Computational Approaches to Solving Integer Programs in Parallel

TED RALPHS

ISE Department
COR@L Lab
Lehigh University
ted@lehigh.edu

COR@L
COMPUTATIONAL OPTIMIZATION RESEARCH AT LEHIGH

SIAM Conference on Optimization, Darmstadt, Germany
17 May 2011

Thanks: Work supported in part by the National Science Foundation and IBM

Outline

1. Introduction
2. SYMPHONY
   - Overview
   - Parallelization
3. CHiPPS
   - ALPS: Abstract Library for Parallel Search
   - BiCePS: Branch, Constrain, and Price Software
   - BLIS: BiCePS Linear Integer Solver
4. Computational Experiments
   - Measuring Performance
   - Computational Experiments
   - Conclusions

What Will Future Architectures Look Like?

- Moore's Law has now moved from clock speeds to numbers of cores.
- To take advantage of the capabilities of new hardware, effective parallelization will be the key.
- It seems clear that the next generation(s) of hardware will be clusters (of clusters) of machines with multiple multi-core chips.
- The result will be a memory hierarchy of ever-increasing complexity.
  - Cache memory
  - Main memory (local to core)
  - Main memory (attached to other cores)
  - Local disk
  - Co-located distributed memory
  - Remotely located distributed memory
- How do we deal with this complex hierarchy?

Sources of Parallelism

Parallelization of tree search is easy in principle...but not in practice!

- Tree parallelism: Process different part of the tree simultaneously.
- Decomposition: Apply a decomposition approach in order to parallelize the bounding procedure.
- Task parallelism
  - Special procedures for the ramp-up/ramp-down phases
  - Parallelize primal heuristics
  - Parallelize cut generation
  - Process multiple trees simultaneously

Basic Parallelization Approaches

- Local/shared memory computation (SYMPHONY)
  - Work on one local copy of the tree and use threads to process multiple “chains” simultaneously.
  - Pro: Implementation (load balancing) is much easier.
  - Con: Expensive and limited computational resources.
- Distributed memory computation (CHiPPS)
  - Partition the tree and process subtrees asynchronously.
  - Pro: No limit to hardware access, inexpesive.
  - Con: Efficiency requires load balancing.

Parallel Overhead

- The amount of *parallel overhead* determines the scalability.
- "Knowledge sharing" is the main driver of efficiency.

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)
  - Memory Contention
  - Ramp Up Time
  - Ramp Down Time
- Performance of Redundant Work (cost of *not* sharing knowledge)

- This breakdown highlights the tradeoff between centralized and decentralized knowledge storage and decision-making.

Hybrid Approach

- To take advantage of modern hardware, a hybrid approach is required.
  - Partition the tree and process subtrees in a distributed fashion.
  - Locally, subtrees are processed in parallel using the shared memory approach.
- With this approach, we hope to do much of the parallel computation at the local level.
- This approach *should* require less load balancing.

Brief Overview of SYMPHONY

- SYMPHONY is an open-source software package for solving and analyzing mixed-integer linear programs (MILPs).
- SYMPHONY can be used in three distinct modes.
  - Black box solver: From the command line or shell.
  - Callable library: From a C/C++ code.
  - Framework: Develop a customized solver or callable library.
- Advanced features
  - Warm starting
  - Sensitivity analysis
  - Bicriteria solve
  - Parallel execution

Basic Algorithm

- Core solution methodology is branch and cut.
- Default search strategy is a hybrid depth-first/best-first strategy.
- Default branching scheme is strong branching.
- Cuts can be generated using COIN's Cut Generator Library.

- Clique
- Flow Cover
- Gomory
- Knapsack Cover
- Lift and Project
- Mixed Integer Rounding
- Odd Hole
- Probing
- Simple Rounding
- Reduce and Split
- Two-slope MIR

Enabling Parallelism

SYMPHONY is divided into five independent modules that communicate through shared memory or over a network.

