SymGrid-Par: A Standard Skeleton-Based Framework for Computational Algebra Systems

Phil Trinder The SCIEnce Team The HPC-GAP Team

Dependable Systems Group
School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, UK

25/09/2010

Computational Algebra

- Computational Algebra (CA) systems manipulate symbolic structures like groups, rings, or sets of polynomials, to provide exact mathematical solutions.
- Examples: Maple, Mathematica, GAP, Kant, ...
- Unusual for parallel programming

Computational Algebra

- Computational Algebra (CA) systems manipulate symbolic structures like groups, rings, or sets of polynomials, to provide exact mathematical solutions.
- Examples: Maple, Mathematica, GAP, Kant, ...
- Unusual for parallel programming
  - Neither matrices, nor floating points
  - Relatively complex data structures, e.g. tuples, lists & sets
  - highly dynamic parallelism: tasks created at runtime
  - highly irregular parallelism: task sizes vary by 5 orders of magnitude

CA Community

- Seeking to standardise between multiple CA systems
- Seeking to parallelise large computations
- Seeking to provide distributed access to CA systems
- Willing to adopt high-level approaches like skeletons

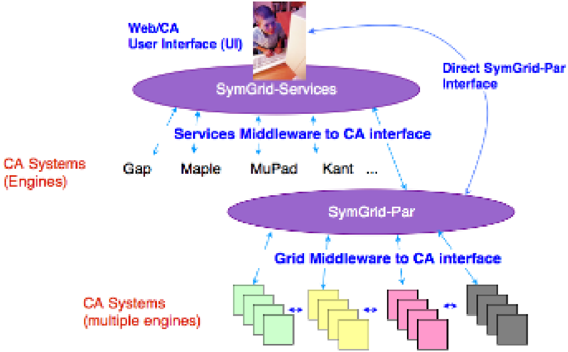
SCIEnce Project

- Symbolic Computation Infrastructure for Europe (SCIEnce) is an EU FP6 project (£4.6M) 2007-2012 to addresse the challenges.
- Partners: St Andrews, T.U. Berlin, CNRS, T.U. Eindhoven, Heriot-Watt, Linz, Paderborn, Timosoara Universities, Maplesoft

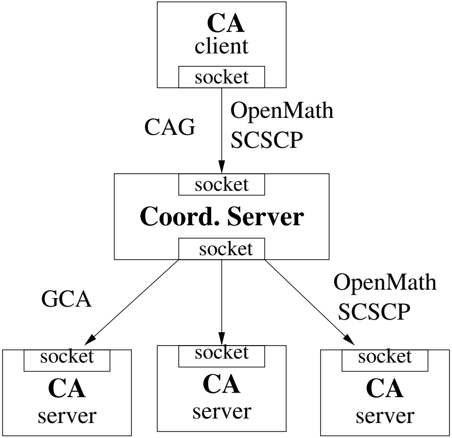
SCIEnce Project

- Symbolic Computation Infrastructure for Europe (SCIEnce) is an EU FP6 project (£4.6M) 2007-2012 to addresse the challenges.
- Partners: St Andrews, T.U. Berlin, CNRS, T.U. Eindhoven, Heriot-Watt, Linz, Paderborn, Timosoara Universities, Maplesoft
- Aim to provide
  - transparent access to complex, mathematical software, through Grid/Web Services
  - interoperation between independent CA systems through OpenMath data format and the SCSCP protocol
  - exploitation of modern parallel hardware with high-level orchestration of parallelism

SymGrid-Services and SymGrid-Par



SymGrid-Par Infrastructure



Components of the Architecture

- Either command-line client or a computer algebra system.
- High-level Coordination Server handling parallelism
  - makes essential use of parallel Haskell dynamic parallelism management to deal with irregularity
  - parallelism is mainly specified and coordinated here
  - automatic resource management on this level (load balancing etc)
- Any SCSCP-based (Symbolic Computation Software Composability Protocol) computer algebra server:
  - tested with GAP and a Haskell-side server
  - servers can themselves use parallelism, but
  - no direct communication between servers

Coordination Server

A Parallel Haskell component that

- implements a collection of (parallel) CA functions
- provides algorithmic skeletons for CA computation
- calls CAs to perform the heavy computation.

