CS566 Spring ‘23

Homework 2

Parallel Calculation ofπ using a Hypercube Reduction Algorithm

Utsav Sharma
3-17-2023

Table of Contents

I. Pseudocode 2
II. Algorithm 3
III. Simulating a Hypercube 5
IV. Test Results 6
V. Analysis 7
a) Theoretical Analysis 7
b) Hypercube Algorithm 11
c) Hypercube Reduction vs MPI_Reduce 18
d) Nodes=1:PPN=P vs Nodes=P:PPN=1 26
VI. Conclusion 27

I. Pseudocode:

1) Initialize MPI environment.
2) Get number of processes and ID of processors.
3) If myid is 0:
    a. get input values for the case of nodes:ppn assignment and exponent from command line arguments.
    b. calculate n =2^exponent.
    c. Record first start time to calculate total runtime.
4) Broadcast n to all processors using MPI_Bcast().
5) If myid is 0:
    a. Record second start time after broadcast.
6) Calculate partial sum for pi:
    a. Calculate h = 1.0 / (double)n.
    b. Calculate sum as the summation of the individual terms using the formula given.
    c. Calculate mypi as sum * h.
7) Use the hypercube algorithm which implements MPI_Send and MPI_Recv to sum up partial results from all processors into the mypi at processor 0.
8) If myid is 0:
    a. Record end time.
    b. If number of processes > 1:
        i. Open output file in read mode.
        ii. Read lines from file and extract runtime for P=1, n=exponent and store in baserun variable.
        iii. Close file.
    c. Open file output file in append mode.
    d. Print runtime, runtime after broadcast, pi value, error, exponent, nodes and ppn to console and append to file.
    e. If number of processors > 1:
        i. calculate and print speedup and efficiency to console and append to file.
    f. Close file
9) Finalize MPI environment.

II. Algorithm

The program starts with defining all the variables that we will use. Once we have done that, we initialize MPI using *MPI_Init(&argc, &argv)*, and then get our values for the number of processors and the ID of each processor using *MPI_Comm_size(MPI_COMM_WORLD, &numprocs)* and *MPI_Comm_rank(MPI_COMM_WORLD, &myid)* respectively. We initialize *MPI_Status* and *MPI_Request* to use later in our send and receive calls. Lastly, since we are assuming a hypercube topology, we need the dimension of the hypercube to perform our communication. To do this, we simply take the $\log_2$ of the number of processors.
To broadcast our value of n to all the processors from processor 0, we need to fetch the value from the arguments of the program. For our result, we also need to print the number of nodes and the processors per node. Therefore, we first read which case we have used to assign nodes and processors, where a indicates 1 node with all processors on that node, b indicates that there are as many nodes as processors, and one processing unit per node, and c indicates that we have half as many nodes and 2 processors per node. Once we have the values for nodes and ppn set, we read the exponent given in the argument, and find the value of that exponent raised to 2, which is assigned to n. We also capture our start time here using *MPI_Wtime()*.
Once the above steps have been completed, we can perform our broadcast with *MPI_Bcast(&n, 1, MPI_LONG, 0, MPI_COMM_WORLD)* where &n indicates the value to be broadcast, 1 is the number of elements in the message, *MPI_LONG* is the datatype of the message, 0 is the source of the message, and *MPI_COMM_WORLD* is the destination of the message. All processors call *MPI_Barrier* to synchronize, before starting their part of the calculation. We also capture the time here since we need to output the runtime of the program after the Broadcast has been performed.
The calculation is perform using the formula pi= (1/n)* S i=1n 4/(1 +((i-0.5)/n)2), where n is the accuracy term. We use *MPI_Barrier* to synchronize again as a precaution, and then proceed to collect the result back at processor 0.

There are two programs in this homework, one that uses *MPI_Reduce*, and the other that implements our own logic to perform the reduction using *MPI_ISend* and *MPI_Recv*. The method for simulating the hypercube for our own implementation is described in the next section.

Once we have collected our result, we need to print it to the output file. This is done by processor 0, since it holds our final result. We start with capturing the end time using *MPI_Wtime()*. Then, we check if we have more than 1 processor because the output needs to be handled differently in that case. If *numprocs*=1, we assume that this is the first run for P=1, and there are no values that need to be calculated for speedup and efficiency, since there are our baseline runs.
If *numprocs* is greater than 1, we open our output file and read the lines in a while loop. We use the boost library to easily split our line based on a delimiter, and save the output to a vector. This means that our line has been split as:

Runtime=0.0158858299255371
Runtime after Broadcast=0.0158784389495850
pi=3.1415926535897891
Error=0.0000000000000040
n=20
nodes=1
ppn=1

Taking the segment at index 4, i.e., the value of n, we split that again into a vector around '='. This gives us:

n
20

We check if the element at index 1 is equal to the exponent we received as argument to the program, and if it is equal, we split the segment at index 0 in the initial vector around '=', giving us:

Runtime
0.0158858299255371

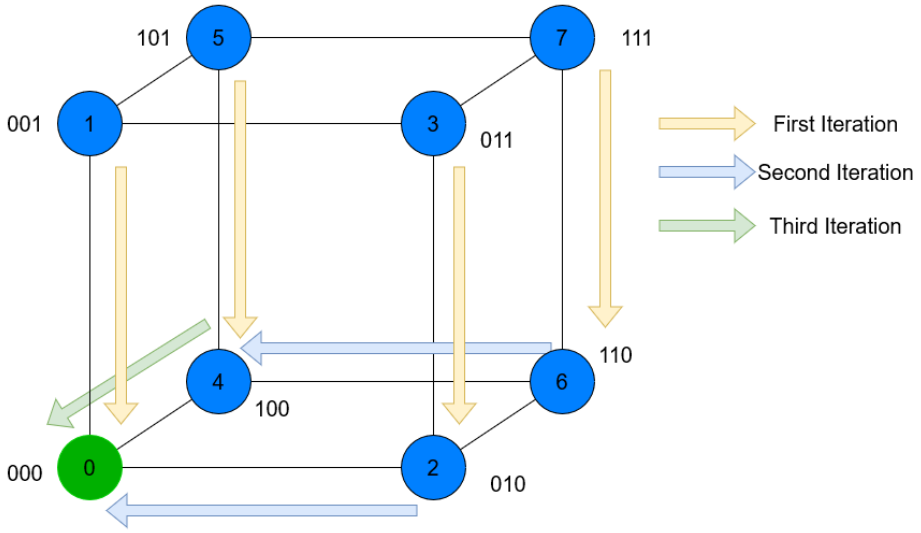
The value we now have at index 1 in this third vector is the runtime for P=1 and n=the command line argument, which we convert to a double from a string and store in *baserun*.
If the element at index 1 is not equal to the exponent, we fetch the next line.

We then proceed to write our output to the screen and the output file. We open our output file in append mode and add the values for Runtime, Runtime after Broadcast, pi, Error, n, nodes and ppn. The same is also printed to the screen. If we have more than 1 processor, it means that we need to write the speedup and efficiency too, so we add the value of speedup as *baserun / (end - start)* and efficiency as *baserun / (numprocs*(end - start))*.
Finally, we close the file and use *MPI_Finalize()* to terminate the MPI environment.

Utsav Sharma
usharm4@uic.edu

III. Simulating a Hypercube

We assume a hypercube topology for our processors, and to perform our reduction, we use an algorithm that works iteratively on each dimension of the hypercube, running from i=0 to dimension-1.
Taking a 3 dimensional hypercube of 8 processors as an example, the reduction works as below:



The reduction works by checking which processors differ in the bit corresponding to the iteration, moving from the least to most significant bit.
In the first iteration i=0, power = 2^0 = 1, and the mask is initialized to 0. All processors whose ID bitwise-AND mask = 0 participate in this iteration. To determine which processors will perform the send and which will perform the receive, we bitwise-AND the ID of the processor with the power calculated above. Those processors whose ID bitwise-AND power does not equal 0 perform the send, and the others perform the receive. That is, the processors whose least significant bit is set to 1 perform the sending, and those with their least significant bit set to 0 perform the receive. To calculate the destination, the ID of the sending processor is Bitwise-XORed with the power, and similarly for the source, the ID of the receiving processor is bitwise-XORed with the power.
In this iteration, processor 1 sends its data to 0, 3 sends its data to 2, 5 sends its data to 4 and 7 sends its data to 6. The ones that are performing the receive (0,2,4,6) will add the values they receive to their own partial calculation of pi.
At the end of the iteration, the mask variable is updated by Bitwise-XORing the mask with the power.
In the second iteration, i=1, power = 2^1 = 2. Again, in this iteration the processors whose ID bitwise-AND mask equals = 0 participate. The processors whose ID bitwise-AND power does not equal 0 perform the send, and the others perform the receive. This means that processor 2 sends to processor 0, and processor 6 sends to processor 4 (second LSB differs). Processors 0 and 4 add the values they receive to their own partial pi calculation, and the mask is updated again by Bitwise-XORing the mask with the power.
In the third and final iteration, i=2, power = 2^2 = 4. The processors 0 and 4 participate in this iteration, since their ID Bitwise-AND mask = 0. Processor 4 sends to processor 0, and processor 0 adds this value to its partial pi calculation.
In this way, all the partial sums have been collected at processor 0, and the variable mypi at processor 0 holds our final result.

IV. Test Results:

a) Hypercube algorithm with MPI_ISend and MPI_Recv:

Nodes	PPN	P	N	Runtime	Runtime After Broadcast	pi	Error	Speedup	Efficiency
1	1	1	20	0.0158858299255371	0.0158784389495850	3.1415926535897800	0.0000000000000040		
			23	0.1295883655548090	0.1295831203460690	3.1415926535893600	0.0000000000004281		
			26	0.9844195842742910	0.9844145774841300	3.1415926535910100	0.0000000000012212		
	2	2	20	0.0082774162292480	0.0082471370697021	3.1415926535898100	0.0000000000000204	1.919177372	0.959588686
			23	0.0664978027343750	0.0664720535278320	3.1415926535898800	0.0000000000000941	1.948761617	0.974380808
			26	0.5234065055847160	0.5233824253082270	3.1415926535893600	0.0000000000004321	1.880793559	0.940396779
	4	4	20	0.0043742656707764	0.0042920112609863	3.1415926535899000	0.0000000000001106	3.631656402	0.9079141
			23	0.0351140499114990	0.0349652767181396	3.1415926535897900	0.0000000000000044	3.690498985	0.922624746
			26	0.2785565853118890	0.2783439159393310	3.1415926535898700	0.0000000000000786	3.534002196	0.883500549
	8	8	20	0.0029766559600830	0.0022156238555908	3.1415926535898500	0.0000000000000666	5.336804165	0.667100521
			23	0.0176074504852295	0.0175046920776367	3.1415926535898000	0.0000000000000075	7.359859718	0.919982465
			26	0.1403150558471680	0.1397118568420410	3.1415926535899000	0.0000000000001070	7.015780155	0.876972519
	16	16	20	0.0017981529235840	0.0011765956878662	3.1415926535898700	0.0000000000000782	8.834526651	0.552157916
			23	0.0089631080627441	0.0088157653808594	3.1415926535897800	0.0000000000000062	14.45797202	0.903623251
			26	0.0699512958526611	0.0698812007904053	3.1415926535897900	0.0000000000000053	14.07292849	0.879558031
2	1	2	20	0.0102484226226807	0.0097873210906982	3.1415926535898100	0.0000000000000204	1.550075608	0.775037804
			23	0.0662305355072021	0.0658197402954102	3.1415926535898800	0.0000000000000941	1.956625665	0.978312832
			26	0.4975857734680170	0.4971175193786620	3.1415926535893600	0.0000000000004321	1.97839174	0.98919587
4	1	4	20	0.0064647197723389	0.0051760673522949	3.1415926535899000	0.0000000000001106	2.457311451	0.614327863
			23	0.0359344482421875	0.0350437164306641	3.1415926535897900	0.0000000000000044	3.606243365	0.901560841
			26	0.2604258060455320	0.2594602108001700	3.1415926535898700	0.0000000000000786	3.780038542	0.945009636
8	1	8	20	0.0052261352539062	0.0030927658081055	3.1415926535898500	0.0000000000000666	3.039689781	0.379961223
			23	0.0193099975585938	0.0177648067474365	3.1415926535898000	0.0000000000000075	6.71094676	0.838868345
			26	0.1335341930389400	0.1318495273590080	3.1415926535899000	0.0000000000001070	7.372041287	0.921505161
	2	16	20	0.0030705928802490	0.0015292167663574	3.1415926535898700	0.0000000000000782	5.173538318	0.323346145
			23	0.0106449127197266	0.0090835094451904	3.1415926535897800	0.0000000000000062	12.17373679	0.760858549
			26	0.0722658634185791	0.0703423023223877	3.1415926535897900	0.0000000000000053	13.62219363	0.851387102

b) Simple algorithm with MPI_Reduce:

Nodes	PPN	P	N	Runtime	Runtime After Broadcast	pi	Error	Speedup	Efficiency
1	1	1	20	0.0159008502960205	0.0158917903900146	3.1415926535897800	0.0000000000000040		
			23	0.1268060207366940	0.1268002986907950	3.1415926535893600	0.0000000000004281		
			26	0.9884655475616450	0.9884600639343260	3.1415926535910100	0.0000000000012212		
	2	2	20	0.0087189674377441	0.0082998275756836	3.1415926535898100	0.0000000000000204	1.823707957	0.911853979
			23	0.0662102699279785	0.0656864643096924	3.1415926535898800	0.0000000000000941	1.915201688	0.957600844
			26	0.5213477611541740	0.5213236808776850	3.1415926535893600	0.0000000000004321	1.895981188	0.947990594
	4	4	20	0.0044693946838379	0.0044059753417969	3.1415926535899000	0.0000000000001106	3.55771898	0.889429745
			23	0.0373673439025879	0.0369012355804443	3.1415926535897900	0.0000000000000044	3.393498373	0.848374593
			26	0.2757003307342520	0.2756650447845450	3.1415926535898700	0.0000000000000786	3.585289669	0.896322417
	8	8	20	0.0023379325866699	0.0022406578063965	3.1415926535898500	0.0000000000000666	6.801244136	0.850155517
			23	0.0181088447570801	0.0176053047180176	3.1415926535898000	0.0000000000000075	7.002435685	0.875304461
			26	0.1396031379699700	0.1395540237426750	3.1415926535899000	0.0000000000001070	7.080539606	0.885067451
	16	16	20	0.0012516975402832	0.0011830329895020	3.1415926535898700	0.0000000000000782	12.70342857	0.793964286
			23	0.0088505744934082	0.0087730884552002	3.1415926535897800	0.0000000000000062	14.32743387	0.895464617
			26	0.0698716640472412	0.0698101520538330	3.1415926535897900	0.0000000000000053	14.14687286	0.884179554
2	1	2	20	0.0099482536315918	0.0095918178558350	3.1415926535898100	0.0000000000000204	1.598355941	0.799177971
			23	0.0661761760711670	0.0658130645751953	3.1415926535898800	0.0000000000000941	1.916188397	0.958094198
			26	0.4965116977691650	0.4956648349761960	3.1415926535893600	0.0000000000004321	1.990820261	0.99541013
4	1	4	20	0.0065805912017822	0.0051350593566895	3.1415926535899000	0.0000000000001106	2.416325495	0.604081374
			23	0.0364022254943848	0.0350058078765869	3.1415926535897900	0.0000000000000044	3.483468909	0.870867227
			26	0.2573063373565670	0.2563958168029780	3.1415926535898700	0.0000000000000786	3.841590369	0.960397592
8	1	8	20	0.0044643878936768	0.0028517246246338	3.1415926535898500	0.0000000000000666	3.561708945	0.445213618
			23	0.0196499824523926	0.0178382396697998	3.1415926535898000	0.0000000000000075	6.45323837	0.806654796
			26	0.1349153518676750	0.1337838172912590	3.1415926535899000	0.0000000000001070	7.326560943	0.915820118
	2	16	20	0.0034549236297607	0.0017170906066895	3.1415926535898700	0.0000000000000782	4.602373887	0.287648368
			23	0.0115828514099121	0.0100715160369873	3.1415926535897800	0.0000000000000062	10.94773785	0.684233615
			26	0.0721869468688965	0.0701713562011719	3.1415926535897900	0.0000000000000053	13.69313415	0.855820885

V. Analysis:

a) Theoretical Analysis:

i) With Broadcast:

From our problem statement, the message-passing time for 1 unit of data is 10x the time for atomic/scalar arithmetic operations. Taking each step of the program one by one:

The broadcast to P processors would take $\log_2(P)$ time. Since the time for communication is 10x, this is multiplied by 10.

For the partial calculations of pi, we have each processor calculate h, which takes 1 unit of time multiplied by P.

For the actual calculation, there are 2^n terms divided among P processors. There are 2 steps to the summation, and these summations happen in parallel, which means they take $2(2^n/P)$ time.

The final calculation of mypi by multiplying h and sum takes 1 time unit per processors, that is, P time in total.

Similar to the broadcast operation, the reduce operation would take $\log_2(P)$ time. Since this is a communication operation, it is multiplied by a factor of 10.

Therefore, our final term for calculating the time for the program is:

$10(\log_2(P))+P+2(2^n/P)+10(\log_2 P)$

This can be simplified to:

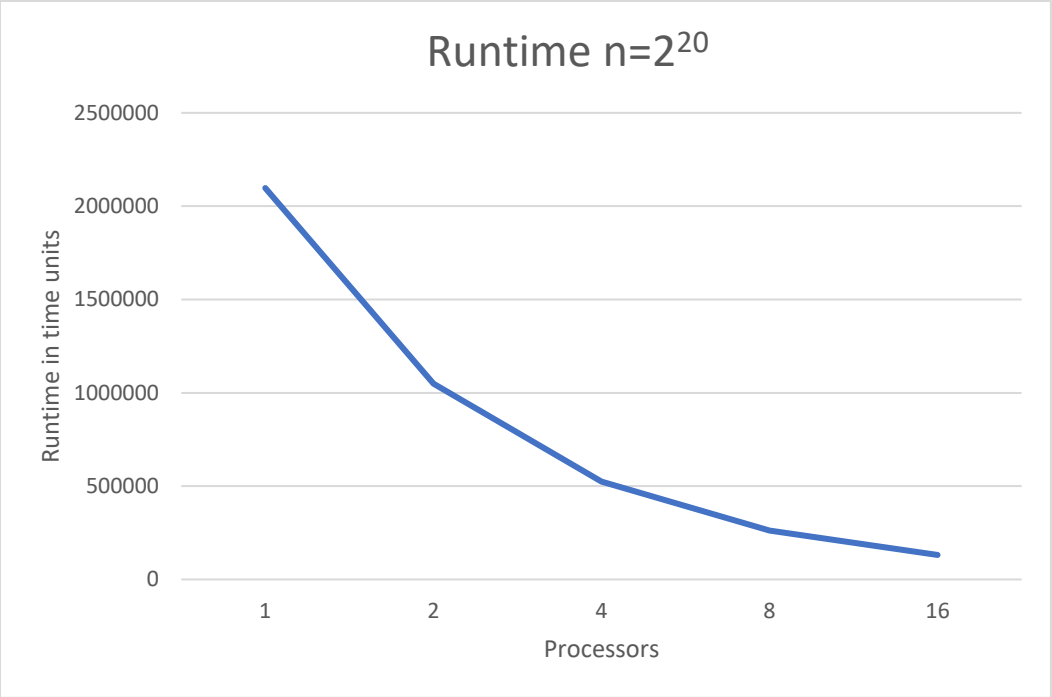
$20(\log_2 P)+(2^{n+1}/P)+P$

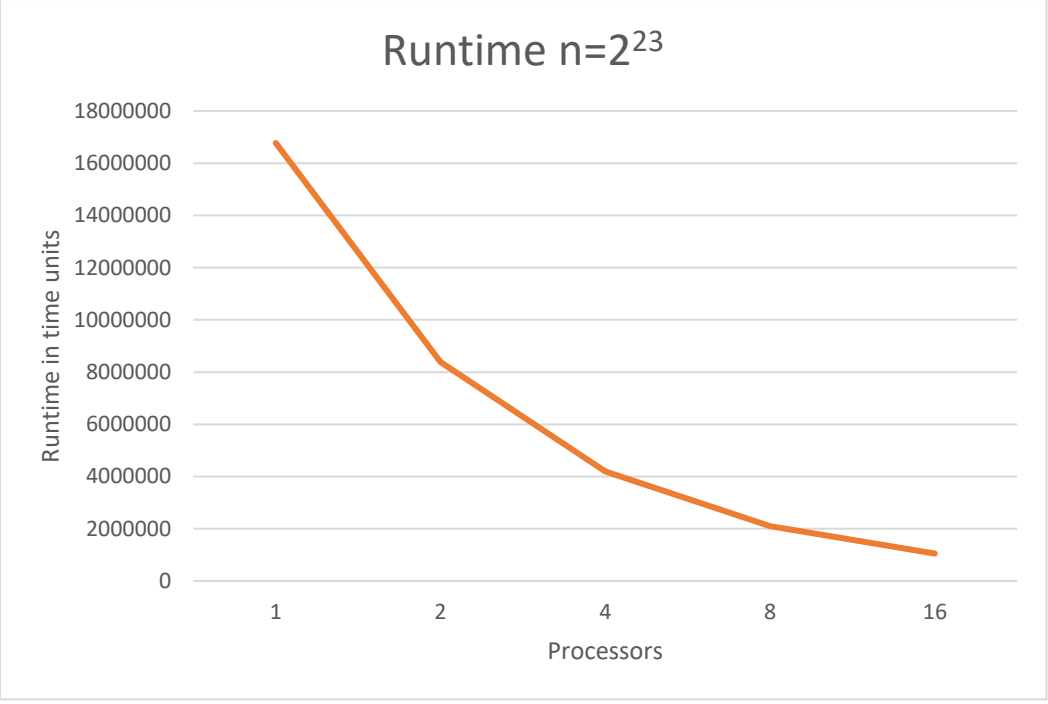
Plugging in values of P and n, we get the below table:

P	n	Time
1	20	2097152
1	23	16777216
1	26	134217728
2	20	1048596
2	23	8388628
2	26	67108884
4	20	524328
4	23	4194344
4	26	33554472
8	20	262204
8	23	2097212
8	26	16777276
16	20	131152
16	23	1048656
16	26	8388688

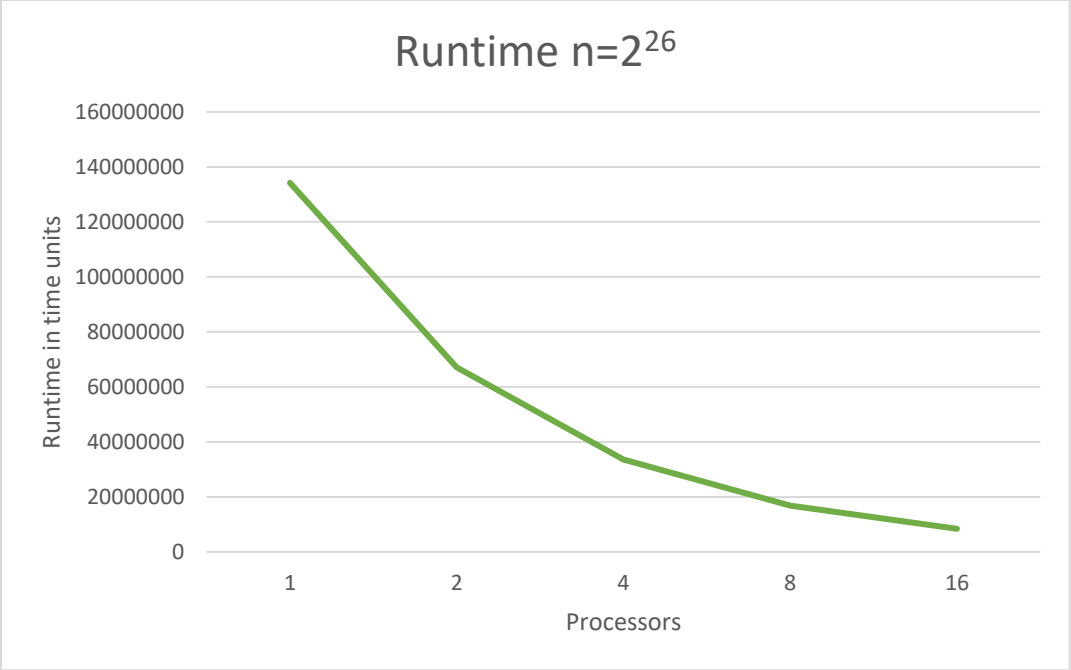
Utsav Sharma
usharm4@uic.edu

Plotting these values keeping n constant, we get the below charts:





Utsav Sharma
usharm4@uic.edu



ii) <u>Without Broadcast:</u>

This calculation is similar to the above calculation, with the only difference that the first $10(\log_2 P)$ term is removed, since we are calculating time from after the broadcast.
In this case, the expression is:

