I/O Friendly Data Parallelization for Spatial Computation

Baoqiang Yan, Philip J. Rhodes

Department of Computer and Information Science, University of Mississippi, University, Mississippi 38677

Introduction

- Two important performance issues involved in efficient data parallelization for HPC on large spatial datasets are the heavy I/O costs and the communication overhead. Scientist-programmers are either not skilled in writing parallel code to solve these issues or find such issues a distraction from the science.
- Compiler based data parallelization relies on either the reference pattern detected through source code analysis or user-specified directives for data partitioning on homogeneous processors without addressing out-of-core I/O issues directly. In GRID environments, I/O performance can be improved through careful data and code placement techniques such as data declustering, data replication and scheduling frameworks. Data parallelization is done via code replication. Except for parallelizing compiler techniques, they all require considerable user involvement in data partitioning.
- Efficient automatic data parallelization is thus desirable to save scientists from onerous efforts. It should and can be informed by the dependency pattern inherent in the application. It should be I/O friendly and adaptive to changes in the dependency pattern when application specific parameters change, such as the view direction in ray casting as shown in figure 1.

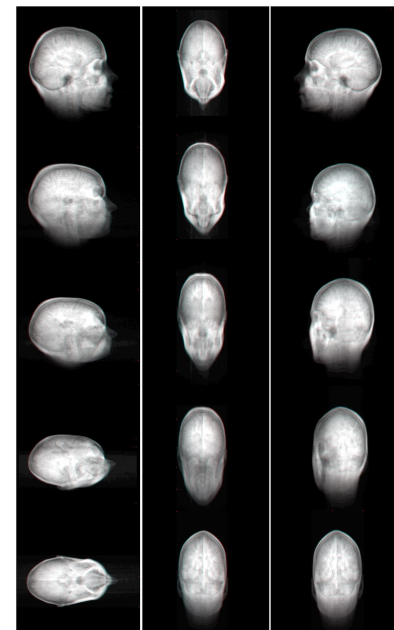

Figure 1. 3Dhead images generated using our automatic data parallelization mechanism for computing clusters. The head data is courtesy of Siemens Medical Systems, Inc.

- We propose an iteration and dependency aware data partitioning mechanism that turns explicit dependency information into suitable access patterns for efficient data loading and partitioning. This mechanism is particularly valuable for the visualization of datasets that are much larger than the aggregate storage capacity of a cluster. It also aggregates cluster I/O requests to minimize the effect of disk and network latency on performance.

Granite Datasource Component

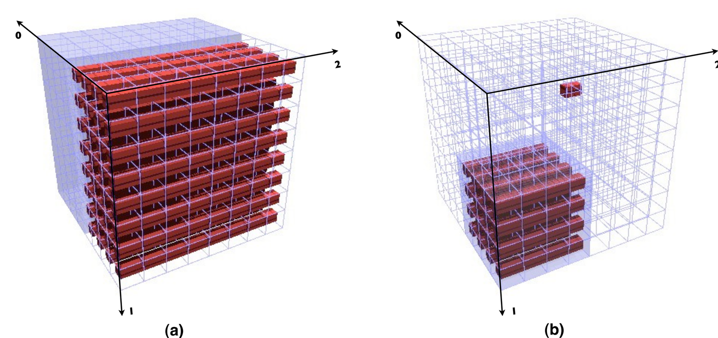

Figure 2. Granite datasource component (a) A 3D data volume stored in {0,1,2} ordering. (b) Datum and subblock queries.

- Datasource — conceptually an *n*-dimensional array that is physically stored in linear fashion.
- Storage ordering — the ordering of axes in the nested loop of its linear storage. {0,1,2} in example of figure 2.
- Rod storage model — a rod is a series of elements that are contiguous in both the data volume and the one dimensional file or memory array. The number and length of rods in a query determine the cost to satisfy the query.
- Basic queries — datum query and subblock query.

Iteration Aware Prefetching and Caching

- Iterator — an object that represents a regular access pattern over a data volume.
- Iteration ordering — determines the direction in which a rectilinear iterator proceeds through an iteration space.
- Prefetching cache — given an iterator, Granite can create a multidimensional cache block that contains all the data needed in current and future iteration steps. It is efficient since the number of disk reads is reduced by extending rods and the cache is well-formed, meaning no reloading of cache blocks is needed.

