

Parallel I/O using standard data formats in climate and NWP

Project ScalES funded by BMBF

Deike Kleberg Luis Kornbluh

Max-Planck-Institut für Meteorologie, Hamburg



Max-Planck-Institut
für Meteorologie

Outline

Introduction

Proposed Solution

Implementation

Results

Conclusions, Input and Outlook



Thanks to

- ▶ Uwe Schulzweida, MPIM
- ▶ Mathias Pütz, IBM
- ▶ Christoph Pospiech, IBM
- ▶ Colleagues at DKRZ



Targets

- ▶ Scalability of Earth System Models
- ▶ Usability of Infrastructure Components
- ▶ Portability of Implemented Solutions
- ▶ Solutions Prepared for Future Computer Development
- ▶ Free Availability for the Modelling Community

Constraints

- ▶ Large amounts of data
- ▶ Comparable small spatial extent ($O(2, 3)$)
- ▶ Large Time extent ($O(8)$)
- ▶ Large Number of variables ($O(3)$)

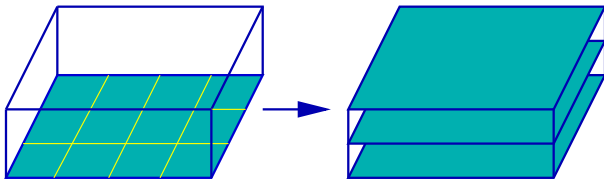
Requirements

- ▶ long term storage of metadata and data
- ▶ standardization
- ▶ compression

Solutions

- ▶ WMO GRIB standard
- ▶ lowest entropy data subsampling
- ▶ two stage compression: lossy entropy based and lossless compression of result *image* — metadata uncompressed!

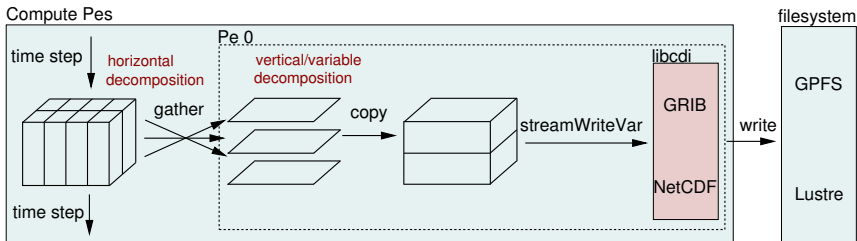
... continued



Problem

- ▶ model decomposition is based on two-dimensional horizontal slicing
- ▶ storage unit of model data is based on vertical slicing
- ▶ requires transpose and data gathering

File writing in ECHAM AS-IS



- ▶ All processes are compute processes.
- ▶ After one I/O timestep all processes gather their data on process 0.
- ▶ Process 0 writes the data to the filesystem.
- ▶ All processes compute till the next I/O timestep.

Problem

All processes wait until process 0 has written the data.

Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)



Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas



Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas
3. instead of doing I/O, copy data to buffer and continue simulation



Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas
3. instead of doing I/O, copy data to buffer and continue simulation
4. collectors collect their respectable data (gather) via one-sided (RDMA based) MPI calls and do the transpose



Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas
3. instead of doing I/O, copy data to buffer and continue simulation
4. collectors collect their respectable data (gather) via one-sided (RDMA based) MPI calls and do the transpose
5. compress each individual record



Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas
3. instead of doing I/O, copy data to buffer and continue simulation
4. collectors collect their respectable data (gather) via one-sided (RDMA based) MPI calls and do the transpose
5. compress each individual record
6. write ... sort of



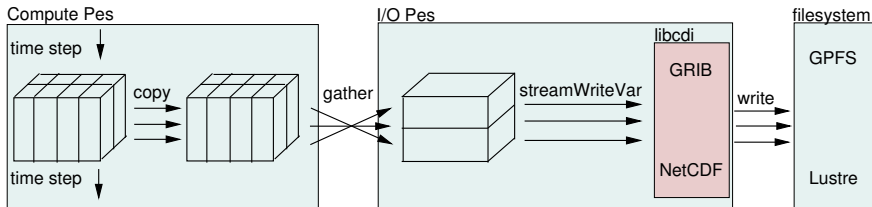
Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas
3. instead of doing I/O, copy data to buffer and continue simulation
4. collectors collect their respectable data (gather) via one-sided (RDMA based) MPI calls and do the transpose
5. compress each individual record
6. write ... sort of

First five steps runs fine, but the last one makes a lot of trouble!



