Mitra Raman (mitrar) & Yingchao Liu (yingchal)
15-418, Assignment #3
February 28, 2013

Program 1: OpenMP

Initial Solutions

Our first instinct was to parallelize over the start cities, making each tree run synchronously. By storing each tree's best solution in an array of solutions, we would then iterate through the array sequentially to find the best solution overall. This approach required storing multiple solutions in an array data structure as well as sequential work to find the best solution overall. We realized that this was not the optimal approach, so we then focused on updating the best solution among all the trees whenever we found it.

This next approach would remove the need to store local solutions for each tree and the sequential iteration of the array after the parallel computation of all the trees. However, since we would be updating one global solution, this approach added contention to whenever a thread needed to update the global solution to the local solution found from its tree. We added additional speedup to the serial and parallel versions of the code by pruning branches of trees even earlier, which was done by comparing the tree's local best solution to the global solution in the serial code itself. If the local solution of the current branch already surpassed the global solution's bound, which could have been calculated from a different tree or branch, then the serial program pruned the branch. Otherwise, it continued with the serial program until there were no more cities to be visited. When there were no more unvisited cities, the program would check if the final bound was less than the global solution and update the global solution accordingly. We placed a "critical" statement at this point to ensure that only one thread updated the global solution at any time.

Although this approach greatly increased our speedup, we realized that it was possible to increase it even more by parallelizing not only over start cities' trees but also over subtrees. We wanted to increase the number of tasks to be done and decrease the number of work for each task to improve the work imbalance. Currently, each thread was assigned to a start city tree, and each tree could be of varying work depending on how early on it was pruned - this could lead to some threads working the entire time while others remained idle after finishing up their tree. So, we wanted to split the trees into subtrees and allow threads to work on multiple subtrees so that more threads would be kept busy for longer time.

Our final algorithm followed this approach, but we manipulated our implementation to run faster and increase the speedup. Initially, we recursively iterate through each level of the tree within a for loop; so when we started subtrees at the second child, we had three nested for loops for the roots, the first children, and the second children. However, the nested for loops slowed down our algorithm because OpenMP does not handle the distribution of threads in nested loops well even if we parallelize the for loops. So, we changed our implementation to only contain one for loop that we dynamically ran in parallel.

Final Algorithm

Our final algorithm involves iterating from 0 to the number of tasks, which we calculated as the total number of subtrees we would solve for. We calculate the parent city, first child city, second child city, and third child city by simple mathematics based on the task ID, and then run the serial code on subtrees starting from the fourth children of each tree. These trees are smaller by four levels, and can also be pruned earlier because they must include the distances from the root, first child, second child, and third child. So, the subtrees will each be executed in parallel and the threads will not be assigned to one tree root. There will be less work imbalance, because the trees are smaller so each thread can compute more trees, and less contention because we are not running the serial code at lower levels of the trees so we are not comparing or updating the global solution at those levels, either. Also, since we are running all threads in only one loop, OpenMP can dynamically allocate each thread to the subtrees.

Experimentation

To reach our final solution, we first experimented with the placement of the “critical” section containing the global solution update. We were debating whether to update the global solution after computing a subtree or after completing a branch, and decided it benefited the speedup best when we updated the global solution after completing a branch.

Secondly, we experimented with how many levels of each tree compute in the parallel code before sending a subtree to the serial code. Additionally, we experimented with parallelizing different levels of the trees versus sequential execution. We found varying results, but none with a large enough impact to make an obvious decision. Our final algorithm contains parallelization across the roots and the first children of the trees, and the second children are computed sequentially.

Another approach we wanted to take but did not find sufficient reason to was to create subtrees at the second children only if there were more processors than current trees to compute. The reason for this additional method was to ensure that a fewer number of start cities would be split up into enough work to keep all the processors busy, and if there were more than enough start cities than processors there was no need to create more work because the processors were all already busy. However, this approach did not have much, or any, improvement from our previous implementation so we did not include it in our final algorithm.

