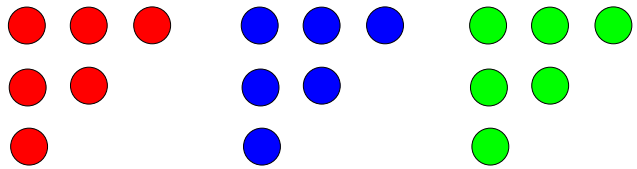


Multiprocessed Parallelism Support in Aldor on SMPs and Multicores

Marc Moreno Maza, Ben Stephenson,

Stephen M. Watt and Yuzhen Xie

ORCCA, UWO, Canada

for Aldor Workshop

August 16, 2008

Motivation

- widely available SMPs and multicores.
- initially, to support the implementation of a component-level parallel solver for triangular decompositions
  - reuse the `BasicMath` library in Aldor for multivariate polynomial arithmetic.
  - reuse the routines of the sequential solver `triade` for parallel execution.
- later, motivated by a reviewer, generalized to a high-level categorical parallel framework to support high-performance computer algebra.

Introduction to Aldor

- many computer algebra systems: Maple, Matlab, MAGMA, NTL, AXIOM, Aldor, ...
- many contributions to parallel computer algebra during 1980s and 1990s: PASAC-2, PACLIB, Piit, ...
- Aldor is an extension of AXIOM:
  - categorical programming
  - a two-level object model of categories and domains
  - allows the implementation of algebraic structures (e.g. rings) and their members (e.g. polynomial domains)
  - ESPRIT Project FRISCO funded by the European Union (1996-1999): `BasicMath` library and `triade` solver
  - interoperable with other languages like C for high-performance computing
  - compiled to stand alone executables!

Outline of this presentation

- A high-level categorical parallel framework in Aldor to support high-performance computer algebra on SMPs and multicores:
  - dynamic process management and user-level scheduling
  - data communication and synchronization
  - packages for serializing and de-serializing high-level Aldor objects
- Benchmarks on performance evaluation

Outline of this presentation

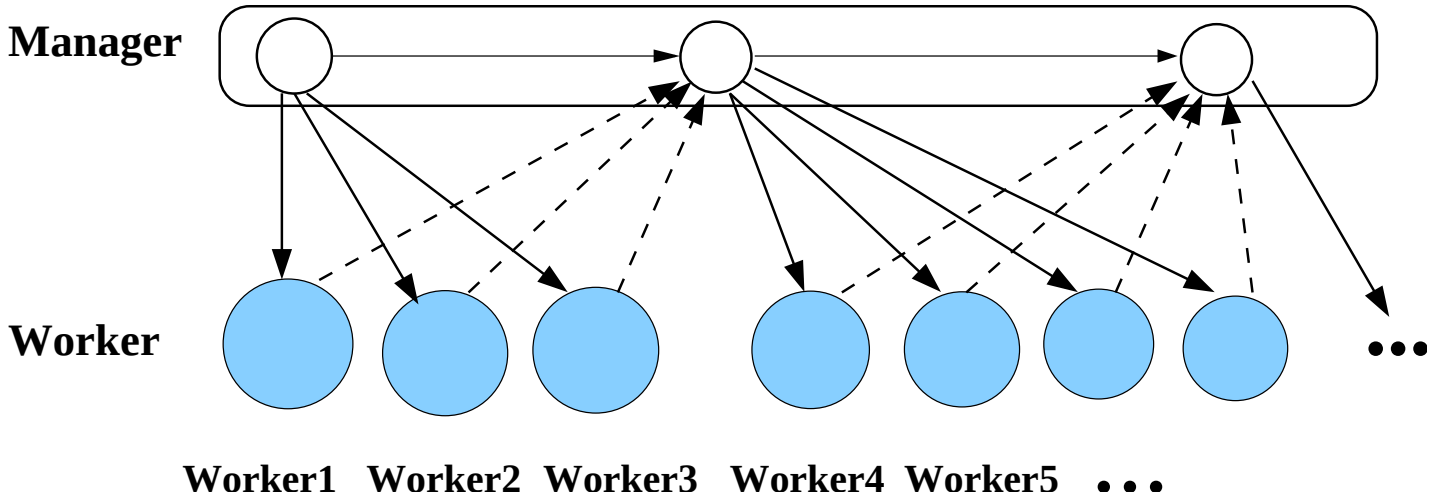
- A high-level categorical parallel framework in Aldor to support high-performance computer algebra on SMPs and multicores:
  - dynamic process management and user-level scheduling
  - data communication and synchronization
  - packages for serializing and de-serializing high-level Aldor objects
- Benchmarks on performance evaluation

