Doing It in Parallel (DIP) with The COIN-OR High-Performance Parallel Search Framework (CHiPPS)

TED RALPHS
LEHIGH UNIVERSITY
YAN XU
SAS INSTITUTE
LASZLO LADÁNYI
IBM T.J. WATSON RESEARCH CENTER
MATTHEW SALTZMAN
CLEMSON UNIVERSITY

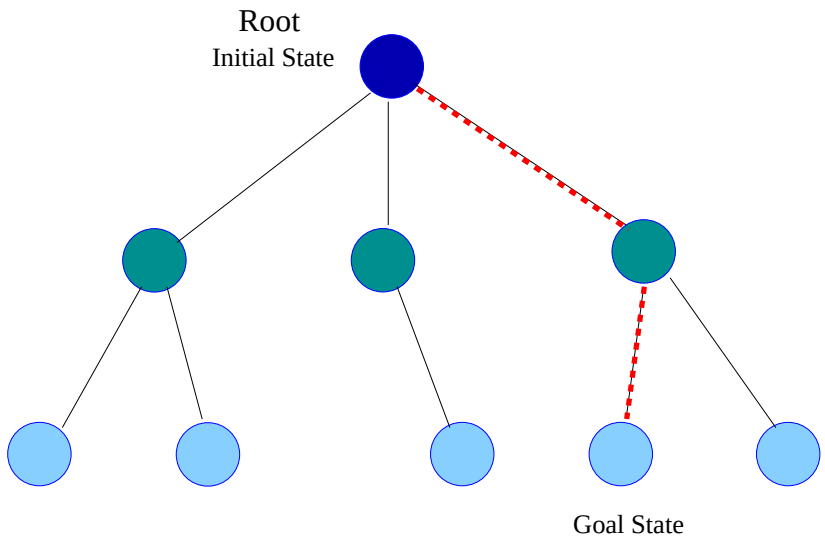
University of Newcastle, May 26, 2009

Thanks: Work supported in part by the National Science Foundation

Outline

Tree Search Algorithms

- Tree search algorithms systematically search the nodes of an acyclic graph for certain *goal nodes*.



- Tree search algorithms have been applied in many areas such as
  - Constraint satisfaction,
  - Game search,
  - Artificial intelligence, and
  - Mathematical programming.

Elements of Tree Search Algorithms

- A generic tree search algorithm consists of the following elements:

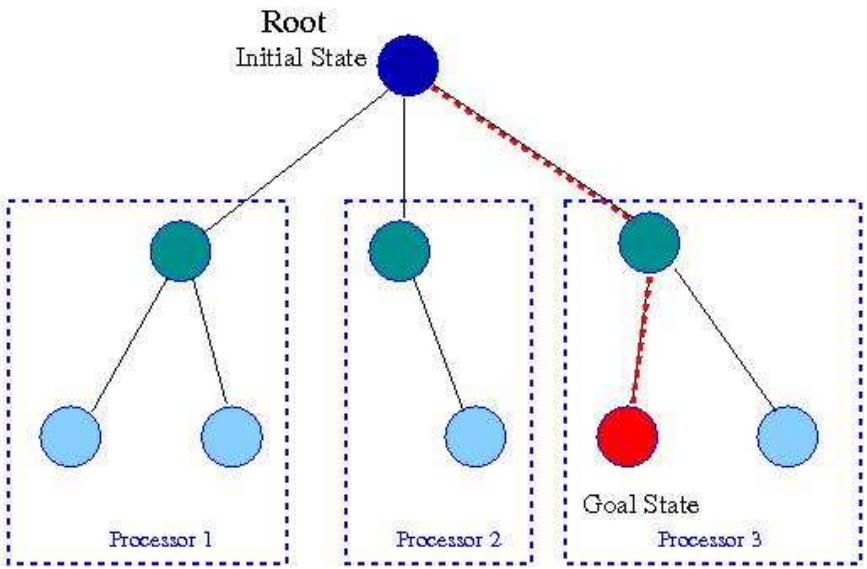
Generic Tree Search Algorithm

- Processing method: Is this a goal node?
- Fathoming rule: Can node can be fathomed?
- Branching method: What are the successors of this node?
- Search strategy: What should we work on next?

- The algorithm consists of choosing a candidate node, processing it, and either fathoming or branching.
- During the course of the search, various information (*knowledge*) is generated and can be used to guide the search.

Parallelizing Tree Search Algorithms

- In general, the search tree can be very large.
- Fortunately, the generic algorithm appears very easy to parallelize.



- The appearance is deceiving, as the search graph is not generally known a priori and naïve parallelization strategies are not generally effective.

Parallel Architectures

- A parallel computer is a collection of processing elements that can cooperate to perform a task.
- Historically, most parallel computers could be considered to belong to one of two broad architectural classes:

Architectures

- Shared memory
  - Each processor can access any memory location.
  - Processing units share information through memory IO.
  - Software scales, hardware doesn't.
- Distributed memory
  - Each processing unit has its own local memory and can only access its own memory directly.
  - Processing units share information via a network.
  - Hardware scales, software doesn't.
  - Typical examples are *massively parallel processors* (MPP), *cluster*, and *computational grids*.

Parallel Programming Tools

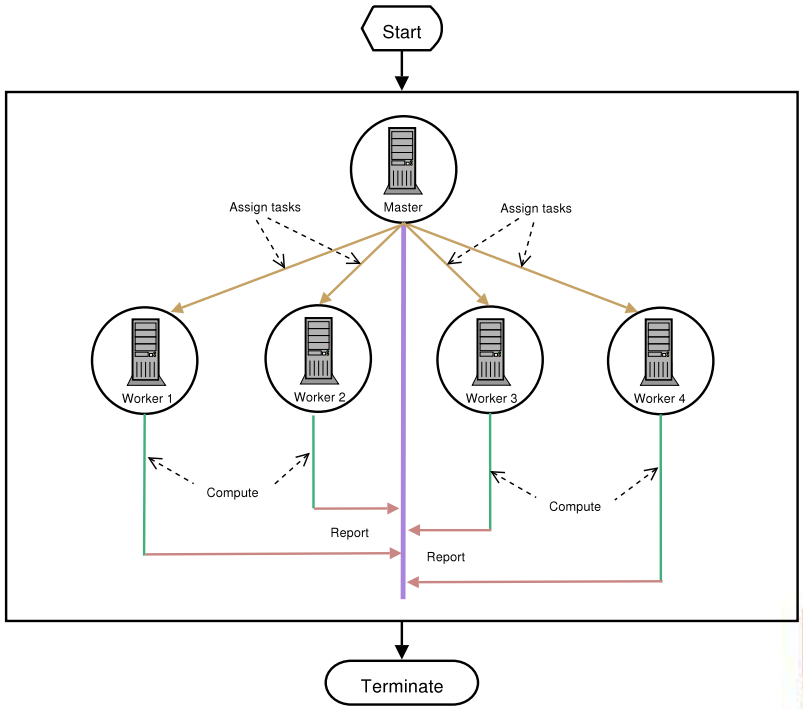
There are a wide variety of tools for parallizing program execution, ranging from low-level, platform-specific to high-level, platform-independent.