To reach the final algorithm, we struggled with whether to include nested for loops or not and with how many levels of the tree to recurse before running the serial code. After experimenting with the nested for loops and discussing with TAs, we realized that OpenMP was not able to handle parallelized nested for loops well and was not dynamically allocating the threads as we had hoped. So, we decided to declare the number of subtrees and levels to iterate through and calculate city indices from the subtree index (which is iterated by the for loop). With these indices we were able to switch the order of unvisited nodes in the unvisited array to show our path so far, and then run the serial code on the subtree. This approach was not difficult to

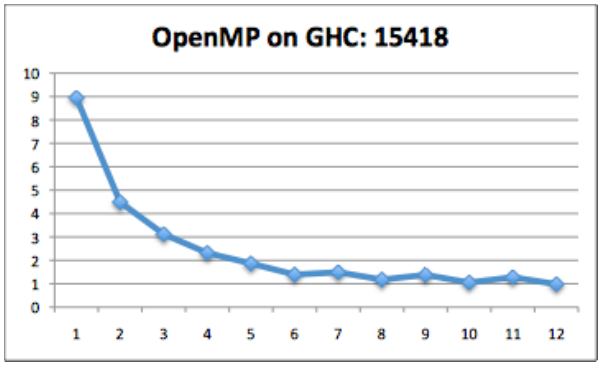
calculate or come up with, but it was a different method of implementing our solution that was not apparent to us until we were not sure how else to increase the speedup.

Execution Time and Speedup

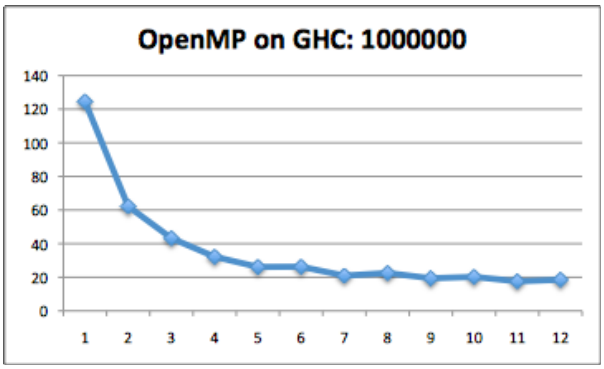
OpenMP on GHC (Number of Virtual Cores vs. Test Scripts)

	15418	**1000000**	**3735928559**	**111111111**	**11**
1	8.95	124.54	164.34	359.614	393.444
2	4.4886	62.1893	81.7968	179.349	197.3289
3	3.12	43.34	57.15	125.19	136.857
4	3.12	32.2003	42.4101	122.703	132.635
5	1.867	26.145	34.267	75.021	82.242
6	1.3908	26.1655	27.4175	51.4852	76.4589
7	1.492	20.952	33.854	60.882	69.283
8	1.1777	22.5237	23.4776	43.7658	65.8738
9	1.379	19.391	25.317	55.262	60.206
10	1.0534	20.2287	20.743	41.6017	64.4696
11	1.279	17.569	23.175	50.229	55.862
12	0.9822	18.5435	19.2501	36.3958	53.1996

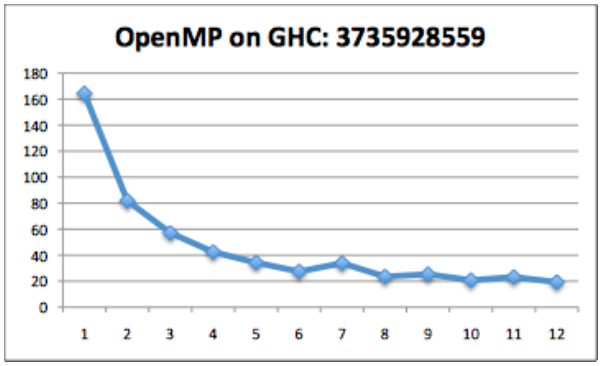
X axis = Number of Virtual Cores
Y axis = Time (in seconds)



