Computers and Geotechnics 89 (2017) 168–178

Contents lists available at ScienceDirect

Computers and Geotechnics

journal homepage: www.elsevier.com/locate/compgeo

Research Paper

Parallel and scalable block system generation

Michael Gardner [a,*], John Kolb [b], Nicholas Sitar [a]

[a] University of California, Department of Civil and Environmental Engineering, Berkeley, CA 94720, United States
[b] University of California, Computer Science Division, Berkeley, CA 94720, United States

ARTICLE INFO

Article history:
Received 27 October 2016
Received in revised form 22 March 2017
Accepted 1 May 2017

Keywords:
Block generation
Fractured rock mass
Parallel computing
Cloud Computing
Open-source software
Linear programming
Apache Spark

ABSTRACT

Generating a realistic representation of a fractured rock mass is a first step in many different analyses. Field observations need to be translated into a 3-D model that will serve as the input for these analyses. The block systems can contain hundreds of thousands to millions of blocks of varying sizes and shapes; generating these large models is very computationally expensive and requires significant computing resources.

By taking advantage of the advances made in big data analytics and Cloud Computing, we have a developed an open-source program—`SparkRocks`—that generates block systems in parallel. The application runs on Apache Spark which enables it to run locally, on a compute cluster or the Cloud. The block generation is based on a subdivision and linear programming optimization as introduced by Boon et al. (2015). `SparkRocks` automatically maintains load balance among parallel processes and can be scaled up on the Cloud without having to make any changes to the underlying implementation, enabling it to generate real-world scale block systems containing millions of blocks in minutes.

1. Introduction

Generating a realistic three dimensional model of a fractured rock mass is the first step in many analyses. Discrete block based methods such as discontinuous deformation analysis (DDA) [1] and the distinct element method (DEM) [2] require a full geometric description of the particles in their initial configuration as a starting point for the computations. Identifying removable blocks in a larger rock mass also requires a complete representation of the orientation of the blocks and discontinuities within the rock mass—as blocks are removed, the stability of newly exposed blocks must also be considered. Similarly, analysis of seepage through fractured rock relies on a complete description of the fractures and how they are connected within the rock.

Fundamentally, this is not a new topic and many researchers have developed algorithms to address this problem. Warburton [3,4] presents a methodology for generating a blocky rock mass based on sequential introduction of discontinuities and stores the generated blocks using a three-level data structure (vertices, edges and faces). Heliot [5] proposes a scheme for generating a blocky rock mass that additionally deals with non-convex blocks by representing a rock block as an assemblage of convex blocks. In Heliot's approach, the blocks are stored using a two-level data structure (vertices and faces). Ikegawa and Hudson [6] developed the so-called directed body concept in which all the discontinuities are introduced simultaneously. The blocky mass is then systematically extracted from the vertices, edges and faces. Additionally, several researchers [7,8] have developed algorithms based on principles from combinational topology. These techniques are able to deal with complex geometry, but require a significant amount of "bookkeeping" when implemented. Recently, Boon et al. [9] presented a block cutting algorithm that is based entirely on linear programming. Instead of explicitly calculating the vertices where the discontinuities intersect, the problem is cast as a linear programming optimization. This makes it possible to represent the rock blocks by a single-level data structure since only information about the faces is needed. The simplicity and the efficiency of the method makes it an attractive candidate for large-scale computations. The algorithm itself is entirely decoupled—once a block has been subdivided into two new blocks, the further subdivision of these new blocks can proceed independently. Consequently, this algorithm is naturally parallel and multiple cuts can be made simultaneously without the need to share information among processes.

2. Apache Spark

The subdivision of the rock mass is an iterative process on the same set of data, making the parallel, open-source framework

* Corresponding author.
E-mail address: mhgardner@berkeley.edu (M. Gardner).

http://dx.doi.org/10.1016/j.compgeo.2017.05.001
0266-352X/

Apache Spark [10] an ideal platform for a blocky rock mass generator using a subdivision-type approach. Spark can run on any platform ranging from laptops and personal workstations with multicore processors to Cloud based computing platforms, such as Amazon Elastic Cloud Compute (EC2). This scalability in the power of the computing environment without having to make any changes to the application code allows for the analysis of extremely large problems requiring large amounts of memory and computing power.

The fundamental abstraction in Spark is Resilient Distributed Datasets (RDDs) [10] that allow it to keep large data sets in memory and perform computations in a fault tolerant manner. By keeping the dataset in memory, Spark is able to do iterative transformations on the data extremely quickly since it avoids writing to disk. Fault tolerance is achieved by tracking the lineage of RDDs—all operations applied to the RDD are represented through a lineage graph. When a new operation is applied to the RDD, a new link is added to the graph. Additionally, RDDs are evaluated "lazily": only when a result is requested does Spark execute the transformations described by the lineage graph to actually materialize the current RDD. In this way, if a process unexpectedly fails the current state can be quickly reconstructed from the lineage contained in the graph.