- Low-level Tools: *Sockets*, *threads*, *remote procedure calls*.
- Parallelizing Compilers: Compilers that automatically parallelize sequential programs.
- APIs: Standard programming interface for threading (primarily *OpenMP* on shared memory computers).
- Parallel Languages: Languages with parallel constructs such as *High Performance Fortran*.
- Message Passing Libraries: *MPI*, *PVM*, etc.
- Grid Tools: Tools for coordinating remote jobs across networks such as *Condor*.

Parallel Programming Paradigms

A programming paradigm is a class of algorithms that have generally the same control structure. Common paradigms include:

- Task-Farming/Master-Worker
- Single-Program Multiple-Data
- Data Pipelining



Measuring Performance of a Parallel System

- *Parallel System*: Parallel algorithm + parallel architecture.
- *Scalability*: How well a parallel system takes advantage of increased computing resources.

Terms

- Sequential runtime: T_s
- Parallel runtime: T_p
- Parallel overhead: $T_o = NT_p - T_s$
- Speedup: $S = T_s/T_p$
- Efficiency: $E = S/N$

- Standard analysis considers change in efficiency on a fixed test set as number of processors is increased.
- *Isoefficiency analysis* considers the increase in problem size to maintain a fixed efficiency as number of processors is increased.

Parallel Overhead

- The amount of *parallel overhead* determines the scalablity.

Major Components of Parallel Overhead in Tree Search

- Communication Overhead (cost of sharing knowledge)
- Idle Time
  - Handshaking/Synchronization (cost of sharing knowledge)
  - Task Starvation (cost of *not* sharing knowledge)
  - Ramp Up Time
  - 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.

Previous Work

Previous tree search codes:

- Game tree search: ZUGZWANG and APHID
- Constraint programming: ECLiPSe, etc.
- Optimization:
  - Commercial: CPLEX, Lindo, Mosek, SAS/OR, Xpress, etc.
  - Serial: ABACUS, bc-opt, COIN/CBC, GLPK, MINTO, SCIP, etc.
  - Parallel: COIN/BCP, FATCOP, PARINO, PICO, SYMPHONY, etc.

However, to our knowledge:

- Few studies of general tree search algorithms, and only one framework (PIGSeL).
- No study has emphasized scalability for *data-intensive* applications.
- Many packages are not open source or not easy to specialize for particular problem classes.

The COIN-OR High-Performance Parallel Search Framework

- CHiPPS has been under development since 2000 in partnership with IBM, NSF, and the COIN-OR Foundation.
- The broad goal was to develop a succesor to SYMPHONY and BCP, two previous parallel MIP solvers.
- It consists of a hierarchy of C++ class libraries for implementing general parallel tree search algorithms.
- It is an open source project hosted by COIN-OR.
- Design goals
  - Scalability
  - Usability

COIN-OR

- The software discussed in this talk is available for free download from the Computational Infrastructure for Operations Research Web site

  `projects.coin-or.org/CHiPPS`

- The COIN-OR Foundation (`www.coin-or.org`)
  - An non-profit educational foundation promoting the development and use of interoperable, open-source software for operations research.
  - A consortium of researchers in both industry and academia dedicated to improving the state of computational research in OR.
- The COIN-OR Repository
  - A library of interoperable software tools for building optimization codes, as well as some stand-alone packages.
  - A venue for peer review of OR software tools.
  - A development platform for open source projects, including an SVN repository, project management tools, etc.

Algorithm Design Elements (Scalability)

For scalability, the main objective is to control overhead. Design issues fall into three broad categories:

- Task Management
  - Task granularity
  - Ramp up/Ramp down
  - Termination detection
- Knowledge Management
  - Sharing
  - Storage
  - Searching
- Load balancing
  - Static (mapping)
  - Dynamic

Algorithm Design Elements (Knowledge Management)

- *Knowledge* is information generated during the search.
  - Knowledge generated during the search procedure guides the algorithm and changes the shape of the tree dynamically.
  - This is essentially what makes *load balancing* difficult.
  - Parallel tree search algorithms differ primarily in the way knowledge is shared (Trienekens '92).
- Knowledge sharing is necessary to achieve good efficiency.
  - Helps eliminate the performance of *redundant work*.
  - Helps avoid "task starvation."
  - Goal is for parallel search to mirror sequential search.
- Knowledge sharing also increases communication overhead.
- This is the fundamental tradeoff.

Algorithm Design Elements (Usability)

- Ease of use
  - Intuitive class structure.
  - No need to understand implementation.
- Generality
  - Minimal algorithmic assumptions in base layer.
  - Specialized methods implemented in derived classes
- Extensibility
  - Mechanism for defining new knowledge types.
  - Ability to develop custom applications.
- Portability
  - Coded in ANSI/ISO C++.
  - No dependance on architecture, operating system, or third-party software.

