Distributed parallel
Gröbner bases computation

Heinz Kredel

ECDS at CISIS 2009, Fukuoka
Overview

• Introduction to JAS
• Gröbner bases
  – problems with parallel computation
  – sequential and parallel algorithm
• Distributed algorithm
  – execution middle-ware
  – data structure middle-ware
  – workload paradox
• Conclusions and future work
Java Algebra System (JAS)

- object oriented design of a computer algebra system
  = software collection for symbolic (non-numeric) computations
- type safe through Java generic types
- thread safe, ready for multi-core CPUs
- use dynamic memory system with GC
- 64-bit ready
- jython (Java Python) interactive scripting front end
Implementation overview

- 200+ classes and interfaces
- plus ~90 JUnit test cases
- uses JDK 1.6 with generic types
  - Javadoc API documentation
  - logging with Apache Log4j
  - build tool is Apache Ant
  - revision control with Subversion
- jython (Java Python) scripts
  - support for Sage like polynomial expressions
- open source, license is GPL or LGPL
Polynomial functionality

**RingElem**
- `isZERO()` : boolean
- `isONE()` : boolean
- `isUnit()` : boolean
- `equals(a : Object) : boolean`
- `hashCode() : int`
- `compareTo(a : C) : int`
- `clone() : C`
- `negate() : C`
- `sum(a : C) : C`
- `subtract(a : C) : C`
- `multiply(a : C) : C`
- `inverse() : C`
- `divide(q : C) : C`
- `remainder(q : C) : C`

**RingFactory**
- `getZERO() : C`
- `getONE() : C`
- `fromInteger(i : long) : C`
- `random(n : int) : C`
- `copy(a : C) : C`
- `parse(s : String) : C`
- `isField() : boolean`
- `isCommutative() : boolean`
- `isAssociative() : boolean`
- `characteristic() : int`

**GenPolynomialRing**
- `GenPolynomialRing(coFac : RingFactory, n : int)`
- `GenPolynomialRing(coFac : RingFactory, n : int, to : TermOrder)`
- `GenPolynomialRing(coFac : RingFactory, n : int, to : TermOrder, v : String[])`
- `contract(i : int) : GenPolynomialRing`
- `extend(i : int) : GenPolynomialRing`
- `toString() : String`
- `random(k : int, l : int, d : int, q : float) : GenPolynomial`

**GenPolynomial**
- `GenPolynomial(r : GenPolynomialRing)`
- `GenPolynomial(r : GenPolynomialRing, c : C, e : ExpVector)`
- `GenPolynomial(r : GenPolynomialRing, m : SortedMap)`
- `leadingBaseCoefficient() : C`
- `leadingExpVector() : ExpVector`
- `leadingMonomial()`
- `length() : int`
- `extend(r : GenPolynomialRing, j : int, k : long) : GenPolynomial`
- `contract(r : GenPolynomialRing) : GenPolynomial`
- `toString() : String`
- `toString(v : String[]) : String`
- `gcd(a : GenPolynomial) : GenPolynomial`
- `modInverse(m : GenPolynomial) : GenPolynomial`
Example: Legendre polynomials

\[ P[0] = 1; \quad P[1] = x; \]
\[ P[i] = \frac{1}{i} ((2i-1) \cdot x \cdot P[i-1] - (i-1) \cdot P[i-2]) \]

```java
BigRational fac = new BigRational();
String[] var = new String[] { "x" };
GenPolynomialRing<BigRational> ring = new GenPolynomialRing<BigRational>(fac,1,var);
List<GenPolynomial<BigRational>> P = new ArrayList<GenPolynomial<BigRational>>(n);
GenPolynomial<BigRational> t, one, x, xc, xn; BigRational n21, nn;

one = ring.getONE(); x = ring.univariate(0);
P.add( one ); P.add( x );
for ( int i = 2; i < n; i++ ) {
    n21 = new BigRational( 2*i-1 ); xc = x.multiply( n21 );
    t = xc.multiply( P.get(i-1) );
    nn = new BigRational( i-1 ); xc = P.get(i-2).multiply( nn );
    t = t.subtract( xc ); nn = new BigRational(1,i);
    t = t.multiply( nn ); P.add( t );
}

int i = 0;
for ( GenPolynomial<BigRational> p : P ) {
    System.out.println("P["+(i++)+"] = " + p);
}
```
Overview