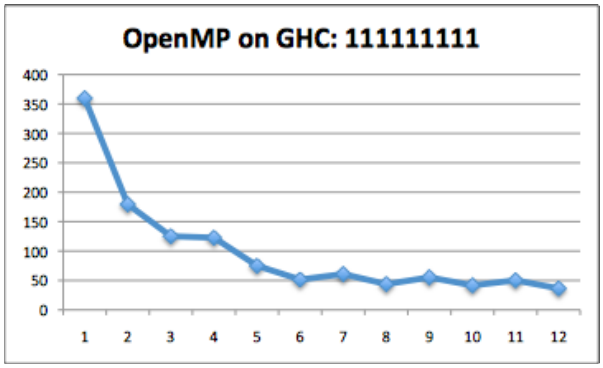
Speedup: 9.14x on 12 cores



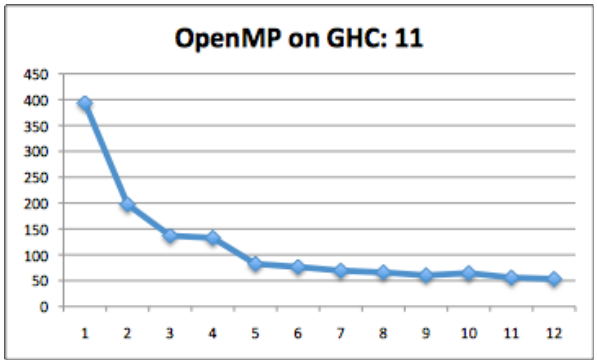
Speedup: 6.716x on 12 cores



Speedup: 8.537x on 12 cores



Speedup: 9.88x on 12 cores

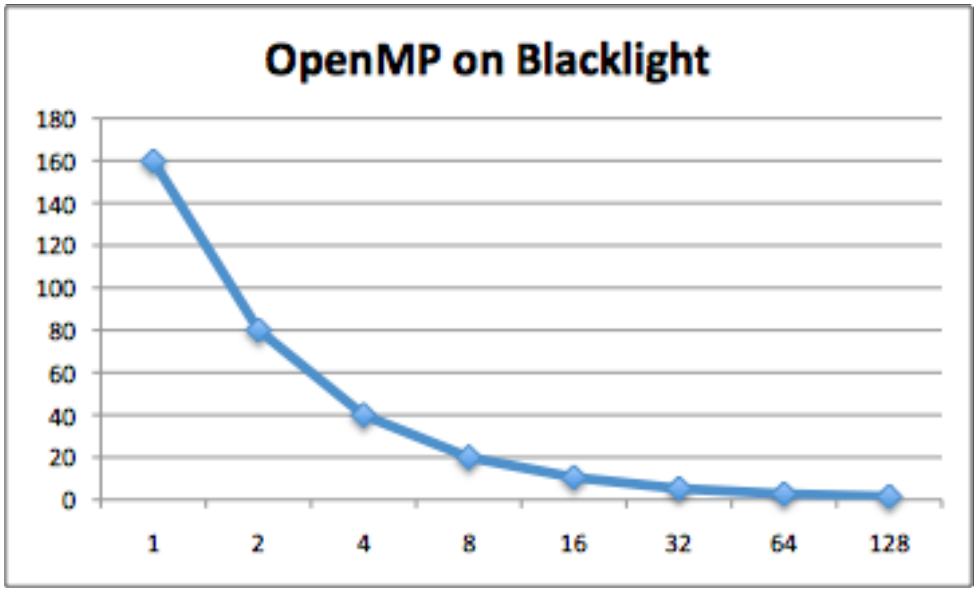


Speedup: 7.396x on 12 cores

OpenMP on Blacklight

Number of Processors	Time (seconds)
1	159.784
2	80.079
4	39.926
8	19.997
16	10.503
32	5.214
64	2.674
128	1.444

X axis = Number of Processors
Y axis = Time (in seconds)



Speedup: 110.96x on 128 processors

Results

Our program currently achieves close to perfect speedup using six threads (depending on the machine in GHC). However, the speedup is not perfect when run on more than 6 threads. These less than ideal results could be due to overhead costs from reinitializing the unvisited array for every path of the tree that we compute before running the serial code. Another source of overhead costs could be from updating the general solution in the serial code whenever we find an appropriate solution that is less than the current bound. Since the general solution is shared among all threads, the update of the solution is in a critical section which even stalls the threads not updating the general solution.