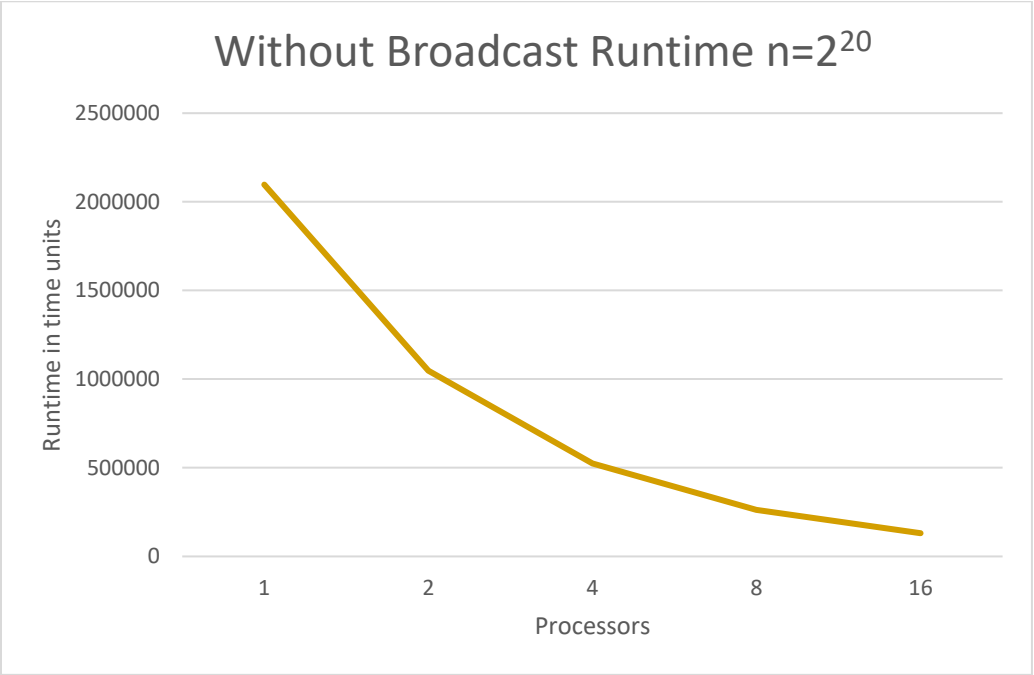
$10(\log_2 P)+(2^{n+1}/P)+P$

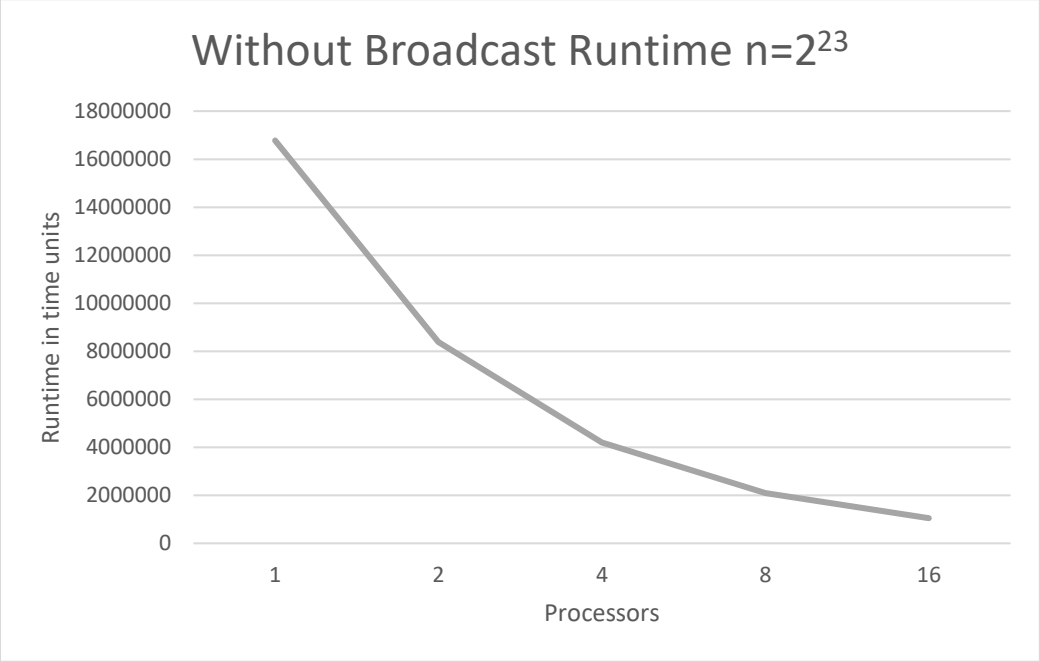
Plugging in values of P and n, we get the below table:

P	n	Time without Broadcast
1	20	2097152
1	23	16777216
1	26	134217728
2	20	1048586
2	23	8388618
2	26	67108874
4	20	524308
4	23	4194324
4	26	33554452
8	20	262174
8	23	2097182
8	26	16777246
16	20	131112
16	23	1048616
16	26	8388648

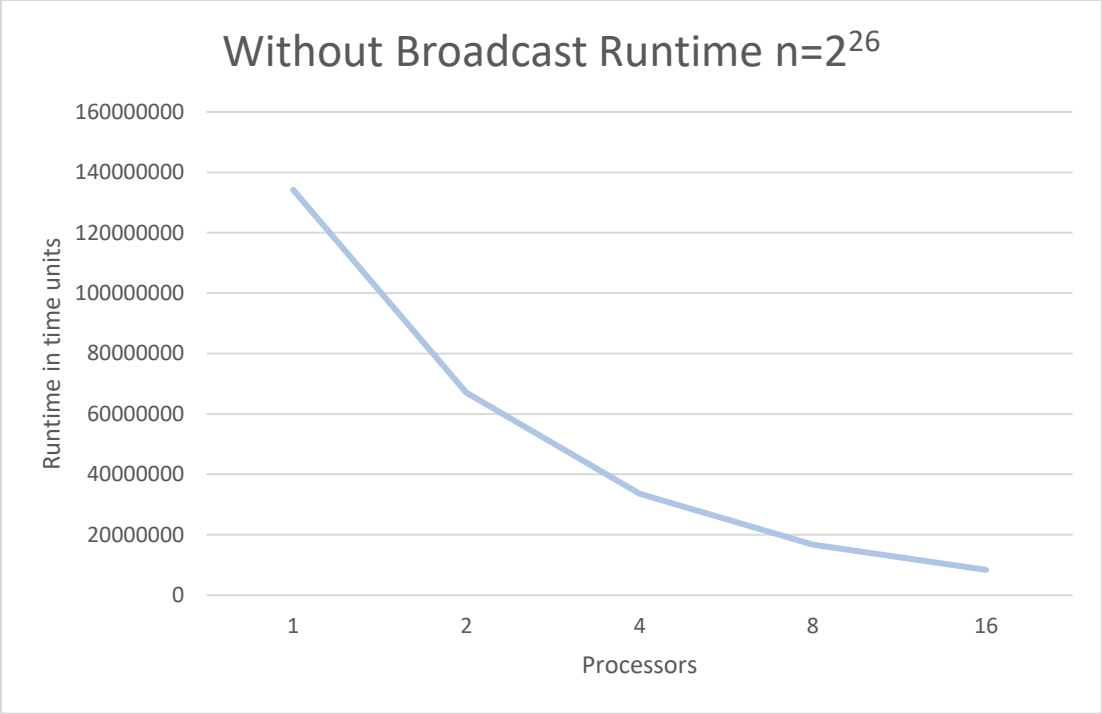
Utsav Sharma
usharm4@uic.edu

Creating the plots for these values gives us the below charts:





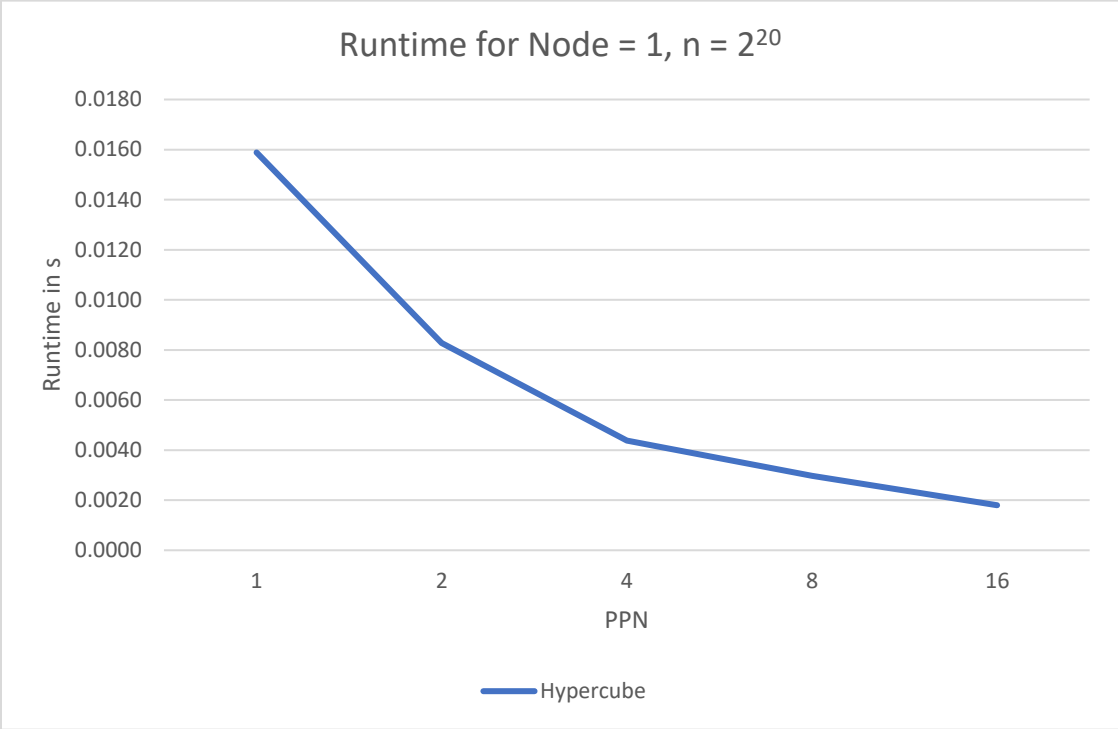
Utsav Sharma
usharm4@uic.edu



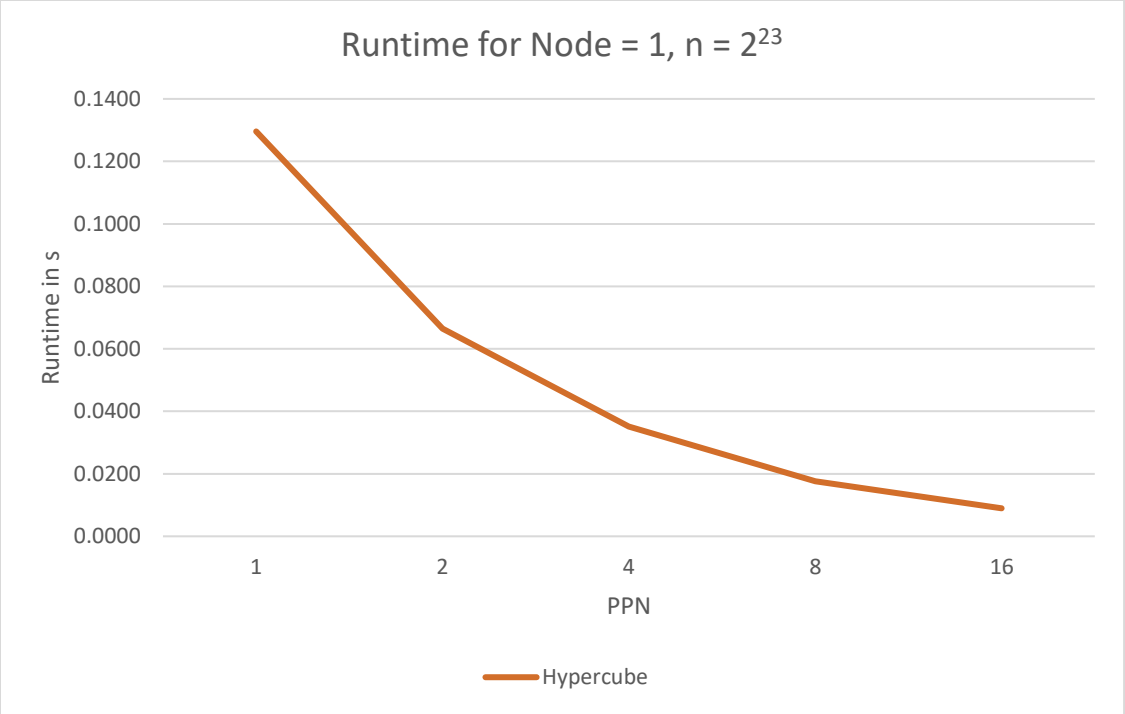
b) Hypercube Algorithm

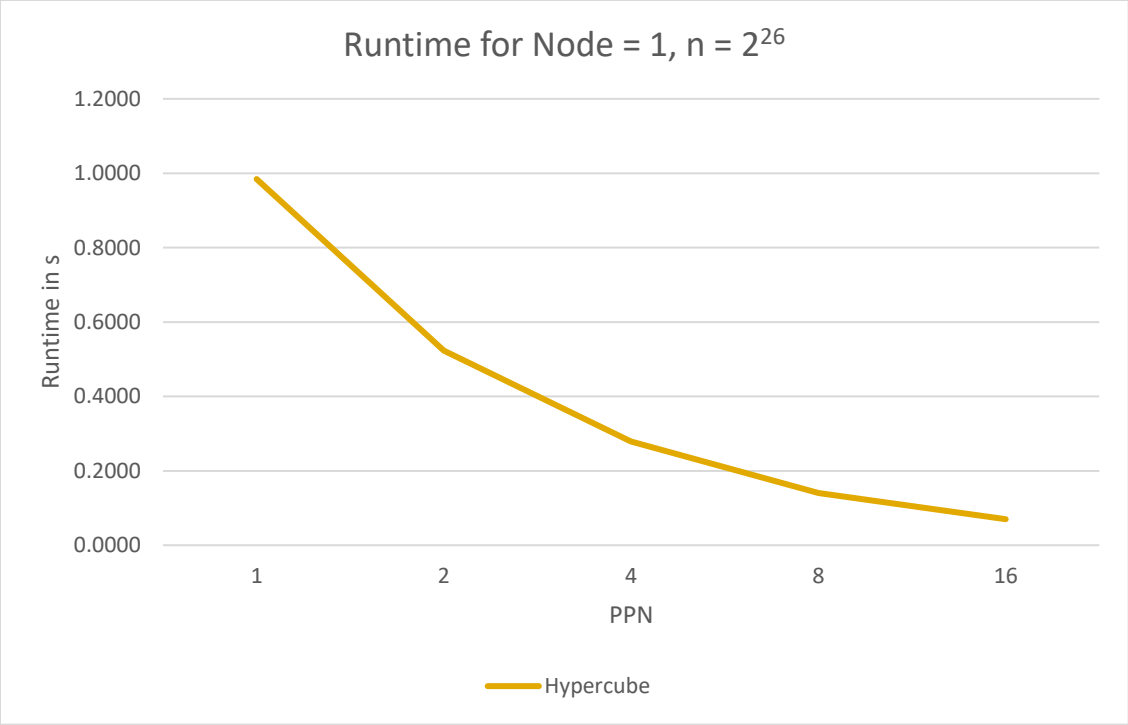
i) With Broadcast:

We plot the actual values for our Hypercube algorithm keeping n constant and get the below charts:



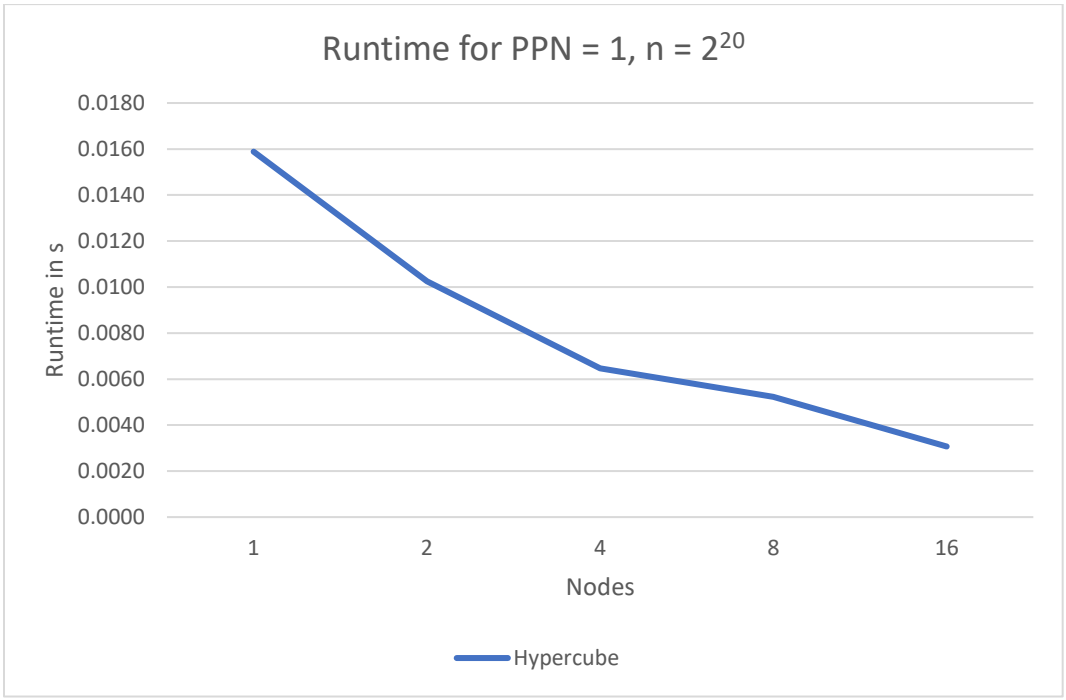
Utsav Sharma
usharm4@uic.edu

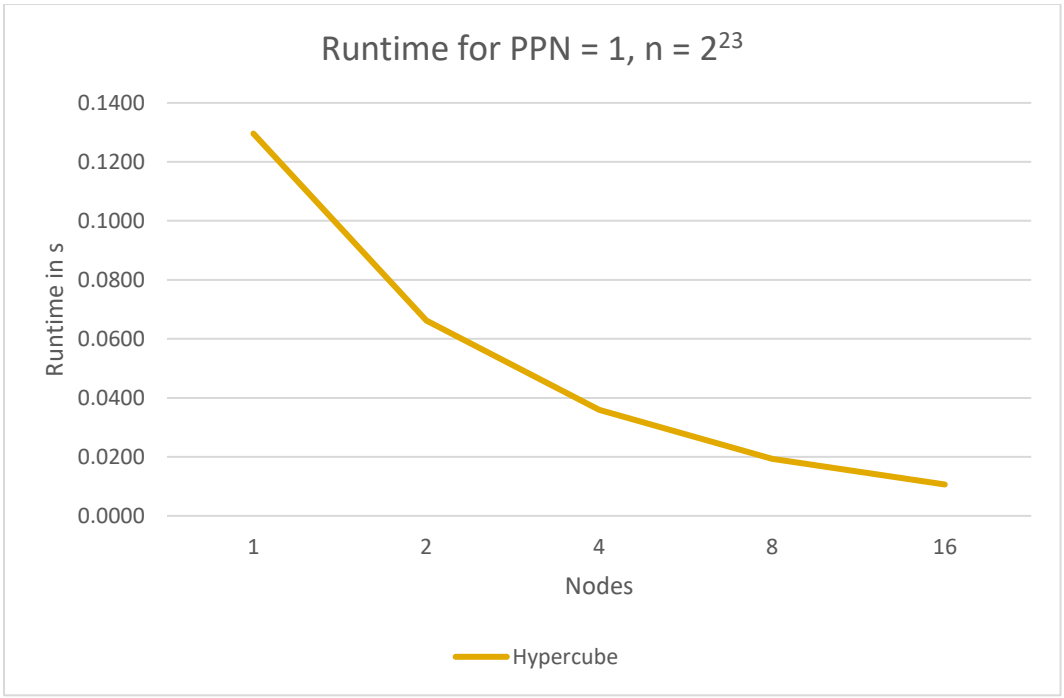




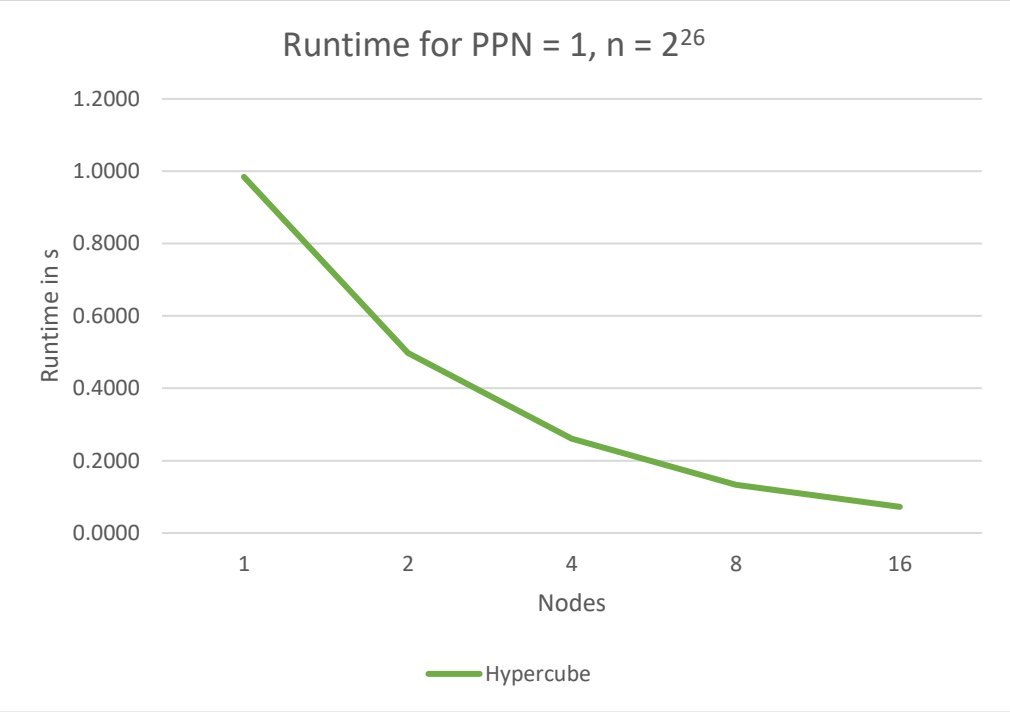
Utsav Sharma
usharm4@uic.edu

Keeping N and PPN constant, we get the below plots:





Utsav Sharma
usharm4@uic.edu

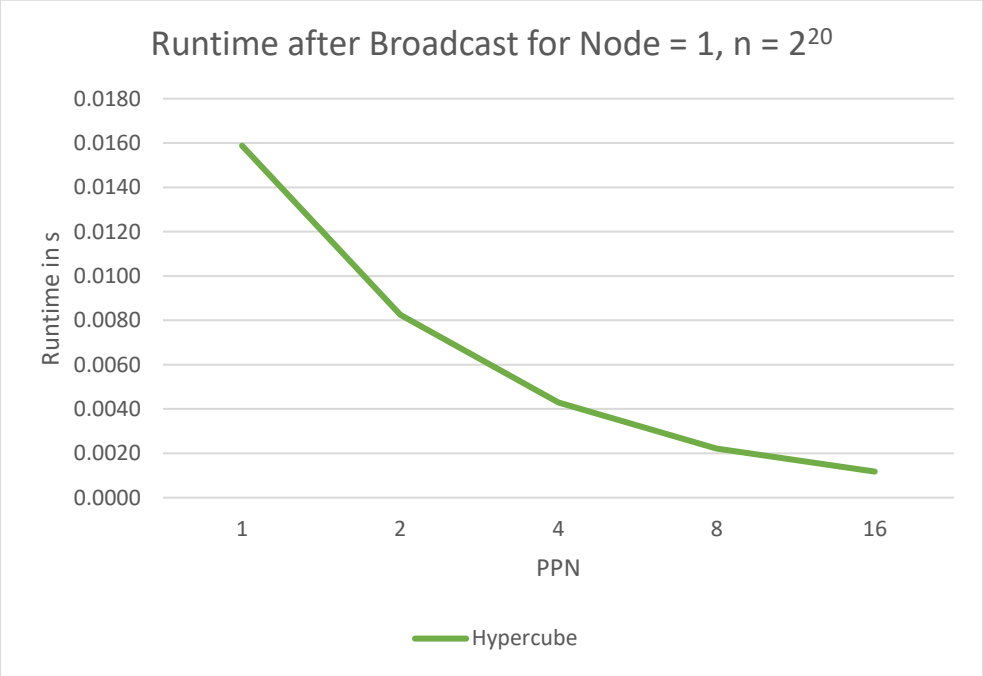


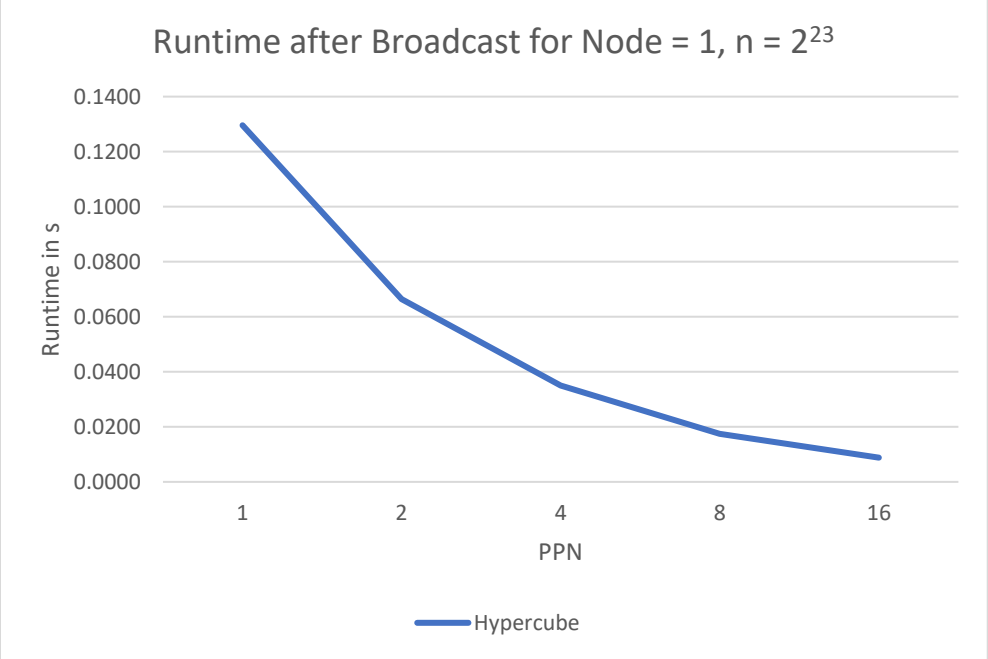
From observing these plots, we can see that these empirical plots are identical to our theoretical plots. This means that our theoretical analysis was correct, and the hypercube reduction algorithm is working as expected.
We do see a minor deviation from our expectations at Nodes=8 PPN=1 for $N=2^{20}$, but that can be caused by a variety of factors such the load on the system at the moment this particular program was running.
While we do not expect the actual numbers of our theoretical analysis to match with our empirical results because the time unit is an arbitrary measure, the line plotted by those numbers is enough to give us the information we need about the behavior of the code.
We see that as expected, the runtime of the program follows the reduction in runtime that we anticipated from our analysis, where doubling the node count halves our runtime, and this trend holds true regardless of the value of the exponent.