CHiPPS Library Hierarchy

ALPS (Abstract Library for Parallel Search)

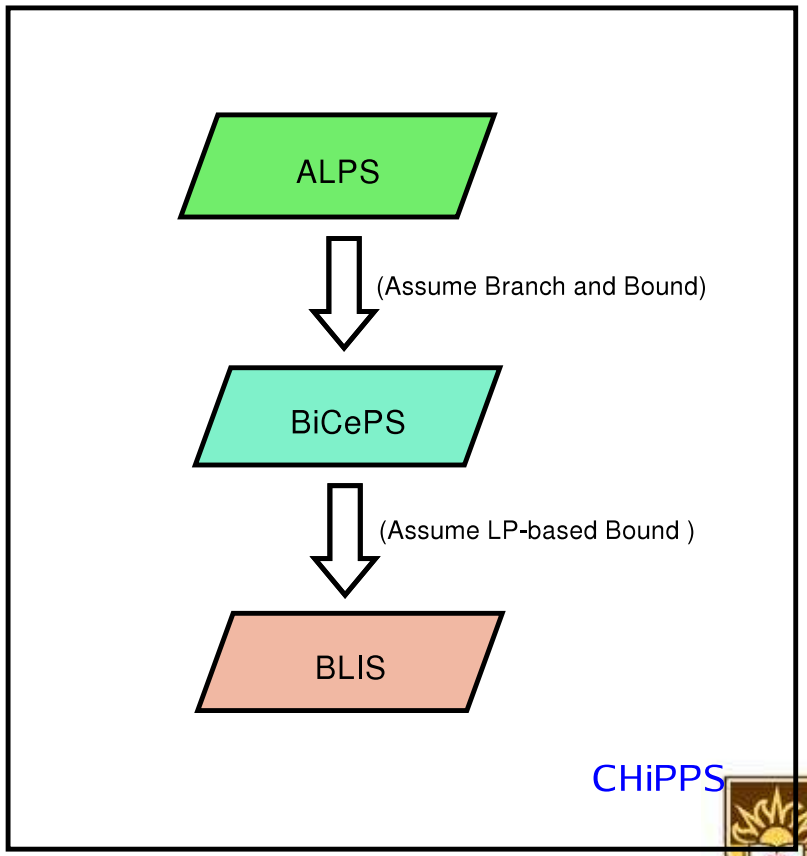
- search-handling layer
- prioritizes based on quality

BiCePS (Branch, Constrain, and Price Software)

- data-handling layer for relaxation-based optimization
- variables and constraints
- iterative bounding procedure

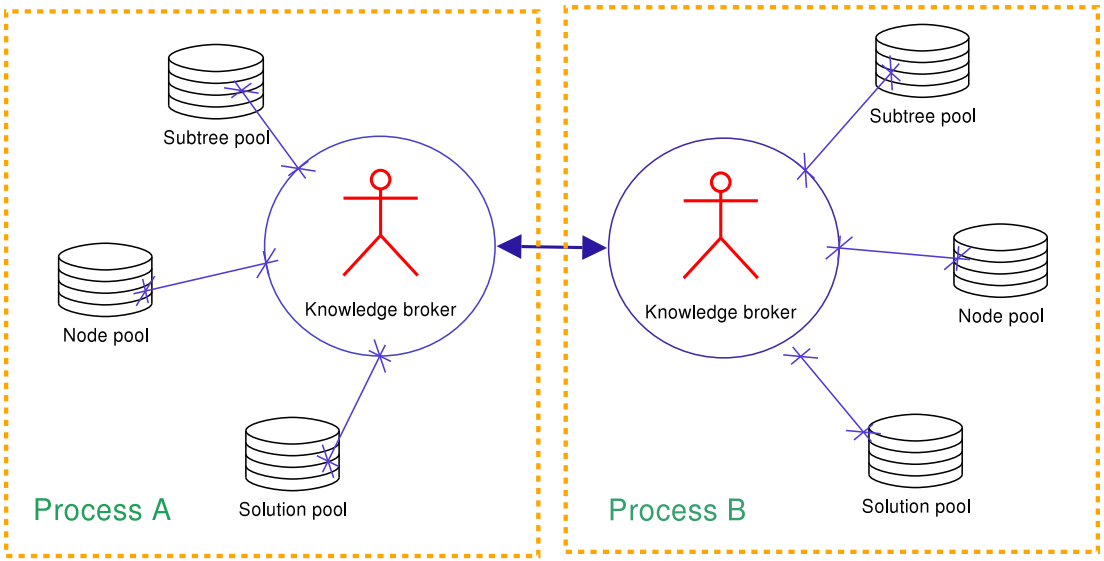
BLIS (BiCePS Linear Integer Solver)