Classes of SymGrid-Par Skeletons

- Problem-Oriented
- Process-Oriented
- Computational Algebra specific

Problem-Oriented Skeletons

Standard, problem oriented higher-order functions:

```
parMap       :: (a->b) -> [a] -> [b]

parFold      :: (a->b->b) -> b -> [a] -> b

parFold1     :: (a->b->b) -> [a] ->  b

parMapFold   :: (a->b) -> (b->c->c) -> c -> [a] -> c

parMapFold1 :: (a->b) -> (b->c->c) -> [a] -> c

parZipWith   :: (a->b->c) -> [a] -> [b] -> [c]
```

Problem-Oriented Examples I

```
  zipWith (+) [1..10] [100,200..1000]

=>

  [101,202,303,404,505,606,707,808,909,1010]
```

Problem-Oriented Examples I

```
mapFold f g n = fold g n . map f
```

Problem-Oriented Examples II

- parMap in GAP:

```
gap> Read("parallel.g");
gap> GAPparMap("Fibonacci",[100..105]);
==== Starting parallel execution on 6 processors ...
====== Setting up Gap around Haskell ....6
====== Setting down Gap around Haskell....
["354224848179261915075","573147844013817084101","
927372692193078999176","1500520536206896083277","
2427893228399975082453","3928413764606871165730"]‘
```

Problem-Oriented Examples II

- parMap in Maple:

```
gap> Read("parallel.g");
gap> mapleParMap("fibonacci",[100..103]);
==== Starting parallel execution on 4 processors ...
====== Setting up Maple around Haskell....4
====== Setting down Maple around Haskell....4
["354224848179261915075","573147844013817084101","
927372692193078999176","1500520536206896083277"]
```

Problem-Oriented Examples III

- A Chinese Remainder Algorithm operating on a list of values and a list of modules

```
cra xs ms = foldl (modSum m') 0 (parZipWith bin_cra xs ms)
  where bin_cra x m = x * b * m0
                    where b =  modInv m m0
                          m0 = m' `div` m
        m' = product ms
```

Process-Oriented Skeletons

- Create common process structures: farm, workpool, Google mapReduce
- A farm statically partitions the work.

```
farm :: Int ->               -- number of workers
        (a -> b) ->          -- worker process
        [a] ->               -- input data
        [b]                  -- output data
```

Process-Oriented Skeletons II: workpool

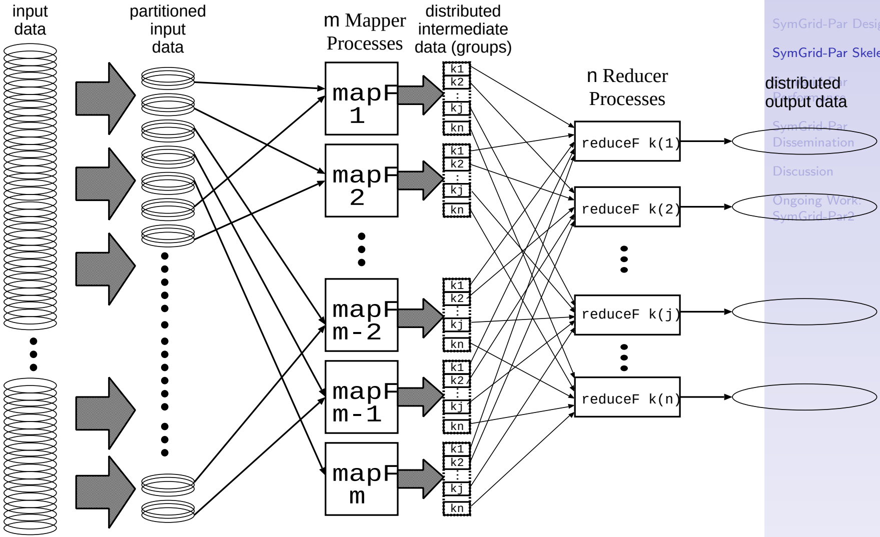
- A workpool dynamically partitions the work.

```
workpool::  Int -> -- number of workers
            Int -> -- prefetch
            (a -> a) -> -- worker
            [a] ->      -- input data
            [a]         -- output data
```

