**Lab-4 Assignment**

**Vaibhav Patel - 201401222**

**Tanmay Patel - 201401409**

**Hardware details:**

Architecture: x86\_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

CPU(s): 12

On-line CPU(s) list: 0-11

Thread(s) per core: 1

Core(s) per socket: 6

Socket(s): 2

NUMA node(s): 2

Vendor ID: GenuineIntel

CPU family: 6

Model: 63

Model name: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz

Stepping: 2

CPU MHz: 1264.218

BogoMIPS: 4804.38

Virtualization: VT-x

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 15360K

NUMA node0 CPU(s): 0-5

NUMA node1 CPU(s): 6-11

**Question 1:**

**1. Context:**

**● Trivia about the serial and parallel implementation:**

**These details are for the case where number of processors are n:**

**For Serial:**

Span=n

Work O(n)

Cost O(n)

**For Parallel:**

Span=2logn

Work O(n)

Cost O(n\*2logn)

**Now,These details are for the case where number of processors are p<<n:**

**For Serial:**

Span=n

Work O(n)

Cost O(n)

**For Parallel:**

Span=2n/p + 2logn

Work O(n)

Cost O(span\*p)

**● Brief Description of the problem :** Compute the inclusive prefix sum (scan) from an input array; store results in output array for different lengths of N (size of the input array).

**● Complexity of the algorithm (serial) :** If N coordinates are generated randomly then the complexity is O(N).

**● Possible Speedup (theoretical) :**

Parallel time(Tp) : 2\*n/p + (logp)

​Possible speedup = n/(Tp)

If p<<n then possible speedup=p/2

**● Optimization strategy and Problems faced in parallelization and possible solutions :** At first we implemented the theoretical model of this span problem and because in practice we don’t have n processors, we made window of length p and moved it along the input array. This operation is very costly because of two main reasons:

**1:** First we should not try to imitate the theoretical model of this problem because we know that this problem is not cost efficient and therefore we will not get more speed up.

**2:**Another reason is that moving a window for only one element will convert into too much parallel overhead. Because we are accessing only one element even if we are loading whole cache line for a given processor. Other thing is that this will create cache coherency problems. Because if we change an element in processor 2’s cache line and even if will not accessed by another processor but if it is in its cache line then it has to be updated. This will make things slower.

Let’s say cache line is of 8 **ints** then in this algorithm it will load the cache line for n times. But in a simple serial algorithm it will load it only n/8 time. Now you can see that this will make the code drastically slow.

But in practice we just have only p elements, That’s why we decided to change the algorithm and use this algorithm when we were reduced to only p elements.

Now we divided the problem into p chunks of size n/p. And then we imitated the serial algorithm for these chunks. Now after that for the last element of each chunk we performed scan for that p elements and gave the output again to every chunk and performed scan again.

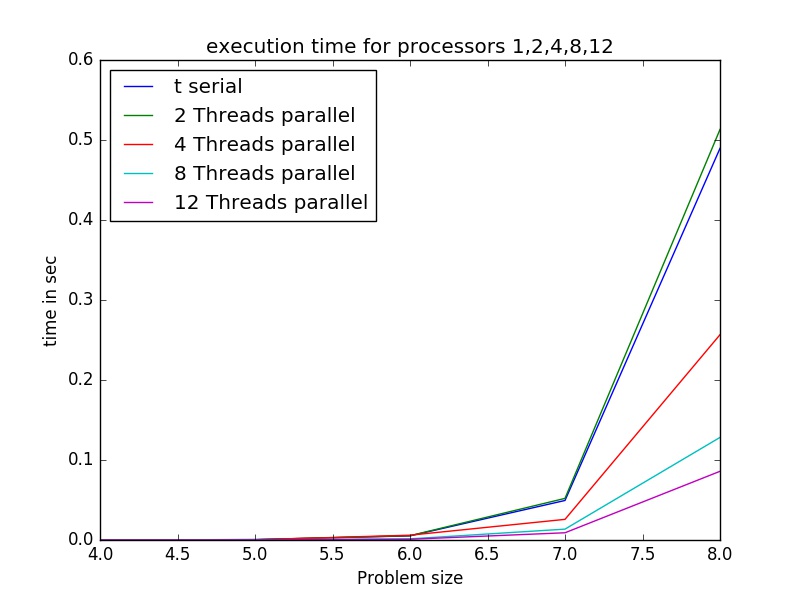
One another solution we are suggesting is that we just find reduction of this n/p chunks and then implement the scan for that part. Essentially it is the same but we don’t have to do scan again and again. So here the store operations are ~n as compared to the above one which has this store operations ~2\*n times.