- concretization of BiCePS
- linear constraints and objective

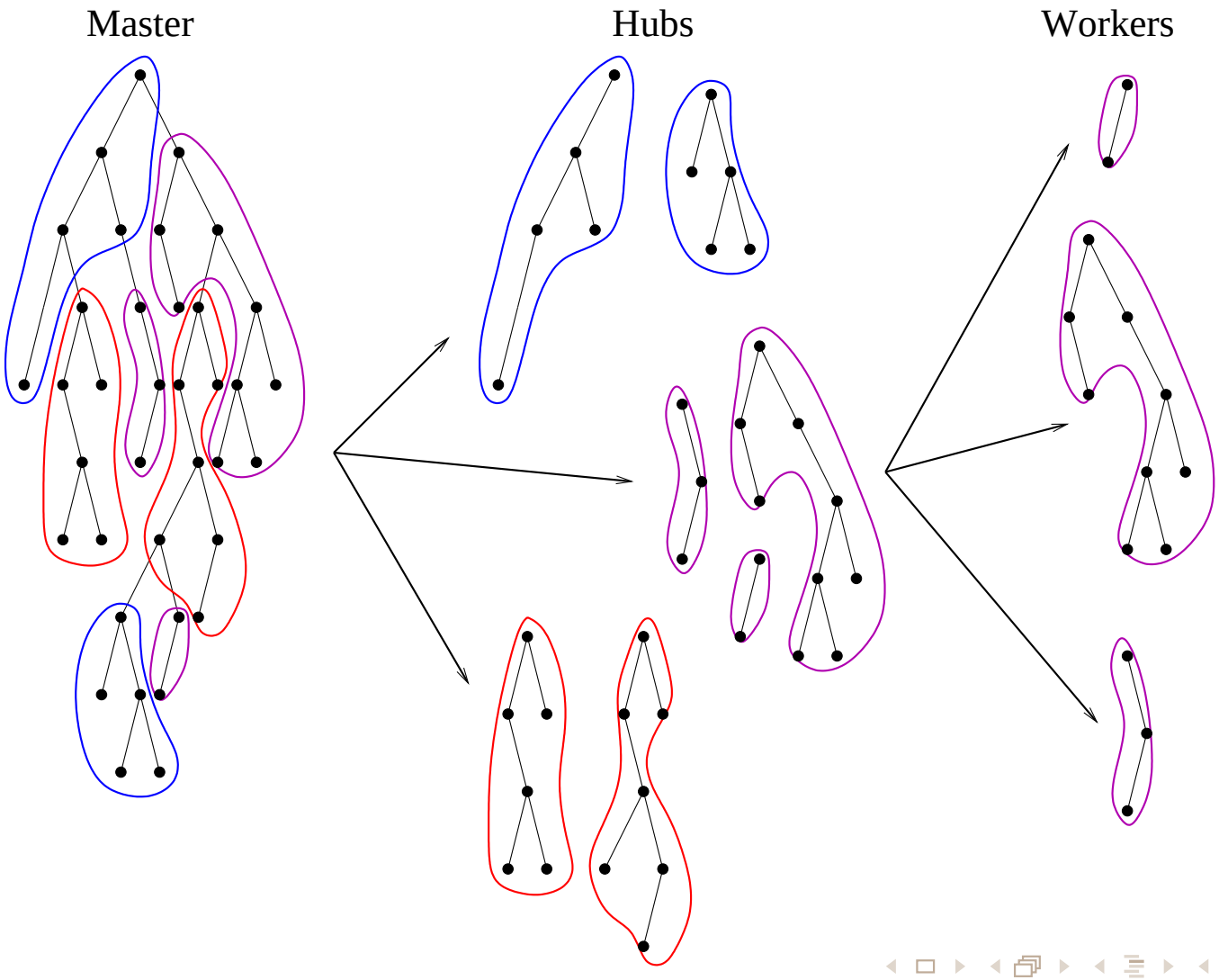


Knowledge Sharing

- All knowledge to be shared is stored in classes derived from a single base class and has an associated *encoded form*.
- Encoded form is used for identification, storage, and communication.
- Knowledge is maintained by one or more *knowledge pools*.
- The knowledge pools communicate through *knowledge brokers*.

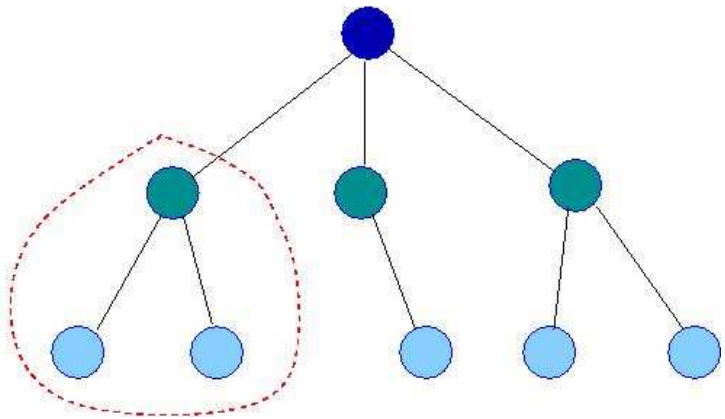


Master-Hub-Worker Paradigm



Task Granularity

- Task granularity is a crucial element of parallel efficiency.
- In CHiPPS, each worker is capable of exploring an entire subtree autonomously.
- By stopping the search prematurely, the task granularity can be adjusted dynamically.
- As granularity increases, communication overhead deacreases, but other sources of overhead increase.