SYMPHONY Modules

- Master: Maintains static data between solves, spawns parallel processes, performs I/O.
- Tree Manager: Controls overall execution by tracking growth of the tree and dispatching subproblems to the NPs.
- Node Processors: Performs processing and branching operations.
- Cut Generator: Generates cuts.
- Cut Pool: Acts as an auxiliary cut generator by maintaining a list of the “most effective” cuts found so far.

SYMPHONY Configurations

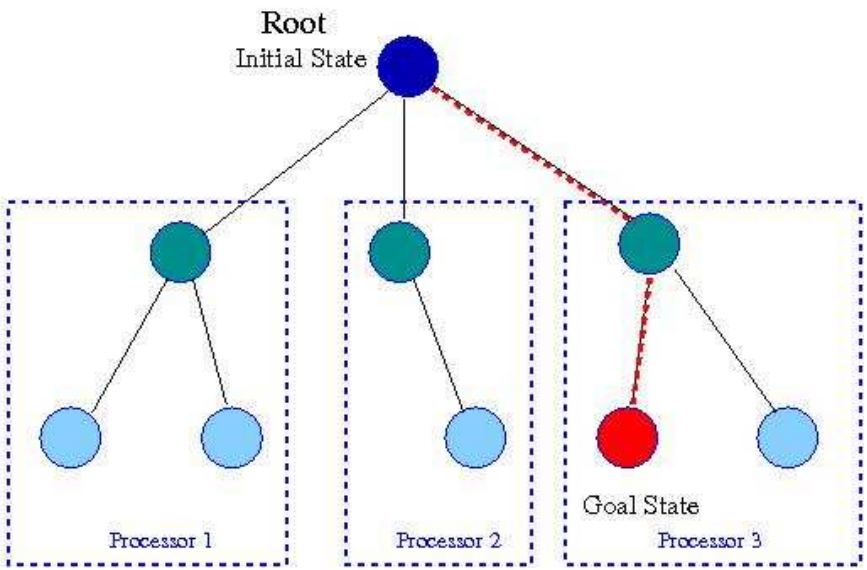
- Each module can be compiled as an independent executable for parallel execution.
- The modules can also be combined in any number of different ways to yield other parallel configurations.
- If all modules are combined together, we get either
  - A sequential executable (with a standard C++ compiler).
  - A shared-memory parallel executable (with an OpenMP-aware C++ compiler).
- The most common distributed-memory parallel configuration is to have two executables.
  - Combined NP/CG executable:
  - Combined Master/TM/CP executable: Storing and distributing generated data (subproblem descriptions and cuts).

Parallelization Approach

- The approach to parallelism in SYMPHONY is very straightforward.
- Search the tree as in the sequential case, but process multiple “chains” simultaneously on different processors/cores.
- The advantage of this approach is its simplicity and the avoidance of redundant work.
- The disadvantage is that it’s not very scalable.
- For relatively small numbers of processors/cores, it works well.

Quick Introduction to CHiPPS

- CHiPPS stands for COIN-OR High Performance Parallel Search.
- CHiPPS is a set of C++ class libraries for implementing tree search algorithms for both sequential and parallel environments.
- The basic goal is to generalize notions from SYMPHONY and enable large-scale computation.



CHiPPS Components

ALPS (Abstract Library for Parallel Search)

- is the search-handling layer (parallel and sequential).
- provides various search strategies based on node priorities.

BiCePS (Branch, Constrain, and Price Software)

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

BLIS (BiCePS Linear Integer Solver)

- is a concretization of BiCePS.
- specific to models with linear constraints and objective function.