• Introduction to JAS

• Gröbner bases
  – problems with parallel computation
  – sequential and parallel algorithm

• Distributed algorithm
  – execution middle-ware
  – data structure middle-ware
  – workload paradox

• Conclusions and future work
Gröbner bases

- canonical bases in polynomial rings $R = C[x_1, \ldots, x_n]$
- like Gauss elimination in linear algebra
- like Euclidean algorithm for univariate polynomials
- with a Gröbner base many problems can be solved
  - solution of non-linear systems of equations
  - existence of solutions
  - solution of parametric equations
- slower than multivariate Newton iteration in numerics
- but in computer algebra no round-off errors
- so guarantied correct results
Buchberger algorithm

algorithm: G = GB( F )
input: F a list of polynomials in R[x1,...,xn]
output: G a Gröbner Base of ideal(F)

G = F;
B = \{ (f,g) \mid f, g \text{ in } G, f \neq g \};
while ( B \neq {} ) {
    select and remove (f,g) from B;
    s = S-polynomial(f,g);
    h = normalform(G,s); // expensive operation
    if ( h \neq 0 ) {
        for ( f in G ) { add (f,h) to B }
        add h to G;
    }
} // termination ? Size of B changes
return G
Problems with GB algorithm

- requires exponential space (in the number of variables)
- even for arbitrary many processors no polynomial time algorithm will exist
- highly data depended
  - number of pairs unknown (size of B)
  - size of polynomials s and h unknown
    - size of coefficients
    - degrees, number of terms
- management of B is sequential
- strategy for the selection of pairs from B
  - depends moreover on speed of reducers
Gröbner bases classes

GroebnerBase
+ isGB(F : List<GenPolynomial>) : boolean
+ isGB(modv : int, F : List<GenPolynomial>) : boolean
+ GB(F : List<GenPolynomial>) : List<GenPolynomial>
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>
+ extGB(F : List<GenPolynomial>) : ExtendedGB
+ extGB(modv : int, F : ExtendedGB) : ExtendedGB
+ minimalGB(G : List<GenPolynomial>) : List<GenPolynomial>

Reduction
+ normalform(F : List<GenPolynomial>, p : GenPolynomial) : GenPolynomial

GroebnerBaseAbstract
+ GroebnerBaseAbstract(red : Reduction)
+ isGB(F : List<GenPolynomial>) : boolean
+ isGB(modv : int, F : List<GenPolynomial>) : boolean
+ GB(F : List<GenPolynomial>) : List<GenPolynomial>
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>
+ extGB(F : List<GenPolynomial>) : ExtendedGB
+ extGB(modv : int, F : List<GenPolynomial>) : ExtendedGB
+ minimalGB(G : List<GenPolynomial>) : List<GenPolynomial>

GroebnerBaseParallel
+ GroebnerBaseParallel(threads : int, red : Reduction)
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>

GroebnerBaseDistributed
+ GroebnerBaseDistributed(threads : int, red : Reduction, port : int)
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>

GroebnerBaseSeq
+ GroebnerBaseSeq(red : Reduction)
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>
Overview

- Introduction to JAS
- Gröbner bases
  - problems with parallel computation
  - sequential and parallel algorithm
- Distributed algorithm
  - execution middle-ware
  - data structure middle-ware
  - workload paradox
- Conclusions and future work
bwGRiD cluster architecture