For large scale problems, Spark clusters can be deployed on EC2 using Amazon's Elastic MapReduce (EMR) framework, which automatically allocates and configures a cluster of EC2 instances to execute Spark tasks submitted by the user. Amazon EC2 falls under the greater umbrella of Cloud Computing—applications delivered as services over the Internet and the associated software and hardware that provide those services [11]. Running large scale computations on the Cloud offers users several advantages. First, resources can be scaled on demand to meet the computing requirements of the problem at hand. Second, it is no longer necessary to invest large amounts of capital in computational hardware and the associated management and maintenance. Lastly, usage can be scaled up or down as needed so users only pay for what they use and only use what they need. This makes it possible for anyone to run large scale computations since it is no longer necessary to physically own a computer cluster. Hence, Cloud Computing essentially opens the door to High Performance Computing (HPC) for anyone willing to step through it.

2.1. SparkRocks

By taking advantage of Spark's ability to run on any computer system and the scalability of Cloud Computing, we developed a parallel block cutting program, `SparkRocks`[1], that is capable of generating large numbers of blocks very quickly. The code is open source and the necessary inputs to generate a fractured rock mass are based on parameters that are obtained from field observations, allowing users to quickly translate field measurements into a three-dimensional model.

The program was tested on different systems—a laptop, desktop workstation, and Amazon EC2—to illustrate its ability run on different platforms and to verify its scalability. Results show that we can generate approximately 8 million blocks in roughly 9 minutes.

3. Block cutting algorithm

The block cutting algorithm uses a sequential subdivision approach based on linear programming optimization introduced by Boon et al. [9]. Each discontinuity is introduced individually and checked for intersection. If it intersects the block, two new blocks are generated. The process continues sequentially until all discontinuities have been introduced, yielding a representation of the fractured rock mass. Many block cutting algorithms require a significant amount of bookkeeping in terms of vertices, edges, faces and how all of these elements are connected. From an implementation perspective, this can be extremely tedious and may not be as robust in terms of floating point error. The linear programming optimization approach introduced by Boon et al. greatly simplifies how block cutting is implemented and how each block is represented in terms of data structure. We give only a brief overview of this rock cutting algorithm since the details are presented in [9].

The orientations of joints in a fractured rock mass are described by strike and dip, as shown in Fig. 1. The block cutting algorithm uses the normal vector of the plane containing the joint and the distance of that plane to some origin. The strike and dip define the normal vector of the joint. The distance of the joint plane from the origin is determined by projecting a vector connecting the origin to a point in the joint plane onto the normal vector. The global $+x$, $+y$ and $+z$ axes are oriented North, West and upward.

Using only the joint normal and distance, it is possible to completely describe a polyhedral block, as shown in Fig. 2. A block bounded by N planes is described by the following equation:

$$a_i x + b_i y + c_i z \leqslant d_i, \quad i = 1, \ldots, N \tag{1}$$

The coefficients (a_i, b_i, c_i) represent the normal vector to the ith plane bounding the block and d_i is the distance of that plane from some local origin. In vector notation this becomes:

$$\boldsymbol{a}_i^T \boldsymbol{x} - d_i \leqslant 0, \quad i = 1, \ldots, N \tag{2}$$

In order to subdivide a block, it is necessary to establish whether the block is intersected by the discontinuity being considered. The novelty in the algorithm presented in [9] is recasting this problem as a linear program:

$$\begin{aligned} &\text{minimize} \quad s \\ &\boldsymbol{a}_i^T \boldsymbol{x} - d_i \leqslant s, \quad i = 1, \ldots, N \\ &\boldsymbol{a}_{\text{new}}^T \boldsymbol{x} - d_{\text{new}} = 0 \end{aligned} \tag{3}$$

Here N represents the number of planes that define the block and the discontinuity being considered is represented by the equality. If $s < 0$, there is an intersection and the parent block is split into two child blocks. The child blocks inherit all the parent block's planes as well as the intersecting discontinuity with opposite signs for the discontinuity normal vector for each child block.

As the subdivision continues, some of the discontinuities may become geometrically redundant. It is not necessary to remove these redundancies after each intersection check; instead they can be removed at a later time as discussed in Section 4.2.3. Again, this can be done by solving a linear program:

$$\begin{aligned} &\text{maximize} \quad \boldsymbol{c}^T \boldsymbol{x} \\ &\boldsymbol{a}_i^T \boldsymbol{x} \leqslant d_i, \quad i = 1, \ldots, N \end{aligned} \tag{4}$$

Here, $\boldsymbol{c}$ is the normal vector specific to the discontinuity being checked for redundancy and with associated distance d. If $| \boldsymbol{c}^T \boldsymbol{x} - d | < \varepsilon$ the discontinuity is not redundant, where ε represents a numerical tolerance close to zero.

Additionally, we take advantage of two major optimizations to the block cutting process that are presented in [9]. The first optimization draws on an idea common to contact detection in particle methods (for example, see [12]): the complex geometry of the polyhedral blocks is enclosed in a simpler shape, in this case a bounding sphere. This enables a simple and fast check for intersection to determine if a more thorough but computationally expensive check is necessary. The second optimization is to control the size and aspect ratio of the blocks that are generated during the

[1] Available at https://github.com/cb-geo/spark-rocks.