**2. Input parameters and Output:** Input is an array of n elements. And output is a prefix array operated on that array.

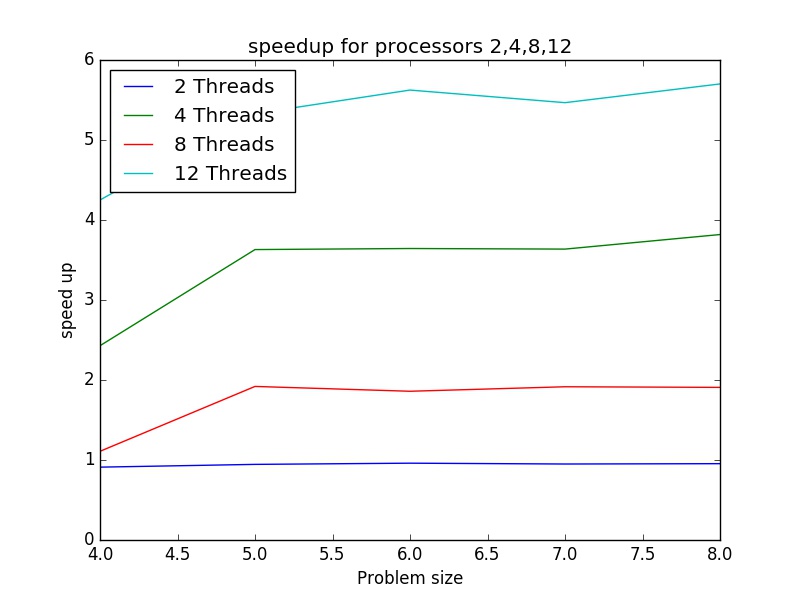
**3. Parallel overhead time:**

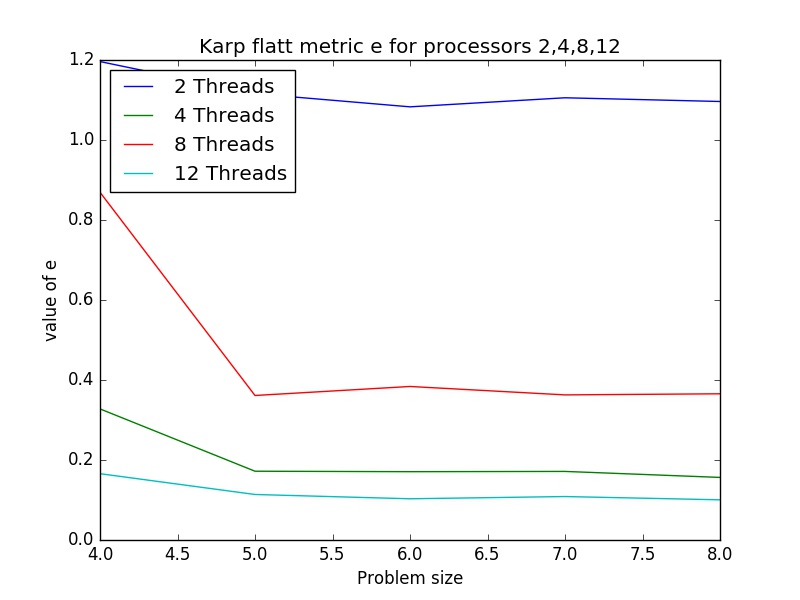
**0.0438sec**

**4 Problem size vs time :**

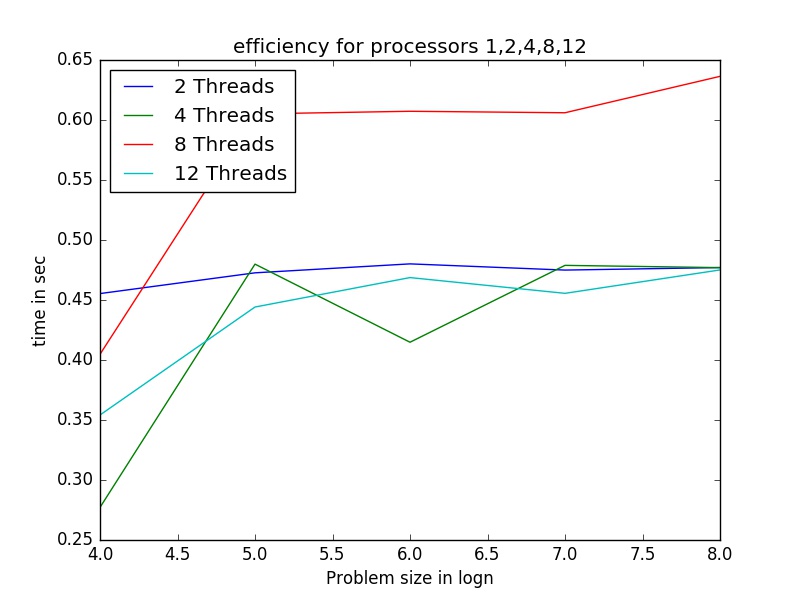


**5. Problem Size vs Speedup:**



**6. Problem size vs Value of Karp Flatt Metric e :**

**7. Problem size vs Efficiency:**



**Question 2:**

**1. Context:**

**● Trivia about the serial and parallel implementation:**

**This problem is actually an application of scan operation. So all the details (Trivia, Implementation notes, problem faced were same in this case too.)**

**These details are for the case where number of processors are p<<n:**

**For Serial:**

Span=n

Work O(n)

Cost O(n)

**For Parallel:**

Span=4n/p + 2logn

Work O(n)

Cost O(span\*p)

**● Brief Description of the problem :** **[**efficient parallel filter**]**

**(**do the following with both serial and parallel implementation using the above implementations):

Given an array **input**, produce an array **output** containing **only elements** such that

