realized.

Conclusion

Each of the improvements provided the expected improvements – the most dramatic being the implementation of the tiling. For smaller N and a block size of 32K, the results of the tiling were not seen until N became larger – but in the runs of N=100M, the improvement was greatly increased. Eliminating the communication provided a moderate improvement – even with replacing it with a set of work – and reducing the amount of work each processor had to do by half reduced the running time by half. Improvements to the √N computation provided little improvement and probably aren't worth the extra effort of programming unless the effort is small (as these were).

Overall conclusion – the improvements provided the desired improvements but each generated corner cases that made the programming more problematic.

Discussion of block data decomposition

After performing exercise 5.2 per the instructions in the homework assignment, we were to compare and contrast the two methods. Both methods compute N/P and assign these to each of the processors, the first method assigns the leftovers to the lower id number processors, the second scatters them out among the processors.

The major difference between the two is the ease of computation of the size and/or index into the address space of each processor. The first method has to identify how many lower id processors were assigned an additional space each time where the second just says the first element within the address space of each processor is ID * N / P. Since this computation would be done each and every time an element is accessed – savings in computation steps for the second method is favorable.