Process-Oriented Skeletons III: `mapReduce`

- A variant of the `parMapFold`
- Core of Google/Hadoop Web engines
- SymGrid-Par provides scalability, but not reliability

Process-Oriented Skeletons III: mapReduce



Process-Oriented Skeletons III: mapReduce

```
parMapReduce :: Ord k2 =>
                Int ->                      -- No. of partitions
                (k2 -> Int) ->              -- key partitioning
                (k1 -> v1 -> [(k2, v2)] ->  -- mapF function
                (k2 -> [v2] -> Maybe v3 ->  -- local reduceF function
                (k2 -> [v3] -> Maybe v4 ->  -- global reduceF function
                [Map k1 v1] ->              -- distributed input data
                [Map k2 v4]                 -- distributed output data
```

Computational Algebra Skeletons

- Capture CA-specific patterns of parallelism, currently:
  - Transitive Closure
  - Orbit
  - Critical-Pair-Completion
  - Multiple Homomorphic Images

Computational Algebra Skeletons

- Capture CA-specific patterns of parallelism, currently:
  - Transitive Closure
  - Orbit
  - Critical-Pair-Completion
  - Multiple Homomorphic Images
- Support irregular task sizes
- Combine data & task parallelism
- May have shared or distributed memory implementations

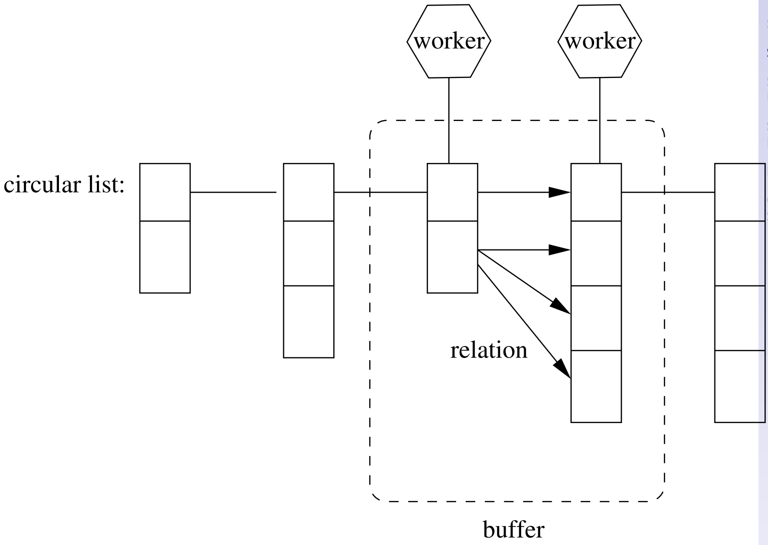
Transitive Closure

- Calculate the transitive closure of a relation from a set of initial elements.

```
transcl :: (Eq a) => (a -> [a]) -> -- relation
                     [a] ->        -- seed elements
                     [a]           -- all reachable elements
```