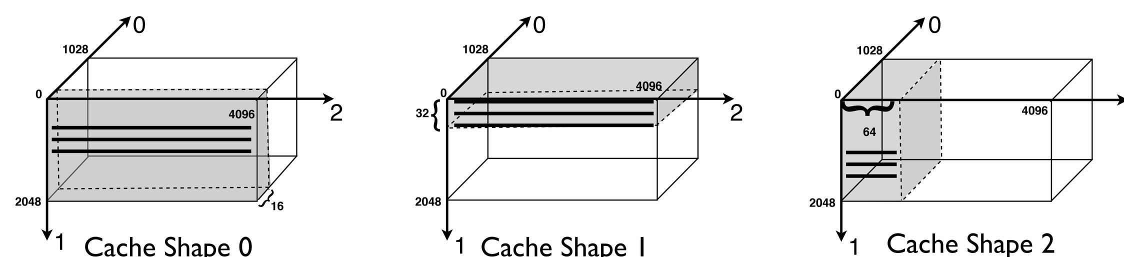

Figure 3. Different cache shapes and their cache block dimensions for a 1028*2048*4096 data volume using 128MB cache memory. Although cache shapes 0 and 1 have longer rods, requiring fewer reads to fill the block from disk, shape 0 has somewhat better locality. Shape 2 is the worst case, since the shorter rods mean that a larger number of reads are required to fill the block.

- Cache shapes — determines how fast the data is brought into memory. The shape that best matches the storage layout of the datasource brings the best I/O performance, such as cache shape 0 shown in figure 3.
- Incomplete access pattern — iteration ordering is not predetermined.
- Pattern Converter — changes incomplete access pattern to a complete one that best suits the storage ordering of data source.

Dependency aware caching and partitioning

- Dependency descriptor — represents the input application dependency pattern and is specified using two bitsets, B_e for existence and B_d for direction. For example, B_e {1,0,1} and B_d {0,*,1} mean positive dependency along axis 0, no dependency along axis 1 and negative dependency along axis 2.
- Efficiency Rules in Data Parallelization

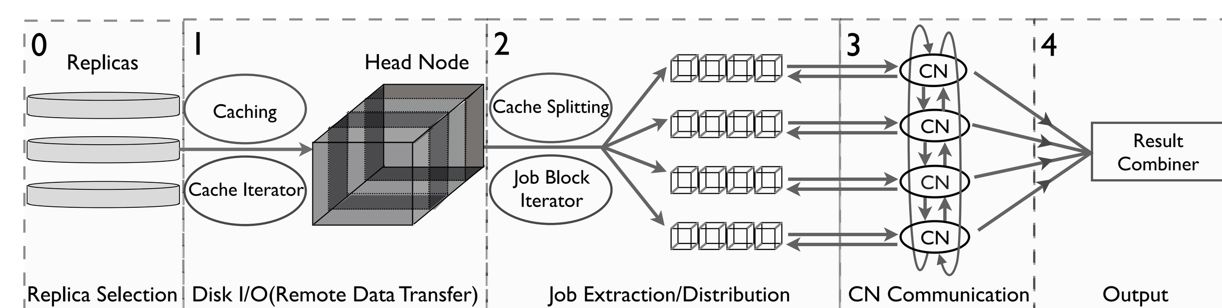

Figure 4. Five stages of a planned cluster computing system for large datasets. CN - Compute Node.

