

Disk-Based Parallel Computing: A New Paradigm

Gene Cooperman

Director, Institute for Complex Scientific Software
<http://www.icss.neu.edu/>

Head of High Performance Computing Lab

Daniel Kunkle

Xiaoqin Ma

Viet Ha Nguyen

Michael Rieker

Eric Robinson

Vlad Slavici

Ana Visan

Northeastern University
Boston, MA / USA



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

1. Simple, but Flexible Task-Based Computing
2. Marshalling
3. Checkpointing
4. Disk-Based Computation



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

1. Simple, but Flexible Task-Based Computing

(a) TOPC-C/C++ (Task Oriented Parallel C/C++)

<http://www.ccs.neu.edu/home/gene/topc.html>

(b) ParGAP (Parallel GAP) <http://www.ccs.neu.edu/home/gene/pargap.html>

(c) ParGCL (Parallel GNU Common LISP) <http://www.ccs.neu.edu/home/gene>

(d) ParGeant4 (Parallel Geant4)

Geant4 is million line C/C++ toolkit for simulation of particle-matter interaction; Designed and written at CERN and over 100 physicists at 10 national high energy physics lab

2. Marshalling

3. Checkpointing

4. Disk-Based Computation



First-Ever Computations

	<p><i>Baby Monster</i> perm. rep. (deg. $\approx 1.3 \times 10^{10}$) (over $GL(4370, 2)$)</p> <p>J_4 perm. rep. (deg. 173,067,389) (over $GL(1333, 11)$)</p> <p>Parallelization of GNU Common Lisp (GCL)</p> <p>Parallelization of GAP (Groups, Algorithms and Programming)</p> <p>Parallelization of Geant4</p>	<p><i>Th</i> condensation (from perm deg. 976,841,775 to matrix dim. 1,403) (over $GL(248, 2)$)</p> <p>J_4 condensation (from perm deg. 173,067,389 to matrix dim. 5,693) (over $GL(112, 2)$)</p> <p><i>Ly</i> perm. rep. (deg. 9,606,125) (over $GL(111, 5)$)</p>
TOP-C (shared mem.)	TOP-C (dist. mem.)	
POSIX threads	MPI (Message Passing Interface)	

TOP-C from the Command Line

```
./topcc --mpi myapp.c
```

```
[ OR:  ./topcc --pthread myapp.c
```

```
OR:  ./topcc --seq myapp.c ]
```

```
./a.out --TOPC-help
```

```
./a.out --TOPC-trace --TOPC-stats --TOPC-num-slaves=50
```

```
    --TOPC-aggregated-tasks=5  <APPLICATION_PARAMS>
```

G. Cooperman, “TOP-C: A Task-Oriented Parallel C Interface”, 5th *International Symposium on High Performance Distributed Computing (HPDC-5)*, 1996, IEEE Press, pp. 141–150



Running TOP-C

```
./topcc -c -g -O2 /tmp/topc-2.5.0/examples/parfactor.c  
./topcc -g -O2 parfactor.o  
./a.out 123456789
```

```
FACTORING 123456789  
master -> 1: 2  
master -> 2: 1002  
master -> 3: 2002  
master -> 4: 3002  
master -> 5: 4002  
1 -> master: TRUE  
    UPDATE: TRUE  
master -> 1: 5002  
...  
2 -> master: FALSE  
3 -> master: FALSE  
3 3 3607 3803
```