- Shared-memory implementation

Transitive Closure: with Parallel Buffer



Orbit

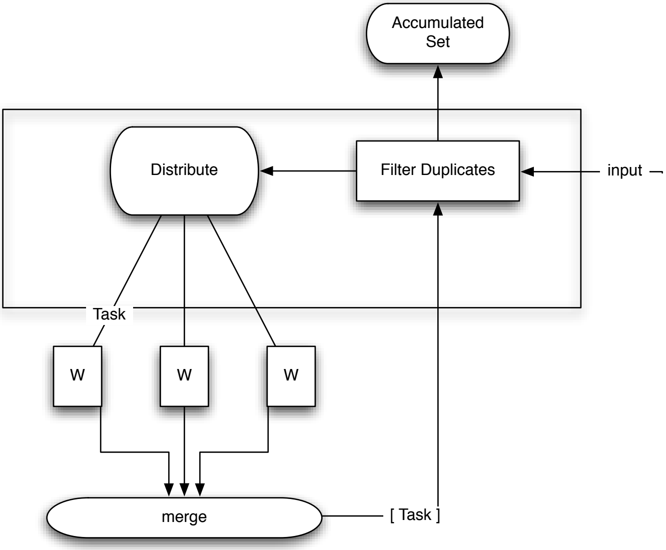
- Recent CA pattern (2001)
- Similar to transitive closure, except that new elements are generated from existing elements by a set of generator functions.

Orbit Skeleton

```
orbit :: (Eq a, Ord a) =>
         [ a -> a ] -> -- generator functions
         [ a ] ->      -- initial set
         [ a ]         -- result set
```

Orbit Implementation

- Currently only a shared-memory implementation [TFP'10]
- Design of a distributed-hash table implementation in progress


Accumulated Set
Distribute
Filter Duplicates
input
Task
W
W
W
merge
[Task]

Critical Pair Completion

- Starts with an arbitrary specification of an algebraic structure, and transforms it to an algebraic structure with nicer properties, typically common decision problems are far simpler.
- Formally: Given a set of objects T and a reduction relation defined as a set of rules $G \subseteq T \times T$, decide whether two elements $s, t \in T$ are in the reflexive, symmetric, transitive closure of G.

Critical Pair Completion Implementation


```
parCritPairCompletion ::
  (NFData a) =>
  ([a] -> a -> a -> a) ->    -- f: operation on pair
  ([a] -> [a] -> [a]) ->     -- f': normalization of the pool
  ([a] -> a -> ([a], [(a,a)])) -> -- g: construct new pool and pairs
  ([a] -> a -> Bool) ->      -- predicate, guarding addition to pool
   [a] ->                    -- pool
   [a] ->                    -- worklist
   [a]                       -- result
```

- Application: Implementing Buchberger's Gröbner Basis algorithm

Multiple Homomorphic Images

- Often a problem can be solved faster in a simpler domain because the data structures are smaller
- The multiple homomorphic images approach:
  1. map the input data into several homomorphic images (`h`)
  2. compute the solution in each image (`f`)
  3. combine the results of all images to a result in the original domain (a fold of `g`).

Multiple Homomorphic Images Skeleton

```
multHomImg ::
 (p -> b -> b') ->  -- map input to homomorphic images
 (p -> b' -> c') -> -- solve the problem in the hom. imgs.
 ((p, c) -> (p, c') -> (p, c)) ->  -- combine the results
 (p,c) ->           -- neutral elem in the overall domain
 Strategy [c'] ->   -- strategy to generate parallelism
 b ->               -- input
 (p,c)              -- result
```

- Application: exact linear system solver

SymGrid-Par Performance

- Most performance results reported here use the prototype Coordination Server
- Hardware:
  - 28 node Beowulf cluster, 3GHz Intel Pentium 4, 512MB RAM
  - 8 core Dell PowerEdge 2950, 2.7GHz Intel Xeon, 16GB shared RAM

sumEuler Example: Haskell

- Sum the Euler totient function over arange of integers.
- Moderately irregular as the totient function takes longer to calculate for larger integers

```
sumTotient :: Int -> Int -> Int -> Int
sumTotient lower upper c =
sum (parMap (euler) (splitAtN c [lower..upper]))

euler :: Int -> Int
euler n = length (filter (relprime n) [1..n-1])

relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1
```

sumEuler Example: GCA


```
sumTotient_GAP :: Int -> Int -> Int -> Int
sumTotient_GAP lower upper c =
 sum(parMap (euler_GAP) (splitAtN c [lower..upper]))

euler_GAP :: Int -> Int
euler_GAP n = gapObject2Int(gapEval ‘‘euler’’ [int2GAPObject n])
```