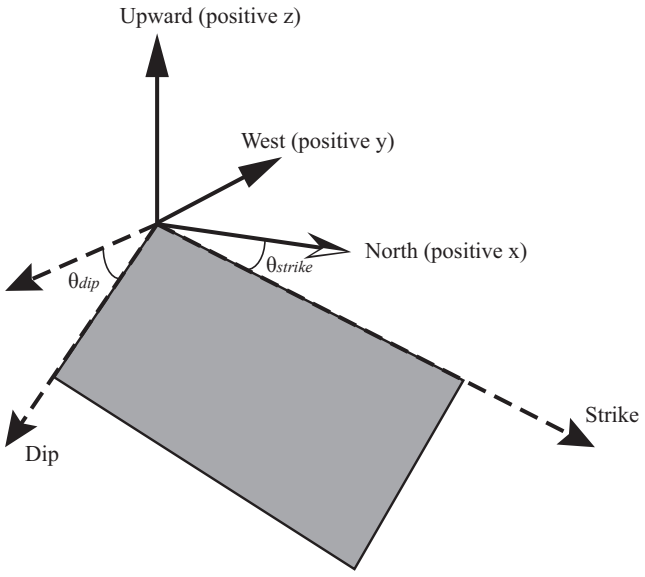


Fig. 1. Description of strike and dip, illustrating relationship to global coordinates.

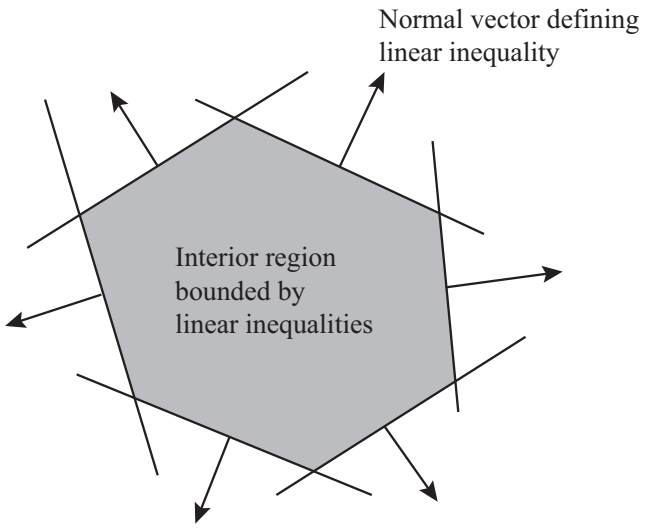


Fig. 2. Volume bounded by set of inequalities. The shaded region represents the block that is defined by this set of linear inequalities (after [9]).

slicing process. Unrealistically small or thin blocks can contaminate the generated blocks, leading to undesirable side effects in the subsequent analyses that use the fractured rock mass as an input. Both of these optimizations involve the construction and solution of linear programs, the details of which are fully described in [9].

4. Implementation on Spark

4.1. Translation into a parallel problem

As already stated, the serial approach to cutting blocks can readily be modified to run in parallel. Once two child blocks are cut from their parent by a particular joint, they can be treated independently for the remainder of the block cutting process. In other words, one child's intersection with subsequently introduced joints and future subdivisions into further child blocks has no effect on the subdivision matters of its sibling. This gives rise to a tree structure of relationships between blocks, depicted in Fig. 3. Therefore, while processing each joint in a rock mass, the joint's intersection with each block cut so far can be computed independently. We take advantage of this property to construct and solve the linear programs described in the previous section in parallel and independently on each processor. It is important to note that the tree structure described is not unique to [9], and any other block cutting algorithm with this property would lend itself to parallelization.

In practice, it is necessary to effectively distribute work to the multiple central processing unit (CPU) cores and nodes that are available. By expressing the current set of blocks cut from a rock mass as an RDD in the Spark context, it is possible to seamlessly perform parallel operations on these blocks and scale the associated computation to different quantities of CPU cores and nodes without changing any of the underlying rock slicing logic. To split up responsibilities for all of the required rock slicing, we select a small subset of joints to break the overall rock volume into a group of initial blocks of roughly equal volume. Each block, and all of that block's descendants, are then processed independently as illustrated from a high level in Fig. 4.

4.2. Implementation

A straightforward translation of the method described in [9] into code does not lead to an efficient parallel rock slicing implementation. This section describes important features and refinements necessary to achieve good performance when dealing with problems at a large scale.

4.2.1. Load balance

In terms of performance, maintaining load balance among parallel processes as well as minimizing communication between them is of primary concern. A solution that achieves good load balance may not necessarily feature a reasonable level of communication overhead, while another solution may have low communication overhead but poor balancing of work among parallel processes. It is therefore necessary to find a strategy that achieves a balance between these two demands by taking into account the characteristics of the underlying framework on which computations are done.

In our case, the initial thought was to focus on load balance. Artificial joints are introduced to divide the rock volume into equal pieces so that near-perfect load balance is achieved between parallel processes. However, in order to remove the artificial joints at the end of the slicing process, all blocks sharing an artificial joint must be recombined in order to remove that artificial joint from the final rock mass. This involves an exchange of blocks over the network among all nodes, which induces a high amount of communication overhead that slows down the overall rock slicing process and offsets the gains achieved through load balancing. Spark does not have the necessary communication primitives to directly manage communication which, in this case, leads to excessive communication to the point that the majority of the computation time is spent on the removal of artificial joints rather than the introduction of real joints.