ALPS: Design Goals

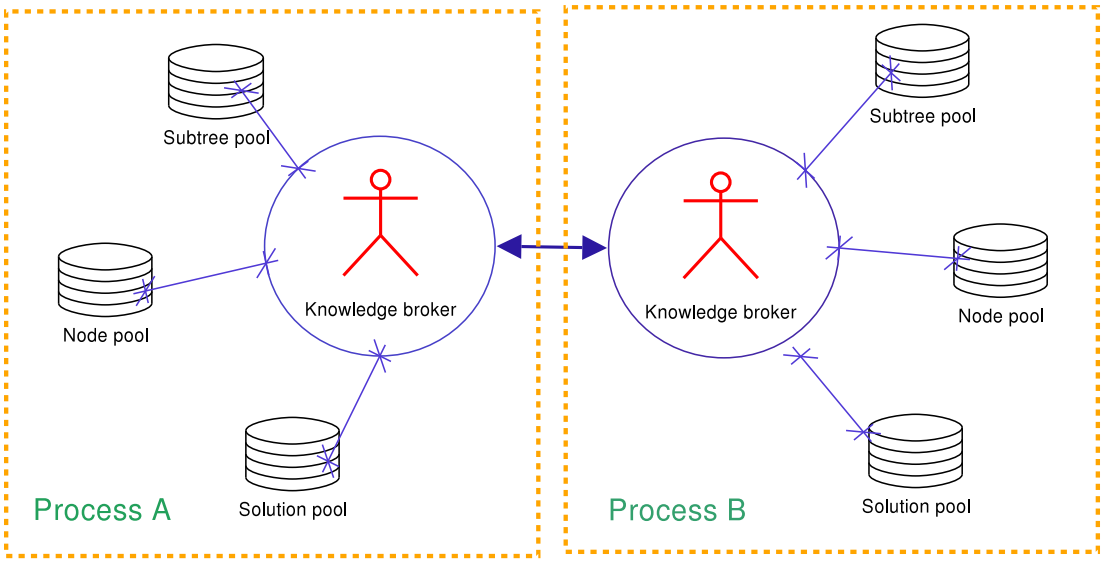
- Intuitive object-oriented class structure.
  - `AlpsModel`
  - `AlpsTreeNode`
  - `AlpsNodeDesc`
  - `AlpsSolution`
  - `AlpsParameterSet`
- Minimal algorithmic assumptions in the base class.
  - Support for a wide range of problem classes and algorithms.
  - Support for constraint programming.
- Easy for user to develop a custom solver.
- Design for *parallel scalability*, but retain ability to operate effectively in a sequential environment.
- Explicit support for *memory compression* techniques important for implementing optimization algorithms.