Cache Loading Cost in Stage One

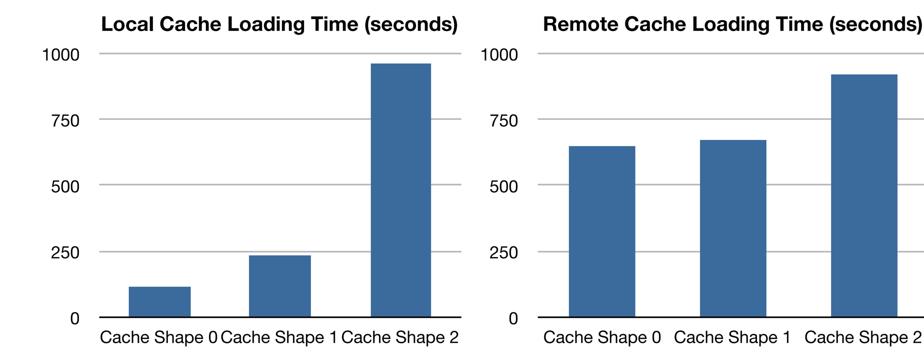

Figure 5. The cache loading time using 256MB of cache memory and different cache shapes, using the Orion cluster located at the University of Mississippi. a) Cache loading time for local disk. b) Remote cache loading time, accessing a fast server located at the University of New Hampshire.

Rule 1: Always use cache blocks that result in the fewest reads from disk.

Job Extraction Cost in Stage Two

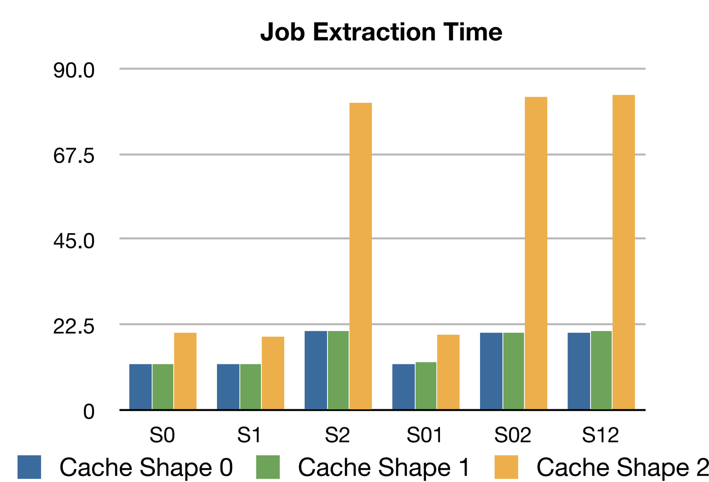

Figure 6. The job extraction time from different cache shapes of the same cache memory size 256MB. S01 - split along axes 0 and 1. Times are in seconds.

Rule 2: Avoid splitting the cache rod axis.

Job Distribution Cost in Stage Two

Rule 3: Split along as few axes as allowed by the memory on the compute nodes.

Job Splitting	Dep {0,1}	Dep {0,2}	Dep {1,2}
S0	302 ▲	303 ▲	68 ★
S1	92	69 ★	90
S2	75 ★	90	90

Table 1. The combined time of stage two and three for cache shape 0 and single axis splittings. Time unit - second.

Job Splitting	Dep {0,1}	Job Splitting	Dep {0,2}	Job Splitting	Dep {1,2}
S01LUC	87	S01NC	71	S01NC	71
S01MUC	88	S01YC	89	S01YC	86
S02NC	79	S02LUC	86	S02NC	78
S02YC	89	S02MUC	88	S02YC	86
S12NC	78	S12NC	78	S12LUC	89
S12YC	87	S12YC	86	S12MUC	87

Table 2. The combined time of stage two and three for cache shape 0 and double axis splittings. NC/YC - Do Not/Do require communication. LUC/MUC - Less/More Unit Communication Size. Time unit - second.

Communication Overhead in Stage Three

Rule 4: Avoid partitioning a dependency axis among compute nodes. If unavoidable, and R6 does not apply, choose the partitioning with less unit communication (LUC) size.

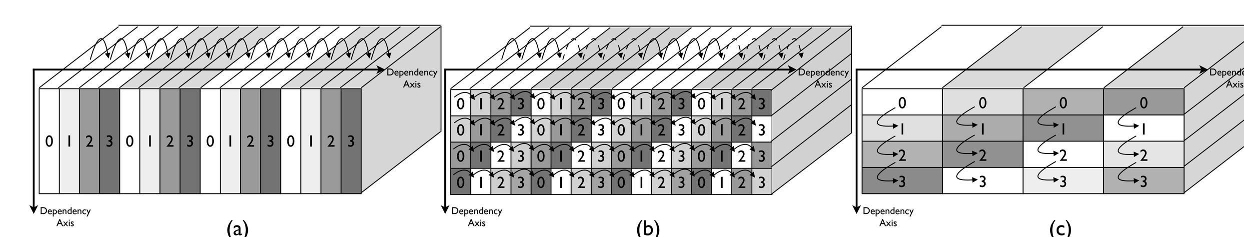

Figure 7. Serialized application among compute nodes and its avoidance. The shading on the top face represents different cache blocks loaded from disk or remote server. Blocks with the same shade of gray on the front face are processed at the same time for each cache block. Numbers indicate compute nodes. (a) Full serialization (b) Partially parallelized with finer jobs (c) Partially parallelized with coarser jobs, the best case.

