Parallel Branch and Bound Skeleton

Using Shared Memory

Supervised by Murray Cole

The Project Purpose

There are many combinatorial optimization problems in the world considering finding the optimal solution from finite but usually very large feasible solutions. To solve such problems, Branch and Bound (B&B) algorithms are frequently used. B&B algorithms try to avoid examining all solutions for a specified problem by discarding some obvious “bad” solutions. The problem space is divided into subspaces (branch) and a bound function is performed on each such subspace to determine whether to divide this subspace or not.

Though B&B algorithm has shorten the search of the optimization problem for optimal solution, it is still impossible to solve a large optimization problem within reasonable time, and the time is exponentially increasing while problem grows larger. So it is needed to parallelize the B&B algorithms to improve the performance. Also, the B&B algorithm itself is naturally parallelizable because the exploration of individual subspace is independent. And this is perfect for parallelization by letting different processors work on different subspaces.

Implementing a parallel B&B algorithm to solve a particular problem can be complicated. As the programmer not only needs to deal with the sequential B&B component such as branching, but also needs to deal with the parallel components and some optimization strategies (such as loading balance). This complexity may easily leads to high rate of coding error and performance inefficiency.

When solving different combinatorial optimization problems using the Branch and Bound algorithms, it is noticed that there are many common operations repeatedly being used. It is desirable to reuse these common operations for different applications. Such operation set is the core of the B&B algorithms and forms a template that can apply to different applications providing some problem-specific information. Therefore, these commonly used operations or templates can be developed as skeletons or framework in order to enable reusing and simplify the implementation process. User will only need to specify the problem-specific issues such as data representation and bound function, and leave the rest to be handled by the framework system.

Project Goals & Background

This project intends to capture the common operations needed in typical Branch and Bound algorithms, and encapsulate them into a basic B&B skeleton. When a problem is described to the system, the system should be able to solve the problem by applying
Informatics Research Proposal  
Parallel B&B skeleton  
Yang Xu  
S0455739

the B&B skeleton on the input information.

The system should be able to:

1. Use Branch and Bound algorithm to solve typical combinatorial optimization problems (e.g. Traveling Salesman Problem) in parallel correctly.
2. Simplify the implementation work for the programmers and is easy to use.
3. Solve the problem efficiently

As the programmers no longer need to deal with the complexity of parallel B&B algorithms and the skeleton is reusable for different applications, the programming errors will be significantly reduced. And because the skeleton part is reusable, the implementation work is simplified to only dealing with the problems representation. The representation includes several parts such as bound function, branch method and application data. This information will be passed into the system as parameters through interface. All these would make the implementation of parallel B&B much easier and make the system easy to use.

There have been many strategies developed aiming to optimize the performance of parallel B&B algorithms. Our B&B skeleton will try to express some of these new strategies in the implementation and thus make the skeleton efficient. The performance is expected to be comparable to the stand alone parallel B&B algorithms and existing frameworks (skeletons).

Many efforts have been made to capture the generality of parallel B&B algorithms as skeletons or frameworks. Roman Batoukov and Tor Sorevik have developed “A Generic Parallel Branch and Bound Environment” [1] in 1999. This framework is based on a network of workstations and aims to design a low communication cost, branch and bound algorithm with a coarse grain parallelism and an efficient dynamic load balancing on scheme. Also the authors try to ease the programming burden by creating a generic framework for a class of problems. Herbert Kuchen [2] has developed a skeleton library which includes a branch and bound skeleton. The skeleton is implemented based on the message passing model of parallelism. The aim of the library is also to simplify the parallel programming by offering typical efficiently implemented patterns.

**Methods and Implementation Issues**

As we mentioned in the previous section, there are several existing skeletons for the parallel B&B algorithms. Most of them aim to implement the skeleton based on the message passing model. This project intends to develop a B&B skeleton using a threaded (i.e. shared memory) model of parallelism. Java is used to implement this skeleton. In this section, some implementation issues will be discussed.
**Basic architecture**

![Figure 1: Basic architecture of the B&B framework](image)

The basic architecture consists of an interface, an initialization component and a typical B&B pool-worker model. The problem specific information is inputted through the interface and the initialization component starts the skeleton to compute the problem. Workers fetch sub-problems from the work pool and put newly generated ones in. When the computation finishes, the solution will reside in the work pool (or an output mechanism will print it out).

**Interface**

The interface allows user to specify the problem to be solved as well as some configuration parameters. The problem specific information includes the branch method, bound function and data structure, so that the branch method can divide the data structure into sub-structures and the bound function can compute the lower bound value of a given data structure.

Configuration parameters relate to how the skeleton will be running. User will be able to specify the number of workers to be used, the number of work pools (we will discuss this issue later) and other strategy choices related to multiple pools.

The implementation issues here include how to represent the input and how to make these inputs available to the system.

**Initialization**

Based on the user input, in this phase we initialize a number of workers and the work pool. The branch method and the bound function are passed to each worker and the problem (data representation) is put into the pool together with its bound value (computed using the bound function). Also an initial current best solution (CBS) is put into the pool. This initial value can be simply set to infinite or, we could use a heuristic method to obtain a “good” solution so that the search space can be significantly reduced.