Given these constraints, we sacrifice some of the load balance in order to minimize communication. Since the block cutting process is entirely decoupled and can be done independently, once each node receives a portion of the initial rock volume it can complete the cutting process without communicating with other nodes. We exploit this by selecting joints from the input joint sets that divide the initial rock volume into approximately equal volumes. The blocks generated by cutting the initial rock volume with the selected joints are used to seed the initial RDD. This idea is illus-

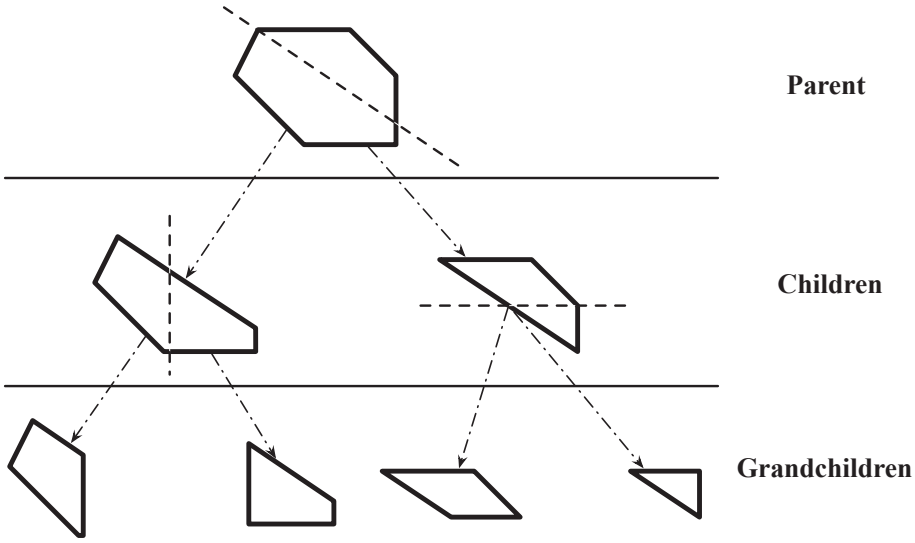


Fig. 3. Three generations of blocks cut from a common ancestor.

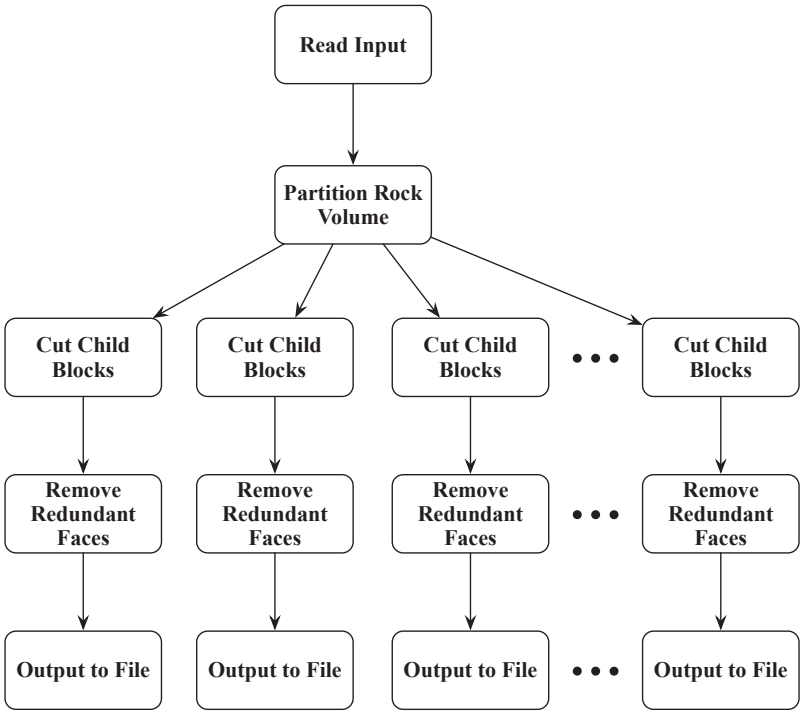


Fig. 4. Steps in the parallel rock slicing process.

trated in Fig. 5. The different colors represent which portions of the rock mass were processed by which node. In this example four nodes were used. Each node processed one portion of approximately equal volume and the last, much smaller volume was processed by the node that completed its subdivision first.

Since the joints used to generate the seed blocks are real joints taken from the input joint sets, it is not possible to achieve perfect load balance. In some instances it may not be possible to find adequate seed blocks from a single joint set, so it becomes necessary to select joints from multiple joint sets. This complicates finding the exact same number of seed blocks as the number of nodes in the analysis. In most cases, more seed blocks are generated than what is requested. This is especially the case when selecting joints from multiple joint sets. Since Spark performs dynamic load balancing internally, having more seed blocks provides flexibility in managing and maintaining load balance.

4.2.2. Lineage

Spark internally tracks the transformations applied to each RDD in a lineage graph. This allows it to defer the materialization of an RDD until its contents are actually needed by traversing a path from a previously materialized RDD to the required RDD, applying the necessary transformations along the way. However, Spark fails when lineage chains in this graph grow too long. Specifically, this occurred in the initial version of the rock slicing code depicted in Fig. 6a, which iterated through each joint in the rock mass, checked for intersections with any members of the current block RDD, and produced a new RDD in which any blocks intersecting the joint