File writing in ECHAM TO-BE



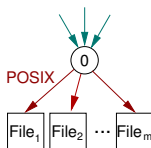
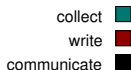
- ▶ After calculating one I/O timestep the compute processes copy their data to a buffer and go on calculating till the next I/O timestep.
- ▶ I/O processes fetch the data using MPI one sided communication.
- ▶ Gather and transpose of the data is based on callback routines supplied by the model.

Most important property

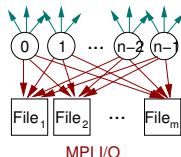
Compute processes are not disturbed by file writing.



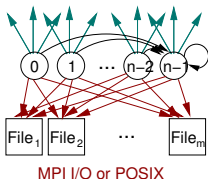
Algorithm alternatives



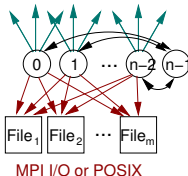
(a) *Classic serial,*
no comm.



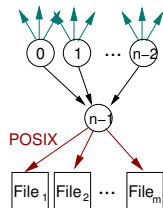
(b) *MPI writer,*
no visible comm.



(c) *Offset sharing,*
comm. offsets

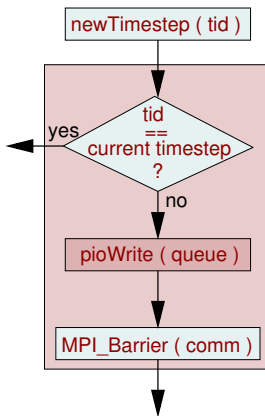


(d) *Offset guard,*
comm. offsets



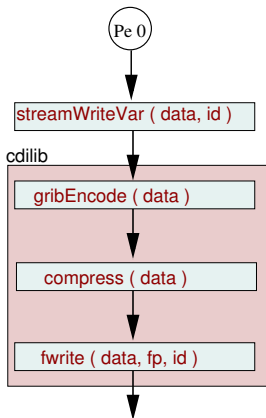
(e) *POSIX writer,*
comm. data

Timesteps



Characteristics

- ▶ Within files data have to be ordered by time.
- ▶ In case of serial writing is a natural barrier between the timesteps.
- ▶ When writing in parallel buffering of data is necessary for performance.
- ▶ Buffers have to be flushed after each I/O timestep to keep data in right time order.



Collect and write serial using `fwrite`.

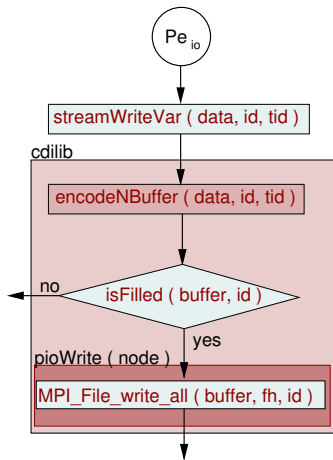
Advantages

- ▶ No communication is needed.
- ▶ Simple algorithm.
- ▶ Simple multi-file handling.

Disadvantages

- ▶ Does not scale.

Program flow (2), MPI writer (1)



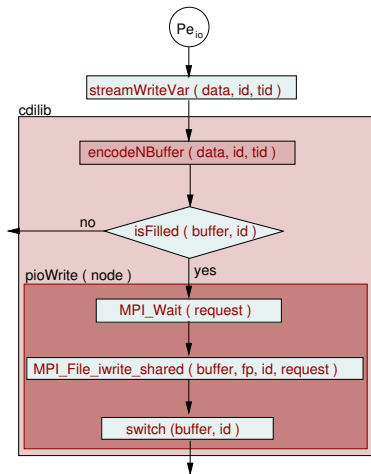
Collect and write parallel using `MPI_File_write_all`.

