IDEA — An API for Parallel Computing with Large Spatial Datasets

Baoqiang Yan
Department of Computer Science, Mathematics and Physics
Missouri Western State University
(01) 816-2714372
byan@missouriwester.edu

Philip J. Rhodes
Department of Computer and Information Science
University of Mississippi
(01) 662-9157082
rhodes@cs.olemiss.edu

Abstract — We describe IDEA, an API designed specifically for the parallel processing of large spatial datasets on a cluster. Because such datasets present special challenges for efficient I/O and communication, it is especially valuable to provide an API that frees the user from the burden of partitioning the data among the processors. IDEA allows the user to address a communication to neighboring blocks of data, rather than processes or nodes. In addition to being very natural for the user, this data-centric view allows communication to a data block before it has been assigned a process. This is a key ability when handling data sets larger than the aggregate memory capacity of the cluster, since the dataset must be processed in a piecewise fashion.

Keywords — cluster; parallelization; spatial dataset; dependency; data partitioning

I. INTRODUCTION

As designers run up against the limits of physics in their quest for smaller and faster processors, the importance of parallelism in scientific computing is steadily increasing. Unfortunately, existing programming environments such as MPI, CUDA, pthreads, KeLP and PETSc can be difficult to use partly because they require the scientist to manually partition the data among processors or cluster nodes [1, 7, 18, 19, 20]. From the scientist’s point of view, thinking about processors or nodes is an artifact of the parallel platform, and distracts from the science at hand. The IDEA API for parallel spatial computation described in this paper provides a data-centric view of the computation that hides the details of partitioning data among nodes.

IDEA is intended to address the difficulties encountered when processing large spatial datasets. Spatial data presents special challenges because data elements that are contiguous in the n-dimensional data space may be far away in a one dimensional memory array or disk file. Very large data volumes exacerbate this problem, and present difficulties of their own. When the size of a dataset exceeds the aggregate memory capacity of a cluster, we must find efficient ways to load the data from disk or remote server into the cluster in a piecewise fashion. However, this piecewise strategy must accommodate the case where the value of a data element depends upon a neighboring element that does not correspond to an active process. The IDEA repository is able to store results computed by a node and forward them to future computations that require them.

The IDEA system asks the scientist to write a kernel — the code that processes a single block of data carved from the larger data volume. Supported by the repository, kernel code is able to communicate with neighboring blocks using either relative coordinates (e.g. left, right, up, down, etc.) or absolute coordinates (e.g. block {10,1,4}) denoting another block’s location in the partitioned data volume. These spatial relationships are most natural to the scientist, and preferable to the process numbers or thread IDs used in other systems.

IDEA also asks the scientist to write two other modules. The result combiner performs a generalized reduction on processed data, while the dependency descriptor allows the user to specify the dependencies inherent in the calculation in a spatial manner. Often, the dependencies are entirely static, but we also support the important case where dependencies vary with application parameters. For example, in a visualization application, dependencies may vary with view direction, so when the user relates this parameter to the dependency descriptor, IDEA can make dynamic partitioning decisions as the view direction changes.

After a brief review of background and related work, we will give an overview of the IDEA system in section III. Section IV discusses two example applications implemented in IDEA, while section V compares computational performance with MPI. Section VI contains conclusions and future work.

II. BACKGROUND

Many popular parallel programming APIs leave the job of partitioning the data in the hands of the user. For example, MPI [7] usually requires the user to map a process identifier to a section of the dataset. Though MPI's grid communicator defines a topology for a group of processes and allows the user to refer to neighboring processes, MPI still leaves the data partitioning to the user. The CUDA API for GPU programming similarly provides thread identifiers, and requires the user to determine what data a thread will process [1]. CUDA's grid scheme provides a way to divide a computation into blocks of threads, where each block runs concurrently on the device. It is entirely up to the user to relate the block ID and...
thread ID to elements of the data set. OpenMP, another popular system, doesn’t require the user to map data to processors, but has no special facilities for spatial data or for data stored on disk or remote server [2]. *Kelp* [18,19] provides a set of runtime communication abstractions for irregular blocked data. The programmer uses intuitive geometric constructs to describe dependence patterns among data and express complex communication patterns through a high-level interface. However, *Kelp* leaves the data partitioning completely to the programmer. *PETSc* [20] is a set of libraries that facilitate the implementation of large-scale application codes on high-performance computers. Essentially, *PETSc* uses the MPI standard for communication. The users of *PETSc* usually face a steep learning curve and also have to perform the data partitioning and communication by themselves.