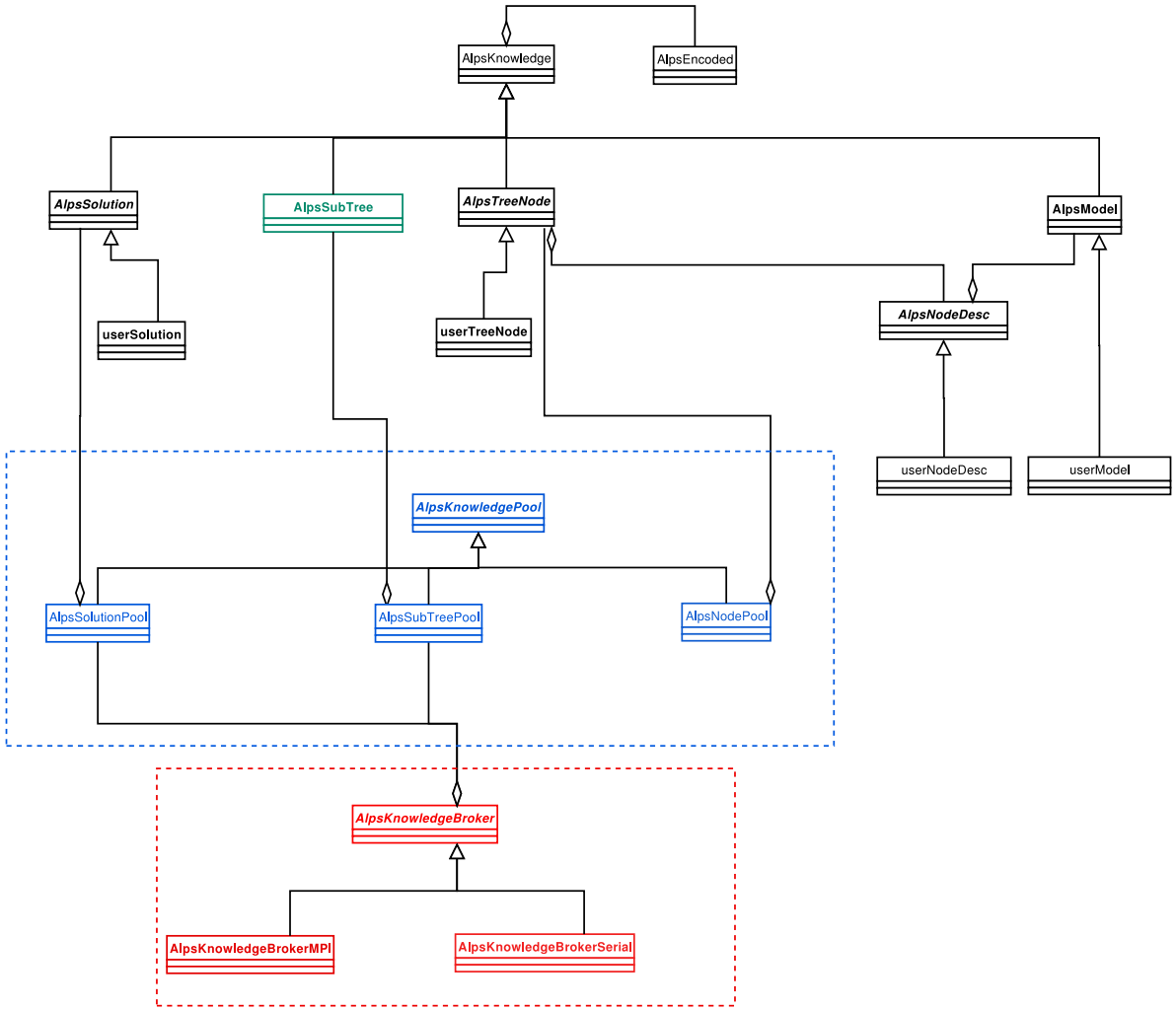
Synchronization

- As much as possible, we have eliminated handshaking and synchronization.
- A knowledge broker can work completely asynchronously, as long as its local node pool is not empty.
- This asynchronism can result in an increase in the performance of redundant work.
- To overcome this, we need good load balancing.

Load Balancing

- Static
  - Performed at startup
  - Two types
    - Two-level root initialization.
    - Spiral initialization.
- Dynamic
  - Performed periodically and as needed.
  - Balance by quantity and quality.
  - Keep subtrees together to enable differencing.
  - Three types
    - Inter-cluster dynamic load balancing,
    - Intra-cluster dynamic load balancing, and
    - Worker-initiated dynamic load balancing.
  - Workers do not know each others' workloads.
  - Donors and receivers are matched at both the hub and master level.
  - Three schemes work together to ensure workload is balanced.

Alps Class Hierarchy



BiCePS: Basic Notions

- BiCePS introduces the notion of *variables* and *constraints* (generically referred to as *objects*).
- Objects are abstract entities with *values* and *bounds*.
- They are used to build mathematical programming *models*.
- Search tree nodes consist of subproblems described by sets of variables and constraints.
- Key assumptions
  - Algorithm is relaxation-based branch-and-bound.
  - Bounding is an iterative procedure involving generation of variables and constraints.

BiCePS: Differencing Scheme

- Descriptions of search tree nodes can be extremely large.
- For this reason, subtrees are stored using a *differencing scheme*.
- Nodes are described using differences from the parent is this description is smaller.
- Again, there is a tradeoff between memory savings and additional computation.
- This approach requires keeping subtrees wholeas much as possible.
- This impacts load balacing significantly.

BLIS: Branch, Cut, and Price

MILP

$$\min \quad c^T x \tag{1}$$
$$\text{s.t.} \quad Ax \leq b \tag{2}$$
$$x_i \in \mathbb{Z} \quad \forall\, i \in I \tag{3}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, $I \subseteq \{1, 2, \ldots, n\}$.

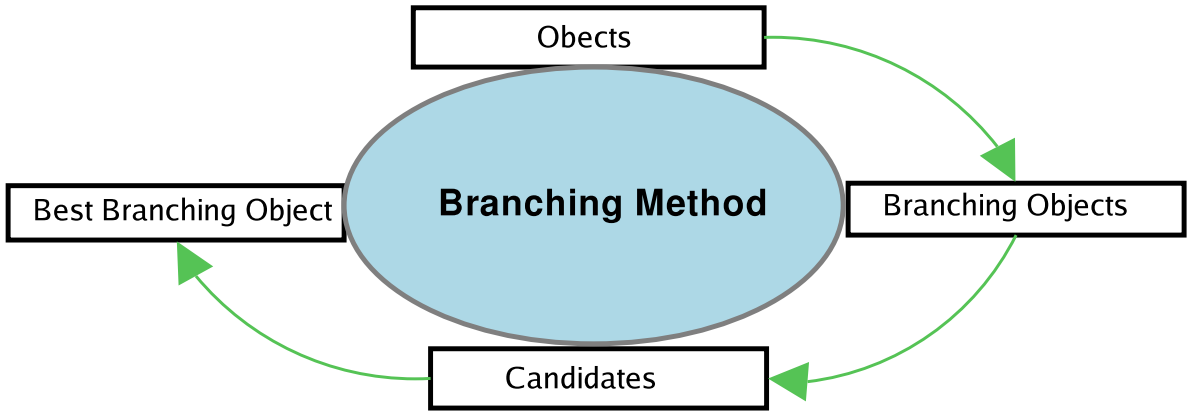
Basic Algorithmic Elements

- Search strategy.
- Branching scheme.
- Object generators.
- Heuristics.

BLIS: Branching Scheme

BLIS Branching scheme comprises three components:

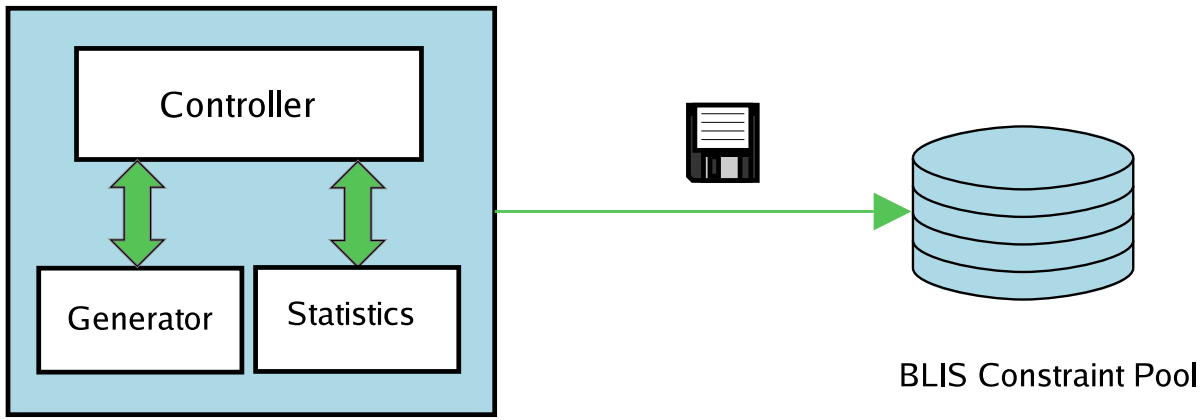
- Branching object: has feasible region and can be branched on.
- Branching candidate:
  - created from objects not in their feasible regions or
  - contains instructions for how to conduct branching.
- Branching method:
  - specifies how to create a set of branching candidates.
  - has the method to compare objects and choose the best one.