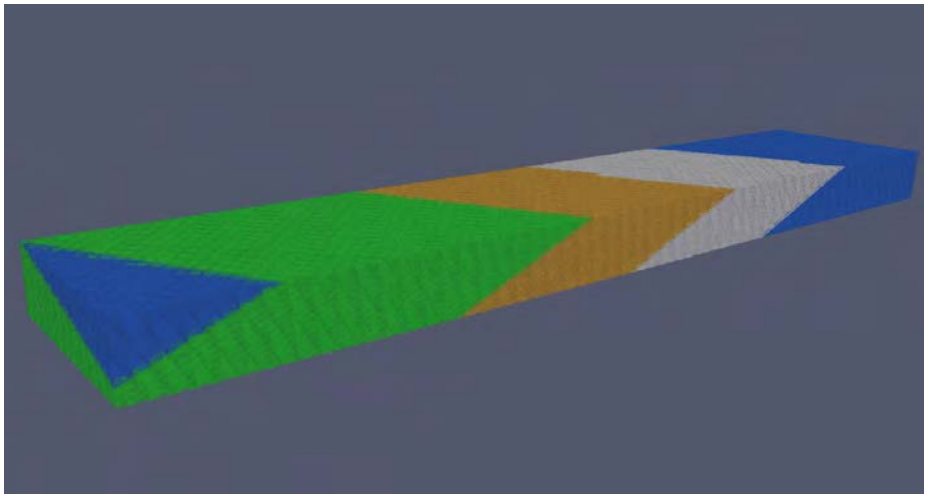
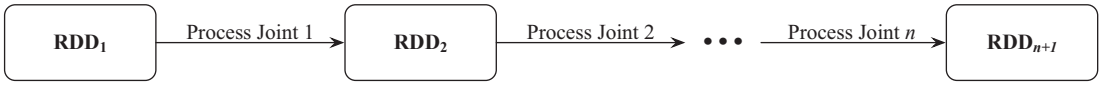
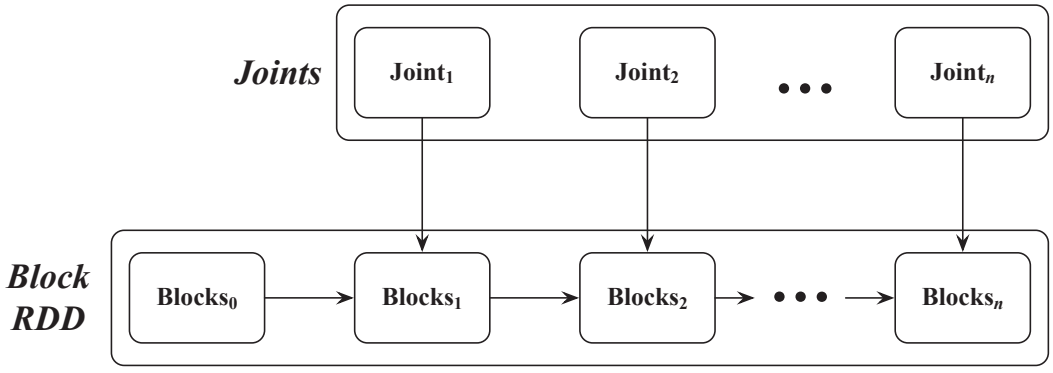


Fig. 5. Example of load balance. Here, at least 4 partitions were requested. The different colors represent which pieces were processed by which node. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)



(a) Iteratively Creating RDDs



(b) Using `fold` to Process all Joints

Fig. 6. Lineage chains in Spark.

were cut into two child blocks. Thus, a new RDD was created for each joint, and a lineage chain formed with a length proportional to the total number joints, which becomes unwieldy when the number of joint is large.

We resolved this issue by taking advantage of Spark's `fold` primitive, as shown in Fig. 6b. This operation individually examines each joint and creates an intermediate collection of blocks that are the cumulative result of processing all joints seen so far. The operation repeats until all joints have been processed, and only the final result is retained. In more detail, `fold` starts with an initial element (the seed blocks used for load balancing), and element i is produced by applying an operation to element $i-1$ and the next joint, which in this case is an intersection check and the necessary slicing of parent blocks into child blocks. Spark treats a `fold` as a single transformation and therefore a single link in the lineage chain. This replaces the original lineage chain, with a length proportional to the number of joints, with a lineage chain of length one.

4.2.3. Redundant faces

Two child blocks that are cut from a parent inherit all of the parent's faces as well as a face along the discontinuity that separates the blocks from each other. Many of these faces are geometrically redundant and can removed without compromising the integrity of the block. Boon et al. [9] advocate deferring the removal of geometrically redundant faces until all blocks have been cut. However, retaining a large number of redundant faces in blocks during the slicing process increases the size of the linear programs that must

be solved when checking for intersection between blocks and joints. The increase in linear program size leads to degraded performance as more child blocks are cut from the initial rock volume.

To avoid this deterioration in performance, we periodically remove redundant faces during the block cutting process. The frequency at which this removal is performed represents a tradeoff. Removing redundant faces is somewhat expensive because it involves solving a linear program for each face of a block, so it cannot be done too often. We found that eliminating redundant faces for every 200 joints processed keeps the linear programs reasonably sized without adding excessive overhead from geometric redundancy checking.

An important consideration in this scheme is the fact that many, if not most, blocks will not change when additional joints are introduced. Therefore, examining these blocks is unnecessary work and a source of significant inefficiency. To address this problem, we index all joints by the order in which they are processed, i.e. the first joint checked for intersection against all blocks is assigned index 1, the second joint checked is assigned index 2, and so on. Each block is augmented to track its *generation* – the index of the joint that cut the block from its parent, and blocks that were not cut from their parents by any of the 200 most recently introduced joints are skipped.