Direct/Recursive **sumEuler** Variants

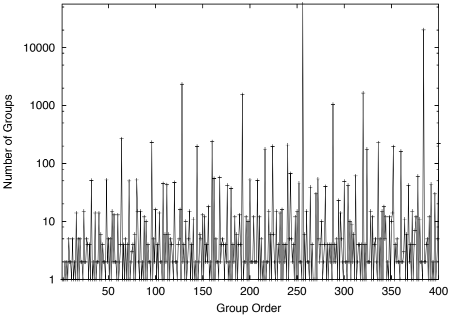
- **sumEuler** would normally be written directly using iteration in GAP
- A recursive GAP version is more directly comparable with the Haskell and GCA versions.

	GpH/GUM	GAP direct	GAP recursive
Runtime	164s	500s	2298s

Table: Sequential **sumEuler** 32000

More Typical Example: **smallGroup**

- Searches for *groups* of a given order, with some property: e.g. average order of their elements is an integer
- Two levels of irregularity
    - the number of groups of a given order varies by 5 orders of magnitude!
    - the time to compute the conjugacy classes of each group.



GCA smallGroup

```
smGrpSearch :: Int -> Int [(Int,Int)]
smGrpSearch lo hi = concat(map(ifmatch)(predSmGrp [lo..hi]))

predSmGrp :: ((Int,Int) ->(Int,Int,Bool)) -> Int ->[(Int,Int)]
predSmGrp (i,n) = (i,n,(gapObject2String (gapEval "IntAvgOrder"
    [int2GapObject n, int2GapObject i])) == ‘‘true’’)

ifmatch :: ((Int,Int) -> (Int,Int,Bool)) -> Int -> [(Int, Int)]
ifmatch predSmGrp n= [(i,n) | (i,n,b) <-(masterSlaves predSmGrp
     [(i,n) | i<- [1 nrSmGrps n]]),b]

nrSmGrps :: Int -> Int
nrSmGrps n=gapObject2Int(gapEval "NrSmallGroups" [int2GapObject n])
```

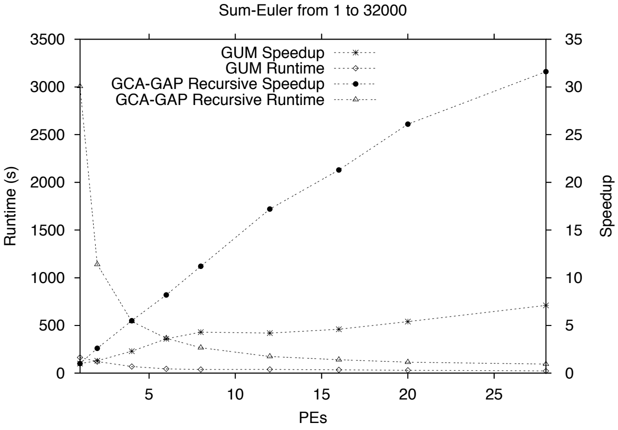
Comparison with bespoke parallel CA systems

- SymGrid-Par works with multiple CA systems: Maple, Kant, GAP, MuPAD etc
- How does it compare with bespoke parallel CA systems?

sumEuler [1..32000]