BLIS: Constraint Generators

BLIS constraint generator:

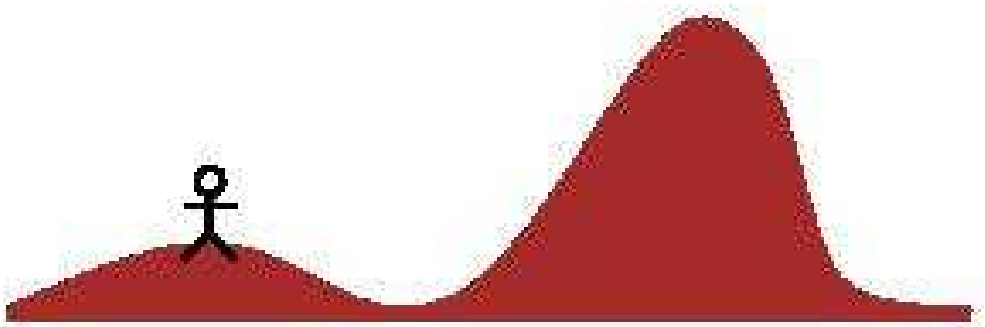
- provides an interface between BLIS and the algorithms in COIN/Cgl.
- provides a base class for deriving specific generators.
- has the ability to specify rules to control generator:
  - where to call: root, leaf?
  - how many to generate?
  - when to activate or disable?
- contains the statistics to guide generating.



BLIS: Heuristics

BLIS primal heuristic:

- defines the functionality to heuristically search for solutions.
- has the ability to specify rules to control heuristics.
  - where to call: before root, after bounding, at solution?
  - how often to call?
  - when to activate or disable?
- collects statistics to guide the heuristic.
- provides a base class for deriving specific heuristics.



Implementing a Knapsack Solver

- As a demonstration application, we implemented a solver for the knapsack problem using ALPS.
- The solver uses the closed form solution of the LP relaxation as a bound.
- Branching is on the fractional variable.
- Implementation consists of deriving a few classes to specify the algorithm.
  - `KnapModel`
  - `KnapTreeNode`
  - `KnapSolution`
  - `KnapParams`
- Once the classes have been implemented, the user writes a `main` function.
- The only difference between parallel and serial code is the knowledge broker class that is used.

Sample main() Function

```
int main(int argc, char* argv[])
{
    KnapModel model;
#if defined(SERIAL)
    AlpsKnowledgeBrokerSerial knap(argc, argv, model);
#elif defined(PARALLEL_MPI)
    AlpsKnowledgeBrokerMPI knap(argc, argv, model);
#endif
    knap.registerClass("MODEL", new KnapModel);
    knap.registerClass("SOLUTION", new KnapSolution);
    knap.registerClass("NODE", new KnapTreeNode);
    knap.search();
    knap.printResult();
    return 0;
}
```

The Vehicle Routing Problem

The VRP is a combinatorial problem whose *ground set* is the edges of a graph $G(V, E)$. Notation:

- V is the set of customers and the depot (0).
- d is a vector of the customer demands.
- k is the number of routes.
- C is the capacity of a truck.

A feasible solution is composed of:

- a partition $\{R_1, \ldots, R_k\}$ of V such that $\sum_{j \in R_i} d_j \leq C,\ 1 \leq i \leq k$;
- a permutation σ_i of $R_i \cup \{0\}$ specifying the order of the customers on route i.

Standard IP Formulation for the VRP

VRP Formulation

$$\begin{array}{lcll} \sum_{j=1}^{n} x_{0j} & = & 2k & \\ \sum_{j=1}^{n} x_{ij} & = & 2 & \forall i \in V \setminus \{0\} \\ \sum_{\substack{i \in S \\ j \notin S}} x_{ij} & \geq & 2b(S) & \forall S \subset V \setminus \{0\},\ |S| > 1. \end{array}$$

- $b(S)$ = lower bound on the number of trucks required to service S (normally $\lceil (\sum_{i \in S} d_i)/C \rceil$).
- The number of constraints in this formulation is exponential.
- We must therefore generate the constraints dynamically.
- A solver can be implemented in BLIS by deriving just a few classes.

Implementing the VRP Solver

- The algorithm is defined by deriving the following classes.
  - `VrpModel`
  - `VrpSolution`
  - `VrpCutGenerator`
  - `VrpHeuristic`
  - `VrpVariable`
  - `VrpsParams`
- Once the classes have been implemented, the user writes a `main` function as before.

Computational Results: Platforms

Clemson Cluster

Machine:	Beowulf cluster with 52 nodes
Node:	dual core PPC, speed 1654 MHz
Memory:	4G RAM each node
Operating System:	Linux
Message Passing:	MPICH

SDSC Blue Gene System

Machine:	IBM Blue Gene with 3,072 compute nodes
Node:	dual processor, speed 700 MHz
Memory:	512 MB RAM each node
Operating System:	Linux
Message Passing:	MPICH

KNAP Scalability for Moderately Difficult Instances

- Tested the 10 instances in the moderately difficult set on the Clemson cluster.
- The default algorithm was used except that
  - the static load balancing scheme is the two-level root initialization,
  - the number of nodes generated by the master is 3000, and
  - the size of a unit work is 300 nodes.

P	Node	Ramp-up	Idle	Ramp-down	Wallclock	Eff
4	193057493	0.28%	0.02%	0.01%	586.90	1.00
8	192831731	0.58%	0.08%	0.09%	245.42	1.20
16	192255612	1.20%	0.26%	0.37%	113.43	1.29
32	191967386	2.34%	0.71%	1.47%	56.39	1.30
64	190343944	4.37%	2.27%	5.49%	30.44	1.21

- Super-linear speedup observed.
- Ramp-up, ramp-down, and idle time overhead remains low.

KNAP Scalability for Very Difficult Instances

- Tested the 26 instances in the difficult set on the Blue Gene system.
- The default algorithm was used except that
  - the static load balancing scheme is the two-level root initialization,
  - the number of nodes generated by the master varies from 10000 to 30000 depends on individual instance,
  - the number of nodes generated by a hub varies from 10000 to 20000 depends on individual instance,
  - the size a unit work is 300 nodes; and
  - multiple hubs were used.