- 8-core CPU nodes @ 2.83 GHz, 16GB, 140 nodes
- shared NFS/Lustre home directories
- InfiniBand and 1 G Ethernet interconnects
- managed by PBS batch system with Maui scheduler
- running Java 64bit server VM 1.6 with 4+GB memory
- start Java VMs with daemons on allocated nodes
- communication via TCP/IP interface to InfiniBand
- no Java high performance interface to InfiniBand
- alternative Java via MPI not studied
- other middle-ware ProActive or GridGain not studied
Distributed GB computation

- main method GB()
- distribute list G via distributed hash table (DHT)
- start ReducerServer threads
- method clientPart() starts ReducerClients
- select pair and send to distributed client
  a) send polynomials them-selves
  b) send index of polynomial in G
- client performs S-polynomial and normalform computation sends result back to master
- master eventually inserts new pairs to B and adds polynomial to G in DHT
mtype = { Get, Fin, Pair, Hpol };  
proctype ReducerServer (chan pairs) {  
do  
:: idler++; pairs ? Get;  
if  
:: (! nextPair && idler == PROCNUM) -> pairs ! Fin; break;  
:: (! nextPair) -> skip; // sleep delay  
:: else skip;  
fi;  
idler--; getPair(); /* take pair from queue */  
pairs ! Pair; /* send to client */  
progress: skip;  
pairs ? Hpol; /* receive result */  
addPair(); /* add new pairs to queue */  
od;  
}  
proctype ReducerClient (chan pairs) {  
do  
:: pairs ! Get;  
if  
:: pairs ? Fin -> break;  
:: pairs ? Pair -> /* compute h-pol */ pairs ! Hpol;  
fi  
od  
}
Middle-ware overview

GBDist

Distributed ThreadPool

Reducer Server

GB()

DHT Client

DHT Server

ExecutableServer

Distributed Thread

Reducer Client

clientPart()

DHT Client

master node

InfiniBand

client node
Execution middle-ware (nodes)

- on compute nodes do basic bootstrapping
  - start daemon class `ExecutableServer`
  - listens on connections (no security constrains)
  - start thread with `Executor` for each connection
  - receives (serialized) objects with `RemoteExecutable` interface
  - execute the `run()` method
  - communication and further logic is implemented in the `run()` method
- multiple processes as threads in one JVM
Execution middle-ware (master)

- on master node
  - **start** DistThreadPool **similar to** ThreadPool
  - starts threads for each compute node
  - list of compute nodes taken from PBS
  - starts connections to all nodes with ExecutableChannel
  - can start multiple tasks on nodes to use multiple CPU cores via `open(n)` method
  - **method** `addJob()` on master
  - send a job to a remote node and wait until termination (RMI like)
Execution middle-ware usage

- Gröbner base master `GBDist`
- initialize `DistThreadPool` with PBS node list
- initialize `GroebnerBaseDistributed`
- `execute()` method of `GBDist`
  - add remote computation classes as jobs
  - `execute clientPart()` method in jobs
    - is `ReducerClient` above
  - calls main `GB()` method
    - is `ReducerServer` above
Data structure middle-ware

- sending of polynomials involves
  - serialization and de-serialization time
  - and communication time
- avoid sending via a distributed data structure
- implemented as distributed list
- runs independently of main GB master
- **setup in** `GroebnerBaseDistributed constructor` and `clientPart()` **method**
- then only indexes of polynomials need to be communicated
Distributed polynomial list

- distributed list implemented as distributed hash table (DHT)
- key is list index
- class `DistHashTable` similar to `java.util.HashMap`
- methods `clear()`, `get()` and `put()` as in `HashMap`
- method `getWait(key)` waits until a value for a key has arrived
- method `putWait(key, value)` waits until value has arrived at the master and is received back
- no guaranty that value is received on all nodes
DHT implementation (1)

- implemented as central control DHT
- client part on node uses TreeMap as store
- client DistributedHashTable connects to master
- master class DistributedHashTableServer
- put() methods send key-value pair to a master
- master then broadcasts key-value pair to all nodes
- get() method takes value from local TreeMap
DHT implementation (2)