PE	SymGrid-Par/GCA				ParGAP				GUM	Spd
	Recur	Spd	Dir	Spd	Recur	Spd	Dir	Spd		
1	3006s	1	500s	1	3288	1	686s	1	164s	1.0
2	1139s	2.6	320s	1.5	2117	1.5	481	1.4	121s	1.3
4	542s	5.5	154s	3.2	1175	2.8	233	2.9	69s	2.3
6	365s	8.2	107s	4.6	690	4.7	137	4.8	45s	3.6
8	267s	11.2	84s	5.9	490	6.7	104	6.6	38s	4.3
12	174s	17.2	59s	8.4	310	10.6	62	11.0	39s	4.2
16	141s	21.3	51s	9.8	223	14.7	44	15.5	35s	4.6
20	115s	26.1	45s	11.1	166	19.8	34	20.1	30s	5.4
28	95s	31.6	40s	12.5	115s	28.5	25s	27.4	23s	7.1

Recursive **sumEuler** Speedups



smallGroup [256]

PE	GCA	Spdup	Spdup (GAP)	ParGAP	Spdup	ParGAP/ GCA
1	829s	1	1.1	1312s	1	1.5
2	416s	1.9	2.1	1421s	0.92	3.4
4	206s	4.0	4.4	549s	2.39	2.6
8	104s	7.9	8.7	185s	7.09	1.7
12	70s	11.8	13.0	105s	12.5	1.5
16	53s	15.6	17.2	79s	16.6	1.5
20	42s	19.7	21.7	64s	20.5	1.5
24	36s	23.0	25.3	53s	24.7	1.5
28	31s	26.7	29.4	45s	29.1	1.4

- Performance of generic SymGrid-Par is comparable with bespoke parallel CA systems

smallGroup [1..400]

PE	GCA	Spd	Spd GAP
1	1,377s	1	1.2
2	698s	1.9	2.3
4	360s	3.8	4.6
6	243s	5.6	6.8
8	186s	7.4	8.9
12	132s	10.4	12.5
16	105s	13.1	15.7
20	89s	15.4	18.6
28	73s	18.8	22.7

- The skeletons deliver speedups even for problems with high levels, and multiple levels, of irregularity.

Generating New Knowledge: Summatory Liouville

- Search for positive values of the Summatory Liouville function
- The *Liouville* function $\lambda(n) = (-1)^{r(n)}$, where $r(n)$ is the number of prime factors of n.
- The *summatory Liouville's function*, $L(x)$, is the sum of values of $Liouville(n)$ for all n from $[1..x]$.

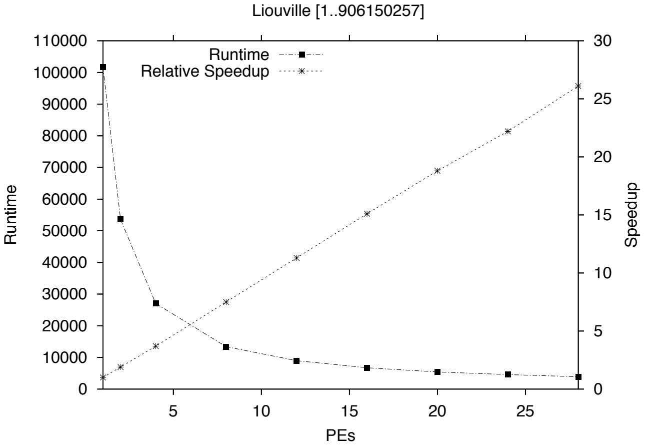
Summatory Liouville

```
L :: Integer -> Integer -> Int -> [(Integer,Integer)]
L lower upper c = sumL (myMakeList c lower upper)

sumL :: [(Integer,Integer)] -> [(Integer,Integer)]
sumL mylist = mySum ((masterSlaves liouville) mylist)

liouville :: (Integer,Integer) ->
             [((Integer, Integer),(Integer,Integer))]
liouville  (lower,upper) =
    let
    l  = map gapObject2Integer (gapEvalN "gapLiouville"
     [integer2GapObject lower,integer2GapObject upper])
    in
    ((head l, last l), (lower,upper))
```