P	Node	Ramp-up	Idle	Ramp-down	Wallclock	Eff
64	14733745123	0.69%	4.78%	2.65%	6296.49	1.00
128	14776745744	1.37%	6.57%	5.26%	3290.56	0.95
256	14039728320	2.50%	7.14%	9.97%	1672.85	0.94
512	13533948496	7.38%	4.30%	14.83%	877.54	0.90
1024	13596979694	8.33%	3.41%	16.14%	469.78	0.84
2048	14045428590	9.59%	3.54%	22.00%	256.22	0.77

- KNAP scales well even when using several thousand processors.

The Performance of Serial BLIS on Generic MILPs

Test Bed

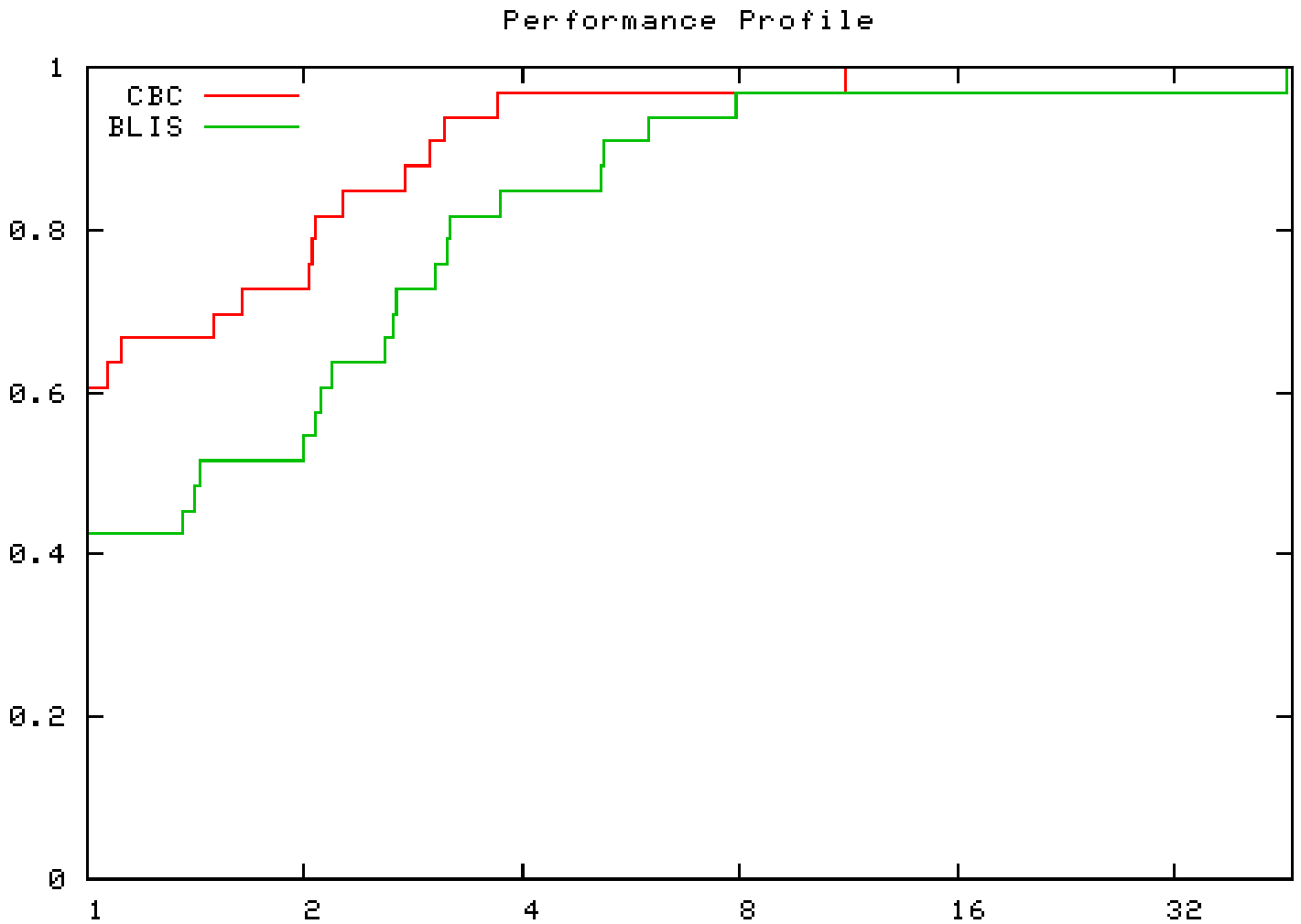
- Test Machine: PC, 2.8 GHz Pentium, 2.0G RAM, Linux
- Test instances: Selected 33 instances from Lehigh/CORAL and MIPLIB 3 that both solvers can solve in 10 minutes.

- BLIS (serial version)
  - Branching strategy: Pseudocost branching.
  - Cuts generators: Gomory, Knapsack, Flow Cover, MIR, Probing, and Clique.
  - Heuristics: Rounding.
- COIN/Cbc
  - Branching strategy: Strong branching.
  - Cut generators: Gomory, Knapsack, Flow Cover, MIR, Probing, and Clique.
  - Heuristics: Rounding and Local search.

Running Times

Problem	Row	Column	Nonzero	Time-BLIS	Time-CBC
22433	198	429	3408	95.00	37.59
23588	137	368	3701	118.75	108.75
air03	124	10757	91028	36.45	7.84
aligninq	340	1831	15734	356.76	181.22
bell3a	123	133	347	49.21	38.49
dcmulti	290	548	1315	18.68	13.84
dsbmip	1182	1886	7366	55.30	38.66
...	...	...	...	...	...
qnet1	503	1541	4622	20.17	41.80
rgn	24	180	460	20.06	62.61
roy	162	149	411	23.79	7.56
stein27	118	27	378	25.15	9.78
vpm1	234	378	749	1.45	16.24
TOTAL				1642.61	1238.32

Performance Profile



Does Differencing Make a Difference?

A Simple Test

- 38 MILP instances from Lehigh/CORAL and MIPLIB3.
- Solved to optimality by using BLIS in 10 minutes.
- PC, 2.8 GHz Pentium, 2G RAM, Linux, COIN/Clp.