Example: **input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]**

**filter: is element > 10**

**output [17, 11, 13, 19, 24]**

Choose an Input with length (>10^6); fill the vector with random numbers (integer) between 0-100, and do the above operation.

Here we have to find the elements which are greater than a given number v.

**● Complexity of the algorithm (serial) :** If N coordinates are generated randomly then the complexity is O(N). Which is very trivial algorithm to make.

**● Possible Speedup (theoretical) :**

For making the boolean array n/p

Scan operation on it. 2\*n/p +logp

And then finding the elements from that n/p

Parallel time(Tp) : 4\*n/p + (logp)

​Possible speedup = n/(Tp)

If p<<n then possible speedup=p/4

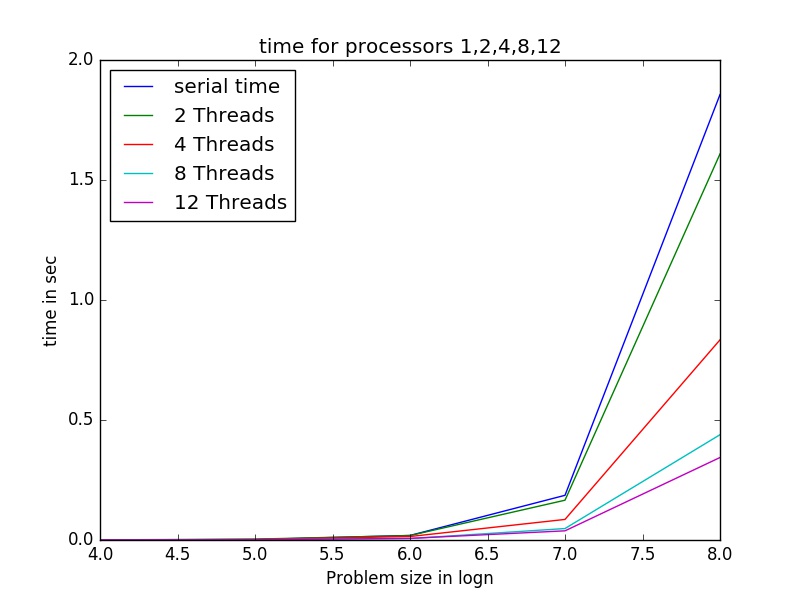
**● Optimization strategy and Problems faced in parallelization and possible solutions :** This part is same as the above question because the main bottleneck for the problem was the scan operation.