5. Performance

Performance evaluation was done on Amazon EC2 with Amazon Elastic MapReduce (EMR). EMR can seamlessly configure a Spark cluster on EC2, which makes deploying applications written for Spark easy to run. All testing was done on compute-optimized instance types, which are specifically designed for compute-intensive HPC applications. An instance in the context of EC2 is a single node. For example, a four node cluster comprises four instances. The different instance types give the user a choice in the hardware configuration of the nodes. Testing was done with `c3.xlarge`, `c3.4xlarge` and `c3.8xlarge` instances. These three instance types span a range of computational power and hardware configurations, shown in Table 1.

In order to maintain control over the exact number of blocks that are generated, the input rock volume consists of a rectangular prism and input joints are defined such that they divide the rock volume into a specific number of cubes. This allows us to easily interpret how well `SparkRocks` scales.

5.1. Partitioning

Load balance among the different nodes is maintained by partitioning the initial rock mass into approximately equal volumes, as described in Section 4.2.1. The number of partitions—seed blocks—used in block cutting has a significant impact on the efficiency. Figs. 7 and 8 show the total elapsed times for the three different instance types with 4,000 blocks and 32,000 blocks per node, respectively. Instances with 64,000 blocks per node showed similar trends. The most important result is that efficiency is highly sensitive to the number of initial partitions used to seed the RDD. Seeding the RDD with more partitions gives Spark more freedom in managing parallel execution, as indicated earlier. This trend is observed independent of the number of nodes. Each node can execute computations in parallel locally; however, if it receives only a single partition, computations will be serial as demonstrated by the runs executed on a single node.

When more than one node is used with too few partitions, Spark cannot effectively share the computational load across all members of the cluster, and some nodes end up doing much more work than others. By seeding the RDD with more partitions initially, each node will receive many seed blocks. This allows Spark to locally exploit parallelism and greatly increase efficiency. This is seen more clearly for tests with more blocks, such as shown in Fig. 8 where execution times for large node counts are slower than smaller node counts for the same number of partitions.

Table 1
Amazon EC2 instance types used in testing.

	`c3.xlarge`	`c3.4xlarge`	`c3.8xlarge`
vCPU	4	16	32
Memory (GB)	7.5	30	60
SSD Storage (GB)	2×40	2×160	2×320

However, when the input data set is too small, larger clusters perform poorer regardless of the number of partitions. For example, when each node only has 4,000 blocks, as shown in Fig. 7, the data set is too small to benefit from the greater computational power of larger clusters. The communication overhead required to manage more nodes dominates total execution time.

Interestingly, in some instances, there is a slight increase in execution time for larger partition counts. Apparently, with too many partitions the cost of communication among the nodes begins to outweigh the load balancing benefits, leading to higher execution times. However, as can be seen, great speedup is attainable even if the most optimal partition count is not selected.

5.2. Instance type

Fig. 9 shows the total execution time of the rock slicing process on a four-node cluster for the three EC2 instance types and for three different problem sizes (4,000 blocks per node, 32,000 blocks per node, and 64,000 blocks per node). Clusters of two, eight, and sixteen nodes exhibited similar results. The least powerful instance, `c3.xlarge`, is affected by small initial partition counts far more than the other types. A low partition count prevents all nodes in the cluster from fully participating, which accentuates the disparity in computational power between the `c3.xlarge` and the other instance types. As the partition count increases, the different instance types begin to yield more comparable performance, although `c3.xlarge` clusters still remain noticeably worse than the alternatives. Interestingly, the `c3.4xlarge` and `c3.8xlarge` demonstrate very similar performance characteristics, not just at high partition counts but for all partition counts. This implies that there are diminishing returns to running a well-tuned deployment on more powerful EC2 nodes, and this has important consequences for users seeking to perform rock slicing at large scale on the Cloud. While Amazon's price for a `c3.8xlarge` instance is double that of a `c3.4xlarge` instance, one can use the latter without suffering a compromise in performance.

5.3. Weak scaling

The *weak scaling* of a parallel program is its ability to maintain a constant level of efficiency while increasing the number of nodes involved in its computations. The problem size per node is kept constant, so each node performs the same amount of work as new nodes are added. In the ideal case, the execution time should remain constant as the number of nodes increases. To test the weak scaling capabilities of `SparkRocks`, we performed rock slicing on clusters of increasing size while proportionally increasing the total number of blocks that are sliced, e.g. a cluster with twice as many nodes slices twice as many blocks. The relevant results are included in Fig. 10, presented both in terms of total execution time and scaling efficiency. When processing 4,000 blocks per node, `SparkRocks` demonstrates good weak scaling behavior, although