Summatory Liouville [1 ..906150257] Speedups



- Result almost immediately superceded by [ISSAC09]

Performance Portability

- SymGrid-Par designed for distributed memory architectures
- Many CA problems have large thread granularity and hence perform well on multicores [DAMP09]

8-core Summatory Liouville $[1 \ldots 25 \times 10^6)]$

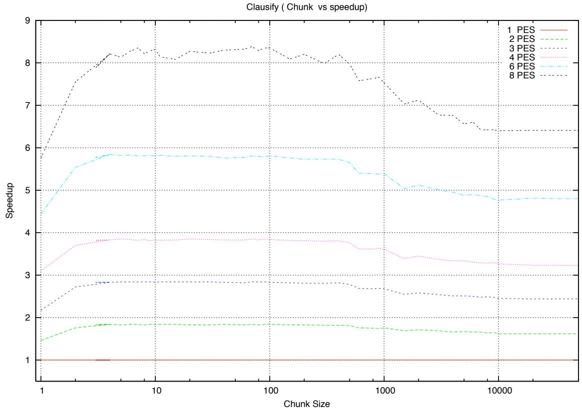
No PEs	Rtime	Spdup	CPU Utilis.
1	526s	1	92.8%
2	264s	1.9	89.6%
3	178s	2.9	93.1%
4	132s	3.9	92.0%
5	106s	4.9	90.7%
6	89s	5.9	90.7%
7	76s	6.9	89.4%
8	68s	7.7	88.9%

8-core **smallGroup** [1. . .350]

No PEs	Rtime	Spdup	CPU Utilis.
1	480s	1	96.0%
2	246s	1.9	96.0%
3	165s	2.9	98.6%
4	125s	3.8	98.0%
5	104s	4.6	99.2%
6	91s	5.2	98,7%
7	82s	5.8	98.3%
8	76s	6.3	97.0%

Identifying Required Task Granularity

- Q: How coarse grained must the threads be to deliver a speedup?
- A: For some reasonable assumptions: approx. 70 Mcycles (approx. 30 ms on PowerEdge)
- Considered control & data parallel programs [DAMP09]



SymGrid-Par Dissemination

- Documentation
- Downloads
- Demos at CA Conferences, e.g. ISSAC'10
- Courses, e.g. Int. Summer School Symbolic Computation Linz 2008/09/10

Discussion

- SymGrid-Par aims to provide standardised parallelism for Computational Algebra systems
- Skeletons are a key part of the SymGrid-Par architecture
- There are problem-oriented, process-oriented, and domain-specific skeletons
- The skeletons are currently available from Maple, GAP, Kant & MuPAD

SymGrid-Par Performance Summary

- Performance of generic SymGrid-Par is comparable with bespoke parallel CA systems
- The SymGrid-Par skeletons deliver speedups even for problems high levels, and multiple levels, of irregularity
- We have generated new CA results using the SymGrid-Par skeletons
- The SymGrid-Par skeletons provide performance portability across clusters and multicores

Ongoing Work

- Developing and disseminating SymGrid-Par
  - Implementing CA skeletons
  - Designing new CA skeletons
  - New applications of existing skeletons
  - Continuing dissemination

Ongoing Work

- Developing and disseminating SymGrid-Par
  - Implementing CA skeletons
  - Designing new CA skeletons
  - New applications of existing skeletons
  - Continuing dissemination
- New project: HPC-GAP
  - Aims, *inter alia* to scale GAP to large architectures
  - £1.8M, 2009 - 13
  - UK EPSRC funded
  - Aberdeen, St Andrews, Edinburgh, Heriot-Watt Universities

SymGrid-Par2 Design Goals

- Scale: designing for 10^6 cores
  - Core failure every 5 minutes ⇒ reliability
  - Topology awareness
- Rapidly evolving architectures ⇒ performance portability is crucial
- Layered design: explicit parallelism at top layer