Analyzing the Speedup and Efficiency, we can see that for our experiment parameters we see low speedups for low values of P. For higher values of P, we saw the most speedup for $n=2^{23}$ and 2^{26} and P=16, going as high as 13/14. One important thing to note is that high speedup does not correspond to high efficiency, since we can see that for P=16 and $n=2^{20}$, even though we have decent speedups, the efficiency is low. We also do not see efficiency greater than 0.97, which means that we need to make a tradeoff between speedup and efficiency.

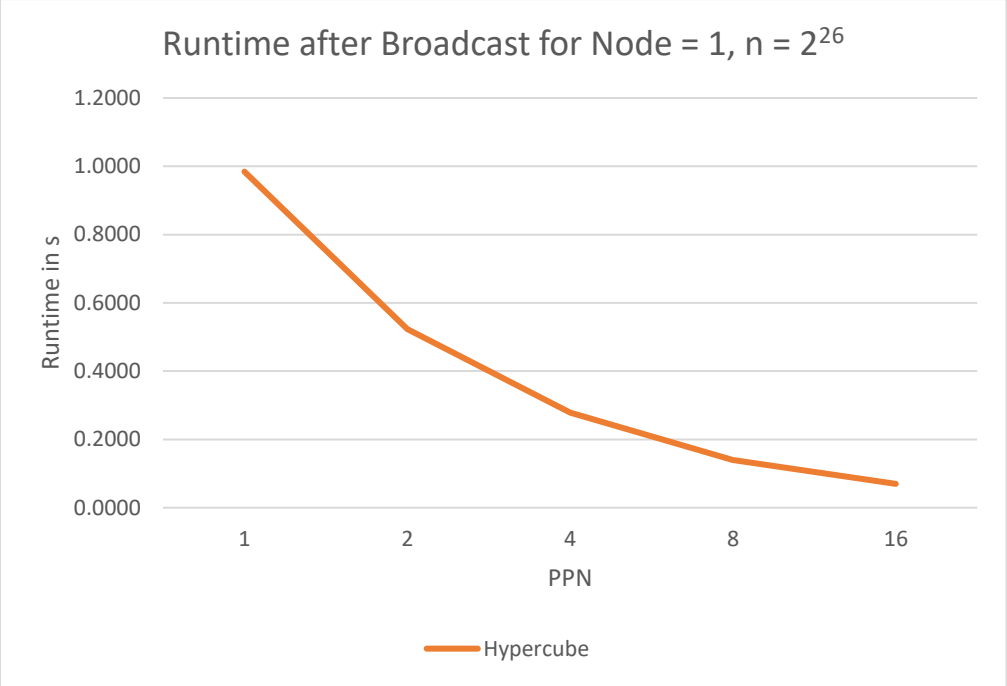
Utsav Sharma
usharm4@uic.edu

ii) <u>Without Broadcast:</u>

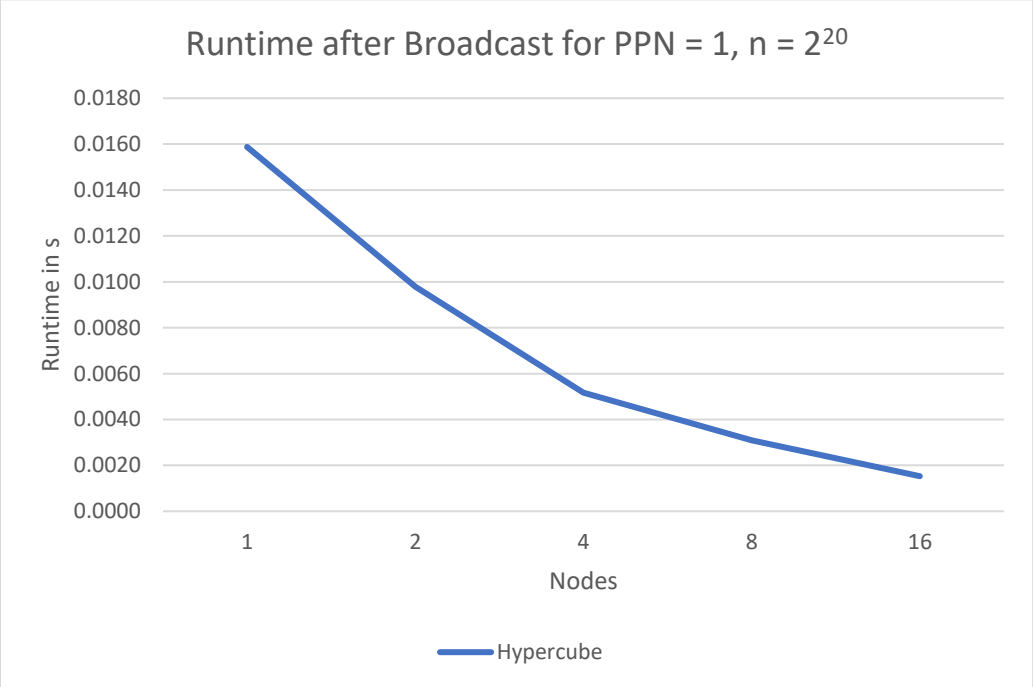




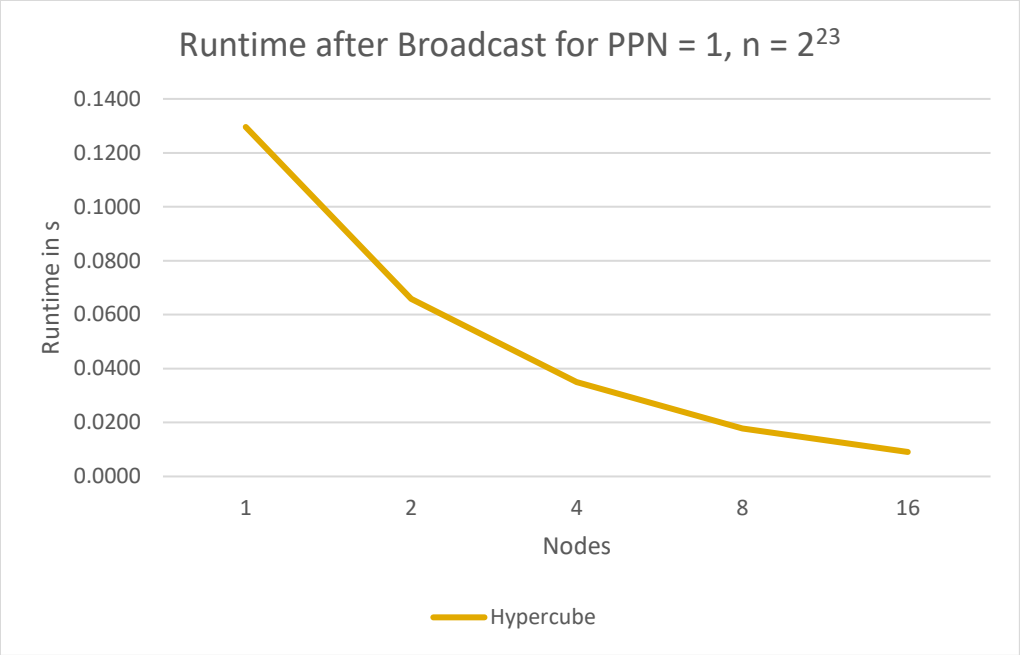
Utsav Sharma
usharm4@uic.edu

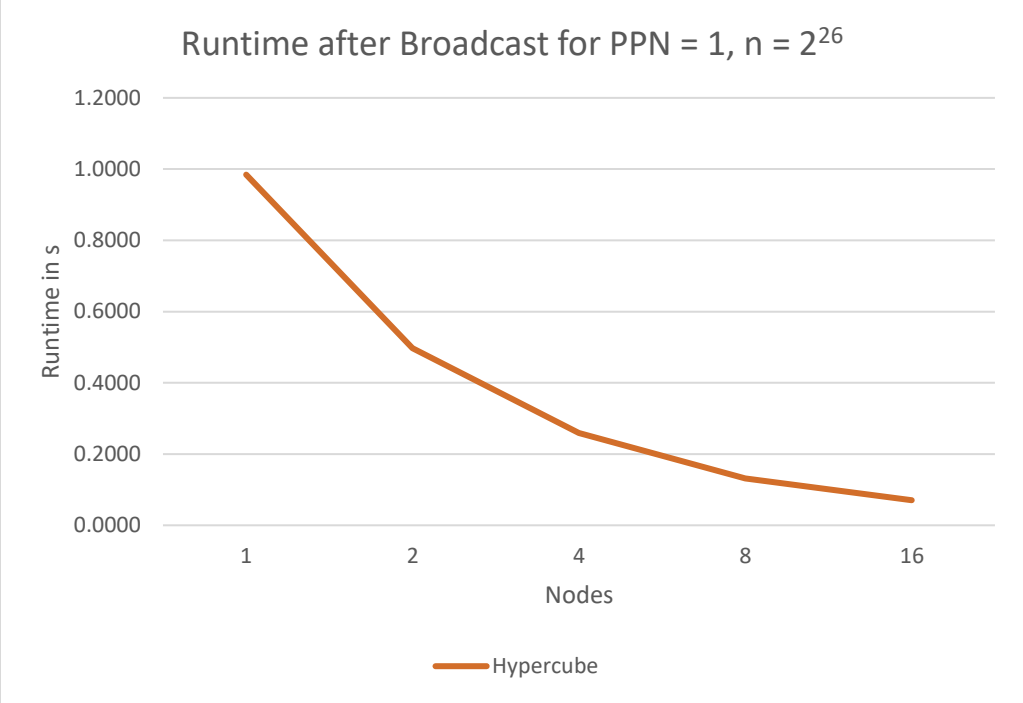


Varying the nodes, we get the below charts:



Utsav Sharma
usharm4@uic.edu





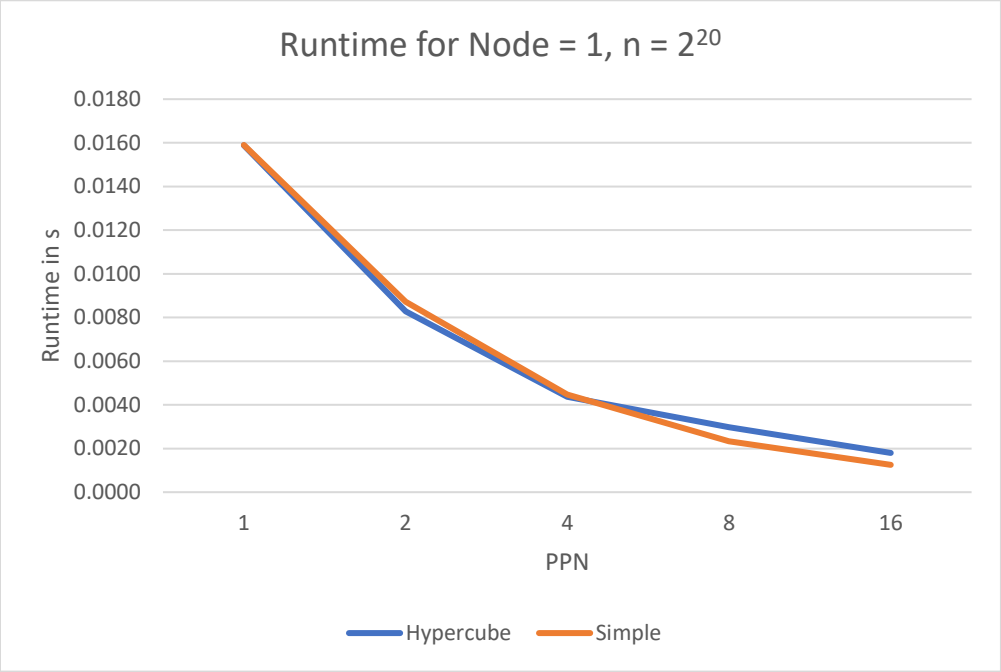
As with the charts for the runtime with broadcast, we can see that the plots here match the expected plots from our theoretical analysis. Thus, we can conclude that our hypercube reduce algorithm is working as expected.

The speedup and efficiency trends follow those of the above section, where higher P counts increase our speedup, and higher speedup does not equal higher efficiency.