Getting Help with TOP-C

```
gene@auditor:/tmp/topc-2.5.0/bin$ ./a.out --TOPC-help
```

```
TOP-C Version 2.5.0 (September, 2004); (distributed (mpi) memory mod
```

```
Usage: ./a.out [ [TOPC_OPTION | APPLICATION_OPTION] ...]
```

```
--TOPC-stats[=<0/1>]          display stats before and after
```

```
[default: false]
```

```
--TOPC-verbose[=<0/1>]       set verbose mode
```

```
[default: false]
```

```
--TOPC-num-slaves=<int>      number of slaves (sys-defined default)
```

```
--TOPC-aggregated-tasks=<int> number of tasks to aggregate
```

```
[default: 1]
```

```
--TOPC-slave-wait=<int>      secs before slave starts (use w/ gdb attach)
```

```
--TOPC-slave-timeout=<int>  dist mem: secs to die if no msgs, 0=never
```

```
[default: 1800]
```

```
--TOPC-trace=<int>          trace (0: notrace, 1: trace, 2: user trace)
```

```
--TOPC-procgroup=<string>   procgroup file (--mpi) [default: "./p
```

```
--TOPC-safety=<int>        [0..20]: higher turns off optimizations,
```

```
add `TOPC_OPT_trace = 0;` before `TOPC_init(&argc, &argv);`  
Try `--TOPC-help --TOPC-verbose` for more information.
```



TOP-C on the Grid

(with H. Casanova, J. Hayes and T. Witzel)

- TOP-C was used to demonstrate parallelization of Geant4 over the Grid.
- This did not require any new code in the parallelization of Geant4. Instead, a new TOP-C communication library was written.
- The new library exists alongside existing communication libraries that support
 1. a distributed memory model (over TCP/IP sockets or MPI);
 2. a shared memory model (over POSIX threads); and
 3. a sequential model (simulating TOP-C in a sequential process, for easy debugging)

G. Cooperman, H. Casanova, J. Hayes and T. Witzel, “Using TOP-C and AMPIC to Port Large Parallel Applications to the Computational Grid”, *Proc. of 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, IEEE Press, 2002, pp. 120–127

ParGAP



```
gap> # This assumes your procgroup file includes two slave processes.
gap> PingSlave(1); #a 'true' response indicates Slave 1 is alive
true
gap> # Print() on slave appears on standard output
gap> # i.e. after the master's prompt.
gap> SendMsg( "Print(3+4)" );
gap> 7
gap> # A <return> was input above to get a fresh prompt.
gap> #
gap> # To get special characters (including newline: '\n')
gap> # into a string, escape them with a '\\'.
gap> SendMsg( "Print(3+4, \"\\n\")" );
gap> 7

gap> # Again, a <return> was input above after the 7 and new-line
gap> # were printed to get a fresh prompt.
gap> #
gap> # Each SendMsg() is normally balanced by a RecvMsg().
gap> SendMsg( "3+4", 2);
gap> RecvMsg( 2 );
7
gap> # The following is equivalent to the two previous commands.
gap> SendRecvMsg( "3+4", 2);
7
```

2

```
gap> # As with Print() the result of Exec() appears on standard
gap> # output. Print() and Exec() are each 'no-value' functions,
gap> # and so the result of a RecvMsg() in these cases
gap> # is "<no_return_val>".
gap> SendRecvMsg( "Exec(\"pwd\")" ); # Your pwd will differ :- )
/home/gene
"<no_return_val>"
gap> # Put default slave into an infinite loop.
gap> SendMsg("while true do od");
gap> # Default slave can't execute the next command until it's
gap> # finished with the previous command.
gap> SendMsg("Print(\"WAKE UP\\n\")");
gap> # Check to see if a message is waiting to be collected but
gap> # return immediately (i.e. don't get blocked by waiting for
gap> # a message to appear). A 'false' response indicates the
gap> # infinite loop hasn't terminated and produced a value yet!
gap> ProbeMsgNonBlocking();
false
gap> # Send an interrupt to each slave, slave 1 will see the
gap> # following command and print 'WAKE UP', and then all
gap> # pending messages are flushed.
gap> ParReset();
... resetting ...
WAKE UP
0
gap> # The return value, 0, from ParReset() indicates there
gap> # were 0 pending messages flushed, confirming correctness
gap> # of ProbeMsgNonBlocking() when it returned "false"
gap> SendRecvMsg( "a:=45; 3+4", 1 );
```

```
gap> RecvMsg( 2 ); # No value for last SendMsg() command
"<no_return_val>"
gap> RecvMsg( 1 );
45
gap> myfnc := function() return 42; end;;
gap> # Use PrintToString() to define myfnc on all slave processes
gap> BroadcastMsg( PrintToString( "myfnc := ", myfnc ) );
gap> SendRecvMsg( "myfnc()", 1 );
42
gap> FlushAllMsgs(); # There are no messages pending.
0
gap> # Execute analogue of GAP's List() in parallel on slaves.
gap> squares := ParList( [1..100], x->x^2 );
[ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
  289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841,
  900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600,
  1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601,
  2704, 2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844,
  3969, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329,
  5476, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056,
  7225, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025,
  9216, 9409, 9604, 9801, 10000 ]
gap> # Ensure problem environment is read into master and slaves.
gap> # Try one of your GAP program files instead.
gap> ParRead( "/home/gene/myprogram.g" );
```