Dynamic process management

- `Spawn(command, argument)`:
  Aldor's `run()`, `system()` in C on UNIX
- can be used within a process (say running program A) to launch one or more additional processes that will run other programs independently.
- User defined **Task** with **virtue process identifier (VPID)**, analogous to a processor's rank in MPI
- This VPID is used to allow the process to communicate with other spawned processes. This VPID is also used to create the unique keys to the two shared memory segments (tag and data, see later)

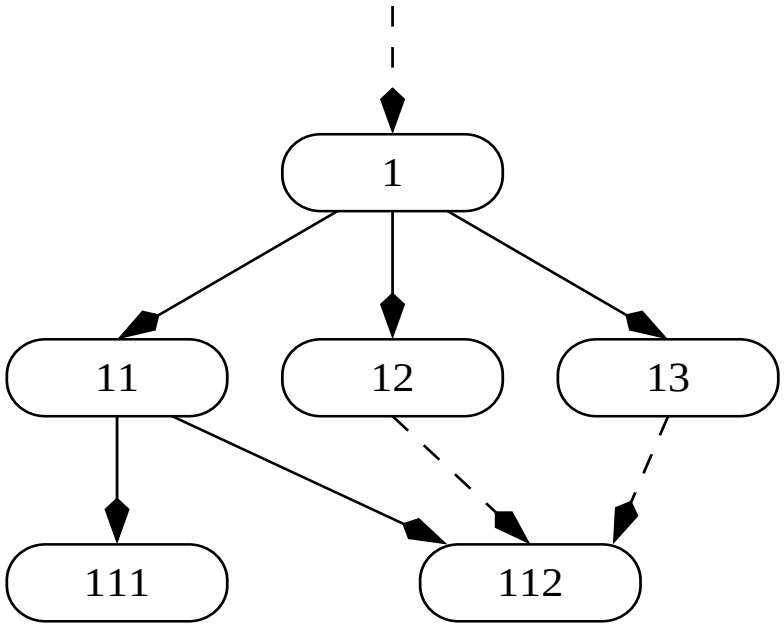
Dynamic process management and user-level scheduling

- Task farming scheme: easy to apply greedy scheduling



Dynamic process management and user-level scheduling

- Dynamic fully-strict task processing



- This solution is akin to the scheme for handling the rank of spawned processes in MPICH2

Dynamic process management: an example

- `LazySolve()` process in the component-level parallel solver
- VPID of Manager = 0
- Manager:

  `Send(0,1,data-1);`

  `Spawn(‘‘/local/bin/LazySolve’’, 1);`

  `Send(0,2,data-2);`

  `Spawn(‘‘/local/bin/LazySolve’’, 2);`

  ...

  result-i = `Receive(i,j);`

  ...
- A worker for `LazySolve()`:

  if $myVPID = j$ then `Receive(0,j);`

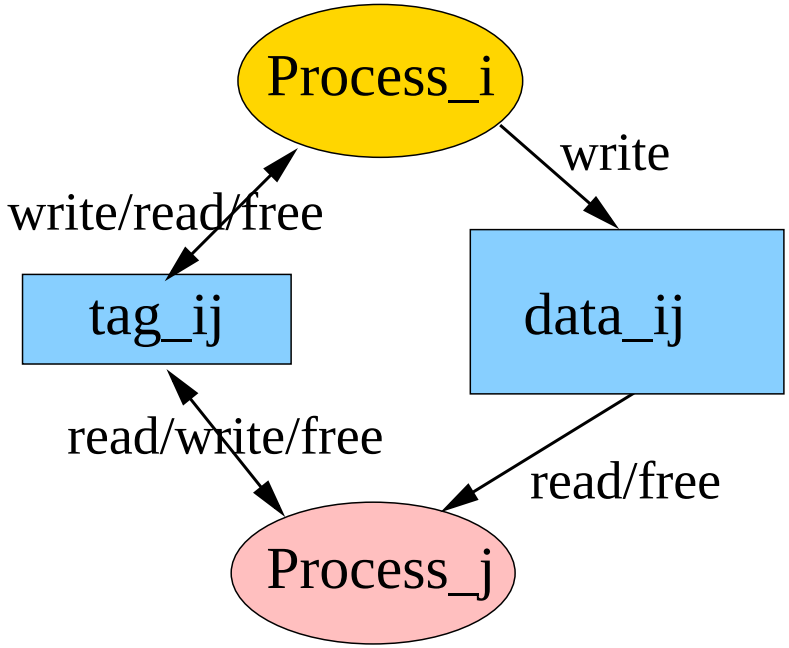
  do ...;

  `Send(j,0,result-j);`

Data communication and synchronization (1/4)

Synchronization protocol:

- Sending:
  if $tag\_ij = 0$
  then write data into $data\_ij$;
  $tag\_ij \leftarrow \ sizeOfdata$;
- Receiving:
  if $tag\_ij > 0$ then
  $sizeOfdata \leftarrow \ tag\_ij$;
  $tag\_ij \leftarrow \ -1$;
  Read data from tag_ij;
  $tag\_ij \leftarrow \ 0$;



Data communication and synchronization (2/4)

