A UT/ORNL PARTNERSHIP
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES
NICS

A Pragmatic Approach to Improving the Large-scale Parallel I/O Performance of Scientific Applications

Lonnie D. Crosby, R. Glenn Brook, Bhanu Rekapalli, Mikhail Sekachev, Aaron Vose, and Kwai Wong

A Pragmatic Approach

- Data Movement
  - I/O is fundamentally data movement between the application and file system.
- Data Layout
  - I/O patterns are informed by the layout of data within the application and within files.
- I/O Performance
  - Although performance is very dependent on data layout within the application and within files,
  - the method by which these layouts are mapped to one another is a substantial contributor to performance.

Optimization of I/O Performance

- Data Layout
  - within the application is difficult to change (in some circumstances) due to domain decomposition and algorithmic constraints.
  - within files are easier to change; however, changes the manner in which post-processing or visualization occur.
  - will remain constant during the I/O optimization process.
- Mapping between Data Layouts
  - Best performance usually seen when the data layout within the application and within the files are similar.
  - Differences in data layout create constrains that may inform poor I/O implementations.

Goals of Study

- Show how I/O performance considerations are utilized in "real" scientific applications to improve performance.
- I/O Performance Considerations
  - Limit the negative impact of latency and maximize the beneficial impact of available bandwidth
    - Perform I/O in as few large chunks as possible.
  - Limit file system interaction overhead
    - Perform only the file opens, closes, stats, and seeks which are absolutely necessary.
  - Write/read data contiguously whenever possible.
  - Take advantage of task parallelism
    - Avoid file system contention

Kraken (Cray XT5)

- **Contains 9,408 compute nodes,**
  - **each containing dual 2.6 GHz hex-core AMD "Istanbul" processors, 16 GB RAM, and a SeaStar 2+ interconnect.**
- **Lustre file system**
  - **48 OSSs and 336 OSTs**
  - **30 GB/s peak performance**

Applications

- **PICMSS (The Parallel Interoperable Computational Mechanics Simulation System)**
  - **A computational fluid dynamics (CFD) code used to provide solutions to incompressible problems. Developed at the University of Tennessee's CFD laboratory.**
- **AWP-ODC (Anelastic Wave Propagation)**
  - **Seismic code used to conduct the "M8" simulation, which models a magnitude 8.0 earthquake on the southern San Andreas fault. Development coordinated by Southern California Earthquake Center (SCEC) at the University of Southern California.**
- **BLAST (Basic Local Alignment Search Tool)**
  - **A parallel implementation developed at the University of Tennessee, capable of utilizing 100 thousand compute cores.**

Application #1

- Computational Grid
  - 10,125 x 5,000 x 1,060 global grid nodes (5,062 x 2,500 x 530 effective grid nodes)
  - Decomposed among 30,000 processes via a process grid of 75 x 40 x 10 processes. (68 x 63 x 53 local grid nodes)
  - Each grid stored column-major.
- Application data
  - Three variables are stored per grid point in three arrays, one per variable (local grid). Multiple time steps are stored by concatenation.
- Output data
  - Three shared files are written, one per variable, with data ordered corresponding to the global grid. Multiple time steps are stored by concatenation.

Optimization

Original Implementation

- **Derived Data type created via MPI_Type_create_hindexed**
  - Each block consists of a single value placed by an explicit offset.

Optimized Implementation

- **Derived Data type created via MPI_Type_create_subarray**
  - Each block consists of a contiguous set of values (column) placed by an offset.

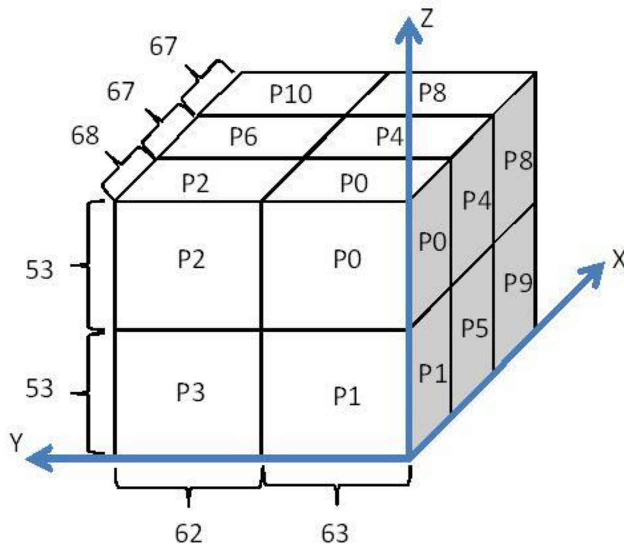


Figure 1: The domain decomposition of a 202x125x106 grid among 12 processes in a 3x2x2 grid. The process assignments are listed P0-P11 and the numbers in brackets detail the number of grid nodes along each direction for each process block.

