A Framework for Scalable Parallel Tree Search

Yan Xu
SAS Institute

Ted Ralphs
Lehigh University

Laszlo Ladányi
IBM T.J. Watson Research Center

Matt Saltzman
Clemson University

CORS/INFORMS Joint Int'l Meeting, Banff, Alberta, Canada, Sunday, May 16, 2004

Outline

- Overview of parallel tree search
  - Tree search
  - Scalability
  - Knowledge sharing
- The Abstract Library for Parallel Search (ALPS)
  - Design overview
  - Class hierarchy
  - Knapsack solver
  - Generic MIP solver
- Future work

Tree Search

- Tree search algorithms systematically search the nodes of a directed, acyclic graph for one or more *goal nodes*.
- Generally, the graph is not known a priori, but is constructed as the algorithm progresses.
- A generic tree search algorithm consists of the following elements:
  - Processing method: Is goal achieved?
  - Search strategy: What is node priority?
  - Fathoming rule: Can node can be fathomed?
  - Branching method: What are the successors?
- Each node has an associated *description* and a *status*.
  - `candidate`: Ready to be processed.
  - `active`: Currently being processed.
  - `processed`: Successors generated (not a leaf node).
  - `fathomed`: No successors generated (a leaf node).
- The algorithm consists of choosing a candidate node, processing it, and either fathoming or branching.

Parallelizing Tree Search

- Tree search is a "divide and conquer" approach, so it is conceptually easy to parallelize.
- As the number of processors is increased, it becomes increasingly difficult to manage the solution process.
- *Scalability* measures how well a parallel system takes advantage of increased computing resources.
- Terms

Sequential runtime	T_s
Parallel runtime	T_p when using p processes
Parallel overhead	$T_o = pT_p - T_s$
Speedup	$S = T_s/T_p$
Efficiency	$E = S/p$

- Good scalability is the goal of any parallel algorithm.
- Achieving it involves a number of tradeoffs.

Knowledge Generation and Sharing

- *Knowledge* is information generated during the course of the search that guides the search.
  - Knowledge generation changes the shape of the tree dynamically.
  - The primary way in which parallel tree search algorithms differ is the way in which knowledge is shared (Trienekens '92).
- Sharing knowledge helps reduce overhead by guiding the search.
  - If all processes have "perfect knowledge," then no process will have an empty task queue and no redundant work will be performed.
  - The goal is for the parallel search to be executed in roughly the same manner as the sequential search.
- However, knowledge sharing also increases communication overhead and idle time.
- This is a fundamental tradeoff between centralization and decentralization of knowledge.

Parallel Overhead

- The main contributors to parallel overhead in tree search are
  - Communication Overhead (cost of sharing knowledge)
  - Idle Time
    - Handshaking/Synchronization (cost of sharing knowledge)
    - Task Starvation (cost of *not* sharing knowledge)
    - Ramp Up Time (cost of sharing knowledge)
    - Ramp Down Time
  - Performance of Redundant Work (cost of *not* sharing knowledge)
- Knowledge sharing is the main driver of efficiency.
- This breakdown highlights the tradeoff between centralized and decentralized knowledge storage and decision-making.

Load Balancing

- The most fundamental knowledge generated during the search are the node descriptions.
- The node descriptions represent work in the system.
- It is critical to keep this work evenly distributed, both in terms of quantity and of quality.
- Static load balancing
  - Determines the initial task distribution.
  - In dynamic search algorithms, this can be difficult.
  - The main source of ramp-up time.
- Dynamic load balancing
  - Used periodically to redistribute the tasks.
  - Critical in dynamic search algorithms.

ALPS Project

- ALPS is a framework for implementing parallel tree search developed in partnership with the COIN-OR project, IBM, and NSF.
- A number of such frameworks and solvers already exist.
  - Commercial: CPLEX, Lindo, Xpress, OSL, and SAS/OR.
  - Serial: ABACUS, MINTO, MIPO, bc-opt, MPSARX, COIN/SBB, bonsaiG.
  - Parallel
    * MIP: SYMPHONY, COIN/BCP, PARINO, FATCOP, PICO
    * Branch and Bound: BoB, PPBB-Lib, and PUBB.
- Why another one?
- Our goal is to build upon and improve previous work.
  - Provide a general framework for parallel tree search.
  - Provide a base layer upon which to build more specialized frameworks.
  - Provide support for *data-intensive* algorithms.
  - Provide improved scalability.
  - Operate effectively in sequential environments.