*Loci* [17] is a rule-based software framework that provides an automatic system for generating the deadlock-free communication and synchronization in parallel computation. Loci employs a relational abstraction of data and allows the designer to use SQL queries to describe intended access patterns. All data from a query operation will then be automatically brought in without specific parallel instructions from the developers. However, Loci does not consider the efficient access to data on disk or a remote server either. Recent work by Falco et al. describes and investigates the utility of *skeletons* for parallel programming [4, 5]. Their *Quaff* system asks the user to break the computation into logical components that can be automatically parallelized. They point out that Google’s *MapReduce* [3] system can be viewed as an example of a parallel skeleton. The IDEA system can also be viewed as a skeleton, but tailored specifically for large spatial datasets. Furthermore, IDEA performs the partitioning on behalf of the user, taking into account the various costs. The user can think of their computation as being performed on blocks of undefined dimension, and how that computation relates and interacts with neighboring blocks. The notion of a process is removed entirely.

IDEA is being developed as a component of the *Granite Scientific Database System* [8], which is designed to provide efficient access to spatial datasets. An unusual feature of the Granite system is *Iterator Aware Prefetching (IAP)*, which allows a highly specialized cache to be constructed using knowledge provided by an iterator [9]. Since the iterator completely specifies the access pattern, Granite caches can flawlessly prefetch data, and never have to load data more than once [9].

IDEA takes advantage of this same mechanism by transforming a user-supplied description of dependencies into a iterator that traverses the dataset. Granite constructs the associated cache, which IDEA divides into blocks that are distributed to the nodes. The cache is effectively aggregating the many I/O requests of the cluster nodes into a few large requests, minimizing the number of times disk or network latency costs are paid. Our initial efforts used volume visualization as a motivating application [11], but our current implementation is able to handle a wider variety of applications.

We recently completed a feasibility study for this system, developing a series of rules for choosing the best manner in which to partition the data among the compute nodes [12]. The rules take into account not only disk and network latency, but also other factors such as communication costs between nodes and the cost of copying data in memory. The effectiveness of the rules was verified experimentally using data stored on disk in *linear format* (i.e. “plane-by-plane, row-by row”). However, we expect our work to be easily extendable to chunked files and space-filling curves [10, 13]. It also can be extended to work with declustering [6, 14, 15] for parallel I/O and to multiple aggregators, although we currently use a single dedicated machine to aggregate cluster I/O requests.

The work described in this paper extends our feasibility study by completing a prototype version of our

---

![Figure 1. Overview of IDEA architecture. The major components of IDEA are the partitioning manager and the repositories, communication repository and job repository. The user needs to define the application specific dependency detector function, the kernel function and the result combination function. The dotted lines represent communications with other nodes.](image-url)
API for parallel computation with spatial data and demonstrating the usefulness of that API for two important but distinct types of spatial computation. In addition to variants of send() and recv() functionality, we have implemented several repositories, used for storing communication between compute nodes and time steps for iterative applications. Together, the new repositories and API provide a natural and easy to use parallel programming environment for data intensive spatial applications. The IDEA system chooses efficient ways of accessing data from disk or remote datasource and partitioning data among compute nodes on behalf of the programmer.

III. SYSTEM OVERVIEW

Figure 1 on the previous page demonstrates the overview of IDEA. The system side of IDEA is composed of two major components, the partitioning manager and the repositories. The user needs to define three functions, a dependency detector function, the kernel and the result combination function.