Utsav Sharma
usharm4@uic.edu

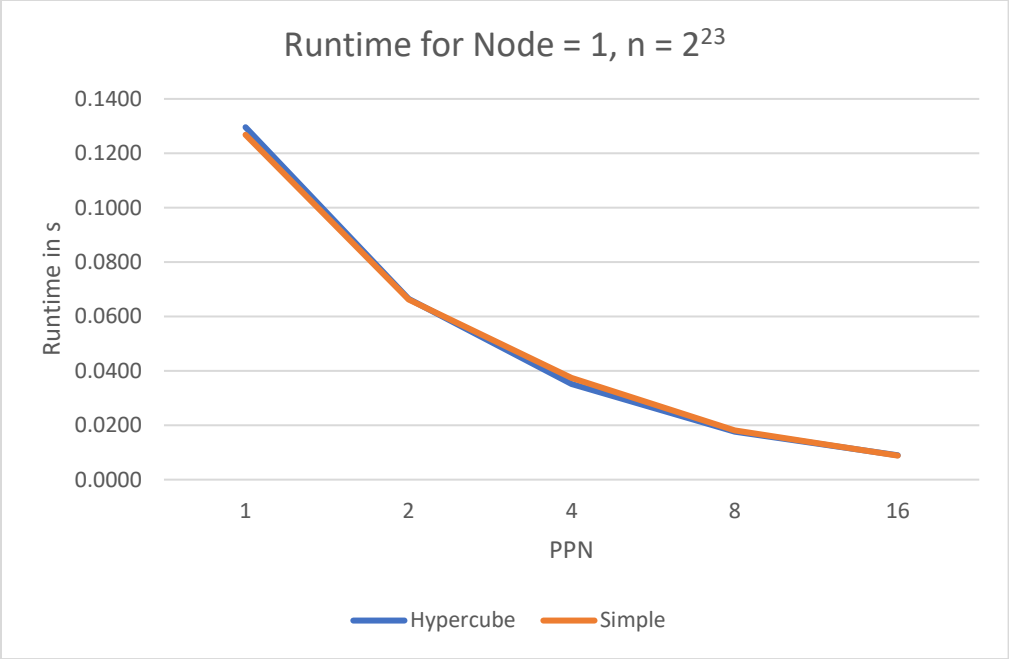
c) Hypercube Reduction vs MPI_Reduce

To perform a one-to-one analysis between our algorithms, we will be looking at various plots which fix certain parameters of our experiment and vary all the other parameters. These plots have been split to make them easier to decipher, since they have varying scales and trying to add all plots in one graph would make it incomprehensible.
Plotting the values for runtime for the first case where we have Nodes fixed at 1, and PPN varying between 1,2,4,8 and 16, we get the below charts:

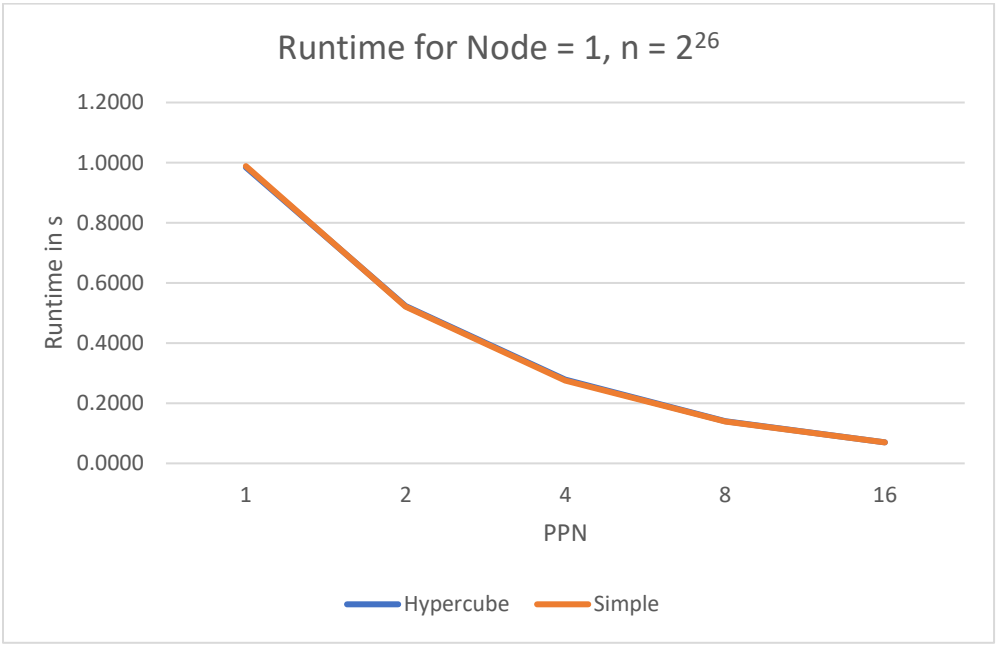


From the above chart, we can see that there is barely any difference between the runtime for the simple program using *MPI_Reduce* and the Hypercube *MPI_ISend* and *MPI_Recv* program. At Nodes = 1 and PPN = 2, the Hypercube algorithm is marginally slower (a difference of 0.0004415512084961 seconds), while at PPN = 8 and 16, the hypercube algorithm is marginally faster (a difference of 0.0006387233734131 seconds and 0.0000796318054199 seconds respectively).

Utsav Sharma
usharm4@uic.edu



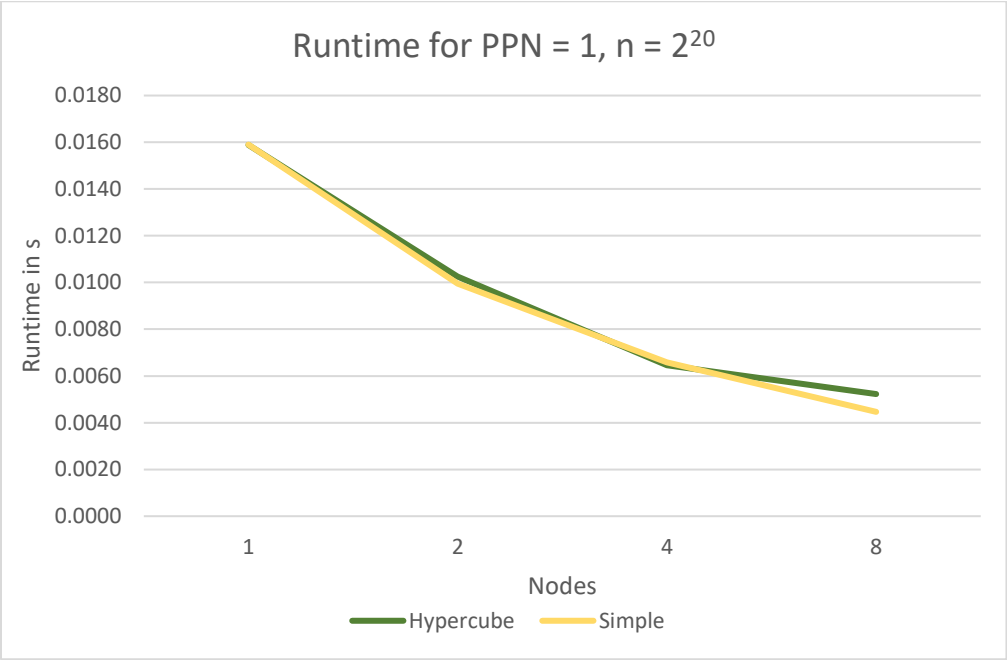
Analyzing the results for Node = 1, $n=2^{23}$, we can see that there is barely any difference between the runtimes of the Hypercube and the simple algorithm, and the story continues when we analyze Nodes =1, $n = 2^{26}$ in the chart below. The runtimes in this case are near identical, to the extent that the plots almost overlap with each other.



From analyzing these charts, we can form a hypothesis that our Hypercube algorithm works just as well as the MPI_Reduce function. To support our hypothesis better, we will look at more comparisons below, which will help us establish a pattern to base our hypothesis on.

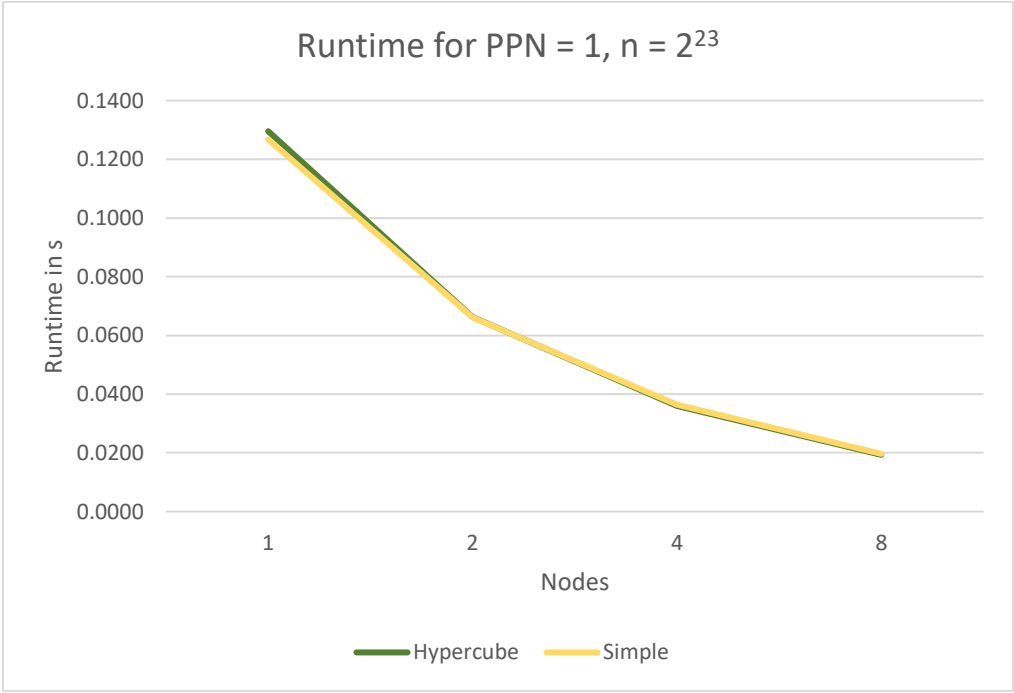
Utsav Sharma
usharm4@uic.edu

For the second case, where we fix PPN = 1 and vary nodes between 1,2,4 and 8, we get the below plots:



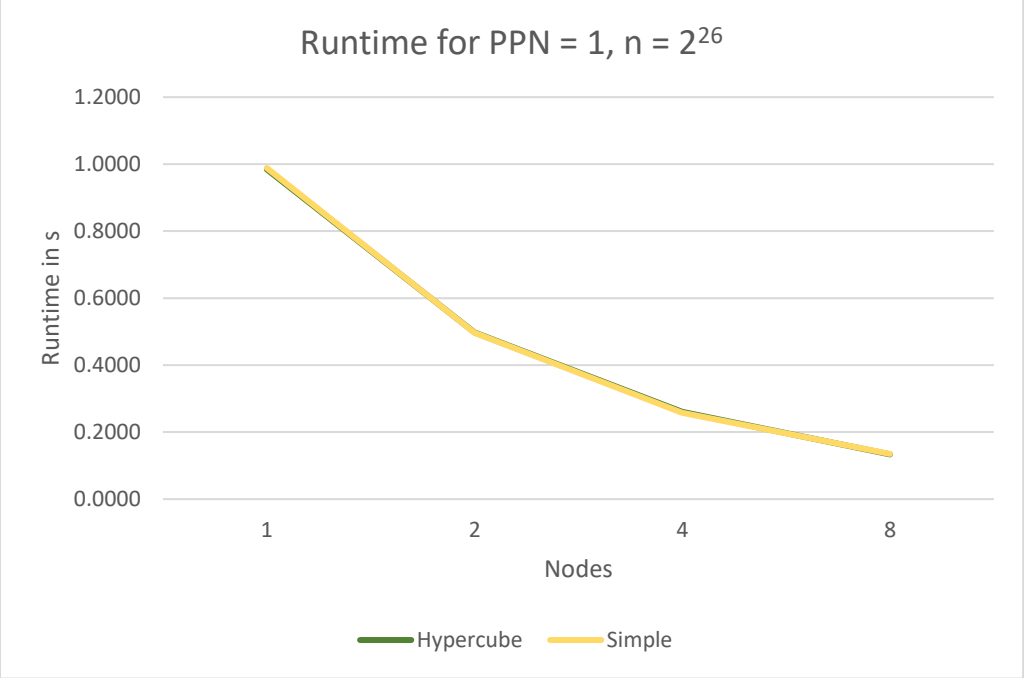
In the above chart for $n=2^{20}$, we again see that at Nodes 1, 2 and 4, the algorithms are identical, while at Nodes = 8, the simple algorithm is slightly faster than our hypercube algorithm (by 0.0006387233734131 seconds).

For the chart below that shows us the results for $n = 2^{23}$, the plots are near identical, with the simple algorithm marginally faster at Nodes = 1 (by 0.0027823448181150 seconds).



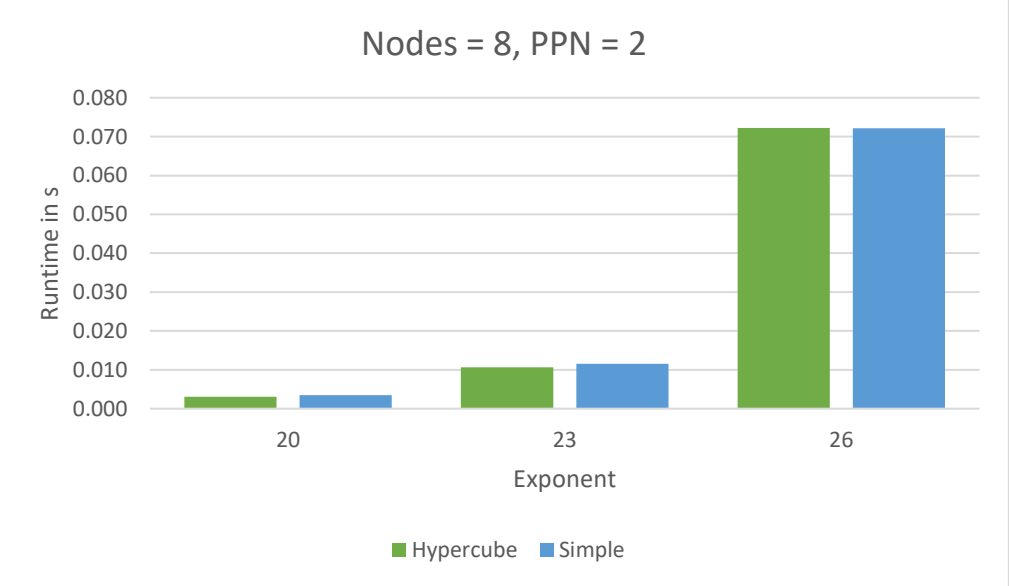
Utsav Sharma
usharm4@uic.edu

At n = 2^{26}, the story is the same, with the plots identical for the hypercube and the simple algorithm.