Global Picture

ALPS

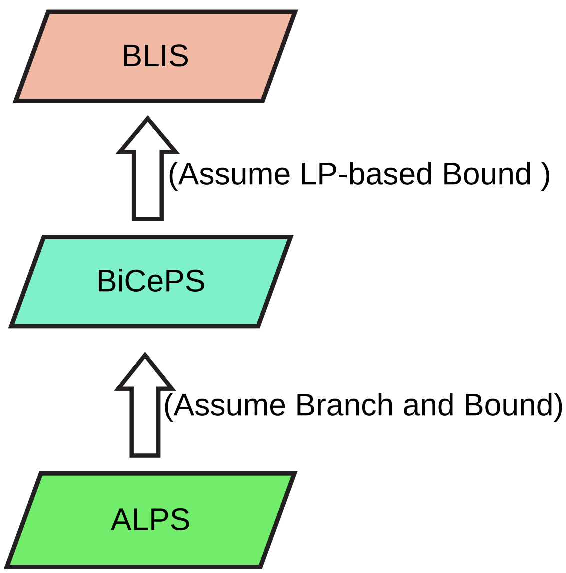
- search handling layer.
- prioritizes nodes based on quality.

BiCePS Branch, Constrain, and Price Software

- data handling layer for optimization.
- adds notion of variables and constraints.
- assumes iterative bounding process.