A. Partitioning Manager

The partitioning manager performs two tasks. First, it selects the proper replicas and shapes the cache blocks so that the data can be loaded efficiently piece by piece if it is too large for the cluster’s local storage capacity. Second, the cache blocks are split into jobs (a data block processed by a compute node) before being distributed among the compute nodes. The partitioning manager also shapes the job blocks to reconcile the costs in job extraction and communication so that the overall best application performance can be realized [12].

Both tasks are done by constructing corresponding spatial iterators. An iterator is an object that both represents the access pattern and performs the iteration on a dataset, either on disk (local and remote) or in memory. IDEA constructs a global cache iterator and sends to each caching node a subset cache iterator for parallel data loading which is performed on the data server nodes and initiated by job misses. IDEA also constructs a global iterator for the job assignment among compute nodes and separate job iterators local to each caching node for job extraction. The synchronization between the global and local job iterators is coordinated through the job-source map created by the partitioning manager. The job-source map describes where each job block should be transferred from. IDEA's job receivers on compute nodes will first query the job source map before contacting the data caching nodes for job transfer. In the single local data source scenario that we assume, the cluster’s head node serves as both a data server and the caching node. The data partitioning manager also generates a job-node map which describes the mapping of job blocks to compute nodes. This map is distributed to all compute nodes to facilitate the communications. In both maps, the jobs are identified by its global coordinates within the whole application iteration space. They will be rebuilt and redistributed if a repartitioning is needed due to changes in attributes such as key application parameters or data storage layout. However, these maps are used only by IDEA internally and invisible to the user. The user’s application codes do not need to be changed.

Selection from among several replicas of spatial data with different storage layout on disk is a topic for future research. In this paper we assume a single data source with a single linear storage layout, but symmetrical arguments can be made for other storage layouts.

B. Repository Mechanism

IDEA has a special repository mechanism to handle the internode communications and time-stepped applications. IDEA's repositories are divided into two categories, communication repositories and job repositories.

1) Communication Repositories

The communication repositories are designed for dependent data transfer among compute nodes, such as the ray’s blending results in parallel volume rendering and the ghost values in parallel Jacobi iteration. Unlike other parallel programming libraries [1, 7] that require explicitly referring to the process or thread ids, IDEA allows the user to initiate the communications by specifying the source or destination job block’s coordinate or location within the application’s data space, either absolutely or relatively as shown in figure 2. The user does not know...
the mapping of job blocks to processors. This feature of IDEA lends itself to the ability of dynamic data partitioning in response to the changes in application parameters. IDEA hides the parallelization details and the user is insulated from the complexity of parallel programming. This transparency makes it unnecessary for the user to change their code using this API even if a repartitioning is performed.

Once an inter-job data transfer is requested, it will return immediately without having to wait until the data is actually sent or received. Instead, a send request will put the data to transfer in the send repository and a receive request will be queued up until the requested data is ready in the receive repository. The non-blocking feature of IDEA’s repository mechanism leaves chances for the data processing to be overlapped with the communications. It also allows the communications with job neighbors that haven’t been assigned to any active processors. This happens when the dataset is larger than the aggregate memory of a cluster and need to be processed in a piecewise fashion. The repositories will hold the data temporarily for future computation.

Associated with the send/receive repositories are the sender/receiver thread pools which work in a client/server manner. The thread pools are created when the compute nodes are started and will be woken up when a data transfer is requested. They will use the job-node map to figure out the source and destination nodes and execute the requests. If the requested data turns out to be on the same node, it can be directly moved from the send repository to the receive repository.

2) Job Repository

The job repository is designed for applications that require multi-step iterations over the dataset, such as Jacobi iteration and similar finite differences methods. Typically, these applications need temporary buffers to hold the new values for the next step. However, if the target dataset is too large to keep all the new values in memory, we have to store them on local disk or externally, perhaps using network attached storage. The job storing operations are performed automatically and keep the order in which they were initially visited so that the iteration can be repeated.

C. Dependency Descriptor

To use IDEA, the client application needs to specify parameters such as the target dataset, the region of interest (ROI), the available memory for caching, the number of compute nodes to use, the maximum job size, and the memory for the job repository. In addition, the user specification of the application dependency is very important for IDEA to realize efficient parallelization. This information helps IDEA to make the right choice of data partitioning during parallelization to avoid higher communication overhead.