Program 2: OpenMPI

Initial Solutions

Initially, we planned to use static mapping through MPI and apply the same trick from our OpenMP solution which was to split up the work into subtrees and make the workload evenly distributed among processors. In this approach we would create subtrees by traversing down paths from the root of a tree to the first and second children through parallel for loops, and each path would be statically assigned to a processor. However, we found out that it is really hard to statically map work to each processor since there are no shared variables among threads. Additionally, statically assigning work would result in much more communication and message passing between threads than was necessary.

So, we switched our approach for the solution of MPI. The MPI standard is more suitable to work as distributed system model, since different processors communicate with each other through messages. We ended up with the master-slave model where we use one processor as a

master node, dedicated to splitting up the work and giving the work to the slaves. The master also keeps track of the best solution of the overall tree. The other processors then work as slaves to finish the work given to them by the master. If there is only one processor running the program, then we run the sequential algorithm on this one processor and abandon the master-slave model.

Final Algorithm

When the program starts, we run processor 0 as the master, and the other processors as slaves. The master receives the requests from slaves and sends different responses based on the tags. If the tag is GET_TREE_TAG and the master has work in the queue, the master then sends a message with a subtree to work on to the slave. If there is no more work, the master sends back a message to the processor with a DIE_TAG to let indicate that this process will die. If the tag from the processor to the master is PUT_BEST_SOLUTION, then the master compares the processor's solution to the global best solution and updates it accordingly. On the slave side, it keeps sending requests to the master to ask for work until it receives the die tag. When slaves work on a subtree, if it finds a better solution it sends the solution back to the master.

The program returns when all of the slaves get the die tag from the master and the master also returns.

Experimentation

At the start of the program, every MPI call is blocking the send and receive messages, which causes a lot of unnecessary latency between processors. For example, when one processor finishes one path, it tries to send the best solution to the master. This sending can be done non-synchronously, since the slaves do not need to make sure the message is received by the master at the current stage because there is a gap between this and the time they need to modify the sending buffer again. Actually, they only need to check to make sure that the previous message is received by the master. During this checking gap, the slaves can continue working on another path, which saves a lot of time in performance. The same trick applies to the master. When the master sends work to slaves, it is not necessary for the master to wait for confirmation from the slaves that they received the message . By unblocking the send message, the master is able to perform other tasks such as updating the global best solution from the solutions received from other slave processors.

In this model, the other thing we tried is to let the master split up the work based on different task IDs and send the actual work as solution type to each slave. However, we thought it might push too much work to the master and the message size would be too big, which would slow down the communication between threads. Therefore, as the number of processors increases, the master would become a bottleneck. So, instead of the master splitting work itself, it gave the task ID to the slaves and let them split down to the levels. This approach also reduced the size

of the message that is passed around to as small as possible. Initially, we used a mpi_sulotion_type which contains one integer and a buffer of size MAX_N. The integer field is to store the best distance so far. So when the master sends the work back to the slaves, the slaves can get the best current distance, and use it to prune trees when it goes into the serial code.

The second thing we were trying to figure out was the trade-off to reduce the communication cost between the master and slaves. One of the benefits from the communication is that if the best distance can be sent back to the master in time, a lot of tree branches can be pruned. However, the communication for the update can potentially make the master too busy to assign work to the waiting slaves. It turns out that the way of communication between the master and slaves is not efficient enough. To largely reduce the communication cost, we tried to only update the best solution when a slave finishes an entire tree, which turns out to have really bad speedup since there is no information in time to prune trees more aggressively. We then placed this updating operation inside the serial code where each slave tries to update the best solution after finishing an entire path. This gives us a decent speedup compared with the previous approach.

So far, communication benefits speedup more than negatively affect the cost. In fact, we think our speedup is largely limited because we cannot make the communication between the master and slaves and slave to slave more efficient. We observed that the serial code occupies a large amount of time in each slave processor compared to the time that they use to communicate with the master. We think this is caused by lack of communication inside the serial code. If the slave can receive more real-time information when it is working on a branch, it would get more restrictive value to prune a tree aggressively, which reduces the time that the slaves stay in the serial code.