- in future implement DHT with decentralized control
- in future implement with generic types
- in master process de-serialization of polynomials should be avoided
- broadcast to clients in master serializes polynomials for every client again
- master is co-located to master of GB computation on same compute node
- this doubles memory requirements on master node
- this increases the CPU load on the master
  - limits scaling of master for more nodes
Performance

- multi-threaded computation
  - scales well to 8 CPU cores
  - 0.4 % overhead on one thread to sequential
- distributed computation
  - scales only to 4 compute nodes
  - absolute computing times comparable to multi-threaded case for up to 4 nodes
    - not too much communication overhead
    - can use multiple cores on nodes
  - InfiniBand is essential

workload paradox, selection strategies
Multi-threaded Gröbner basis

GBs of Katsuras example on a grid cluster

Mon Mar 09 17:23:21 2009

GBs of Katsuras example on a grid cluster
Distributed Gröbner basis

GBs of Katsuras example on a grid cluster

Mon Mar 09 17:34:15 2009

GBs of Katsuras example on a grid cluster
Workload paradox

- parallel: 135 - 154 polynomials, 663 - 686 pairs
- distributed: 171 - 338 polynomials, 699 - 862 pairs
- possible pairs 9.045 – 56.953
  - rest avoided with 'criterions' and strategies
- different computation times in parallel reduction
  - pair polynomial size varies
  - size of polynomials in list varies
- different order of new pairs inserted in B
- different order of pairs removed from B
Table 1. Distributed timings, Katsura 6.

<table>
<thead>
<tr>
<th>algo.</th>
<th>#threads</th>
<th>#VM</th>
<th>time</th>
<th>put</th>
<th>rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>1</td>
<td>1</td>
<td>160.2</td>
<td>70</td>
<td>327</td>
</tr>
<tr>
<td>par</td>
<td>1</td>
<td>1</td>
<td>157.0</td>
<td>70</td>
<td>327</td>
</tr>
<tr>
<td>par</td>
<td>2</td>
<td>1</td>
<td>82.2</td>
<td>72</td>
<td>329</td>
</tr>
<tr>
<td>dist</td>
<td>1</td>
<td>1</td>
<td>177.2</td>
<td>77</td>
<td>334</td>
</tr>
<tr>
<td>dist</td>
<td>2</td>
<td>2</td>
<td>92.2</td>
<td>90</td>
<td>347</td>
</tr>
<tr>
<td>dist</td>
<td>4</td>
<td>2</td>
<td>56.2</td>
<td>112</td>
<td>369</td>
</tr>
<tr>
<td>dist</td>
<td>8</td>
<td>2</td>
<td>58.9</td>
<td>255</td>
<td>516</td>
</tr>
<tr>
<td>dist</td>
<td>4</td>
<td>4</td>
<td>51.2</td>
<td>117</td>
<td>374</td>
</tr>
<tr>
<td>dist</td>
<td>6</td>
<td>4</td>
<td>43.7</td>
<td>129</td>
<td>386</td>
</tr>
<tr>
<td>dist</td>
<td>8</td>
<td>4</td>
<td>62.9</td>
<td>259</td>
<td>519</td>
</tr>
</tbody>
</table>

Computing times in seconds on a 32 CPU Intel Xeon SMP computer running at 2.7 GHz and with 32 GB RAM. JVM 1.4.2 started with AggressiveHeap and UseParallelGC. Columns: #VMs = number of distinct Java virtual machines. put = number of polynomials put to pair list, rem = number of pairs removed from pair list.
### Table 2. Multi-threaded timings, Katsura 7.