The application dependency is represented using a dependency descriptor, as shown in figure 3. It specifies the iteration-space-wide dependency pattern in terms of dependency axes, directions and transfer sizes. The transfer size in the dependency descriptor defines the amount of data in bytes that needs to be transferred along each axis. If the transfer sizes are all zeros on all axes, the application is embarrassingly parallel. Otherwise, the dependency pattern can be unidirectional or bidirectional. This determines the legal start scope of an application, namely which parts of the dataset the program can start processing. For unidirectional dependency, a positive direction indicates that values with smaller coordinates for that axis must be visited before values with larger coordinates. Bidirectional dependency means that data transfer needs to be performed in both directions. However, the transfer sizes in these two directions could be uniform or nonuniform. For example, Jacobi iteration demonstrates a uniform bidirectional dependency pattern.

Figure 3. Graphical representations of example unidirectional dependency descriptors for ray casting on a 3D dataset. The view direction vectors are (a) <0,1,0> (b) <1,-2,0> (c) <1, 2, -3> respectively.
since the ghost areas on both sides of each job block on an axis have the same volume. However, in particle tracing, the position of a particle in next step is unpredictable. In this case, the dependency pattern would usually be bidirectional and nonuniform. We are currently dealing with either unidirectional dependencies or bidirectional dependencies with uniform transfer sizes. However, IDEA would still be able to serve applications with nonuniform bidirectional dependency patterns as long as they are data intensive applications on large spatial datasets since I/O is the main bottleneck and that is what IDEA addresses. Any penalty on computational performance would very likely be overwhelmed by the I/O performance gains brought by IDEA’s collective prefetch caching[12].

For IDEA to perform the proper data partitioning in different application scenarios, the programmer needs to define a dependency detector function. IDEA requires that this function return a dependency descriptor no matter what the input parameters are, as shown in figure 4.

Knowledge of dependency pattern helps IDEA choose the right data caching and partitioning methods that allow the application to start, realize high-performance I/O and avoid higher communication overhead.

As demonstrated in figure 5a, once the partitioning is determined, IDEA will automatically start the main function on the compute nodes which will initialize the send/receive repositories, invoke the job receiver threads to get data from the job distributors on the caching nodes and feed them to the kernel by calling the system function ProcessJobs(), as shown in figure 5b.

### D. Kernel Function

The kernel function is used to perform job processing and initiate communications among compute nodes. As mentioned in figure 5b, it takes a JobBlock as input. The JobBlock contains information such as the dimensions of the job block, its scope and position within the whole data space, current time step, and the data values. IDEA maintains a byte view of the data all the way from data loading to feeding jobs to the kernel. This strategy not only avoids the unnecessary type conversions during the job transfer over the network, but also provides a general uniform interface to the user. The application specific type conversions, if needed, are left to the application programmer.

Figure 6 demonstrates the general structure of a kernel function for IDEA users. The first thing is to figure out which neighbors to contact for dependent data and submit the corresponding requests. This is based upon the information contained in the job block and requires

```c
int main(int argc, char *argv[]){
    initJobBuffs(NUMJOBBUFFS);
    initJobRecvThreads(NUMJOBBUFFS);
    ...
    initRepos();
    initReposSendThreads(numThreads);
    initReposRecvThreads(numThreads);
    ...
    ProcessJobs(kernelFunc);
    ...
    //wait for the repository to be //purged out and terminate
}
```

```c
typedef void kernel(JobBlock *);

ProcessJobs(kernel *kfunc){
    JobBlock *jblk;
    while(moreJobsToReceive() || moreJobsToProcessInBuffs()){
        //iterate through buffers
        jblk = nextJob();
        kfunc(jblk);
    }
}
```

Figure 5. IDEA’s code snippets of the main function on compute nodes and how it feeds data to the kernel function. The nextJob() function gets the jobs one after another from the job buffers.

```c
SubmitDepDataRequest(jblk, nbList);
SendToNeighbor(rData, rSize, nCoord);
SendToResultCombiner(rData, rSize, myCoord);
...
//WaitRecvAll(neighborCoords);
while(WaitRecvAny(&neighborCoords, &numRemain)){
    while(GetNext(&data, &dataSize)){
        //process for dependent part
        //and send results
        
    }
    //process for non-dependent part
    //and send results
}
```