- standard UNIX System V shared memory segments:

```
key_t ftok(const char *pathname, int proj_id);
int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- Aldor domain: `SharedMemorySegment`

Data communication and synchronization (3/4)

Aldor domain: `InterProcessSharedMemoryPackage`:
Send(i,j,data), *Receive(i,j)*

- **Process_i** (Sending)
  1. Create files "*/tmp/data_ij*" and "*/tmp/tag_ij*"
  2. Generate the data segment and tag IPC keys, *data_ij* and *tag_ij* from the integer i and the files "*/tmp/data_ij*" and "*/tmp/tag_ij*" respectively
  3. Create or connect to the tag segment, setting the permission to allow both reads and writes
  4. Repeat until the value of the tag segment is 0
  5. Create the data segment with sufficient size to hold the values being sent to `Process_j`
  6. Write the data to the data shared memory segment
  7. Detach from the data segment
  8. Write the size of the data to the tag segment

Data communication and synchronization (4/4)

Aldor domain: `InterProcessSharedMemoryPackage`:
Send(i,j,data), *Receive(i,j)*

- **Process_j** (Receiving)
  1. Create files "*/tmp/data_ij*" and "*/tmp/tag_ij*"
  2. Generate the data segment and tag IPC keys, *data_ij* and *tag_ij* from the integer i and the files "*/tmp/data_ij*" and "*/tmp/tag_ij*" respectively
  3. Create or connect to the tag segment, setting the permission to allow both reads and writes
  4. Repeat until the value of the tag segment, t, is greater than 0
  5. Write -1 to *tag_ij*
  6. Detach from the tag segment
  7. Connect to the data segment using key *data_ij*
  8. Read t integers from the data segment
  9. Detach from the data segment
  10. Delete the data segment; 11. Write 0 to the tag segment

Data communication: an example

- `LazySolve()` process in the component-level parallel solver
- VPID of Manager = 0
- Manager:

  `Send(0,1,data-1);`

  `Spawn(‘‘/local/bin/LazySolve’’, 1);`

  `Send(0,2,data-2);`

  `Spawn(‘‘/local/bin/LazySolve’’, 2);`

  ...

  result-i = `Receive(i,j);`

  ...
- A worker for `LazySolve()`:

  if $myVPID = i$ then `Receive(0,j);`

  do ...;

  `Send(j,0,result-j);`

Serialization of high-level objects

`BasicMath` library: $g = 5x^2y^3 - 8x^2 - 7y^2 + 4$

- `SparseMultivariatePolynomial` (SMPLY): suitable for Tri. Domp. $(4 - 7y^2) + (-8 + 5y^3)x^2$ for $x > y$;

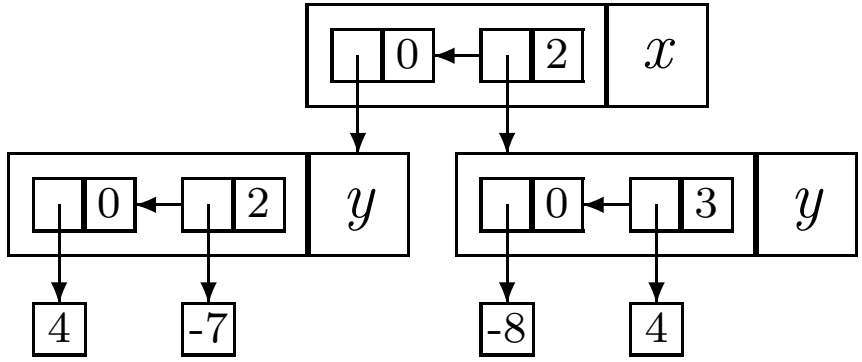


Figure 1: SMPOLY representation of g

- `DistributedMultivariatePolynomial` (DMPOLY): suitable for GB

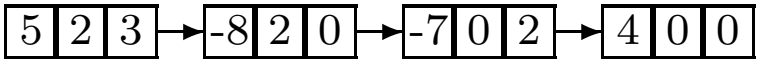


Figure 2: DMPOLY representation of g

- Both SMPOLY and DMPOLY are for efficient representation and manipulation of sparse multivariate polynomials

Serialization of high-level objects

Aldor package: `Serialization`. $g = 5x^2y^3 - 8x^2 - 7y^2 + 4$

- `SerializeSMPbyKronecker()`:
  $g \to \{5, 0, 0, 0, -7, 0, 0, 0, -8, 0, 4\}$
- `SerializeSMPbyDMP()`:
  $g \to \{5, 2, 3, 8, 2, 0, 7, 0, 2, 4, 0, 0\}$

Benchmarks: overhead of the parallel constructs

- By the component-level parallel solver, we measure the costs of:
  - process spawning
  - uses of tag segments
  - data communication/serialization for
    - write by `SerializeSMPbyKronecker()`;
      read by `UnserializeSMPbyKronecker()`;
    - write by `SerializeSMPbyDMP()`;
      read by `UnserializeSMPbyDMP()`

Features of the Examples

Sys	Name	n	d	p	Sequential (s)
1	eco6	6	3	105761	4.00
2	eco7	7	3	387799	727.95
3	CNogues2	4	6	155317	476.16
4	CNogues	4	8	513899	2162.40
5	Nooburg4	4	3	7703	4.14
6	UBikker	4	3	7841	866.20
7	Cohn2	4	6	188261	305.24

Parallel timings for the two serializing methods

Sys	CPUs	Kron. (s)	DMP (s)	Kron. Speedup	DMP Speedup