Avoid Serialization of Application

Rule 5: For a single-axis partitioning, if the split axis and cache iteration axis are the same axis k, then only use this partitioning if k is not a dependency axis.

	MSA == CRA	MSA != CRA
MSA ∈ DA	I Worse Job Extraction Inter Compute Node Communication	II Better Job Extraction Inter Compute Node Communication
MSA ∉ DA	III Worse Job Extraction No Inter Compute Node Communication	IV Better Job Extraction No Inter Compute Node Communication

MSA - Major Split Axis   CRA - Cache Rod Axis   DA - Set of Axes with Dependency

Figure 8. Cost evaluation of different job splitting cases based upon the relation between the major split axis, cache rod axis and the axes with dependence.

A Conflict Resolution Rule

Rule 6: If job blocks for a candidate partitioning are sufficiently short along the cache rod axis, choose a new partitioning even if it incurs communication.

- Scalability with no first stage I/O cost

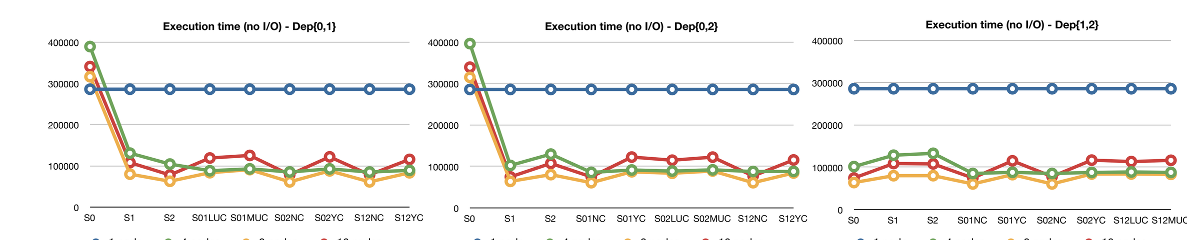

Figure 9. Execution time on an increasing number of nodes for different partitioning strategies and dependency patterns. A 128MB disk cache was used. Times are in milliseconds.

Ray Casting Performance Results

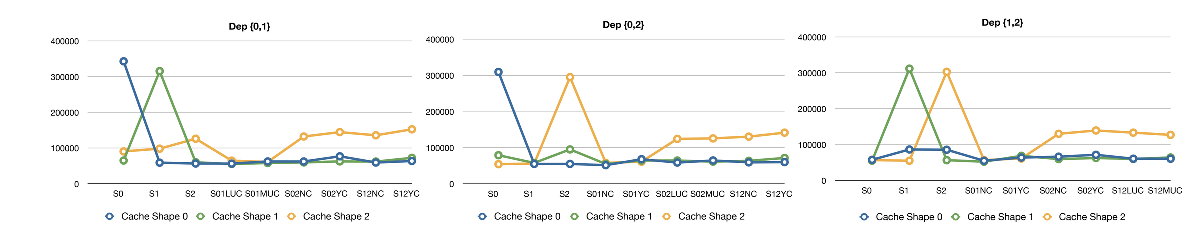

Figure 10. Ray casting application performance results on Orion cluster using 8 nodes. Time unit - millisecond.

Conclusion

Built upon Granite iteration aware spatial prefetching and caching techniques, our data parallelization scheme takes an explicit specification of data dependency, identifies the best feasible access patterns for efficient cache loading and data partitioning respectively. This scheme prioritizes but reconciles the I/O costs in the different stages of a planned cluster application to achieve the overall best I/O performance while maintaining fair scalability.

Acknowledgment

This work was supported in part by the National Science Foundation under grant CCF-0541239