Figure 7. Communications and within the kernel.
domain specific expertise. `SubDepDataRequest()` returns immediately. IDEA’s repository mechanism will take over and carry out the communications while the kernel continues to process the part that does not require dependent data. Once some results are generated, the kernel can either forward them to other neighbor jobs by calling `SendToNeighbor()` or send them to the result node for combination by calling `SendToResultCombiner()`, as demonstrated in figure 7 on the previous page. The repository send threads will be woken up to execute the requests. The kernel then checks whether the previously requested data is ready or not. Depending upon the application’s requirement, the user can choose to continue the job processing as soon as part of the requested data is ready or wait until all of them are gathered. These can be done by calling `WaitRecvAny()` and `WaitRecvAll()` respectively. However, the `WaitRecvAny()` function allows quick continuation of the kernel program while the rest of dependent data is transferred and thus could potentially improve the application performance. Both functions take a list of job coordinates of the neighbors. In addition, `WaitRecvAny()` also takes a pointer to an integer to keep track of how many neighbors are left. The neighbor list is updated every time the `WaitRecvAny()` function returns. This process is repeated until all the dependent data are received and processed.

E. Result Combination Function

IDEA allows the programmer to define their own result combination function customized to specific application. Similar to the kernel, it will be invoked within the main function on the result combiner node and fed with data received from the compute nodes. The result combiner function must produce a single final result. However, this could be done through either a reduction or gather, or some combination of both.

Similar to other parallel programming libraries, IDEA provides a general reduction function called `IDEA_AllReduce()`. Not all applications require this functionality in their kernels, but it is essential for some. For example, it is not needed in ray casting but is required in Jacobi iteration to get the global difference value for proper termination. Implementation of reduction on a cluster is well understood. In future, we may delegate this to MPI, at least for 1D data. It is not clear whether MPI is suitable for more generalized n-dimensional reductions.

IV. APPLICATION EXAMPLES

We use two applications to illustrate how IDEA can be used for automatic parallel computing, volume rendering by ray casting and Jacobi iteration. As mentioned earlier (figure 3), ray casting demonstrates a unidirectional but dynamic dependency pattern while Jacobi iteration has a bidirectional dependency pattern with uniform transfer sizes on each axis.

A. Volume Visualization

Ray casting is a typical technique to render transparent volumes. It requires a back-to-front data processing order along each ray during the blending process. This data dependency imposes cross-node communications in parallel rendering if the rays pass through multiple blocks that are assigned to different compute nodes. Furthermore, its dependency pattern usually changes when the view direction changes. For IDEA to correspondingly partition the data with low communication overhead, the visualization programmer can define the dependency detector function with parameters such as the shape of the data volume to render, the view direction, and the up vector. These three parameters are used to compute the 2D projection of the bounding box of the 3D volume. In addition, the horizontal and vertical sampling rates can also be specified to determine the density of sample rays. With this information, we can determine the number of rays that pass through each face of the dataset and therefore the transfer size along each axis. When these application specific parameters change, the returned dependency descriptor likely also changes.

The kernel function for ray casting follows the general kernel structure as shown in figure 8.

```
void kernelFunc(JobBlock *jblk){
    //calculate rays that intersect with this job block
    ...
    //submit non-blocking requests for blend results (rays)
    //from neighboring job blocks
    int *nbList = GetPrevRaySrcJobs(jblk);
    SubDepDataRequest(jblk, nbList);
    
    //job blocks on the outside of the data volume can
    //process initial intersecting rays here and send
    //the results
    ...
    //WaitRecvAll(nbList);
    while(WaitRecvAny(&nbList,&numNbs))
    {
        //process dependent rays
    }
}
```

Figure 8. Kernel function snippets for ray casting.