Results

- **Collective MPI-IO calls are utilized along appropriate Collective-buffering and Lustre stripe settings.**
  - **Stripe count = 160   Stripe Size = 1MB**

Table 1: Comparison between Original and Optimized I/O in Application 1

		Bandwidth (GB/s)	
Time Step	Data (TB)	Original	Optimized
20	1.46	1.05	2.03
40	1.46	1.06	1.66
60	1.46	1.40	2.16
80	1.46	1.16	1.94
100	1.46	1.27	2.30
120	1.46	1.55	2.06
140	1.46	0.98	1.70
160	1.46	1.30	2.65
180	1.46	0.87	3.34
200	1.46	1.13	2.99

- **Given amount of data**
  - **Optimization saves 12 min/write**
  - **Over 200 time steps, savings of about 2 hours.**

Application #2

- Task based parallelism
  - Hierarchical application and node-level master processes who serve tasks to node-level worker processes
  - Work is obtained by worker processes via a node-level master process. The application-level master provides work to the node-level master processes.
  - I/O is performed per node via a dedicated writer process.
- Application data
  - Each worker produces XML output per task. These are concatenated by the writer process and compressed.
- Output data
  - A file per node is written which consists of a concatenation of compressed blocks.

Optimization

Original Implementation

- On-demand compression and write.
  - When the writer process receives output from a worker it is immediately compressed and written to disk.
- Implications
  - Output files consist of a large number of compressed blocks each with a 4-byte header.
  - Output files written in a large number of small writes.

Optimized Implementation

- Dual Buffering
  - A buffer for uncompressed XML data is created. Once filled, the concatenated data is compressed.
  - A buffer for compressed XML data is created. Once filled, the data is written to disk.
- Implications
  - Output files consist of a few, large compressed blocks each with a 4-byte header.
  - Output files written in a few, large writes.

Results

- Benchmark case utilizes 24,576 compute cores (2,048 nodes) Optimized case utilizes 768 MB buffers.
  - Stripe count = 1   Stripe Size = 1MB

- Compression Efficiency
  - Compression ratio of about 1:7.5
  - Compression takes longer than the file write.
  - With optimizations, file write would take about 2.25 seconds without prior compression.

Table 2: Comparison between Original and Optimized I/O in Application 2

	Original	Optimized
Average Compression Time (s)	11.93	8.85
Std. Dev. (s)	0.79	0.46
Bandwidth (MB/s)	25.75	34.63
Average Write Time (s)	10.47	0.30
Std. Dev. (s)	8.36	1.06
Bandwidth (MB/s)	16.04	466.67

Application #3

- Computational Grid
  - 256^3 global grid nodes
  - Decomposed among 3,000 processes via XY slabs in units of X columns. The local grid corresponds slabs of about 256 x 22 nodes.
  - Six variables per grid node is stored.
  - Each grid stored column-major.
- Application data
  - A column-major order array containing six values per grid node.
- Output data
  - One file in Tecplot binary format containing all data (six variables) for the global grid in column-major order and grid information.

Optimization

Original Implementation

- File open, seek, write, close methodology between time steps.
- Headers written element by element. Requires at least 118 writes.

Optimized Implementation

- File is opened once and remains open during run.
- Headers written by data type or structure. Requires 6 writes.