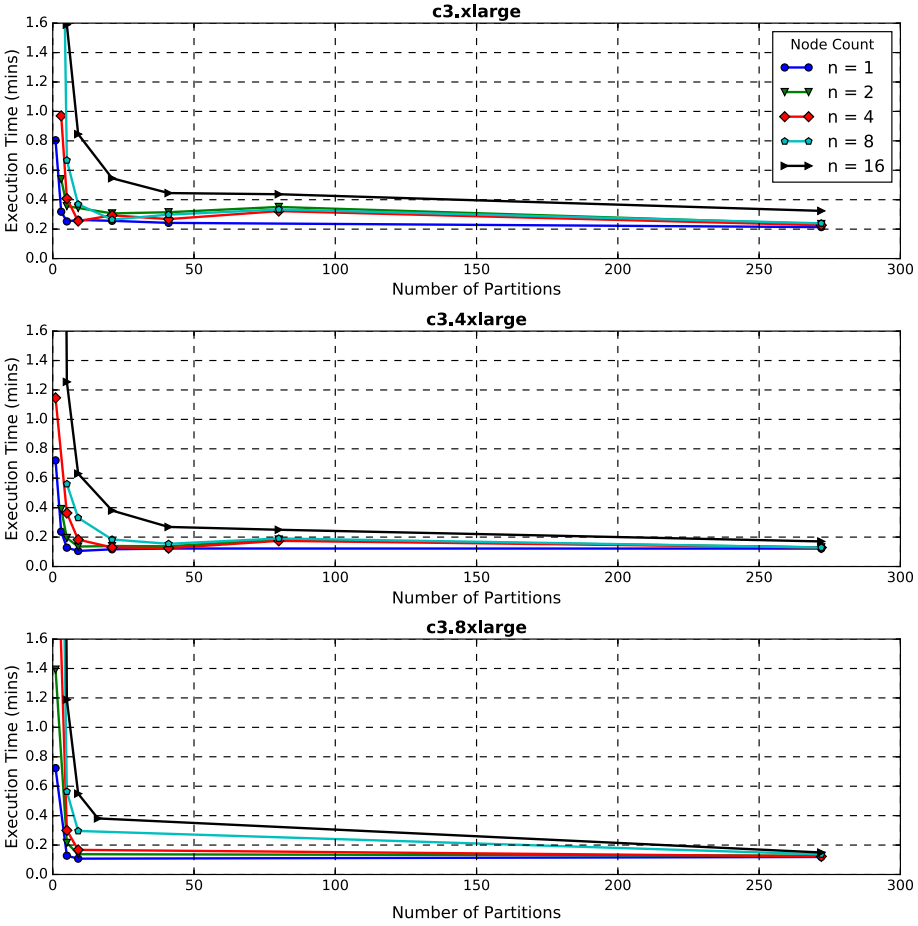


Fig. 7. Execution time vs. Number of initial partitions in the rock volume – 4000 blocks per node.

it can be argued that there are not enough blocks in these experiments to seriously challenge the system's scaling abilities. Total execution time slowly increases as cluster size increases, probably due to the additional communication costs that are introduced by adding more nodes. Again, `c3.4xlarge` clusters achieve performance that is comparable to that of `c3.8xlarge` clusters.

When processing 32,000 and 64,000 blocks per node, the results become more complicated. As with 4,000 blocks per node, there is only a small performance difference between `c3.8xlarge` clusters and `c3.4xlarge` clusters, particularly at larger cluster sizes. Moreover, the performance gap between these two instance types and the `c3.xlarge` clusters also decreases as clusters become larger. Larger cluster sizes are therefore able to mask some of the differences in the capabilities of the underlying hardware. Diminishing scaling returns begin to appear at the larger cluster sizes, where execution time either decreases very slightly or increases. This is probably because communication costs start to become the dominant factor in scaling behavior, as is typical for parallel computing applications.

Fig. 10 also illustrates a pattern in which execution time *decreases* in certain places as cluster size increases, e.g. when moving from two to four nodes. In some sense, this is better than "perfect" scaling where execution time remains constant as the number of nodes increases. This behavior is particularly difficult to analyze because Spark gives the user little control over how their jobs are executed on the underlying cluster of machines. Spark divides a job into a group of tasks, each of which is completed by an *executor* – an abstraction for an independent unit of processing. On Amazon's Elastic MapReduce platform, Spark by default dynamically assigns tasks to executors and increases or decreases the number of executors devoted to a job based on internal heuristics. The improved performance when increasing cluster size is likely due to the fact that with more machines, and therefore with more blocks to partition among these machines, Spark has more freedom to balance load across the cluster and is consequently able to achieve better execution times.

5.4. Strong scaling

Strong scaling is measure of the speedup efficiency when increasing the number of nodes for a fixed problem size—using more nodes should yield shorter execution times. Based on the above mentioned results, strong scaling tests were only performed using `c3.4xlarge` and `c3.8xlarge`. These instance types clearly outperformed `c3.xlarge` and would be reasonable to use when attempting larger analyses. Fig. 11 shows the results of these tests, both in terms of execution time and scaling efficiency. When comparing the execution times, it is clear that both instances perform equally well with more nodes. The largest difference in execution times is seen when using 1 and 2 nodes. This makes sense since the problem size is getting sufficiently large to accentuate the difference in computational power between `c3.4xlarge` and `c3.8xlarge` instances. Using fewer nodes limits parallelism and the more powerful instance wins out. However, considering the difference in cost and the fact that performance is very similar when using 4 nodes or more, `c3.4xlarge` seems to be a better starting choice in terms of instance type.