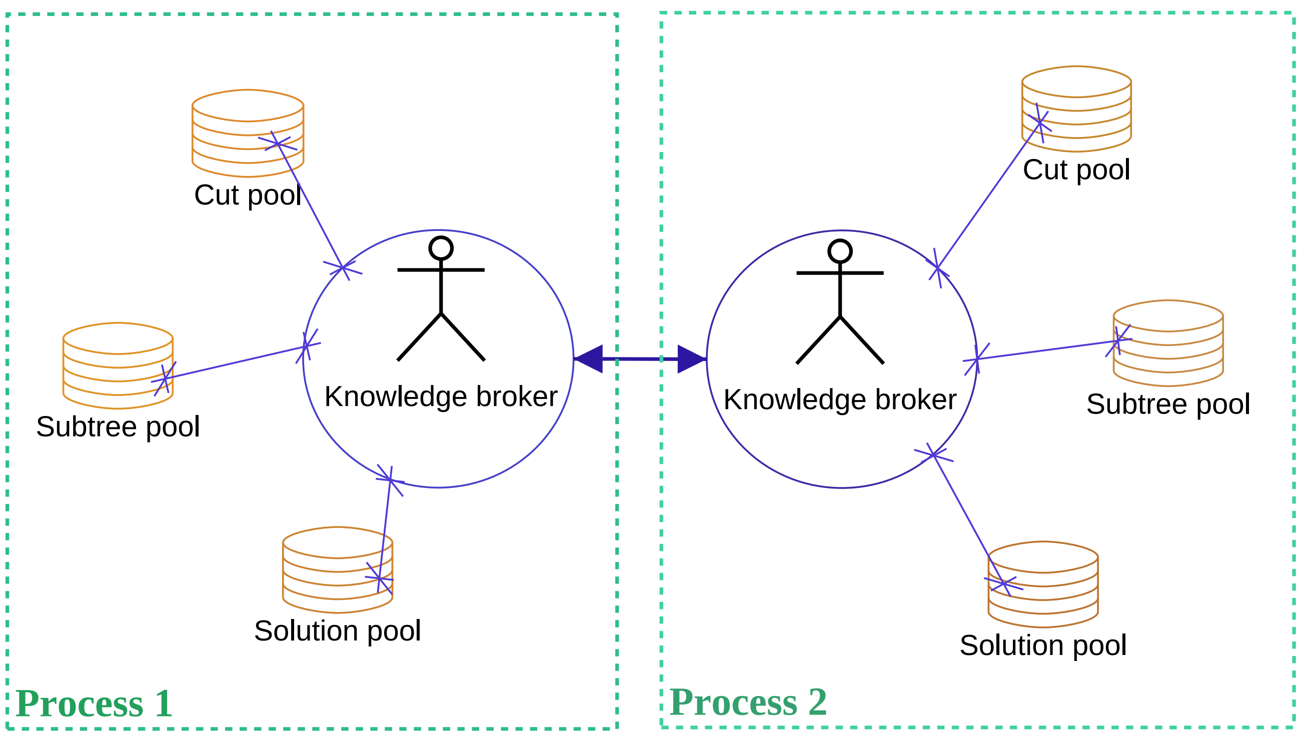
BLIS BiCePS Linear Integer Solver

- concretization of BiCePS.
- constraints are linear functions.



ALPS: Knowledge Sharing Scheme

- All knowledge to be shared is considered as AlpsKnowledge, and has its own type, e.g., AlpsSubTree, AlpsTreeNode, AlpsSolution, and AlpsModel.
- AlpsKnowledge is managed by one or more AlpsKnowledgePools.
- The knowledge pools communicate through AlpsKnowledgeBrokers.



ALPS: Knowledge Handling

- Need to deal with potentially HUGE amounts of knowledge.
- Duplication and efficient storage are a big issue.
- All knowledge has an *encoded form* that contains only the data from the class in the form of a string.
  - This representation is memory-efficient for storage.
  - This form is also appropriate for message-passing.
  - Provides a type-independent representation.
  - Allows easy detection of duplication.
- Detecting duplicate knowledge:
  1. From encoded form, obtain a hash value.
  2. Object is looked up in hash map.
  3. If it does not exist, then it is inserted.
  4. A pointer to the unique copy in the hash map is added to the list.

ALPS: Increased Granularity

- One easy way to decrease communication overhead is to increase granularity.
- In ALPS, the basic unit of work is a subtree.
- Subtrees can be stored efficiently using differencing.
- ALPS tried to keep subtrees together as a unit.
- Pros:
  - less communication.
  - more compact storage via differencing.
  - Easy to use local valid knowledge, like locally valid cuts.
- Cons:
  - more possibility of redundant work being done.

ALPS: Master-Hub-Worker paradigm

The easiest load balancing approach is a single central node pool, but this does not scale. Instead, we use the following architecture:

Master

- has global information about the system status.
- balances load between hubs (quantity *and* quality).