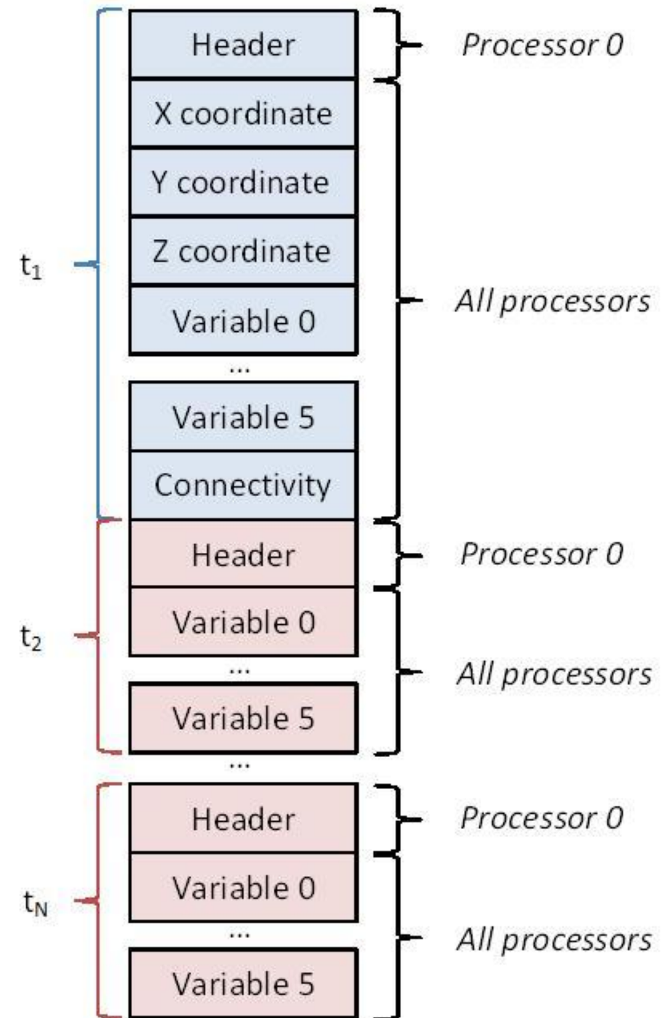


Figure 2: A representation of the Tecplot binary output file format.

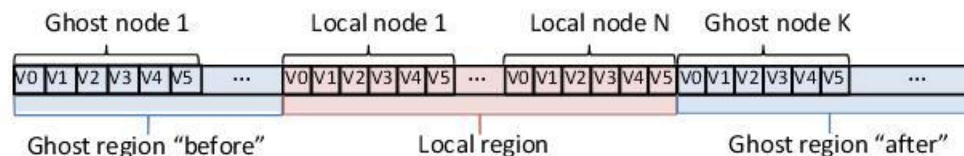
Optimization

Original Implementation

- Looping over array indices to determine which to write within data sections.
  - Removal of ghost nodes
  - Separation of variables
- Use of explicit offsets in each data section.

Optimized Implementation

- Use of derived data types to select portion of array which contains only local region and appropriate variable.
- Use of derived data type to place local data within data section.



- **Figure 3:** A representation of the local process's data structure. The local and ghost nodes are labeled.

Results

- **Collective MPI-IO calls are utilized along appropriate Collective-buffering and Lustre stripe settings.**
  - **Stripe count = 160   Stripe Size = 1MB**

- **Collective MPI-IO calls**
  - **Account for about a factor of 100 increase in performance.**
  - **The other optimizations account for about a factor of 2 increase in performance.**

Table 3: Comparison between Original and Optimized I/O in Application 3

		Bandwidth (GB/s)	
Time Step	Data (GB)	Original	Optimized
1	1.62	$3.47\text{x}10^{-02}$	8.15
2	0.75	$2.18\text{x}10^{-02}$	4.22
3	0.75	$1.91\text{x}10^{-02}$	5.39
4	0.75	$1.74\text{x}10^{-02}$	3.28
5	0.75	$2.18\text{x}10^{-02}$	4.54
6	0.75	$2.05\text{x}10^{-02}$	3.23
7	0.75	$2.01\text{x}10^{-02}$	4.85
8	0.75	$1.79\text{x}10^{-02}$	4.56
9	0.75	$2.59\text{x}10^{-02}$	4.79
10	0.75	$2.62\text{x}10^{-02}$	4.03

Conclusion

- Optimization of I/O performance was achieved without
  - changing the output file format.
  - changing the data layout within the application.
- I/O performance optimization allowed
  - an increase in I/O performance of about a factor of 2 for a data-intensive application. Over the course of 200 time steps this saves about 2 hours of I/O time.
  - an increase in I/O performance which may allow the removal of a time consuming data compression step.
  - an increase in I/O performance of about a factor of 200. A performance increase of a factor of 100 is attributed to the use of collective MPI-IO calls which wasn't possible before initial optimization.