Design Document: Branch and Bound

Overall Design Ethos:

Rather than approaching this problem as specific to the TSP case. We tried to continue generalizing the API and overall design to be applicable to other problems. We therefore try to expose functionality that allows for efficient computations, but leave it up to the application programmer to decide what features they wish to exploit. Through constructor parameters and the overriding of methods as needed we minimize the amount of system specific code in the implementation while allowing the application programmer control of the performance of their application. The application programmer has complete control over how their program decomposes and composes, the priority of a given task and where it should be run (the Space, or a ComputeNode), and allow a custom shared global state if they need it.

New Interface: Shared State

To facilitate a common variable among all the ComputeThreads we introduced a SharedState interface. The interface mandates a single **update()** method that takes in a revised SharedState and returns the original if no update needs to be made, the revised state if the new one is in every way better, or a new state entirely if the optimal sate is some combination of the two. When the SharedState object changes it is passed to compute threads on the Space. We allow the swapping of state objects with different implementations rather than having a single state object that takes on new values.

TSP makes use of this SharedState by implementing StateTSP which stores the best upper bound found so far. Each time a new TSP walk is found the update method is called, if the new bound is lower the state is updated to the new bound and propagated to all the other compute threads of the Space.

Changes: Space

The space interface is slightly altered. The **setTask**() method that is used to assign a problem to the space now takes an initial state parameter. This parameter is propagated to all the computers before the task is dispatched. If a problem does not benefit from a shared state a StateBlank object can be passed to the Space (which does not contain any variables, or preform any work). In the case of TSP we create a SpaceTSP object that sets an upper bound that is calculated by a greedy search, this helps remove a large chunk of false leads when doing branch and bound.

The other change in the space is an **updateState()** method which computers call to propagate a new state to the space and the other computers. The state keeps its own state. When it receives an updateState() request it calls **update()** on the stored state to check if the new state is better. If a new state is generated it sends that to all the computers (except the one that originated the request).

Changes: Space – Scheduling

To improve the performance of applications we reworked how tasks are scheduled. Self-identified short running tasks are now exclusively executed on the space. Tasks may now self-report a priority. Tasks are dispatched to the computers based on their priority by way of a priority queue.

For TSP this is an important change as a DFS allows us to improve the upper bound of the search and not schedule many of the tasks whose lower bounds would have exceeded this. To do this TSP Tasks identify priority based on their depth in the DAG which they get from the length of the number of nodes in their input that have a fixed order. This prioritizes scheduling nodes that expand deeper into the tree before nodes that would decompose and generate more tasks.

Changes: Computer

Computers now have to be aware of the space they are running on in order for them to notify the space that they wish to update the SharedState on the space. This is accomplished via an **assignSpace()** method that is called once a computer is registered with a space. A Computer now has an **updateState**() method with a state and **force** parameter that is called by the space to update the local state. If the **force** parameter is set the computer always replaces its current state (this is useful when a brand new task is assigned to the space and memory of a previous run needs to be cleared away) otherwise the Computer compares the new state from the space to its local state and updates as necessary.

Changes: Task

Tasks now have to report their priority via a **getPriority()** method to facilitate scheduling. When a task starts execution it's **call()** method now takes a state parameter which the computer can use to pass in the current SharedState. If at any point this state changes, the Computers can call the task's **updateState()** to update the task as to the new state. If the task wishes to update the state, the call method also takes in an **UpdateStateCallback** which tasks can use to notify the computer they are executing on that they wish to update the Computer's state, which in turn is propagated to the space and then the other computers.

Changes: TaskTsp

The API changes allow a much better implementation of TSP using branch and bound. Each TSP task now takes a **fixedCititiesLength** parameter as an input. This corresponds to the lower bound of the full partial tour that this branch has taken (including the cost to return) and is updated in constant time at each level in the tree. At the start of execution tasks check to see if this lower bound exceeds the global upper bound in **StateTsp**, if it does it immediately returns an infinite value and does not decompose further.

Once the permutation length reaches a size where it is sufficiently broken up to take advantage of paralleling it but where the latency costs do not exceed the computation time of a single machine (we found this to be around 13 cities for branch and bound, and 10 cities for unbound) we stop the decomposition and run local computation.

In our previous implementation we were using a permutation generator to loop through the permutations and calculating costs across them. To take advantage of the bounds we rewrote the local computation to use recursion for the DFS search. At each recursive call the local computation continues to build on the lower bound of the task decomposition, and if at any point it exceeds the global upper bound that branch terminates and continues the DFS. Every time a possible tour has been found the TaskTsp calls the **UpdateStateCallback** to notify the computer it has found an upper bound. If that upper bound is better than the existing upper bound it is propagated to the other threads, space, and in turn other computers.