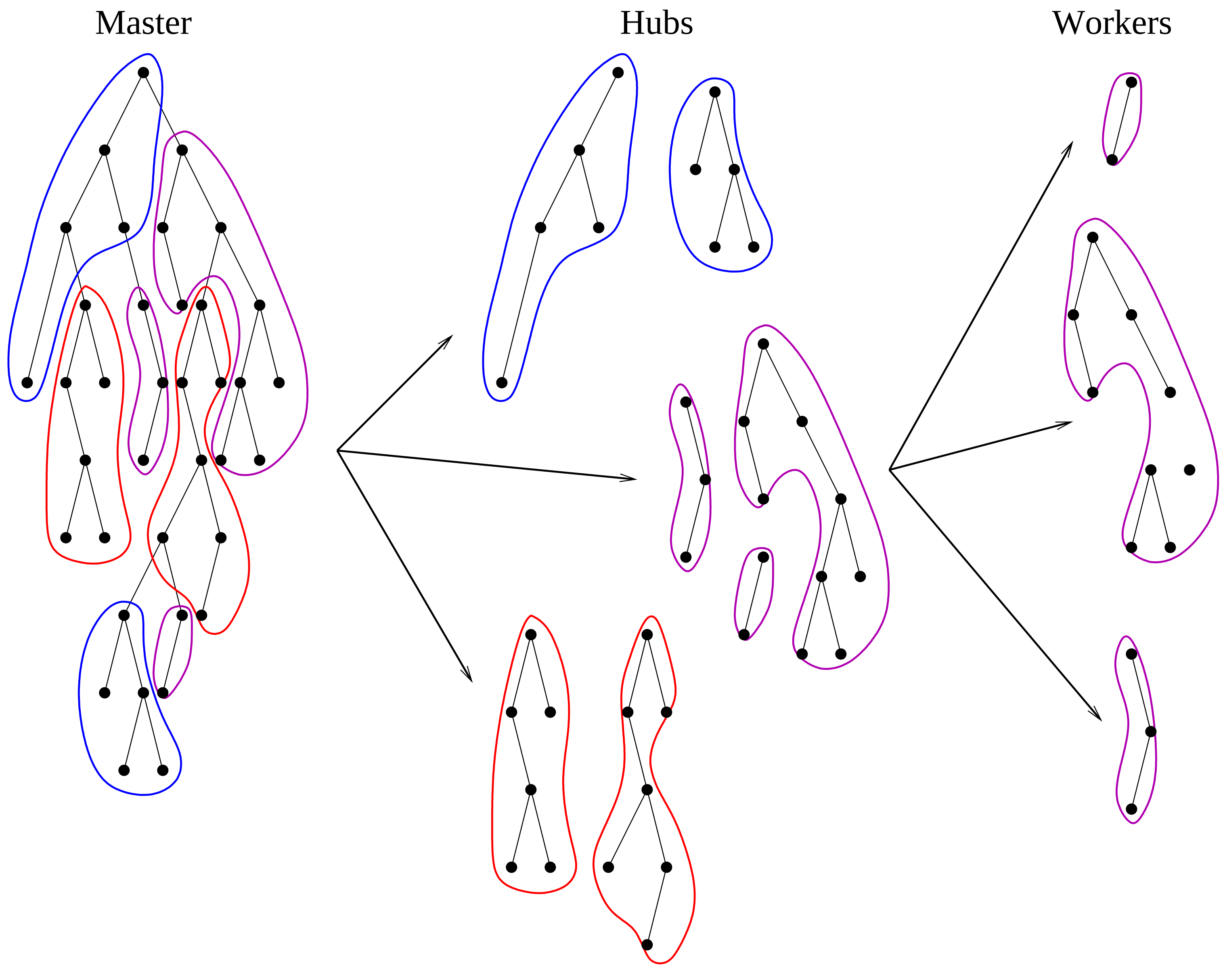
Hub

- manages a cluster of workers. Has information of its cluster status.
- balances load between its workers.

Worker

- explores subtrees.
- hub can interrupt.
- sends workload information to its hub periodically.