**2. Input parameters and Output:** Input is an array of n elements. And output is a different array of size s(number of elements greater than filter) containing those elements.

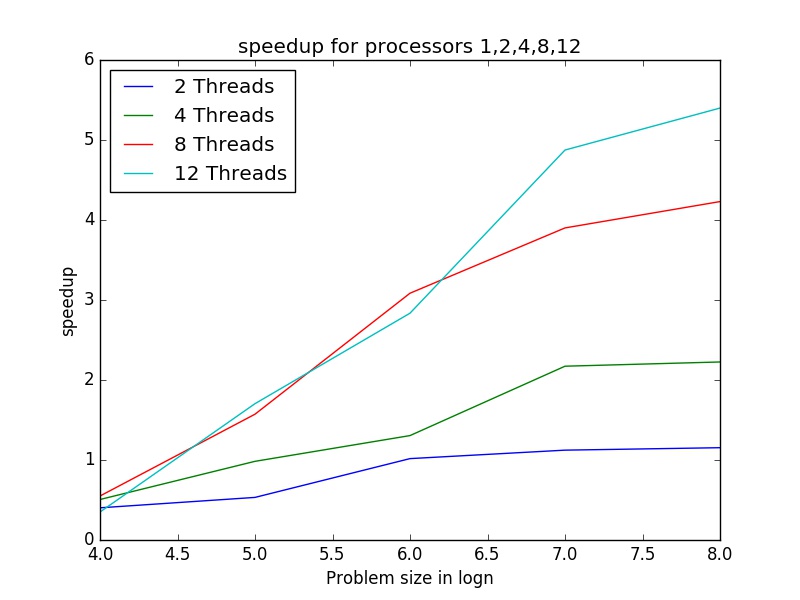
**3. Parallel overhead time:**

**0.0687 sec**

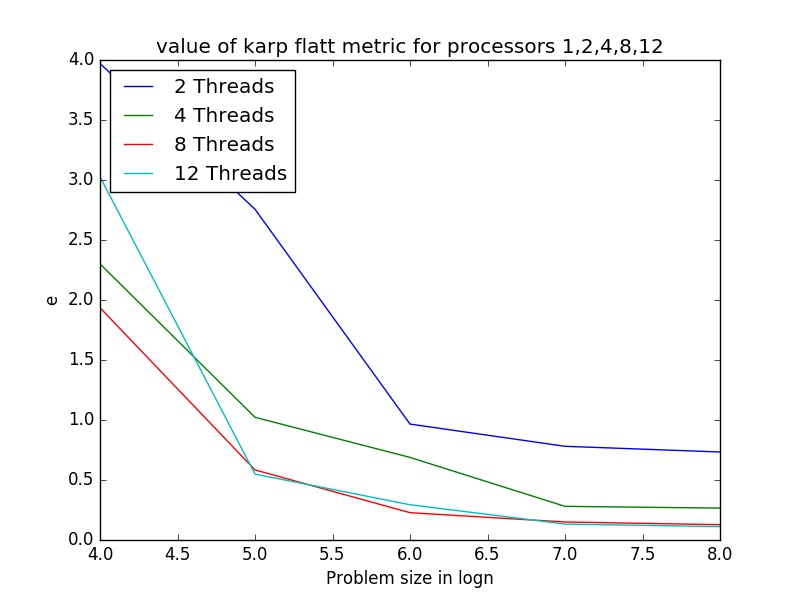
**4. Problem size vs time :**



**5. Problem Size vs Speedup:**



**6. Problem size vs Value of Karp Flatt Metric e :**



**7. Problem size vs Efficiency:**