<table>
<thead>
<tr>
<th># threads</th>
<th>time</th>
<th>speedup</th>
<th>put</th>
<th>rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>7435854</td>
<td>1.0</td>
<td>135</td>
<td>663</td>
</tr>
<tr>
<td>1</td>
<td>7424640</td>
<td>1.00</td>
<td>135</td>
<td>663</td>
</tr>
<tr>
<td>2</td>
<td>4733708</td>
<td>1.57</td>
<td>141</td>
<td>669</td>
</tr>
<tr>
<td>3</td>
<td>3212655</td>
<td>2.31</td>
<td>142</td>
<td>669</td>
</tr>
<tr>
<td>4</td>
<td>2470152</td>
<td>3.01</td>
<td>147</td>
<td>677</td>
</tr>
<tr>
<td>5</td>
<td>1937110</td>
<td>3.83</td>
<td>149</td>
<td>681</td>
</tr>
<tr>
<td>6</td>
<td>1568348</td>
<td>4.74</td>
<td>146</td>
<td>671</td>
</tr>
<tr>
<td>7</td>
<td>1116218</td>
<td>6.66</td>
<td>151</td>
<td>679</td>
</tr>
<tr>
<td>8</td>
<td>1247666</td>
<td>5.95</td>
<td>154</td>
<td>686</td>
</tr>
</tbody>
</table>

Columns: put = number of polynomials put to pair list, rem = number of pairs removed from pair list.
Table 3. Distributed timings, Katsura 7.

<table>
<thead>
<tr>
<th># nodes</th>
<th>time</th>
<th>speedup</th>
<th>put</th>
<th>rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>7435854</td>
<td>1.0</td>
<td>135</td>
<td>663</td>
</tr>
<tr>
<td>2</td>
<td>4260202</td>
<td>1.74</td>
<td>171</td>
<td>699</td>
</tr>
<tr>
<td>3</td>
<td>2990191</td>
<td>2.48</td>
<td>195</td>
<td>726</td>
</tr>
<tr>
<td>4</td>
<td>2385904</td>
<td>3.11</td>
<td>216</td>
<td>745</td>
</tr>
<tr>
<td>5</td>
<td>2243687</td>
<td>3.31</td>
<td>233</td>
<td>764</td>
</tr>
<tr>
<td>6</td>
<td>1784650</td>
<td>4.16</td>
<td>255</td>
<td>786</td>
</tr>
<tr>
<td>7</td>
<td>2684213</td>
<td>2.77</td>
<td>287</td>
<td>814</td>
</tr>
<tr>
<td>8</td>
<td>3522735</td>
<td>2.11</td>
<td>338</td>
<td>862</td>
</tr>
</tbody>
</table>

Columns: put = number of polynomials put to pair list, rem = number of pairs removed from pair list.
Selection strategies

- best to use the same order of polynomials and pairs as in sequential algorithm
- selection algorithm is sequential
  - so optimizations reduce parallelism
- Amrhein & Gloor & Küchlin:
  - work parallel: n reductions in parallel
  - search parallel: select best from k results
- Kredel:
  - n reductions in parallel, select first finished
  - select result in same sequence as reduction is started, not the first finished
Conclusions

- first version of a distributed GB algorithm
- runs on a HPC cluster in PBS environment
- shared memory parallel version scales up to 8 CPUs
- runtime of distributed version is comparable to parallel version
- can the workload paradox be solved?
- developed classes fit in Gröbner base class hierarchy
- new package is type-safe with generic types (with the exception of the distributed hash table)
Future work

- profile and study run-time behavior in detail
- investigate other grid middle-ware
- improve integration into the grid environment
- improve serialization in distributed list
- study other result selection strategies
- develop hybrid GB algorithm
  - distributed and multi-threaded on nodes
- compute sequential Gröbner bases with respect to different term orders in parallel
Thank you

- Questions or Comments?
- http://krum.rz.uni-mannheim.de/jas
- Thanks to
  - Raphael Jolly
  - Thomas Becker
  - Hans-Günther Kruse
  - bwGRiD for providing computing time
  - Adrian Schneider
  - the referees
  - and other colleagues