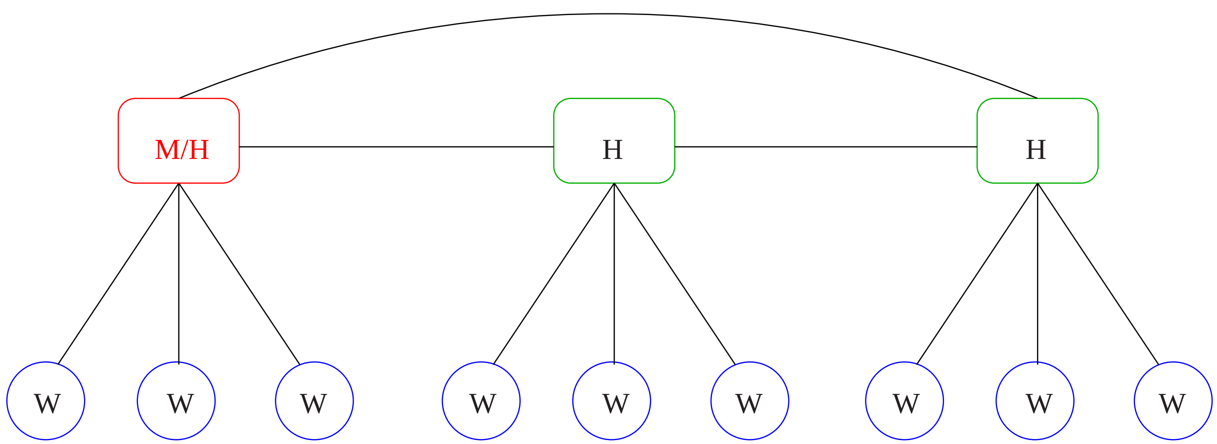
ALPS: Master-Hubs-Workers Paradigm



ALPS: Master-Hubs-Workers Paradigm

- Processes are grouped into different *clusters*.
- Each cluster has one *hub* and at least one *worker*.
- Cluster size S is the minimal integer satisfying

$$\texttt{hubNum} \times S \geq \texttt{processNum}$$



ALPS: Static and Dynamic Load Balancing

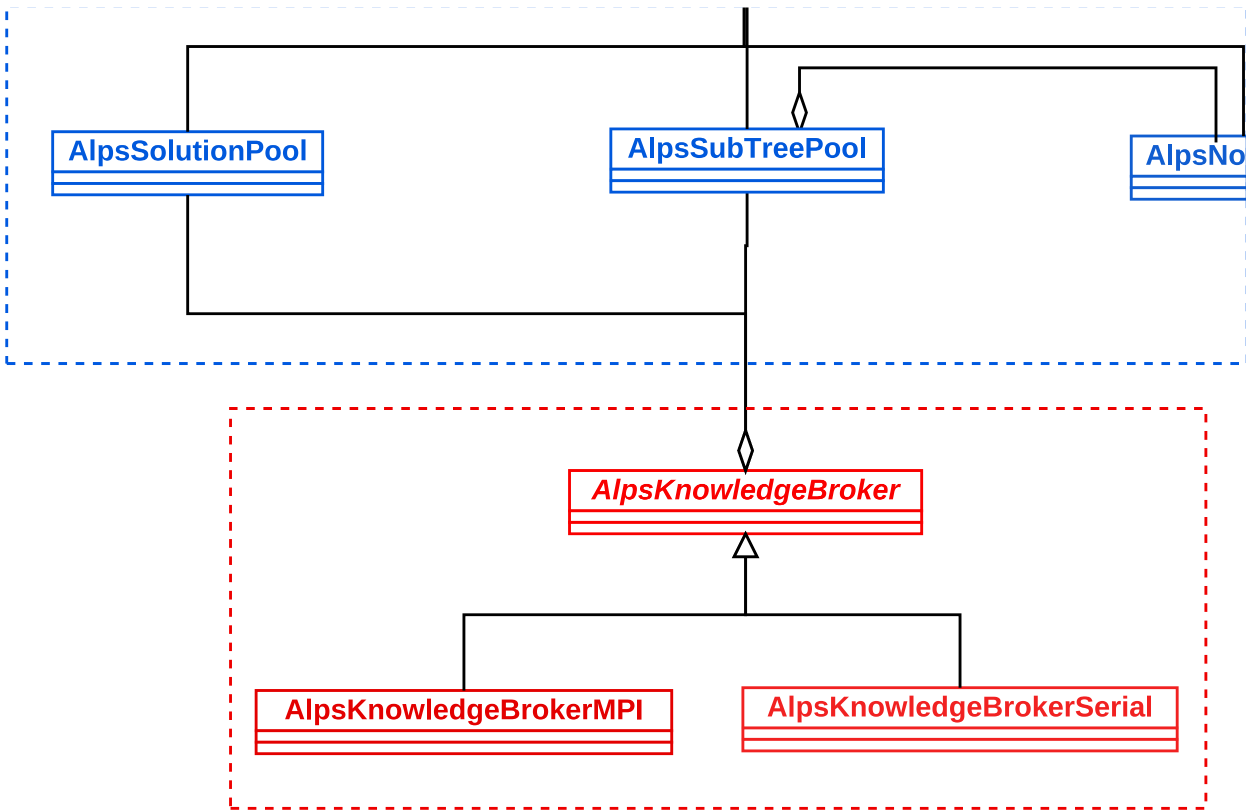
- Load balancing is used to distribute and redistribute tasks.
- Static load balancing (mapping):
  - ALPS uses *two-level root initialization*.
  - The master generates nodes for the hubs.
  - The hubs generate nodes for the workers.
  - This helps reduce ramp-up time.
  - However, it's difficult to predict the work associated with a given node.
- Dynamic load balancing:
  - Periodically redistribute the high priority work.
  - Inter-cluster dynamic load balancing
  - Intra-cluster dynamic load balancing
  - ALPS tries to pass groups of nodes together as subtrees.