If multiple work pools are used, this initialization should be expanded to have more functionality, which we will discuss this later.
The implementation issues here include how to handle the trade-off of using a heuristic since the heuristic itself is time consuming.

**Search strategies**
In the B&B algorithms, the search strategy determines how to select next sub-problem. There are three basic strategies: Breadth First Search (BFS), Best First Search (BeFS) and Depth First Search (DFS).

The BFS examines almost all possible solutions, therefore hardly shorten the search, it is normally omitted.

The BeFS may soon lead to a better solution and updating the current best solution, thus shorten the search. But also it could lead to a breadth first search if nodes in the same level of search tree have lower lower-bounds than nodes in the next level.

The DFS always explores the node (sub-problem) with largest level in the search tree. In this system we will use a combination of BeFS and DFS as the search strategy. The primary strategy is DFS and when selecting nodes from same level, BeSF is employed to get the best one (based on their bound values).

The implementation issue here is how to implement a priority queue in the work pool.

**Work pool**
Work pool is used to store unexamined sub-problems. Also it needs to maintain a current best solution (CBS). This solution is updated when one worker has found a better one.

As the work pool is going to be accessed by several workers, there should be a lock mechanism so that only one worker can access and update the pool at one time.

Another issue is that the pool should be implemented to support the search strategy mentioned before. So for each sub-problem, two extra fields are added. One indicates the bound value and the other for the level of depth. The priority is given to sub-problems with larger depth level or smaller bound value if depth levels are same.

We do not want to sort all sub-problems every time when there is an update to the queue. So the implementation challenge here is how to maintain the priority queue so that workers can efficiently get (delete) or put (insert) sub-problems.

**Workers**
Each worker has a copy of CBS. This CBS may be updated by a better solution during processing a sub-problem. This update should be made visible to all other workers. So each time a worker accessing the pool, it compares its copy of CBS with the pool’s copy and updates these two copies using the smaller CBS.

Each worker branches a problem into sub-problems and calculates the bound values for them. Then promising sub-problems (by comparing bound value to CBS) are added into the pool with their bound values.

In order to support priority queue in the pool, the workers should also compute the depth level for each sub-problem. This is done by simply incrementing the depth level.
Multiple work pools
Obviously the single work pool is a performance bottleneck when the number of workers increases. Thus it is interesting to implement a multi-pool architecture.

![Diagram of multiple work pools]

Work distribution
The initialization phase needs to be expanded to generate initial problems for each pool. This can be done by applying the branch method based on a BeFS strategy, until there are enough initial problems for each work pool. Also the workers should be divided into groups with each attached to one pool.

Communication between pools
Two implementation challenges are how to dynamically balance the workload among different work pools during the computation and how to share the CBS to shorten the search.
We need to enable communication between pools. This can be done using periodical global synchronization or diffusive (local) exchange [3].

Termination detection
Another challenge is how to detect the termination. This can be done in a simple way by defining a global array, and let the last worker collect and print the best solution.

Evaluation
Several real applications addressing different combinatory optimization problems will be implemented using this system. They should work correctly and efficiently to
produce a best solution for given problem.

These applications will be tested using different number of workers (CPUs) and the performance is evaluated in terms of speedup and efficiency. The performance is expected to be comparable to those existing frameworks.

In a single work pool system, the contention to the pool causes performance lost, while in a multi-pool system, load imbalance and communication overhead cause performance lost. In order to explore the trade-off between these two strategies, we will compare the different performance based on single work pool or multiple work pools. Also this evaluation will be done using different number of workers, so that we can show how the number of CPU will affect the trade-off.

Many strategies aiming to improve performance can be employed in the multi-pool architecture. It would be interesting to see how these strategies will affect the performance.

**Platform**

The project will be carried out and evaluated on the 52 processor SunFire parallel machine available through the Edinburgh Parallel Computing Centre (EPCC).

The machine has a single SMP cluster which consists of 52 900 MHz UltraSPARC-III processors in a single cabinet. Each processor has 1 GB of memory associated with it.

The 52 processors are divided into front end with 4 processors and back end with 48 processors. The 48 UltraSPARC-III processors in the back end have 48 Gbyte of shared memory. The nominal peak performance of the back end is 86.4 Gflops (900MHz × 48 processors × 2 flops per cycle).

**Work Plan**

Based on the discussion of methods and implementation issues in previous section, the project can be divided into several sub tasks.

Task T1:
Identify the skeleton components and problem-specific information.

Task T2:
Specify how to represent the problem-specific information.

Task T3:
Design and implement the data structure that supports the search strategy.

Task T4:
Design and implement the pool-worker pattern (includes the implementation of worker and work pool). **Milestone 1**
Task T5: Implement the initialization phase (includes producing initial CBS using heuristic).

Task T6: Integrate system and produce the basic prototype. **Milestone 2**

Task T7: Expand the initial phase to support multi-pool architecture (to create initial problems).

Task T8: Design and implement the interaction for multi-pool architecture (work distribution, communication and termination detection). **Milestone 3**

Task T9: Design system interface to allow user specify number of workers and work pools.

Task T10: Implement the full prototype. **Milestone 4**

Task T11: Evaluation and writing up Thesis. **Milestone 5**

**References**


Figure 3: Dependencies of sub-Tasks and Milestones