ALPS: Overview of Features

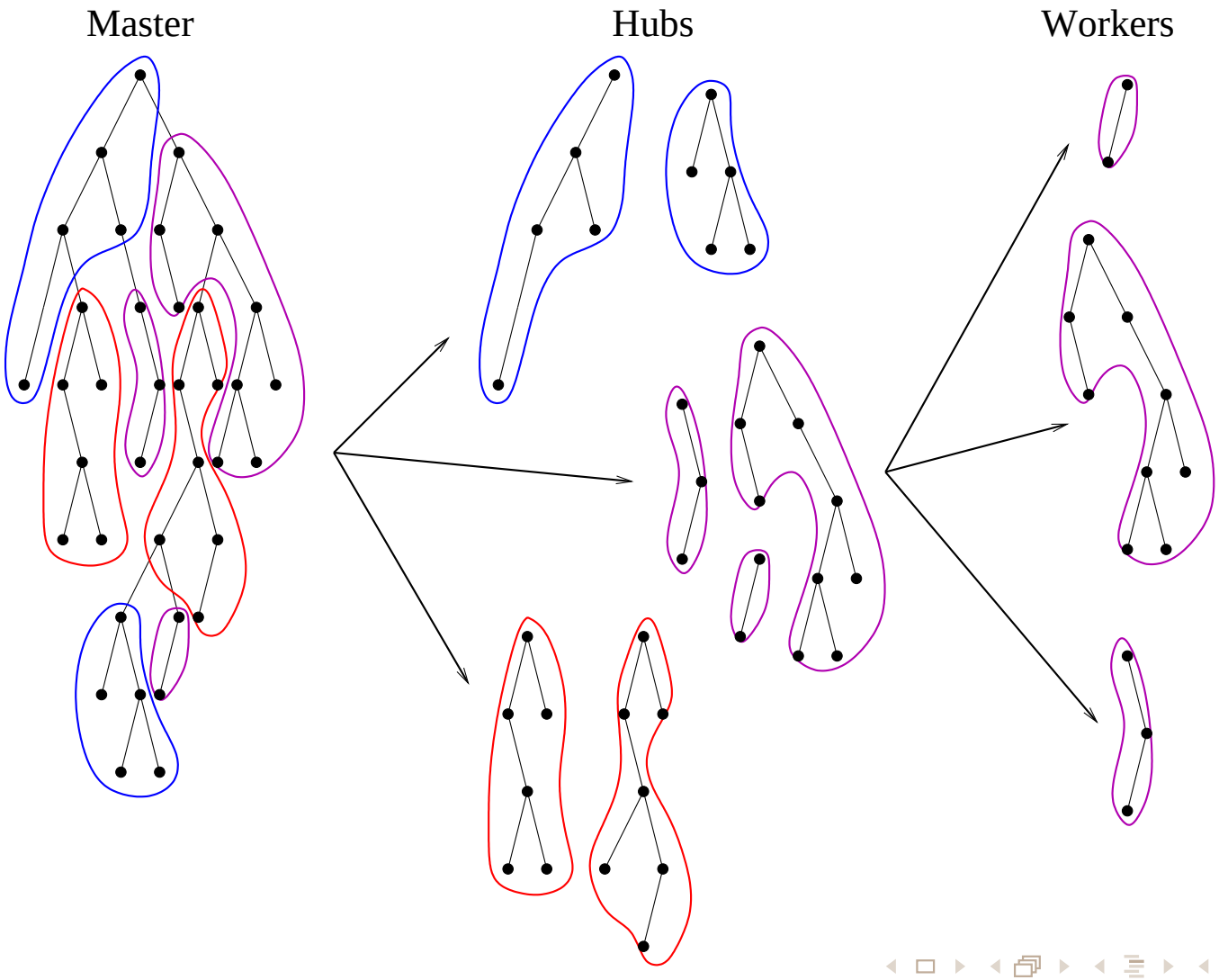
- The design is based on a very general concept of *knowledge*.
- Knowledge is shared asynchronously through *pools* and *brokers*.
- Management overhead is reduced with the *master-hub-worker* paradigm.
- Overhead is decreased using dynamic task granularity.
- Two static load balancing techniques are used.
- Three dynamic load balancing techniques are employed.
- Uses asynchronous messaging to the highest extent possible.
- A scheduler on each process manages tasks like
  - node processing,
  - load balancing,
  - update search states, and
  - termination checking, etc.

Knowledge Sharing

- All knowledge to be shared is 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



Using ALPS: A Knapsack Solver

The formulation of the binary knapsack problem is

$$\max\{\sum_{i=1}^{m} p_i x_i : \sum_{i=1}^{m} s_i x_i \leq c, x_i \in \{0,1\}, i = 1,2,\ldots,m\}, \qquad (1)$$

We derive the following classes:

- `KnapModel` (from `AlpsModel`) : Stores the data used to describe the knapsack problem and implements `readInstance()`
- `KnapTreeNode` (from `AlpsTreeNode`) : Implements `process()` (bound) and `branch()`
- `KnapNodeDesc` (from `AlpsNodeDesc`) : Stores information about which variables/items have been fixed by branching and which are still free.
- `KnapSolution` (from `AlpsSolution`) Stores a solution (which items are in the knapsack).