ALPS: Threads and Scheduler

- Each processor hosts a knowledge broker (KB) and several knowledge pools (KPs), so there must be a scheme for multi-tasking.
- ALPS uses a simple version of threads (ala PICO).
- ALPS processes are message-driven, so the knowledge broker resident at each processor controls the scheduling after initialization.
- The KB receives external messages and forwards them to the appropriate knowledge pool.
- It also receives internal messages from the local knowledge pools and forwards them to the appropriate KB.
- In between execution of the communication threads, the KB schedules computational tasks.
- The granularity of computational tasks is controlled by parameter settings.

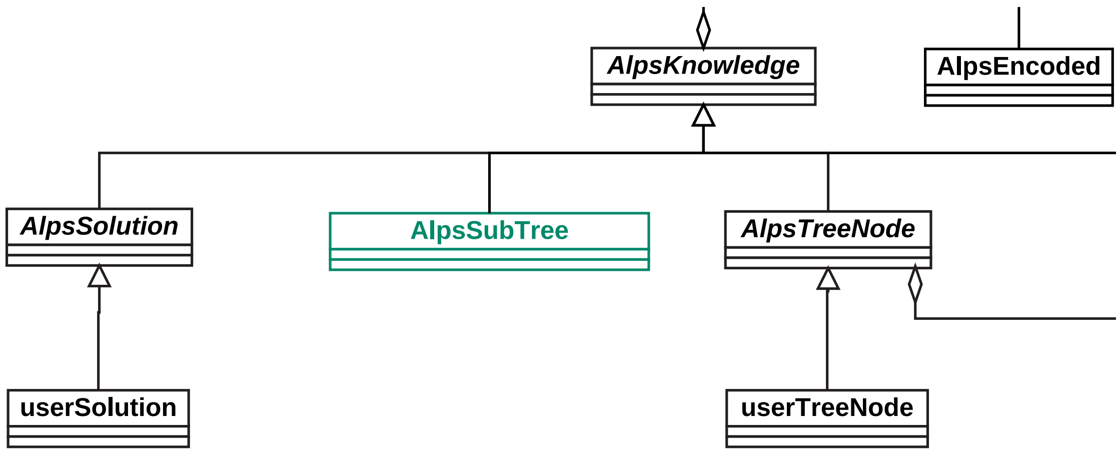
ALPS: Class Hierarchy

- AlpsKnowledgePool and AlpsKnowledgeBroker related classes.



ALPS: Class Hierarchy

- `AlpsKnowledge` related classes.



ALPS: Developing an Application

Mainly comprises two parts:

- deriving the required and auxiliary problem-specific classes
  - `AlpsModel`
  - `AlpsTreeNode`
  - `AlpsNodeDesc`
  - `AlpsSolution`
- writing the `main` function

ALPS: Example Application

```
int main(int argc, char* argv[])
{
    UserModel model;
    UserParams userPar;

#if defined(SERIAL)
    AlpsKnowledgeBrokerSerial broker(argc, argv, model, userPar);
#elif defined(PARALLEL_MPI)
    AlpsKnowledgeBrokerMPI broker(argc, argv, model, userPar);
#endif

    broker.registerClass("MODEL", new UserModel);
    broker.registerClass("SOLUTION", new UserSolution);
    broker.registerClass("NODE", new UserTreeNode);

    broker.search();
    broker.printResult();
    return 0;
}
```

Simple Knapsack Solver

- Four classes are derived from ALPS's base classes:
  - `KnapModel`,
  - `KnapTreeNode`,
  - `KnapNodeDesc`,
  - `KnapSolution`, and
  - `KnapParameterSet`
- It is a pure branch and bound algorithm without advanced techniques such as cutting planes, strong branching, reduce cost fixing, etc.
- Used to test how ALPS scales when node processing times are short.