Geant4



The poster features the Geant4 logo and a URL <http://cern.ch/geant4>. It includes several images and text blocks:

- GLAST Gamma-ray Large Area Space Telescope**: A satellite in space.
- Borexino at Gran Sasso Laboratory**: A circular detector structure.
- ESA XMM X-ray telescope**: A satellite with a large mirror.
- CMS at LHC, CERN**: A large cylindrical detector.
- BaBar at SLAC**: A detector structure.
- High energy μ** : A graph showing a peak at high energy, courtesy of L3.
- Photon attenuation**: A graph showing attenuation length vs energy, with data from Geant4 and NIST.
- Neutrons**: A graph showing neutron flux vs energy, courtesy of CMS.
- Stopping α** : A graph showing stopping power vs energy, with data from Geant4 and experimental data.

Text on the poster: "Geant4 is a toolkit for the simulation of the passage of particles through matter. It has been developed and maintained by a world-wide Collaboration of approximately 100 scientists." and "Its application areas include high energy physics, astrophysics and nuclear physics experiments, medical, accelerator and space science studies."

At the bottom, logos for various institutions are shown: Budker Inst. of Physics, IHEP, Protvino, MPPHI Moscow, Pittsburgh University, Jefferson Lab, PPARC, Stanford Linear Accelerator Center, TERA, and others.

S. Agostinelli et al., “Geant4: A Simulation Toolkit”, *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **506**(3), July 1, 2003, pp. 250–303 (over 100 authors, incl. G. Cooperman)

J. Allison et al., “Geant4 Developments and Applications”, *IEEE Transactions on Nuclear Science* **53**(1), 2006, pp. 270–278, (73 authors, incl. G. Cooperman)

G. Cooperman and V.H. Nguyen and I. Malioutov, “Parallelization of Geant4 Using TOP-C and Marshalgen”. *The 5th IEEE International Symposium on Network Computing and Applications* (NCA-06), IEEE Computer Society Press, 2006, pp. 48–55



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

1. Simple, but Flexible Task-Based Computing
2. **Marshalgen** — <http://www.ccs.neu.edu/home/gene/marshalgen.html>
 - annotate fields of class in .h file with comments: deep copy, shallow copy, pointer to table, etc.
 - preprocessor automatically generates marshalling code
 - supports class inheritance and templates, in addition to marshaling facilities of standard packages such as CORBA
3. Checkpointing
4. Disk-Based Computation



Marshalgen Example: Easy Marshaling/Serialization in Geant4

(with V.H. Nguyen)

```
//MSH_BEGIN
class G4HCofThisEvent
{ ...
  // data members
private:
  G4std::vector<G4VHitsCollection*> *HC; /* MSH: ptr_as_array
    [elementType: G4VHitsCollection*]
    [elementCount: { $ELE_COUNT = $THIS->GetNumberOfCollections(); }]
    [elementGet: { $ELEMENT = $THIS->GetHC($ELE_INDEX); }]
    [elementSet: { $THIS-> AddHitsCollection($ELE_INDEX, $ELEMENT); }]
  */
  // member methods
public:
  inline G4VHitsCollection* GetHC(G4int i) { return (*HC)[i]; }
  inline G4int GetNumberOfCollections() { ... }
  void AddHitsCollection(G4int HCID, G4VHitsCollection * aHC);
  ...
}
```