Advantages

- ▶ No user visible communication between I/O processes is needed.
- ▶ Straight forward implementation.
- ▶ Performs best of all MPI file writing routines.

Disadvantages

- ▶ Collective call, doesn't work for inhomogenous allocation of variables per process/file.



Collect and write parallel using `MPI_File_iwrite_shared`.

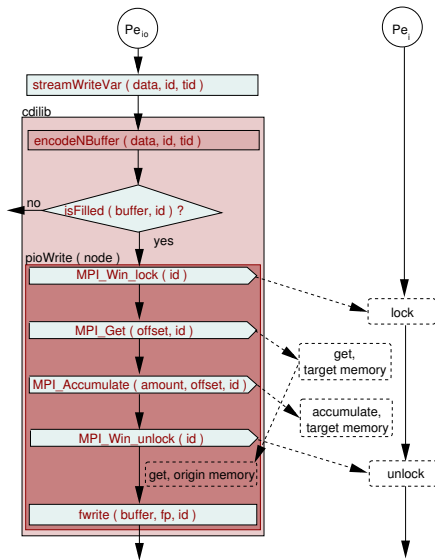
Advantages

- ▶ No user visible communication between I/O processes is needed.
- ▶ Use of double buffering is possible.

Disadvantages

- ▶ Very bad performance on GPFS.

Program flow (4), Offset sharing



Collect and write parallel,
communicate file offsets using
MPI RMA with passive target:

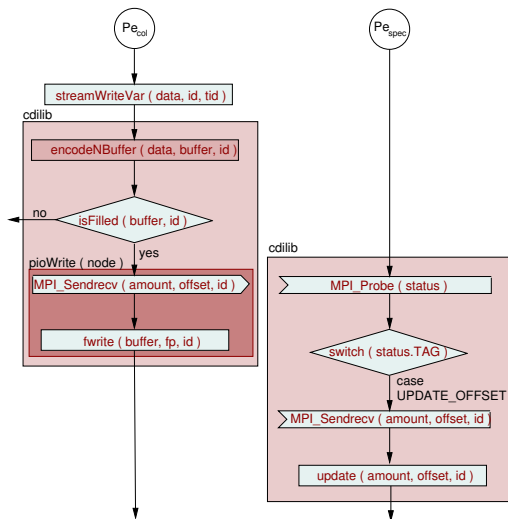
Advantages

- ▶ All collectors write.
- ▶ Use of POSIX AIO possible.

Disadvantages

- ▶ Complex locking is needed,
performance is bad.

Remark: double buffering not
done yet.



Collect and write parallel using one process to administrate file offsets.

Advantages

- ▶ Performs best of tested versions.
- ▶ Use of POSIX AIO possible.

Disadvantages

- ▶ One process is idle most of the time.
- ▶ Problems if I/O PEs span over different nodes.

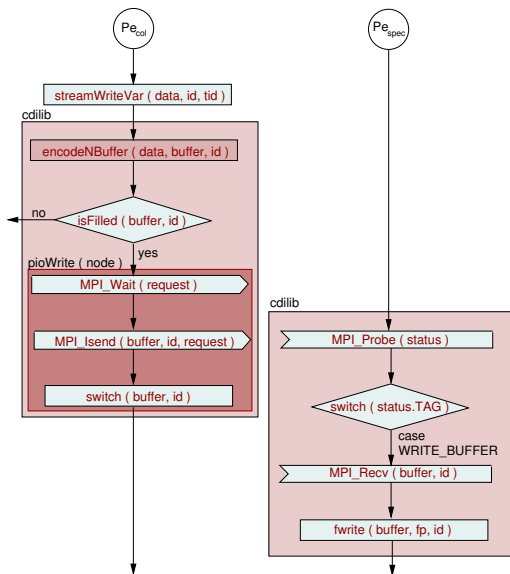
Remark: double buffering not done yet.