So far, all our plots support our hypothesis that the Hypercube algorithm is just as good as the native *MPI_Reduce* call.

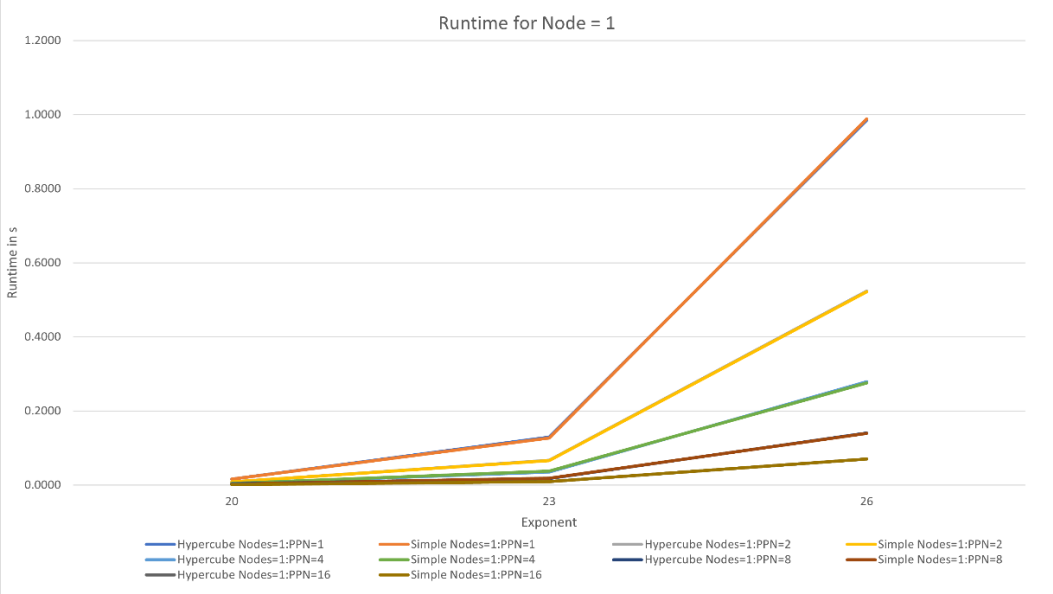
For the third case, where we have P=16, i.e., Nodes = 8 and PPN = 2, we have the below plot:



We see the same trend being followed, where the hypercube and simple algorithms are so close to one another in performance.

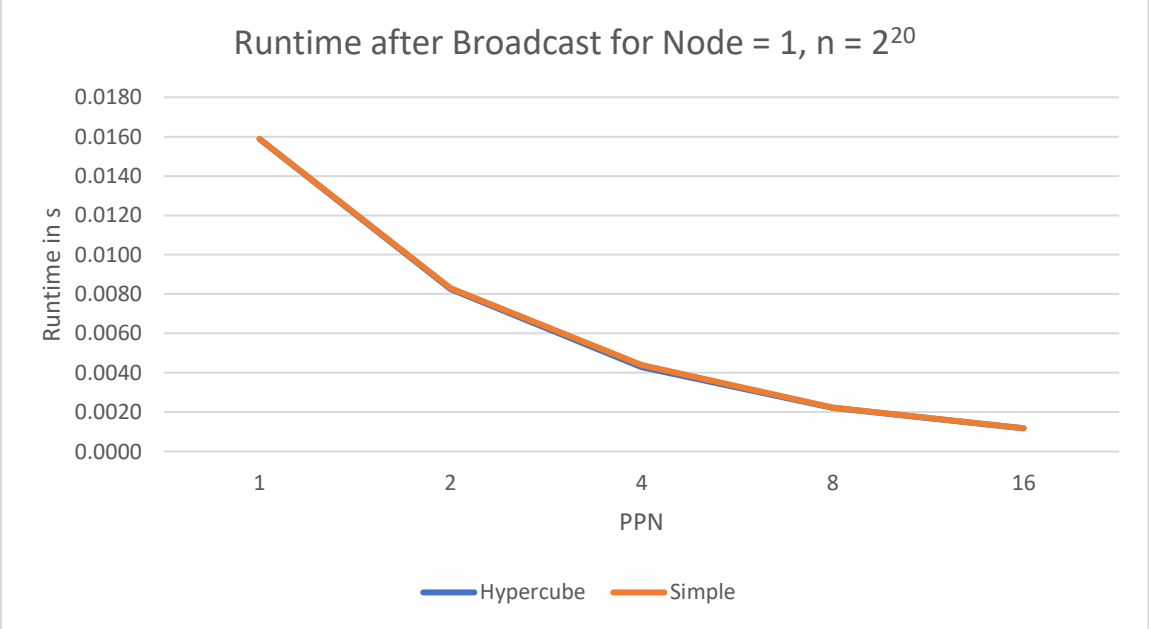
Utsav Sharma
usharm4@uic.edu

Another effect that we can see is of increasing the exponent of n, which is our precision parameter. Across the board, increasing the value of n increases the runtime of the program. However, adding processors does decrease the runtime of the program given a certain exponent.

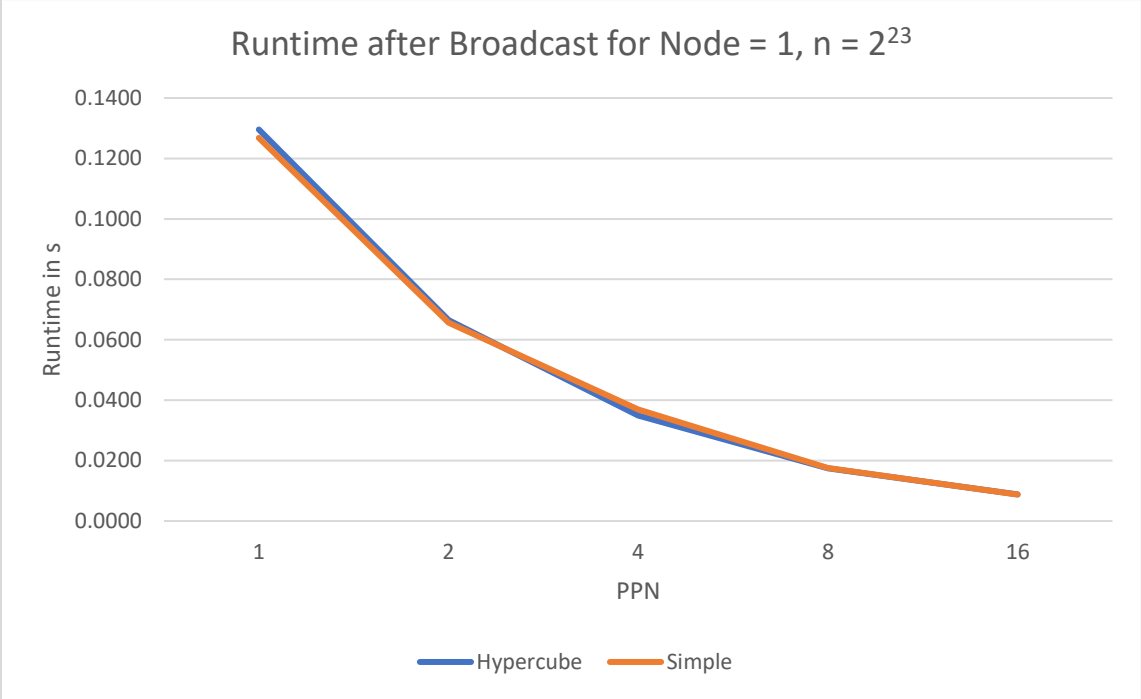


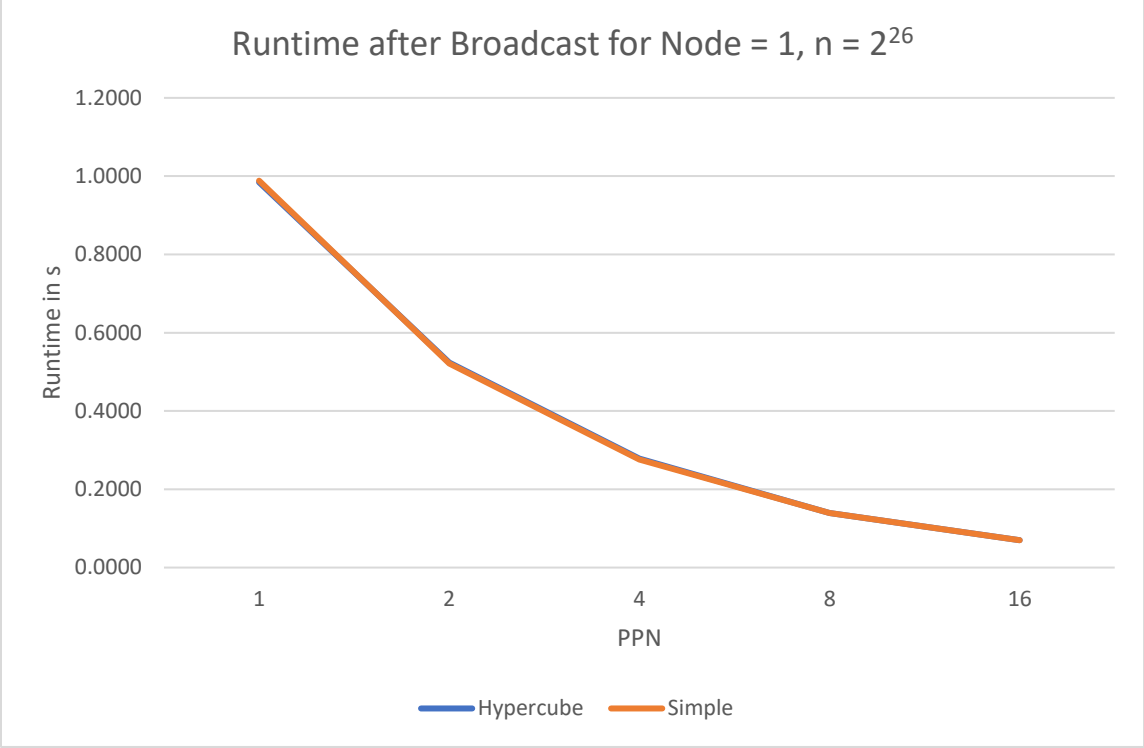
This is most evident at n = 2^{26}, where the runtime of the program is approximately halved when we go from PPN = 1 to PPN = 2, and is halved again when PPN = 4. The trend continues with PPN = 8 and PPN = 16, which further halve the runtimes of their preceding parameters. We can also establish that at higher processor counts, the effect of value of the exponent on the runtime is diminished due to the distribution of the workload among a higher number of processors.

To investigate this even deeper, we will take a look at the charts for the runtime of the programs excluding the broadcast, that is, the time used to perform the reduction operation only.