A job block contains enough information to identify the rays passing through it. In the kernel for ray casting, the programmer can determine the neighbor jobs to send current or receive previous blending results. If there are any previous results to receive, it is suggested to submit the request first. This will initiate the data transfer while continuing to process the initial rays for jobs on the borders of the data space. Then the processing results can be forwarded to neighboring jobs or the final result combiner. It is possible that the blending operation of rays passing through the current job block needs previous results from multiple neighbors, and nothing prevents us
from processing them one after another. The \texttt{WaitRecvAny()} function is preferable here so as to overlap the blending operation with the transfer of other previous results.

When rays are completely blended, they are sent over to result node for combination. This function must be defined by the programmer to perform the filling operation in the final image plane, as shown in figure 9.

\begin{verbatim}
void CombFunc(char*srcdata, int srcsize, char *result)
{
    //fill final rays in the final image plane
}
\end{verbatim}

Figure 9. Result combination function snippets for ray casting.

B. Jacobi Iteration

Jacobi iteration demonstrates a different dependency pattern and brings special storage challenges. It requires neighbors to exchange updated values of their ghost areas and thus has bidirectional dependency with uniform transfer sizes. Its dependency detector function only needs to compute the face areas based upon the shape of the dataset.

In the kernel function as shown in figure 10, the requests for dependent ghost data are submitted from the second iteration. The non-dependent part of the job block would be its interior part and possibly some of its borders that align with the whole data space borders. The \texttt{WaitRecvAny()} is useful since the computations on different borders are independent. The major difference from the kernel of ray casting is that it needs to save the new values as the job blocks for next time step. As mentioned in section III.B.2), there are different ways to handle this issue depending upon the data size, available memory and local disk capacity. However, the application programmer only needs to call function \texttt{SaveJob().} The execution details are hidden in the job repository mechanism and the job reloading is also automatically carried out. As for performance, the more disk or remote accesses needed in these processes, the more the application performance would be degraded. IDEA's job repository for spatial data uses \textit{chunked I/O} [10] which alleviates the latency problems.

Jacobi iteration does not require the application programmer to define a result combination function. Instead, as shown in figure 11, when one round of iteration is completed, they can call \texttt{IDEA\_AllReduce()} to get the global max difference value across all compute nodes. This will repeat until the data reaches a stabilized state.

\begin{verbatim}
//iterate once
IDEA\_AllReduce(diff, &gdiff, MAX);
while(!isStabilized())
{
    //iterate one more time
    IDEA\_AllReduce(diff, &gdiff, MAX);
}
\end{verbatim}

Figure 11. Code snippet for reduction in Jacobi iteration.

It is clear that in both applications, IDEA provides the same skeleton for the kernel function. The user needs to define application specific dependency detector functions and feed the returned dependency descriptor to IDEA. IDEA can automatically choose the right caching, job partitioning without exposing the details to the user. Communications are initiated by specifying the relative coordinates of current job's neighbors and carried out invisibly through a repository mechanism. As demonstrated in the following section, these are realized with reasonable computational and synchronization overhead.

V. PERFORMANCE COMPARISON WITH MPI

We did a series of tests to verify that IDEA will choose the right data partitioning while incurring reasonable computational overhead. We evaluated the performance of Jacobi iteration using IDEA and MPI.

Many datasets are stored in a linear face-by-face, row-by-row fashion. Usually, implementers of parallel Jacobi iteration partition a dataset along the plane axis (axis 0) because it is simpler and can avoid the extra buffer allocation and the related data copy costs during ghost data exchanges. However, programmers may not fully realize the effect of the dataset shape on the communication overhead. Most of the time, cross-node
Figure 12. Three datasets of the same size but with different shapes.

Figure 13. Naive partitioning with MPI does not always bring the best performance. The shapes correspond to those in figure 12.

Figure 14. Comparisons of Jacobi iteration performances using IDEA and MPI on different data block shapes. SA-0 — Split along Axis 0, and so on. Times are measured in seconds.
communications are much more expensive than memory access costs [12]. As verified in our following experimental results, splitting the data to avoid higher data copy costs is not always the best choice to parallelize Jacobi iteration. This conclusion applies to many other parallel computations.