Program flow (6), Posix writer

Timings

Run fwrite

Run aio_write



Collect parallel and write serial using `fwrite`.

Advantages

- ▶ Use of POSIX AIO possible.
- ▶ Use of double buffering possible.
- ▶ Use one writer process for each file possible. This would probably gain performance. Furthermore these processes could be located on different nodes.

Disadvantages

- ▶ The buffer has to be communicated.
- ▶ With one writer process speedup limited.



Parallel performance profile analysis

We made several runs with the tools SCALASCA and TAU. Some of the results follow on the next slides. For these runs we let the processes write 5.9 GB distributed to 10 files.

Remark

The instrumentation of the code increased the runtime of the programs up to 130%!



MPI writer, MPI_File_iwrite_shared

MPI writer

8 Pes

Routines, functions	Time(s)	%	#	$\frac{\text{Time}}{\#\text{Pes}}$ (s)	%	\sum_{Pe_i}
cdiWrite	1370.2	100.0	8	171.3	100.0	100.0
grbEncode	674.2	49.2	8	84.3	49.2	
MPI_File_iwrite_shared	58.9	4.3	8	7.4	4.3	96.7
MPI, other	592.0	43.2	8	74.0	43.2	

16 Pes

Routines, functions	Time(s)	%	#	$\frac{\text{Time}}{\#\text{Pes}}$ (s)	%	\sum_{Pe_i}
cdiWrite	2190.8	100.0	16	136.9	100.0	100.0
grbEncode	611.5	27.9	16	38.2	27.9	
MPI_File_iwrite_shared	550.2	25.1	16	34.4	25.1	97.4
MPI, other	972.5	44.4	16	60.8	44.4	



POSIX writer, aio_write, 5 output streams

POSIX writer

4 Pes

Routines, functions	Time(s)	%	#	$\frac{\text{Time}}{\#\text{Pes}}$ (s)	%	\sum_{Pe_i}
cdiWrite	1582.3	100.0	4	395.6	100.0	100.0
grbEncode	540.4	34.2	3	180.1	45.5	96.6
MPI_Wait	606.3	38.3	3	202.1	51.1	
MPI_Recv	161.3	10.2	1	161.3	40.8	92.9
aio_suspend	206.0	13.0	1	206.0	52.1	

8 Pes

Routines, functions	Time(s)	%	#	$\frac{\text{Time}}{\#\text{Pes}}$ (s)	%	\sum_{Pe_i}
cdiWrite	3695.3	100.0	8	461.9	100.0	100.0
grbEncode	539.5	14.6	7	77.1	16.7	97.5
MPI_Wait	2645.6	70.6	7	373.3	80.8	
MPI_Recv	47.4	1.3	1	47.4	10.3	96.8
aio_suspend	399.6	10.8	1	399.6	86.5	



Offset guard

Offset guard

4 Pes

Routines, functions	Time(s)	%	#	$\frac{\text{Time}}{\#\text{Pes}}$ (s)	%	\sum_{Pe_i}
cdiWrite	875.7	100.0	4	218.9	100.0	100.0
grbEncode	540.4	61.7	3	180.1	82.3	
fseek/fwrite	74.3	8.5	3	24.8	11.3	95.9
MPI_Barrier	15.2	1.7	3	5.1	2.3	
MPI_Probe	218.3	27.9	1	218.3	99.7	99.7

32 Pes

Routines, functions	Time(s)	%	#	$\frac{\text{Time}}{\#\text{Pes}}$ (s)	%	\sum_{Pe_i}
cdiWrite	1646.5	100.0	32	51.5	100.0	100.0
grbEncode	544.1	33.1	31	17.6	34.1	
fseek/fwrite	590.0	35.8	31	19.0	37.0	92.1
MPI_Barrier	333.9	20.3	31	10.8	20.9	
MPI_Probe	50.4	3.1	1	50.4	97.9	97.9



