Distributed Gröbner bases computation with MPJ

Heinz Kredel, University of Mannheim

EOOPS at AINA 2013, Barcelona
Overview

• Introduction to JAS
• Communication middle-ware: sockets and MPJ
  - execution middle-ware
  - data structure middle-ware
  - comparison
• Gröbner bases: sockets and MPJ
  - sequential and parallel algorithm
  - distributed algorithm
  - hybrid multi-threaded distributed algorithm
• 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) and jruby (Java Ruby) interactive scripting front ends
Overview

• Introduction to JAS

• Communication middle-ware: sockets and MPJ
  - execution middle-ware
  - data structure middle-ware
  - comparison

• Gröbner bases: sockets and MPJ
  - sequential and parallel algorithm
  - distributed algorithm
  - hybrid multi-threaded distributed algorithm

• Conclusions and future work
 EOOPS

Socket middle-ware overview

GB()
Reducer Server
DHT Client
DHT Server

Master node

Reducer Client
clientPart()
DHT Client

DistributedThreadPool
ExecutableServer, ExecutableChannel, EC

InfiniBand

a client node
EC execution middle-ware (1)

- on compute nodes do basic bootstrapping
  - daemon class `ExecutableServer`
  - runs thread with `Executor` for each connection
  - receives objects and execute the `run()` method
  - multiple processes as threads in one JVM

- on master start `DistThreadPool`
  - start threads for each compute node
  - starts connections to all nodes with `ExecutableChannel`, gives the name EC
  - can start multiple tasks on nodes: multiple cores
EC execution middle-ware (2)

- client-server programming model
- list of compute nodes taken from PBS
- method `addJob()` on master
- send a job to a remote node and wait until termination
- method `GB()` executed on master
  - schedules `clientPart()` method/class as distributed threads to nodes
  - runs `GBMaster()`
    - starts DHT client
    - initialize communication channels
    - start further threads
MPJ middle-ware overview

- Reducer Server
- GBmaster()
- DHT

- Reducer Client
- clientPart()
- DHT

- 2 MPJ adapter classes

- MPJ middleware

- master node
- a client node
- InfiniBand
MPJ execution middle-ware

- single-program multiple-data (SPMD) programming model
- execution within MPJ runtime environment
- GB() method executed on all nodes
  - rank 0: execute GBmaster()
  - rank > 0: execute clientPart()
- adapters between JAS and MPJ
  - MPJEngine
  - MPJChannel
- ibvdev not thread-safe in FastMPJ V1.0b
JAS to MPJ adapters

- **MPJEngine**
  - `getCommunicator()` delegates to `mpi.MPI.Init()`
  - `terminate()` delegates to `mpi.MPI.Finalize()`
  - `waitRequest()` within a global lock
  - `get*Lock(.)` to obtain global locks

- **MPJChannel**
  - `send()` delegates to `mpi.Comm.Send()`
  - `receive()` delegates to `mpi.Comm.Recv()`
  - also be used for `Isend`, `Irecv` together with `Request.Wait()`
Data structure middle-ware

- sending of polynomials to nodes involves
  - serialization and de-serialization time
  - and communication time
- minimize communication by replicating list on each node in a distributed data structure
- avoid explicit sending in GB to simplify protocol
- distributed list implemented as distributed hash table (DHT)
- key is list index
- implemented with generic types
DHT overview

- class DistHashTable extends java.util.AbstractMap
  - same for EC and MPJ versions
- 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 is received back
- no guaranty that value is received on all nodes
DHT-EC implementation

- client part on node use shared memory TreeMap
- implemented as central control DHT
  - `put()` sends key-value pair to a master
  - master broadcasts key-value pair to all nodes
  - `get()` method takes value from local TreeMap
  - clients to master use marshaled objects
  - no de-serialization in master
  - increases the CPU load on the master
  - doubles memory requirements on master
DHT-MPJ implementation

- class DistHashTableMPJ
- no central control, using MPI broadcast infrastructure
  - put() uses mpi.Comm.Send() to broadcast
  - separate threads use mpi.Comm.Recv() to retrieve message and store key-value pair
  - get() takes value from internal TreeMap
- MPJ must be thread-safe or a global lock must be maintained
Middle-ware comparison (1)

- MPJ simpler to use in PBS environment
  - set of well organized scripts from MPI run-time
- EC more flexible in dynamic task management
  - use of Threads and java.util.concurrent
- TCP/IP Sockets versus mpi.Comm
  - point-to-point with EC, explicit Channel management required, using object streams
  - n-to-n with MPI, all communication connections available via send/recv to MPI rank
Middle-ware comparison (2)

- distributed HT data structure in EC and MPJ
- DHT semantics are different
  - DHT-EC maintains consistent key-value mappings after settling
  - DHT-MPJ can have inconsistent key-value mappings depending on timings
    - can be handled in distributed GB by master
- DHT uses threads and shared memory HT
  - problem with thread safety in MPJ with ibvdev
Overview

• Introduction to JAS
• Communication middle-ware: sockets and MPJ
  - execution middle-ware
  - data structure middle-ware
  - comparison
• Gröbner bases: sockets and MPJ
  - sequential and parallel algorithm
  - distributed algorithm
  - hybrid multi-threaded distributed algorithm
• Conclusions and future work
Gröbner bases

- canonical bases in polynomial rings $R = \mathbb{C}[x_1, \ldots, x_n]$
  - like Gauss elimination in linear algebra
  - like Euclidean algorithm for univariate polynomial greatest common divisors
- 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
Buchberger algorithm