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

1. Simple, but Flexible Task-Based Computing
2. Marshalling
3. **Checkpointing** (currently experimental)

```
checkpoint ./a.out
```

```
restart checkpoint_file
```

Transparent, Library-based Checkpointing:

- user-level, transparent: no modification of kernel; no modific. of application source code
- supports multi-threading
- distributed, socket-based checkpointing: independent of MPI or other message-sending protocol
- checkpointing propagates to forked child processes, and to remotely spawned processes (e.g. via *ssh* or Globus protocols version, *gsissh*)



Checkpointing (previous work)

- Checkpointing of master-worker computations (TOP-C)
- By checkpointing only the master and some state of the workers, time and storage is saved.
- Upon restart, also restart any pending tasks

G. Cooperman, J. Ansel, and X. Ma, *Transparent adaptive library-based checkpointing for master-worker style parallelism*, *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid06)*, pp. 283–291, Singapore, 2006, IEEE Press



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

1. Simple, but Flexible Task-Based Computing
2. Marshalling
3. Checkpointing
4. **Disk-Based Computation**



Disk as the New RAM

Bandwidth of RAM: 3.2 GB/s (PC-3200 RAM, single channel)

Bandwidth of Disk: ~ 50 MB/s

Bandwidth of Cluster of 64 nodes: 3.2 GB/s

Issues: Bandwidth of Network, ability of CPU to keep up



Disk: the New RAM (example)

Initial Testbed: large search and enumeration

- Key data structure: sorted array
- Key algorithm: sorting \Rightarrow *merge, union, intersection*
(sorting on disk done as *external sort*: 4 passes in practice; fewer passes when there are opportunities to pipeline it with previous phase of computation)

Problem: Insertion of new elements

Solution: Defer insertions; sort elements to insert; and merge them into sorted array in large batch



Duplicate Elimination in Baby Monster

Optimization: eliminate duplicate insertions before merge; Use a new hash array in RAM to accumulate elements to insert. Need only store one bit per hash element: 1 = present; 0 = not present

Example: AI search: enumeration of states via open queue, as in breadth-first search

1. If element to insert hashes to 0, it is new; add to *open queue* on disk
2. If element to insert hashes to 1, it is either a hash collision or a duplicate: add to *collision queue* on disk
3. Continue to read from open queue and hash its neighbors: neighbors will also be stored either in open queue or collision queue
4. sort collision queue and eliminate duplicates
5. sort open queue
6. merge collision queue, open queue and original sorted array on disk
7. elements of collision queue that are determined to be new become the next open queue, and we repeat step 1.



Duplicate Elimination in Baby Monster (case 2)

- Hash array too large for RAM; must be stored on disk
 1. All new elements to insert are saved on disk in *open queue*
 2. As neighbors of elements in open queue are expanded, portions of the open queue are transferred into a closed set
 3. The closed set is then externally sorted according to hash index
 4. The closed set is then merged into the existing hash array

NOTE: Both RAM-based and disk-based hash arrays adapt easily to distributed computing. Each node is responsible for a contiguous sequence of hash indexes.



Memory Wall

CPU/RAM	New, Two-Pass Algorithm	Traditional Algorithm
2.66 GHz Pentium 4 DDR-266 RAM	0.042 s	0.159 s
0.6 GHz Pentium III PC-100 RAM	0.131 s	0.097 s

Two-Pass Permutation Multiplication versus Traditional Algorithm (joint work: X. Ma, V.H. Nguyen and C.)

```
Object Z[N], Y[N]; // Object is ``int`` in above experiments
int X[N];
for (i=0; i<N; i++)
    Z[i]=Y[X[i]];
```



Why is Two-Pass Permutation Now Faster?

Two Large Reasons:

1. The Pentium 4 has a longer cache line.

The Pentium 4 has a 128 byte cache line: four times longer than the 32 byte cache line of the Pentium III.