1	5	1.94	1.91	2.1	2.1
2	9	119.44	117.41	6.1	6.2
3	9	207.29	215.28	2.3	2.2
4	9	905.25	1002.56	2.4	2.2
5	9	1.79	1.81	2.3	2.3
6	9	455.21	463.24	1.9	1.8
7	9	96.70	102.55	3.2	3.0

Both methods lead to similar running time.

Dissection of workers' overhead for Kronecker

Sys	Workers (#)	Tags (#)	Read (#*int*∗)	Write (#*int*)	Total (#*int*)	*Zeros* (%)
1	9	9	4131	3586	7717	59
2	24	24	29307	27382	56689	72
3	32	32	57106	55696	112802	73
4	42	42	216000	214217	430217	83
5	14	14	13307	0	13307	72
6	49	49	128983	125162	254145	55
7	44	44	39146	38280	77426	39

- The amount of read and written integers by workers are similar.
- The (intermediate) polynomials are quite dense.

Dissection of workers' overhead for DMPOLY

Sys	Workers (#)	Tags (#)	Read (#*int*)	Write (#*int*)	Total (#*int*)	Zeros (%)
1	9	9	5069	4449	9518	55
2	24	24	36893	35184	72077	57
3	32	32	64106	64106	127304	39
4	42	42	168178	167186	335364	39
5	14	14	12681	0	12681	44
6	49	49	271845	267761	539606	42
7	44	44	104486	103534	208020	40

- The amount of read and written integers by workers is 1 to 3 times larger than with Kronecker's method.
- The serialized DMP polynomials have comparable percentage of zeros.

Dissection of workers' time for Kronecker

Sys	Spawns (ms)	Tags (μs)	Read and Unserialize (ms)	Serialize and Write (ms)	Net Work (s)	Over-head (%)
1	358	1067	492	76	3.80	24.4
2	579	3414	1264	184	660.54	0.3
3	773	4887	9682	623	469.48	2.4
4	1695	7221	68737	491	2164.62	3.3
5	452	1940	488	0	3.57	26.4
6	1558	7773	21762	823	871.04	2.8
7	925	6014	2378	369	289.15	1.3

- Overhead is most of the time negligible, or satisfactory
- Unserializing is much more expensive than serializing

Dissection of workers' time for DMPOLY

Sys	Spawns (ms)	Tags (μs)	Read and Unserialize (ms)	Serialize and Write (ms)	Net Work (s)	Over-head (%)
1	314	1211	347	79	3.71	20.0
2	685	3498	1345	623	611.16	0.4
3	1435	4676	1813	683	474.16	0.8
4	1723	7524	80490	2360	2134.71	3.8
5	552	2224	764	0	3.65	36.2
6	1994	7847	52157	5242	886.59	6.7
7	1110	6673	5881	2063	282.16	3.2

- Overhead is a bit larger than with Kronecker's method.
- Unserializing is even more expensive than serializing for DMPOLY (memory allocation)

Analysis of workers' overhead for Kronecker

Sys	Per Spawn (ms)	Per Tag (μs)	Read and Unserialize (μs per int)	Serialize and Write (μs per int)
1	40	118	119	21
2	24	142	43	6
3	24	152	169	11
4	40	172	318	2
5	32	138	36	-
6	32	158	168	6
7	21	136	60	9
AVG	30	145	130	9

- Unserializing is more expensive than serializing: another illustration.
- Why unserializing per int can vary so much?

Analysis of workers' overhead for DMPOLY

Sys	Per Spawn (ms)	Per Tag (μs)	Read and Unserialize (μs per int)	Serialize and Write (μs per int)
1	35	134	68	17
2	29	146	36	18
3	45	146	180	21
4	41	179	478	14
5	39	158	59	-
6	41	160	192	19
7	26	151	56	20
AVG	37	153	152	18

- Unserializing is more expensive than serializing: another illustration.
- Why unserializing per int can vary so much?

Conclusion and future work

- A multiprocessed parallel framework in Aldor on SMPs and multicores
- It has been used for the successful implementation of component-level parallelization of triangular decomposition.
- Effective in practice
- Practically efficient for coarse-grained parallel symbolic computation
- Advantage of Dynamic process management in multiprogramming (shared) environment
- Near future for Aldor threads:
  - properly handle parametric datatype (Aldor pointer?)
  - apply provable efficient automatic scheduling, like Cilk and KAPPI
- Future: support multi-grained parallelism over clusters of multi-processors

Thank you! Comments and suggestions are more than welcome!