Knapsack: Preliminary Computational Results

- Test Environment:

Machine:	Beowulf cluster with 48 dual-processor nodes
Processor:	1.0 GHz Pentium III
Memory:	512M, 4 nodes has 2G
Operating System:	Red Hat Linux 7.2
Message Passing:	LAM/MPI

- Experiment Design:
  - Randomly select *four* hard knapsack instances that generated based on the rule proposed in Martello('90)
  - Run five trials for each instance, and take the average
  - Use COIN/Sbb (without cuts, heuristics) to produce serial result.

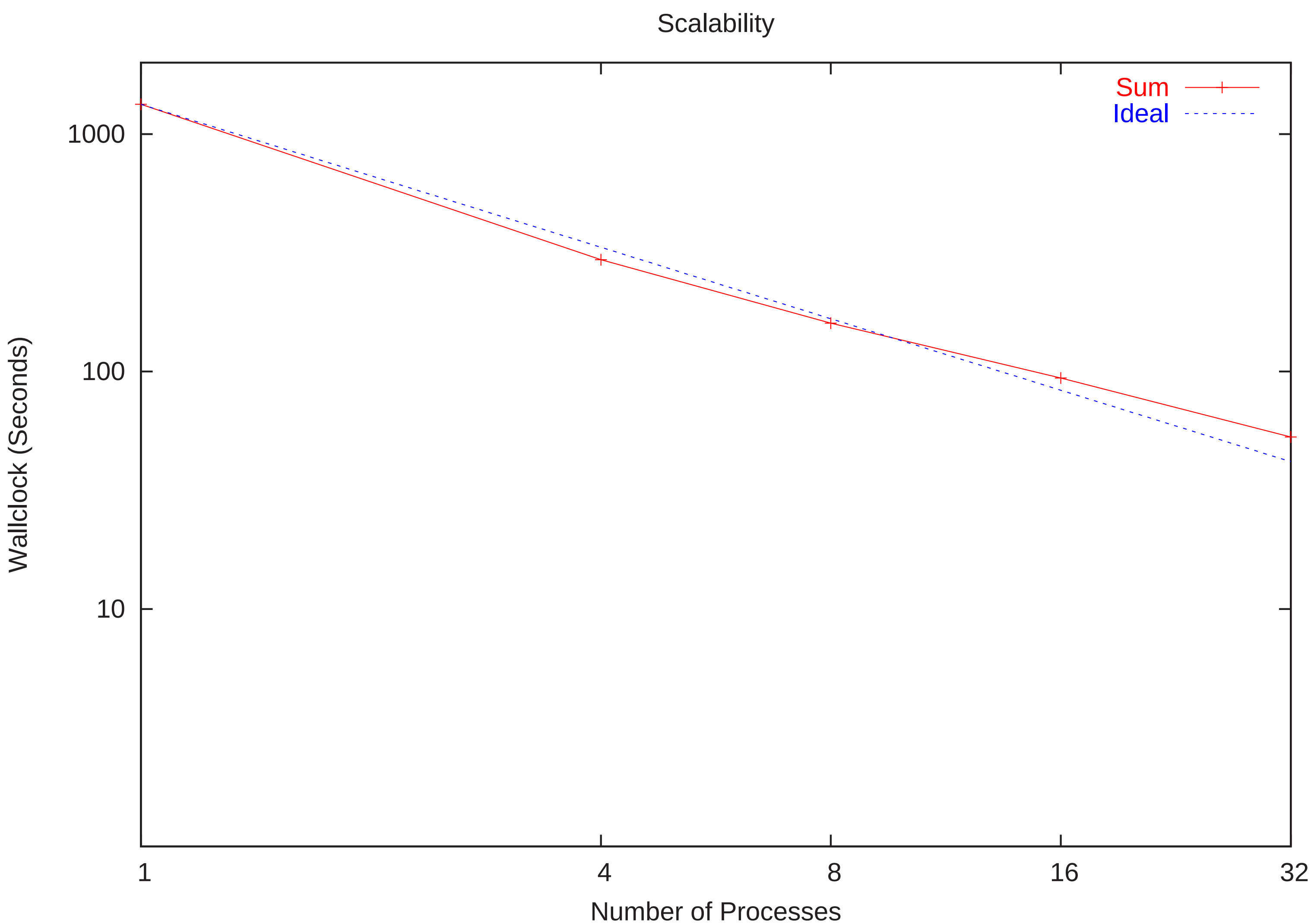
Knapsack: Preliminary Computational Results