Utsav Sharma
usharm4@uic.edu

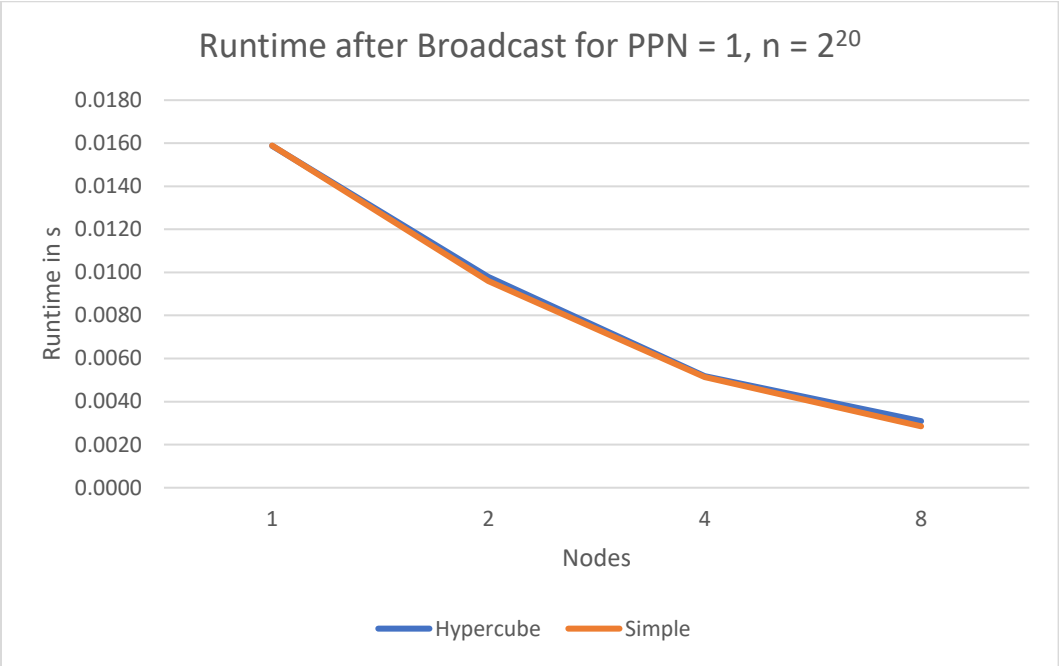


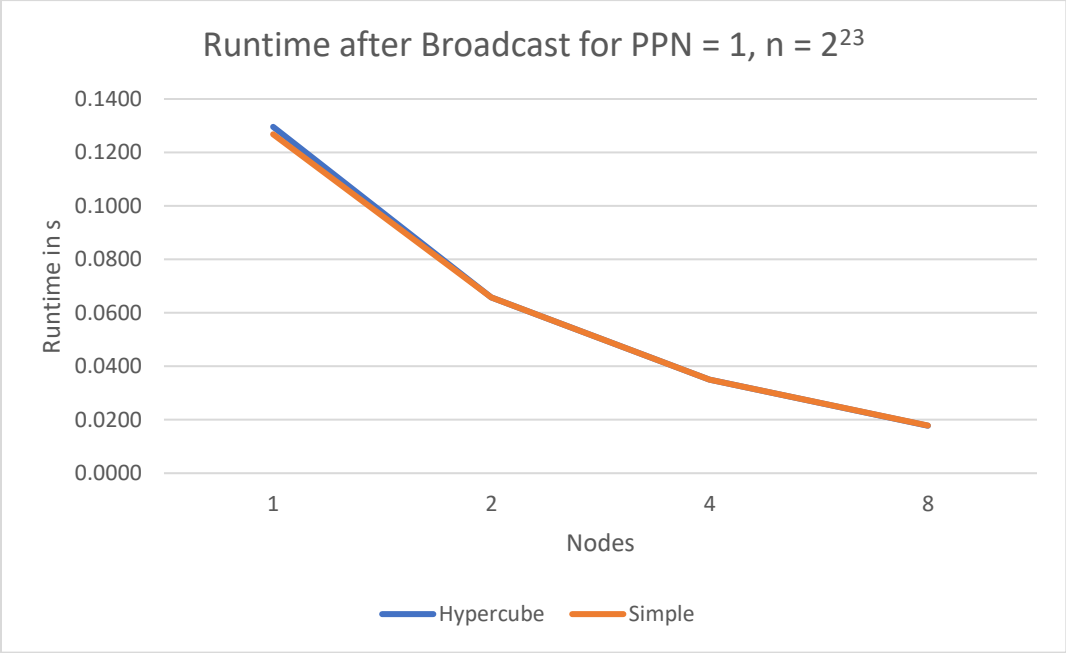


These graphs above corroborate our hypothesis, as the reduction operations take identical times for the hypercube reduction and the native MPI_Reduce function. These plots are similar to the plots above that show the total runtime including the broadcast, which further strengthens our hypothesis.

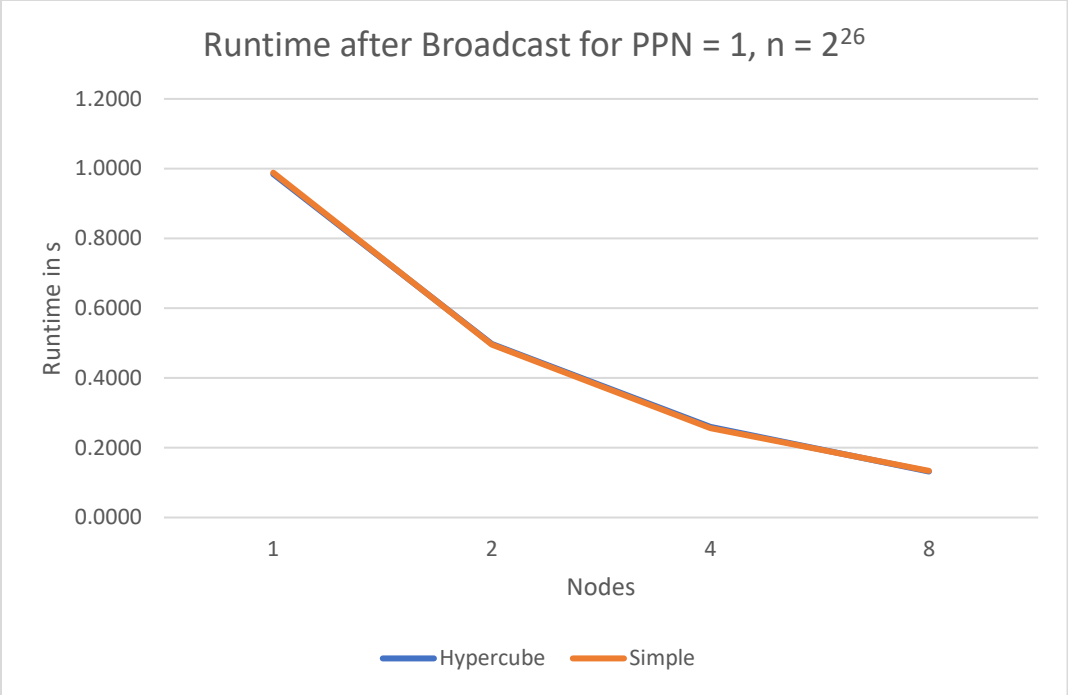
Utsav Sharma
usharm4@uic.edu

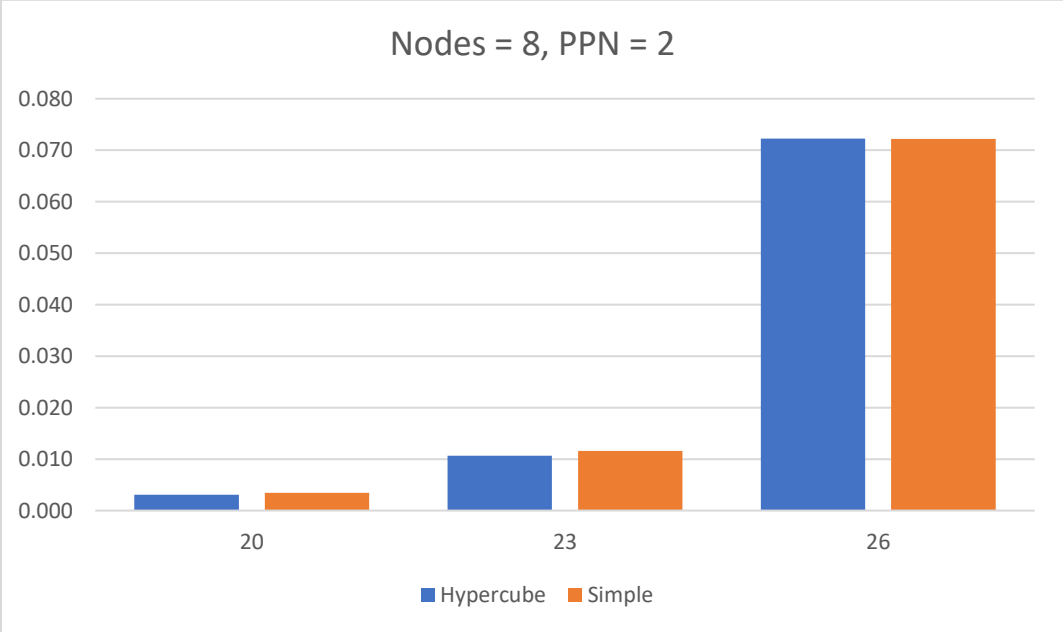
Looking at a few more charts:





Utsav Sharma
usharm4@uic.edu





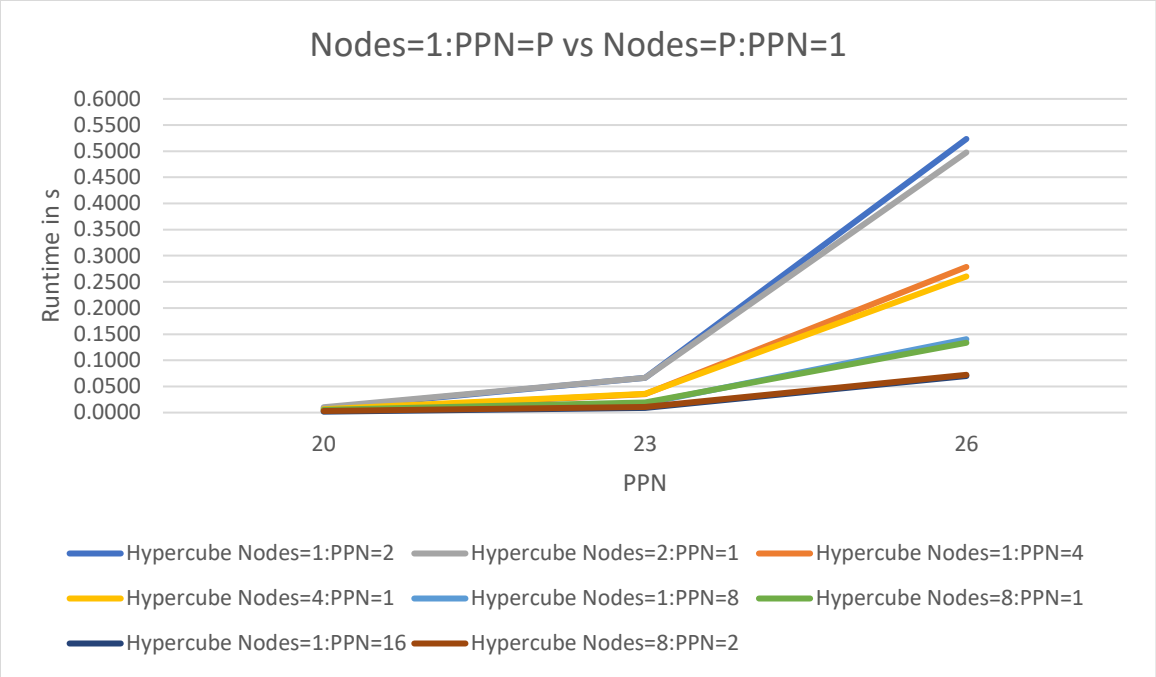
All the above charts are also identical to their counterparts that show us the total runtime.

Thus, we can confidently state that the hypercube reduction algorithm used here is the same in performance as the native MPI_Reduce function, and negligible differences can be chalked down to variance between runs.
One thing to note here is that the Error increases as we increase the precision term. From our discussions in class, the most likely cause of this is the loss of precision due to the higher number of terms in the computation, which causes the smaller terms towards the end of the summation to be lost during addition due to the limitations of the system.

Utsav Sharma
usharm4@uic.edu

d) Nodes=1:PPN=P vs Nodes=P:PPN=1

We can also compare the effect of having Nodes=1:PPN=P and Nodes=P:PPN=1, which is shown in the chart below. For simplicity, we are only comparing the runtimes of the hypercube algorithm, since the simple algorithm produces runtimes that are near identical to it.



Interestingly, in almost every case, we see that having the processors on two separate nodes is measurably faster than having them on the same node. One reason for this measurable difference could possibly be that since the processors on the same node use shared memory to simulate message passing, an additional delay is introduced into the execution, which manifests itself as the difference we see here. Taking $n=2^{26}$, we see that at Nodes=1:PPN=16 vs Nodes=8:PPN=2, the difference is 0.0023152828216553 seconds in favor of Nodes=1:PPN=16. However, this difference is small enough to be down to the variance between executions. At Nodes=1:PPN=8 vs Nodes=8:PPN=1, we see a difference of 0.0067808628082280 seconds, at Nodes=1:PPN=4 vs Nodes=4:PPN=1, we see a difference of 0.0181307792663570 seconds, and at Nodes=1:PPN=2 vs Nodes=2:PPN=1, we see the largest difference yet of 0.0258207321166990 seconds all in favor of Nodes=P:PPN=1.

From this analysis, we can extrapolate that as the precision term becomes larger, the gap in performance between assigning cores on individual nodes vs assigning them to the same node would increase. This increase, however, can be mitigated to some extent by adding more processors to the computation, as we can see that the delta between the runtimes decreases as the number of nodes increases.

VI. Conclusion

In this experiment, we created two programs that calculate pi using a series of summations in parallel. The first program uses a hypercube reduction methodology that we implemented to perform the final reduction step at processor 0, and the second program used the *MPI_Reduce* native function provided in MPICH. We ran the two programs for varying parameters on Nodes, Processors per node and the precision term n, and collected the runtimes of the programs in total and of only the reduction operation.

We then performed a theoretical analysis of our Hypercube reduction algortihm, and created plots of theoretical values that were the expected performance trend of our program. We compared these plots against the empirical values that we collected for our hypercube algorithm and found that our program matches the performance of our theoretical analysis, which indicates that our program is working as expected, and that our theoretical analysis is correct.

We then compared the plots of the MPI_Reduce program with the Hypercube reduction program, and found that the runtime of both the programs is near identical. This indicates that under the hood, the MPI_Reduce function uses a similar methodology of performing reductions. We also noted that while increasing the precision term should be giving us better accuracy, this is not the case for both implementations. From our in-class discussions, this is due to the fact that at higher precisions, there are more terms to add to the summation, which causes the increasingly smaller terms towards the end of the calculation to be lost in the addition.

We also compared the effect of assigning processors on one node versus assigning them to individual nodes. Interestingly, at higher precision values, we can see a measurable difference in runtimes between the two approaches in favor of assigning nodes to individual processors. The probable cause for this is that on the same node, processors use shared memory to simulate message passing, which introduces delays in the execution. However, this can be mitigated to some extent by increasing the number of processors, which diminishes the difference between the assignments.