We carried out two sets of tests on Orion, a small cluster at The University of Mississippi. One set of tests explores how MPI implementations behave with different data partitioning. The other set compares IDEA performance with MPI. In all cases, we used three 3D datasets of the same size but different shapes as shown in figure 12. We assume the job blocks are already distributed to compute nodes when the timing starts. The times are in seconds.

We can see from figure 13 that the data partitioning that seems natural to the Jacobi iteration programmer may not turn out to be the proper one for best performance. Figure 13a shows that partitioning along axis 0 works best only when the dataset is of shape 4098*66*66. Otherwise (figure 13b, 13c), it is wise to choose a different split axis (axis 1 for shape 66*4098*66 and axis 2 for shape 66*66*4098). In all three cases, the partitioning that incurs the smallest communication overhead performs best. However, writing code that always chooses the best partitioning regardless of dataset shape and storage layout requires considerable programming effort and specialized knowledge. Addressing this difficulty is a major contribution of the IDEA system.

In the leftmost case of figure 14, both MPI and IDEA do the same partitioning along axis 0. It is not surprising that MPI implementation outperforms IDEA since IDEA is still in development while MPI is well-developed and highly optimized for performance. However, in the other two cases, IDEA outperforms MPI even with the implementation overhead. The reason is IDEA can always pick the best data partitioning automatically. The programmer only writes the required functions once and it will work for all datasets of different shapes, even with different storage layouts. We expect even more significant performance difference when IDEA’s implementation is optimized.

Lastly, figure 15 shows that IDEA’s performance scales well as the number of processors is increased.

VI. CONCLUSIONS AND FUTURE WORK

The difficulties of parallel programming often distract the scientific researchers from their science, especially when dealing with large spatial datasets. IDEA provides an application level API to help automatically parallelize such computations within a cluster environment. Based upon the application dependency pattern, it can produce a partitioning strategy that minimizes the communication overhead. This process can be done dynamically as application parameters change using a dependency detector function. Furthermore, IDEA provides a data centric view of parallelization. Supported by the communication repository mechanism, it allows the user to address communications to neighboring blocks of data, rather than processes or nodes. Such addressing can even be done before the neighbors have been assigned an active process. This is a key ability when handling datasets that are larger than the aggregate memory capacity of the cluster, since the dataset must be processed in a piecewise fashion. IDEA’s job repository mechanism accommodates applications that produce multiple time steps, by transparently providing results from the previous time step as job blocks for the next time step.

We demonstrated how to use IDEA for parallel computing using two example applications: ray casting and Jacobi iteration. These two applications have different dependency patterns and repository requirements. However, their IDEA implementations are broadly similar, demonstrating the expressiveness of the API. With the complex data partitioning and communication delegated to IDEA, the scientist programmer’s effort in parallelization is minimized.

We experimentally verified that IDEA can automatically choose the right partitioning regardless of the volume shapes. Despite the presence of some implementation specific overhead, IDEA is able to demonstrate significant performance gains over naive MPI
code whenever a naive splitting along the plane axis incurs extra communication costs. Although the current results show the effectiveness of our approach, we expect that future improvements to our implementation will make IDEA's advantages even more compelling.

We see several other directions for future research. First, spatial replica selection would extend IDEA's reach into distributed computing environments. Gutman, et al [13, 16] have addressed how to efficiently pick the replicas that intersect with the ROI using an R-Tree. However, we would like to factor in the actual I/O performance during the selection, especially by taking into account the data storage layout and the communication costs. Second, although we believe IDEA would still perform well with applications that demonstrate a bidirectional non-uniform dependency pattern as long as the application is data intensive, it is still valuable to test how IDEA would perform with various partitioning methods. Third, we would like to extend the job repository to handle extremely large datasets using an external or distributed storage system for spatial data. Lastly, we would also like to compare MPI-IO with IDEA's caching technique for handling spatial data queries on large datasets. Clearly, the IDEA system presents many opportunities for Computer Science research as we work toward a tool that assists scientists and engineers in the pursuit of their own scientific endeavors.

ACKNOWLEDGEMENT

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

REFERENCES