However, it seems hard to receive more information inside the serial code. It is certain that asynchronous receives should be used in this case, otherwise it is very easy to get into deadlock. But, when we tried to put an asynchronous receive call right before we check the if the current branch is prunable, it still causes deadlock and we could not figure out the reason.

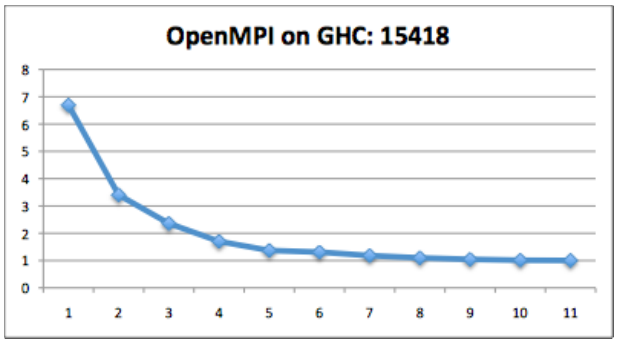
Execution Time and Speedup

OpenMPI on GHC (Number of virtual cores vs. Test Scripts)

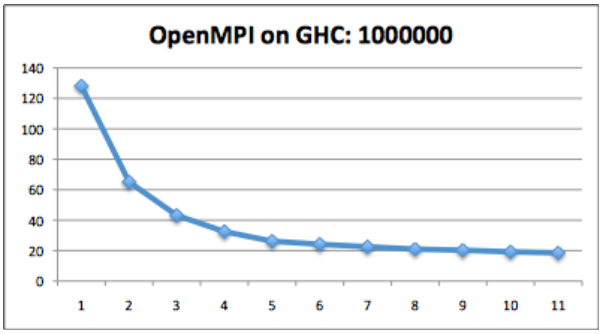
	15418	**1000000**	**3735928559**	**111111111**	**11**
1	6.5111	126.4164	135.4681	235.3703	361.0659
2	6.6945	128.08	127.6192	239.2699	367.2133
3	3.4034	65.0802	66.2424	124.006	188.6132
4	2.3599	43.1338	44.3860	83.3454	125.8645

5	1.7020	32.4910	33.5285	63.0576	94.4328
6	1.3703	26.1759	27.0058	51.2171	76.8041
7	1.3091	24.1537	25.3585	48.5335	69.8822
8	1.1819	22.5265	23.0311	43.1764	68.9143
9	1.0982	20.9741	21.8122	42.0215	60.3874
10	1.0453	20.2549	20.7481	39.4809	57.3905
11	1.0121	19.2018	19.9489	37.4789	55.0147
12	1.0027	18.4895	19.1686	36.3726	53.4825

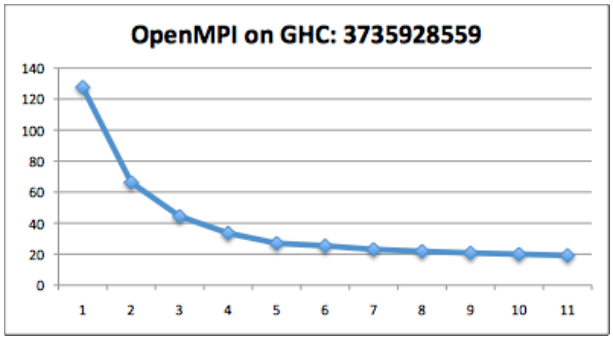
X axis: Number of virtual cores
Y axis: Time (seconds)