2. The bandwidth of DDR-266 (PC-2100) RAM is higher, but the latency is not faster.

- *DDR-266/PC-2100 has a bandwidth of 2.2 GHz, as compared to 1.1 GHz for older PC-100 RAM.*
- *The latency of DDR-266 RAM and PC-100 RAM are both about 25 ns.*

Application: Search and Enumeration Problems

Branch-and-Bound, A* search

Given a state, and a generator/operation, produce a new state

This gives rise to a natural graph in which nodes correspond to states, and edges are labelled by generators or operations. A search/enumeration proceeds by breadth-first search, developing a spanning tree.

Potential applications (some of it is future work):

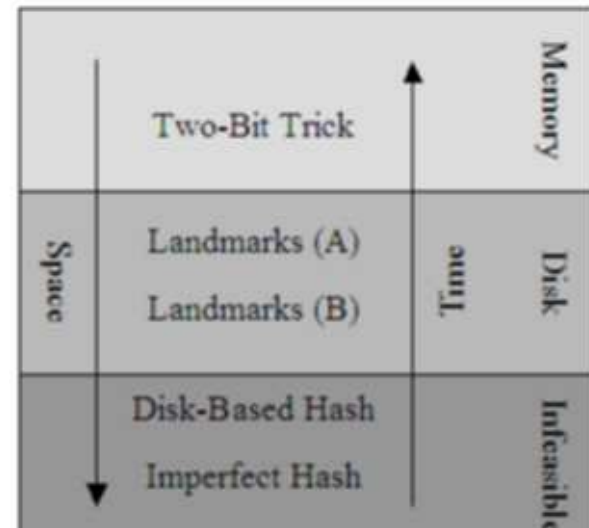
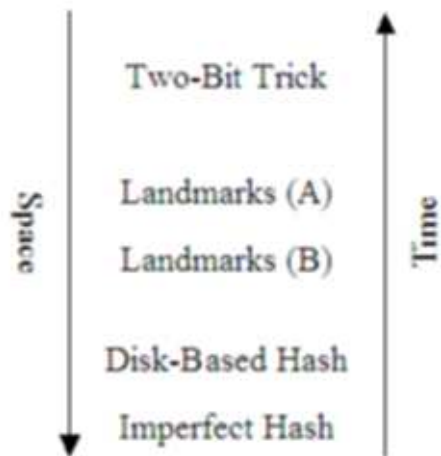
- Enumeration of Orbit Elements
- Orderly Generation of Brendan McKay (symmetry and search)
- Gröbner bases, Knuth-Bendix, similar “completion algorithms”
- SAT (satisfiability) *Example use: VLSI circuit verification*
- Integer Programming
Example use: Travelling Salesman Problem, Airline schedules

Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

1. *Simple, but Flexible Task-Based Computing*
2. *Marshalling*
3. *Checkpointing*
4. **Disk-Based Computation**

General Philosophy in case of Search:





Two-Bit Trick

- Assumes dense, perfect hash function w/ inverses (no hash collisions)
- Breadth-first search, storing level of node *modulo 3* of spanning tree in hash table (2 bits/node)
- Given a node, can now find minimal length path to origin:
 1. Look up level of current state in hash table
 2. Given state, use operators to find all neighbors of node
 3. Look up levels of all neighboring states in hash table
 4. Choose a state whose level is one less than the current level, modulo 3
 5. Repeat on the newly chosen state

Showed Rubik's $2 \times 2 \times 2$ cube (corners, only) always solvable in 11 moves. Used 1 MB on a SUN-3 workstation having only 4 MB of RAM. G. Cooperman, L. Finkelstein, and N. Sarawagi, Applications of Cayley Graphs, Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC-8), Springer-Verlag Lecture Notes in Computer Science **508**, pp. 367–378, 1990. (Also in G. Cooperman and L. Finkelstein. “New methods for using Cayley graphs in interconnection networks”, *Discrete Applied Mathematics*, **37/38**, pp. 95–118, 1992.)