BiCePS: Support for Relaxation-based Optimization

- Adds notion of *modeling objects* (variables and constraints).
- Models are built from sets of such objects.
- Bounding is an iterative process that produces new objects.
- A differencing scheme is used to store the difference between the descriptions of a child node and its parent.

```
struct BcpsObjectListMod
{
    int  numRemove;
    int* posRemove;
    int  numAdd;
    BcpsObject **objects;
    BcpsFieldListMod<double> lbHard;
    BcpsFieldListMod<double> ubHard;
    BcpsFieldListMod<double> lbSoft;
    BcpsFieldListMod<double> ubSoft;
};
```

```
template<class T>
struct BcpsFieldListMod
{
    bool relative;
    int  numModify;
    int  *posModify;
    T    *entries;
};
```

BLIS: A Generic Solver for MILP

MILP

$$min \quad c^T x \tag{2}$$
$$s.t. \quad Ax \leq b \tag{3}$$
$$x_i \in \mathbb{Z} \quad \forall\, i \in I \tag{4}$$

where $(A, b) \in \mathbb{R}^{m \times (n+1)}$, $c \in \mathbb{R}^n$.

Basic Algorithmic Components

- Bounding method.
- Branching scheme.
- Object generators.
- Heuristics.

Traditional Measures of Performance

- *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.

Challenges in Measuring Performance

- Traditional measures are not appropriate.
  - The interesting problems are the ones that take too long to solve sequentially.
  - Need to account for the possibility of failure.
- It's exceedingly difficult to construct a test set
  - Scalability varies substantially by instance.
  - Hard to know what test problems are appropriate.
  - A fixed test set will almost surely fail to measure what you want.
- Results are highly dependent on architecture
  - Difficult to make comparisons
  - Difficult to tune parameters
- Hard to get enough time on large-scale platforms for tuning and testing.
- Results are non-deterministic!
  - Determinism can be a false sense of security.
  - Lack of determinism requires more extensive testing.

Computational Results: Platforms

Clemson Cluster I

Machine:	Beowulf cluster with 52 nodes
Node:	dual core PPC, 1.6 GHz
Memory:	4G RAM each node
Message Passing:	MPICH

SDSC Blue Gene System

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

Computational Results: Platforms (cont.)

COR@L Cluster

Machine:	12 nodes
Nodes:	quad 8-core Opteron, 2 GHz
Memory:	32 G RAM each node

Clemson Cluster II

Machine:	Beowulf cluster with 256 nodes
Node:	dual 8-core Xeon/Opteron mix, 2.33 GHz
Memory:	12-16G RAM each node
Message Passing:	MPICH

Computational Results: SYMPHONY (MIPLIB)

Procs	1		4		16	
Instance	Nodes	Time	Nodes	Time	Nodes	Time
10teams	5823	248	2.23	1.88	2.44	5.28
air04	1659	700	1.18	4.46	1.31	9.46
air05	4637	1037	1.11	3.72	0.85	12.65
arki001	137852	0.02*	5.37	0.78*	18.24	0.82*
bell5	1251108	1812	4.21	0.78	1.76	4.55
danoint	78781	3.85*	4.39	0.83*	18.03	0.67*
fast0507	3800	2.08*	4.41	0.67*	10.76	0.37*
gesa2	18478	492	0.97	4.87	1.01	14.06
gesa2_o	29259	617	1.16	4.31	1.33	10.11
gt2	2536560	0.88*	2.64	0.33*	7.61	0.74*
harp2	145301	0.21*	12.14	2.36*	40.94	2.43*
mas74	2139430	29.44*	3.28	0.12*	9.59	0.03*
mas76	467776	578	1.02	2.41	1.21	6.49
misc07	11665	171	0.73	4.89	1.10	11.40
mod011	2694	235	1.01	3.73	1.07	10.68
modglob	1032435	0.03*	1166086	1138	1.00	2.98
noswot	1455708	4.65*	3.71	1.00*	10.62	1.00*
pk1	578584	1251	0.79	2.11	0.84	3.38
pp08aCUTS	166875	828	0.38	8.20	0.36	22
pp08a	44966	199	0.28	11.06	0.27	25
qiu	110429	6.96*	134178	1373	1.04	4.56
qnet1_o	271107	2.40*	4.94	0.58*	10.76	1.93*
rout	499757	NS	1748134	10.00*	3.76	0.35*
set1ch	144740	22.02*	7.59	0.84*	26.45	0.78*
seymour	7791	6.77*	4.54	0.94*	16.15	0.77*