- The four instances had similar behavior, so we present summary results.

p	Wallclock	Ramp-up	Idle	Speedup	Efficiency	nodes
1	22m15s	–	–	–	–	254,151
4	4m56s	0%	2.9%	4.5	1.13	85 m
8	2m40s	0%	2.6%	8.3	1.04	85 m
16	1m34s	0%	7.8%	14.2	0.89	85 m
32	53s	0%	7.9%	26.3	0.83	85 m

- These indicate reasonable scalability, but for a very limited test.
- For up to 32 processors, 1 hub generally has better results than using 2 or more hubs.

Knapsack: Preliminary Computational Results



Generic MIP Solver

- For an application in which node processing times are much larger, we implemented ALPS Branch and Cut (ABC).
- Consists of the classes:
  - `AbcModel`,
  - `AbcTreeNode`,
  - `AbcNodeDesc`,
  - `AbcSolution`,
  - `AbcParameterSet`, and
  - other auxiliary classes
- Use COIN/Cgl cut generators.
- Use COIN/Sbb rounding heuristic as a primal heuristic.

ABC: Preliminary Computational Results

- Tested ABC using four MIPLIB problems: *gesa3, blend2, fixnet6, cap6000*

Problem	Rows	Cols	Int Vars	Cont Vars
gesa3	1368	1152	720	504
blend2	274	353	264	89
fixnet8	478	878	378	500
cap6000	2176	6000	6000	0

- Search strategy: best-first
- Node selection: strong branching
- Test environment is the same as the previous.

ABC: Preliminary Computational Results

Problem	p	Wallclock	Ramp-up	Idle	*Speedup*	E	nodes
gesa3	1	27m6s	–	–	–	–	403
gesa3	4	10m14s	9.8%	0	2.6	0.66	445
gesa3	8	4m29s	35.1%	0.2%	6.0	0.76	337
gesa3	16	2m41s	49.1%	0.1%	10.1	0.63	247
blend2	1	26m5s	–	–	–	–	2339
blend2	4	4m18s	12.8%	0	6.1	1.53	1019
blend2	8	3m33s	14.0%	0.2%	7.3	0.92	717
blend2	16	2m9s	34.1%	0	12.1	0.76	980
fixnet6	1	45m16s	–	–	–	–	2729
fixnet6	4	11m43s	1.0%	0	3.9	0.98	3598
fixnet6	8	10m26s	3.0%	0.2%	4.3	0.54	4703
fixnet6	16	6m16s	4.6%	0	7.2	0.45	6570
cap6000	1	71m27s	–	–	–	–	6129
cap6000	4	22m24s	0.2%	0	3.2	0.80	9551
cap6000	8	16m52s	0.3%	0	4.2	0.53	12363
cap6000	16	10m40	1.2%	0.2%	6.7	0.42	14121

Future work

- Improving ALPS:
  - Reduction of ramp-up time, and
  - Elimination of redundant work.
- Develop the Branch, Constrain, and Price Software (BiCePS) library
  - Data handling layer
  - Support the implementation of parallel branch and bound algorithms
  - Integrate with OSI 2.
- Develop the BiCePS Linear Integer Solver (BLIS)
  - Add customization features akin to COIN/BCP