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

1. *Simple, but Flexible Task-Based Computing*
2. *Marshalling*
3. *Checkpointing*
4. **Disk-Based Computation**
 - (a) Data Structure: Distributed Database of Key-Value Pairs
 - (b) Building Blocks: Algorithmic Subroutines
 - (c) Integration into General Search Routines
 - (d) Example Large Computations: Baby Monster; Rubik's Cube
 - (e) Other Applications
 - (f) Natural API (in progress)



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

4. Disk-Based Computation

(a) Data Structure: Distributed Database of Key-Value Pairs

i. Goals

A. Key-Values: *Set(key, value); Get(key); Delete(key)*

B. Duplicate Elimination

ii. Data Structures for Database

A. Distributed Hash Array

B. Distributed Sorted Array

C. Double Hashed Array: (hybrid of above two data structures)

(b) Building Blocks: Algorithmic Subroutines

(c) ...



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

4. Disk-Based Computation

(a) Data Structure: Distributed Database of Key-Value Pairs

i. Goals

ii. Data Structures for Database

(b) **Building Blocks: Algorithmic Subroutines**

distributed hashing, sorting, duplicate elimination, binary search, batching of queries, pipelining of computations, striped access to distributed data structures, on-the-fly compression and expansion of data structures, Bloom filters, two-phase commit in support of persistent data, structures, ...

(c) ...



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

4. Disk-Based Computation

(a) Data Structure: Distributed Database of Key-Value Pairs

i. Goals

ii. Data Structures for Database

(b) Building Blocks: Algorithmic Subroutines

i. **EXAMPLE: Bloom filters:** Use hash array with only one bit per hash entry; We wish only to record if key is present or not present in hash table; Use k hash functions, and for a given key, set k bits of hash table (one bit for each hash function); To test presence of key, test all k bits; This greatly reduces hash collisions.

(c) ...



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

4. Disk-Based Computation

(a) Data Structure: Distributed Database of Key-Value Pairs

ii. Data Structures for Database

A. Distributed Hash Array: Good for key-value database

Batching of queries important for efficiency

B. Distributed Sorted Array: Good for duplicate elimination

Given source of new key-values, externally sort it, and compare with original sorted array; Merge on the fly

C. Doubly Hashed Array: Good for duplicate elimination

Key-value pairs stored in buckets, based on high bits of hash index; High bits also determines node to hold bucket;

Key-value pair stored unsorted in bucket; For duplicate elimination, sort elements of bucket in RAM



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

4. Disk-Based Computation

(d) Example Large Computations:

i. **Construction of Permutation Representation of Baby Monster**

GOAL: enumerate all 13,571,955,000 “points”

Each point given as a vector of dimension 4,370 over $GF(2)$
(547 bytes per “point”)

STORAGE: about 7 terabytes ($13,571,955,000 \times 547$ bytes)

TIME: About 750 hours BOTTLENECK: RAM: limited by speed
of reading vectors/matrices from RAM for matrix-vector multiplication

ii. Rubik’s Cube



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

4. Disk-Based Computation

(d) Example Large Computations:

i. Construction of Permutation Representation of Baby Monster

ii. Rubik's Cube

$\sim 4.3 \times 10^{19}$ states

Square subgroup of about 6.6×10^5 elements

SUBGOAL: enumerate all 6.5×10^{13} cosets ($4.3 \times 10^{19} / (6.6 \times 10^5)$)

Reduction: Only enumerate cosets up to symmetries of cube

About 1.5×10^{12} symmetrized cosets

STORAGE: 1 byte per symmetrized coset (1.5 terabytes) times a factor of at least two for frontier expansion in search



Outline

Requirements of Scalable, Parallel Computing for Symbolic Algebra

1. *Simple, but Flexible Task-Based Computing*

2. *Marshalling*

3. *Checkpointing*

4. **Disk-Based Computation**

(e) **Other Applications:**

i. Integer Programming

Example use: Travelling Salesman Problem, Airline schedules

ii. SAT (satisfiability) *Example use: VLSI circuit verification*

iii. Applications to distributed linear algebra

(f) Natural API (in progress)