	No	Yes	Geometric
Total Time	2016 seconds	1907 seconds	1.0
Total Peak Memory	1412 MB	286 MB	4.3

Problem	Time(No)	Memory(No)	Time(Yes)	Memory(Yes)
dcmulti	4.20 s	17.4 MB	4.19 s	1.4 MB
dsbmip	33.28 s	34.1 MB	33.16 s	2.4 MB
egout	0.18 s	0.3 MB	0.18 s	0.2 MB
enigma	6.41 s	13.1 MB	6.16 s	2.3 MB
fiber	6.60 s	17.6 MB	6.56 s	2.1 MB
...	...	...		

BLIS Scalability for Moderately Difficult Instances

- Selected 18 MILP instances from Lehigh/CORAL, MIPLIB 3.0, MIPLIB 2003, BCOL, and markshare.
- Tested on the Clemson cluster.

Instance	Nodes	Ramp-up	Idle	Ramp-down	Comm Overhead	Wallclock	Eff
1 P	11809956	–	–	–	–	33820.53	1.00
Per Node		–	–	–	–	0.00286	
4P	11069710	0.03%	4.62%	0.02%	16.33%	10698.69	0.79
Per Node		0.03%	4.66%	0.00%	16.34%	0.00386	
8P	11547210	0.11%	4.53%	0.41%	16.95%	5428.47	0.78
Per Node		0.10%	4.52%	0.53%	16.95%	0.00376	
16P	12082266	0.33%	5.61%	1.60%	17.46%	2803.84	0.75
Per Node		0.27%	5.66%	1.62%	17.45%	0.00371	
32P	12411902	1.15%	8.69%	2.95%	21.21%	1591.22	0.66
Per Node		1.22%	8.78%	2.93%	21.07%	0.00410	
64P	14616292	1.33%	11.40%	6.70%	34.57%	1155.31	0.46
Per Node		1.38%	11.46%	6.72%	34.44%	0.00506	

Impact of Problem Properties

- Instance `input150_1` is a knapsack instance. When using 128 processors, BLIS achieved super-linear speedup mainly to the decrease of the tree size
- Instance `fc_30_50_2` is a fixed-charge network flow instance. It exhibits very significant increases in the size of its search tree.
- Instance `pk1` is a small integer program with 86 variables and 45 constraints. It is relatively easy to solve.

Instance	P	Node	Ramp-up	Idle	Ramp-down	Wallclock	Eff
input150_1	64	75723835	0.44%	3.38%	1.45%	1257.82	1.00
	128	64257131	1.18%	6.90%	2.88%	559.80	1.12
	256	84342537	1.62%	5.53%	7.02%	380.95	0.83
	512	71779511	3.81%	10.26%	10.57%	179.48	0.88
fc_30_50_2	64	3494056	0.15%	31.46%	9.18%	564.20	1.00
	128	3733703	0.22%	33.25%	21.71%	399.60	0.71
	256	6523893	0.23%	29.99%	28.99%	390.12	0.36
	512	13358819	0.27%	23.54%	29.00%	337.85	0.21
pk1	64	2329865	3.97%	12.00%	5.86%	103.55	1.00
	128	2336213	11.66%	12.38%	10.47%	61.31	0.84
	256	2605461	11.55%	13.93%	20.19%	41.04	0.63
	512	3805593	19.14%	9.07%	26.71%	36.43	0.36

BLIS Scalability for Very Difficult Instances

- Tests on Clemson's palmetto cluster (60 on the Top 500 list, 11/2008, Linux, MPICH, 8-core 2.33GHz Xeon/Opteron mix, 12-16GB RAM).
- Tests use one processor per node.

Raw Computational Results

Name	256	128	64	1
mcf2	926	1373	2091	43059
neos-1126860	2184	1830	2540	39856
neos-1122047	1676	1125	1532	NS
neos-1413153	4230	3500	2990	20980
neos-1456979		78.06%	NS	NS
neos-1461051	396	1082	536	NS
neos-1599274		1500	8108	9075
neos-548047		137.29%	376.48%	482%
neos-570431	1034	1255	1308	21873
neos-611838	712	956	886	8005
neos-612143	565	1716	1315	4837
neos-693347		1.28%	1.70%	NS
neos-912015	538	438	275	10674
neos-933364		6.67%	6.79%	11.80%
neos-933815		6.54%	8.77%	32.85%
neos-934184		6.67%	6.76%	9.15%
neos18		30.78%	30.78%	79344

Speedups

Name	256	128	64
mcf2	46.5	31.36	20.59
neos-1126860	18.25	21.78	15.69
neos-1413153	4.96	5.99	7.02
neos-1599274		6.05	1.12
neos-570431	21.15	17.43	16.72
neos-611838	11.24	8.37	9.03
neos-612143	8.56	2.82	3.68
neos-912015	19.84	24.37	38.81

Efficiency

Name	256	128	64
mcf2	0.18	0.25	0.32
neos-1126860	0.07	0.17	0.25
neos-1413153	0.02	0.05	0.11
neos-1599274		0.05	0.02
neos-570431	0.08	0.14	0.26
neos-611838	0.04	0.07	0.14
neos-612143	0.03	0.02	0.06
neos-912015	0.08	0.19	0.61

Shameless Promotion

In October, 2007, the VRP/TSP solver won the Open Contest of Parallel Programming at the 19th International Symposium on Computer Architecture and High Performance Computing.

ALPS

- Our methods implemented in ALPS seem effective in improving scalibility.
- The framework is useful for implementing serial or parallel tree search applications.
- The KNAP application achieves very good scalability.
- There is still much room for improvement
  - load balancing,
  - fault tolerance,
  - hybrid architectures,
  - grid enable.

BLIS

- The performance of BLIS in serial mode is favorable when compared to state of the art non-commercial solvers.
- The scalability for solving generic MILPs is highly dependent on properties of individual instances.
- Based on BLIS, applications like VRP/TSP can be implemented in a straightforword way.
- Much work is still needed
  - Callable library API
  - Support for column generation
  - Enhanced heuristics
  - Additional capabilities

Obtaining CHiPPS

The CHiPPS framework is available for download at

```
https://projects.coin-or.org/CHiPPS
```

Thank You!

Questions?