POSIX writer, fwrite

POSIX writer

4 Pes

Routines, functions	Time(s)	%	#	$\frac{\text{Time}}{\#\text{Pes}}$ (s)	%	\sum_{Pe_i}
cdiWrite	822.7	100.0	4	206.2	100.0	100.0
grbEncode	540.7	65.6	3	180.2	87.4	
MPI_Wait	46.0	5.6	3	15.3	7.4	95.3
MPI_Barrier	2.8	0.3	3	0.92	0.5	
MPI_Recv	162.0	19.7	1	162.0	78.6	98.9
fwrite	41.8	5.1	1	41.8	20.3	

32 Pes

Routines, functions	Time(s)	%	#	$\frac{\text{Time}}{\#\text{Pes}}$ (s)	%	\sum_{Pe_i}
cdiWrite	3855.1	100.0	32	120.5	100.0	100.0
grbEncode	543.9	14.1	31	17.6	14.6	
MPI_Wait	2626.8	68.1	31	84.7	70.3	97.2
MPI_Barrier	459.9	11.9	31	14.8	12.3	
MPI_Recv	30.5	0.8	1	30.5	25.3	97.1
fwrite	86.5	2.2	1	86.5	71.8	



Timings

Target: 2 GB/s

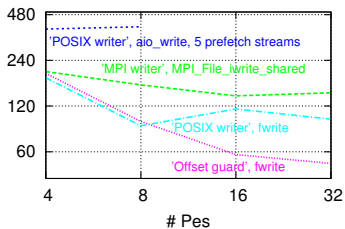
write method	# Pes	# collectors	# writers	[MB/s]
<i>Classic serial</i> , fwrite	1	1	1	148
<i>MPI writer</i> ,	4	4	4	234
MPI_File_ fwrite_shared	8	8	8	281
	16	16	16	338
	32	32	32	322
<i>Offset guard</i> , fwrite	4	3	3	240
	8	7	7	498
	16	15	15	829
	32	31	31	946
<i>POSIX writer</i> , fwrite	4	3	1	257
	8	7	1	450
	16	15	1	411
	32	31	1	482
<i>POSIX writer</i> , aio_write,	4	3	1	122
5 output streams	8	7	1	120

Not all possibilities might be fully optimized.

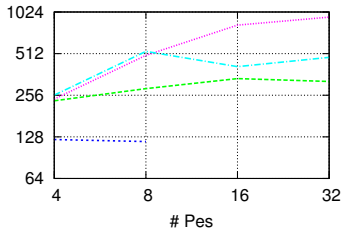


Comparison

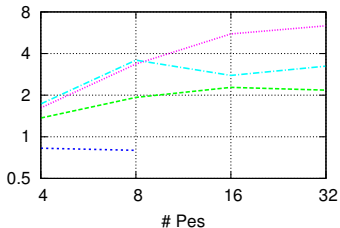
Elapsed time (s), 'Classic serial' 319 s



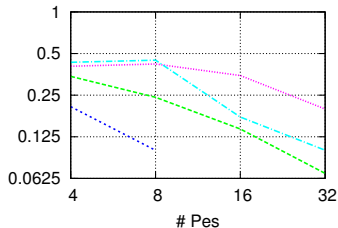
Throughput (MB/s), 'Classic serial' 148 MB/s



Speedup $S_p = T_s / T_p$



Efficiency $E_p = S_p / \# \text{ Pes}$



Some initial conclusions

- ▶ For our kind of models sufficient write speed on a single IBM node to go on for the next months



Some initial conclusions

- ▶ For our kind of models sufficient write speed on a single IBM node to go on for the next months
- ▶ RDMA concept is meeting the expectations - may need more RMA Windows.



Some initial conclusions

- ▶ For our kind of models sufficient write speed on a single IBM node to go on for the next months
- ▶ RDMA concept is meeting the expectations - may need more RMA Windows.
- ▶ Did part of MPI developers work on very unusual usage model of MPI-IO ...



Some initial conclusions

- ▶ For our kind of models sufficient write speed on a single IBM node to go on for the next months
- ▶ RDMA concept is meeting the expectations - may need more RMA Windows.
- ▶ Did part of MPI developers work on very unusual usage model of MPI-IO ...
- ▶ Expect them to take over and beat our results by at least a factor of ?.