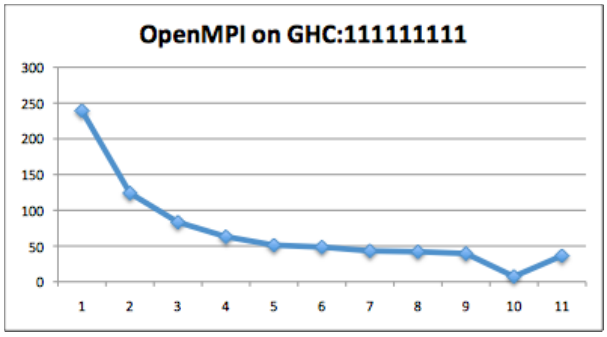
Speedup: 6.494x on 12 cores



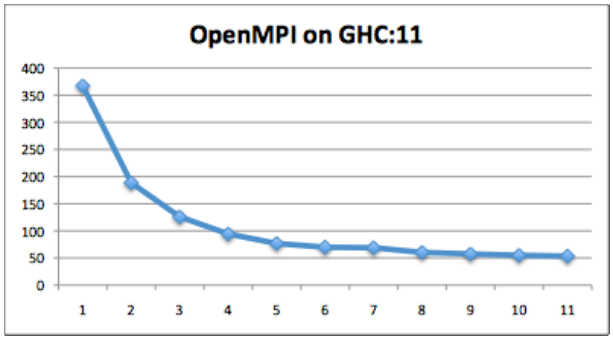
Speedup: 6.837x on 12 cores



Speedup: 7.067x on 12 cores



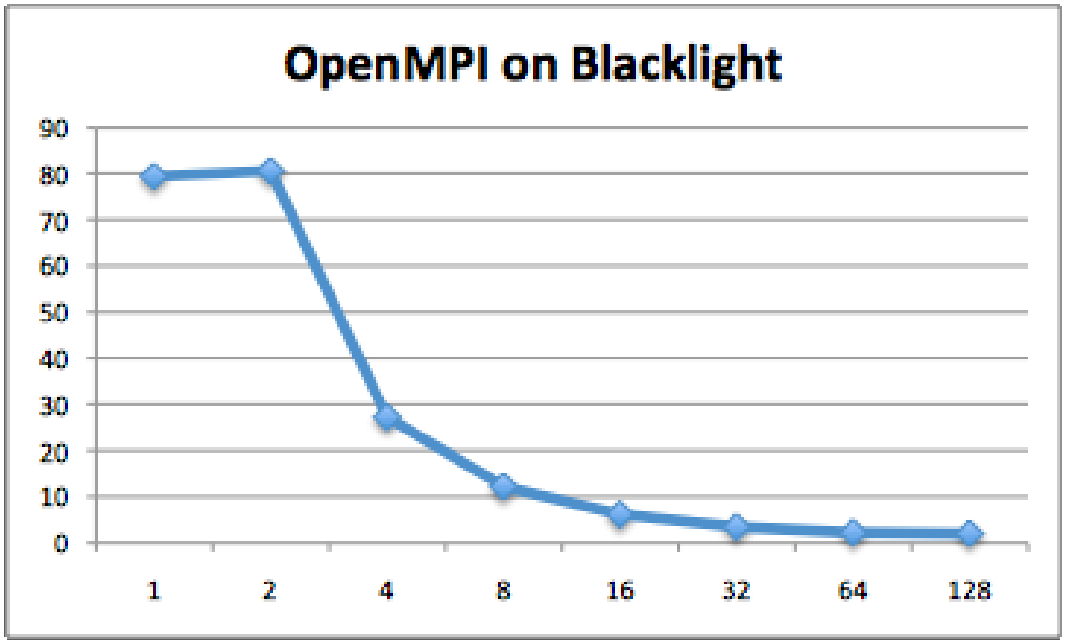
Speedup: 6.471x on 12 cores



Speedup: 6.751x on 12 cores

OpenMPI on Blacklight

Number of Processors	Time (seconds)
1	79.41
2	80.596
4	27.209
8	12.178
16	6.141
32	3.422
64	2.219
128	1.985



Speedup: 40x on 128 processors

Results

Our program currently achieves 6.5X speedup on 12 cores and 40X speedup on blacklight with 128 cores. We also tried to run our program on more than 20 GHC machines with 128 cores, and the highest it can achieve is around 70X speedup. We think reason that our program is more scalable on multiple ghc machines than blacklight is that the GHC machines are connected via ethernet, which is slow. But the computation speed is much faster relative to the ethernet speed, so the master is not really burdened by huge number of requests from the slaves. On the contrary, the communication between blacklight cores is much faster and the master, to some extent, quickly becomes a bottleneck in this program. Besides, we did not figure out a way to let slaves receive more up-to-date information inside the serial code to efficiently cut the work, and the work is not distributed evenly, which might cause some slaves to be super busy while the others are idle.