Results on COR@L cluster with OpenMP version of SYMPHONY

Computational Results: SYMPHONY (VRPLIB)

Procs	1		4		8		16	
Instance	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
A-n54-k7	86993	2.37*	2.87	0.60*	4.63	0.35*	6.98	0.25*
A-n37-k6	12862	292	0.83	3.95	1.31	2.63	1.32	3.60
A-n44-k6	5116	196	0.80	4.26	0.83	4.26	1.20	4.17
A-n45-k7	139189	0.77*	422893	6858	0.77	2.00	0.93	2.23
A-n48-k7	5755	237	0.85	3.04	1.57	1.87	1.11	4.47
A-n55-k9	5851	251	1.07	2.54	1.54	1.93	0.74	6.12
A-n60-k9	85239	4.38*	3.84	0.67*	3.99	0.49*	6.20	0.69*
A-n61-k9	85151	1.40*	3.82	0.35*	4.68	0.62*	6.51	0.37*
A-n62-k8	51287	4.81*	3.07	0.94*	6.43	0.61*	8.61	0.39*
A-n63-k10	60908	3.95*	4.36	0.95*	5.49	0.97*	8.08	1.07*
A-n63-k9	38429	NS	200096	3.45*	1.72	1.03*	1.92	0.82*
A-n64-k9	40262	NS	78811	NS	237099	4.88*	302579	NS
A-n65-k9	37504	3443	0.88	3.16	1.27	3.25	0.71	9.06
A-n69-k9	57843	4.25*	4.44	0.81*	5.65	0.80*	8.29	0.93*
A-n80-k10	21803	NS	84842	NS	144898	NS	183499	NS
B-n50-k8	102800	2.55*	2.94	0.87*	5.57	0.88*	7.95	0.73*
B-n57-k9	88114	4482	1.05	3.37	1.81	2.37	1.09	5.27
B-n67-k10	25517	1678	0.97	4.07	0.68	8.61	1.16	7.88
B-n68-k9	83343	1.56*	3.93	0.52*	5.97	0.34*	7.52	0.49*
B-n78-k10	54386	0.66*	2.93	1.37*	5.41	1.00*	7.98	0.55*

Results on COR@L cluster with OpenMP version of SYMPHONY

Computational Results: KNAP (ALPS)

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

Computational Results: KNAP (ALPS)

- Tested a set of “difficult” instances on SDSC Blue Gene.
- The default algorithm was used except that
  - the static load balancing scheme was the two-level root initialization,
  - nodes generated by the master varied from 10K to 30K.
  - nodes generated by hub varied from 10K to 20K.
  - the size of a unit of work was 3K 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

- Note the inevitable increase in ramp-up and ramp-down.
- KNAP scales well even when using several thousand processors.

Computational Results: BLIS (ALPS)

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

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

Computational Results: BLIS (ALPS)

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

ALPS

- The approach used in SYMPHONY seems quite effective (once a few planned improvements are implemented) for small-scale parallelism.
- The methods implemented in ALPS seem effective in improving scalability for large-scale parallelism.
- Our long-term plan is to hybridize these two approaches.
- We still have a lot of plans for improvement.
  - load balancing,
  - fault tolerance,
  - hybrid architectures,
  - grid implementation.

Obtaining CHiPPS and SYMPHONY

The CHiPPS framework is available for download at

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

The SYMPHONY framework is available for download at

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

Thank You!

Questions?