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

G = F; // needed on all compute nodes
B = { (f,g) | f, g in G, f != g }; 
while ( B != {} ) { 
    select and remove (f,g) from B;
    s = S-polynomial(f,g);
    h = normalform(G,s); // expensive operation
    if ( h != 0 ) {
        for ( f in G ) { add (f,h) to B }
        add h to G;
    }
} // termination ? Size of B changes
return G
Problems with the 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
Sequential and parallel GB

- critical pair list B implemented as thread-safe working queues
- implementations for different selection strategies
  - OrderedPairlist, optimized Buchberger
  - CriticalPairlist, stay similar to sequential
  - OrderedSyzPairlist, Gebauer-Möller version
- selection and removal with `getNext()`
- addition with `put()`
- polynomial list is in shared memory on master
Distributed GB

• master maintains critical pair list and communicates with the distributed workers

• simple version with one JVM process per node
  - can also have multiple JVM processes on a node

• hybrid version with multiple threads per node
  - one channel from master to nodes
  - one DHT per node shared by all threads

• top level GB algorithms same for sockets EC and MPJ
  - only use different middle-wares
Thread to node mapping (EC)
Thread to node mapping (MPJ)
GB comparison

• middle-ware design allows the easy replacement of underlying communication system
• get maximal overlap between communication and computation with DHT data structure
• MPJ less flexible than EC but more easy to use
• FastMPJ uses java.nio and own low-level code
  - niodev is thread-safe, works well with IP over IB
  - ibvdev is not thread safe at the moment
• EC uses Socket from java.io, java.net
  - use IP over IB, plain Ethernet too slow
Performance

- all tests on same hardware, network IP over IB
- same Java version 1.6, different JVM releases
- same example “Katsura 8 modulo $2^{127}-1$”
- improvements over the last two years in JVMs and JAS
  - sequential GB: 20%
  - parallel GB: 40 – 60%
  - distributed hybrid GB: 50%
- EC vs MPJ depends on threads per node
- GB speed-up achieved, EC: 8.9, MPJ: 12.8
GBs of Katsuras example on a grid cluster

seconds

kat8hyb_m1 computing time
kat8hyb_m1 ideal

number of nodes

threads per node

Wed Mar 31 10:25:18 2010
time same EC GB run in 2012

Groebner bases on a grid cluster

seconds

kat8hyb_m1 computing time
kat8hyb_m1 ideal

number of nodes
threads per node

Sun Oct 07 20:44:41 2012
time MPJ GB run in 2012

Groebner bases on a grid cluster

kat8mpjhyb_m1 computing time
kat8mpjhyb_m1 ideal

seconds

number of nodes

threads per node
EOOPS

time EC GB run: different ppn

Groebner bases on a grid cluster, distributed hybrid version

ppn = process / threads per node

So Okt 14 16:36:55 2012
time MPJ GB run: different ppn

ppn = process / threads per node
speed-up MPJ GB: nodes

Groebner bases on a grid cluster, distributed hybrid version

- parallel, n = 1
- distributed, n = 2
- distributed, n = 3
- distributed, n = 4
- distributed, n = 5
- distributed, n = 6
- distributed, n = 7

number of threads
Conclusions (1)

- distributed hybrid GB algorithm
  - communication based on EC sockets or MPJ
  - FastMPJ has support for direkt InfiniBand
- improvements within 2 years of 40-60%
  - JVM more optimized, JAS better optimized
- achieved speed-up with IP over IB on 8 nodes
  - 12.8 for FastMPJ and 5-7 threads per node
  - 8.9 for sockets EC and 4-6 threads per node
- EC for small number of threads per node faster
- FastMPJ is 50% faster for 5-7 threads per node
Conclusions (2)

- both run on a HPC cluster in PBS environment
- reduced communication overhead between nodes, main objects in shared memory
- less memory required on nodes compared to pure distributed version
- both packages are type-safe with generic types
- developed classes fit in Gröbner base class hierarchy
Future work

- fix or work around thread safety issues in FastMPJ
- investigate InfiniBand ibvdev device performance
- profile and study run-time behaviour in detail
- investigate further optimizations of the GB algorithms: F4, F5, GGV, ARRI, ...
Thank you for your attention

Questions?
Comments?

http://krum.rz.uni-mannheim.de/jas/

Acknowledgements

thanks to: Thomas Becker, Raphael Jolly, Werner K. Seiler, Axel Kramer, Dongming Wang, Thomas Sturm, Hans-Günther Kruse, Markus Aleksy

thanks to the referees
more slides
bwGRiD cluster architecture

- 8-core CPU nodes @ 2.83 GHz, 16GB, 140 nodes
- shared Lustre home directories
- 20Gbit InfiniBand and 1Gbit Ethernet interconnect
- managed by PBS batch system, Moab scheduler
- running Java 64bit server VM 1.6 with 4+GB mem
- start Java VMs with daemons on allocated nodes
- communication via TCP/IP over InfiniBand
- other middle-ware ProActive or GridGain not studied
JAS Implementation overview

- 340+ classes and interfaces
- plus ~150 JUnit test classes, 5000+ assertions
- uses JDK 1.6 with generic types
  - Javadoc API documentation
  - logging with Apache Log4j
  - build tool is Apache Ant
  - revision control with Subversion
  - public git repository
- jython (Java Python), jruby (Java Ruby) scripts
  - support for Sage compatible polynomial expressions
- Android version based on Ruboto using jruby
Example: Legendre polynomials

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

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);
}