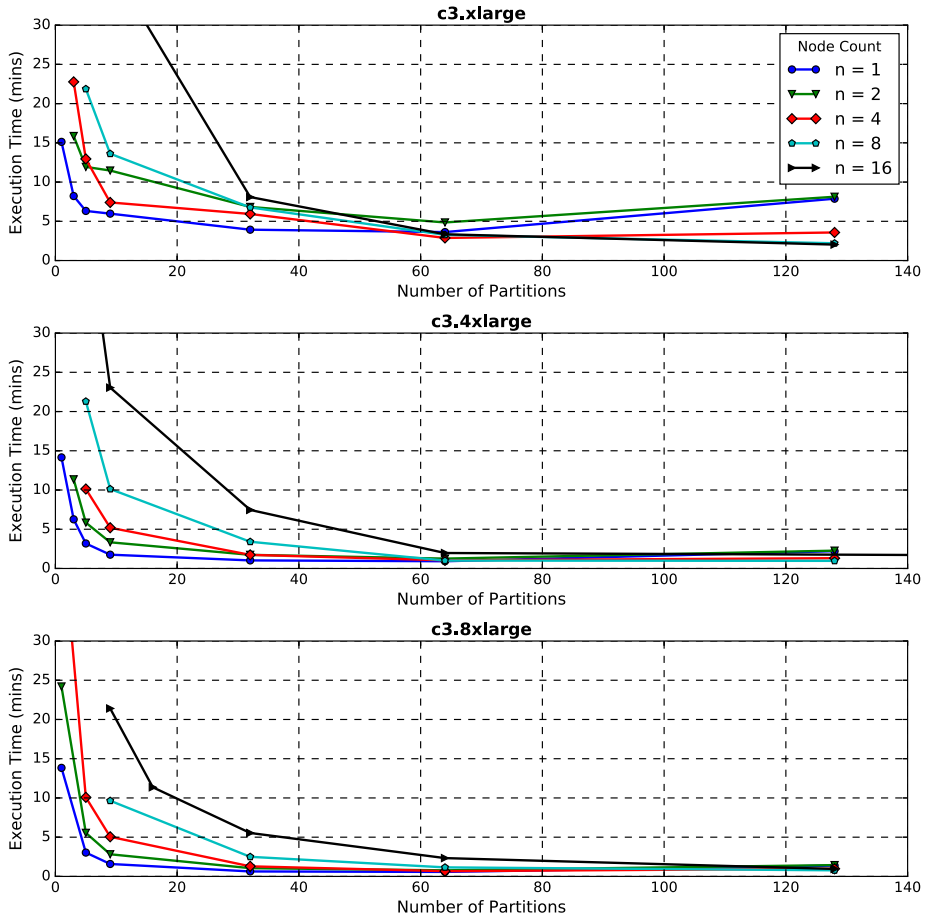


Fig. 8. Execution time vs. Number of initial partitions in the rock volume – 32,000 blocks per node.

In terms of efficiency, both instance types exhibit similar trends though `c3.4xlarge` is somewhat more efficient. If `SparkRocks` had perfect strong scaling, the speedup would be equal to the number of nodes used. Most likely, the increase in communication overhead required to manage the cluster offsets much of the gain in additional resources. Also, the strong scaling tests were performed with the same number of initial partitions, regardless of the cluster size. More in-depth optimization would most likely reveal better strong scaling with respect to varied initial partitioning.

5.5. Practical implications

From a practical perspective, the results in Fig. 11 reveal that, for most cases, using the less expensive `c3.4xlarge` instance type will yield very comparable speedup at a lesser price—even for as many as 8 million blocks. For greater problem sizes, where more memory and computational power are necessary, `c3.8xlarge` is available.

In this context, we compare the results presented here with the computational speed of a single core of a 3.1 GHz Intel Core-2-Duo CPU presented in Boon et al. [9] and `SparkRocks` on both a laptop with an Intel Core i7-4720HQ (2.6 GHz) CPU with 10 GB of memory and a workstation with two Intel Xeon E5-2630 v2 (2.3 GHz) CPUs with 20 GB of memory. Spark was run with 4 cores on the laptop and 12 cores on the workstation. The performance data for EC2 is from an eight-node cluster of `c3.4xlarge` instances, using the best partition count for each problem size. This cluster therefore features a total of 240 GB of memory and 128 vCPUs.

Fig. 12 is a plot of execution time against problem size. It shows that the parallel implementation offers orders of magnitude speedup compared to the serial implementation featured in [9] as the number of available cores and memory increases. In particular, we observe that when we move the execution of `SparkRocks` from a single desktop to an EC2 cluster, running times decrease by about an order of magnitude for problems of the same size, while the cluster can also accommodate much larger problems than the lone server. Running times on the EC2 cluster do not begin to significantly increase until the problem size becomes quite large. If even larger problem sizes were tested, running time on EC2 should scale similarly to the running times seen on the laptop and desktop deployments. Overall, performance on EC2 conforms to our expectations, as the cluster represents about an order of magnitude increase in CPU and memory resources compared to the desktop, and we generally observed that the execution of `SparkRocks` is CPU-bound. While the parallel processes running on different machines within the cluster now have to communicate over the network, our rock slicing algorithm minimizes this communication, keeping overhead small and allowing `SparkRocks` to take nearly full advantage of the additional resources.

Overall, the parallel implementation in `SparkRocks` is capable of generating 8,192,000 blocks in roughly 9 minutes on EC2, while a serial analysis running on a desktop CPU is able to slice only 60,000 blocks in roughly ten minutes [9]. This speed and scalability is made possible by the use of Spark and the abstractions it provides. Expressing the rock slicing process as a series of transformations on a resilient distributed dataset of blocks allows us to spread work and to scale to all of the nodes and CPU cores available. All of

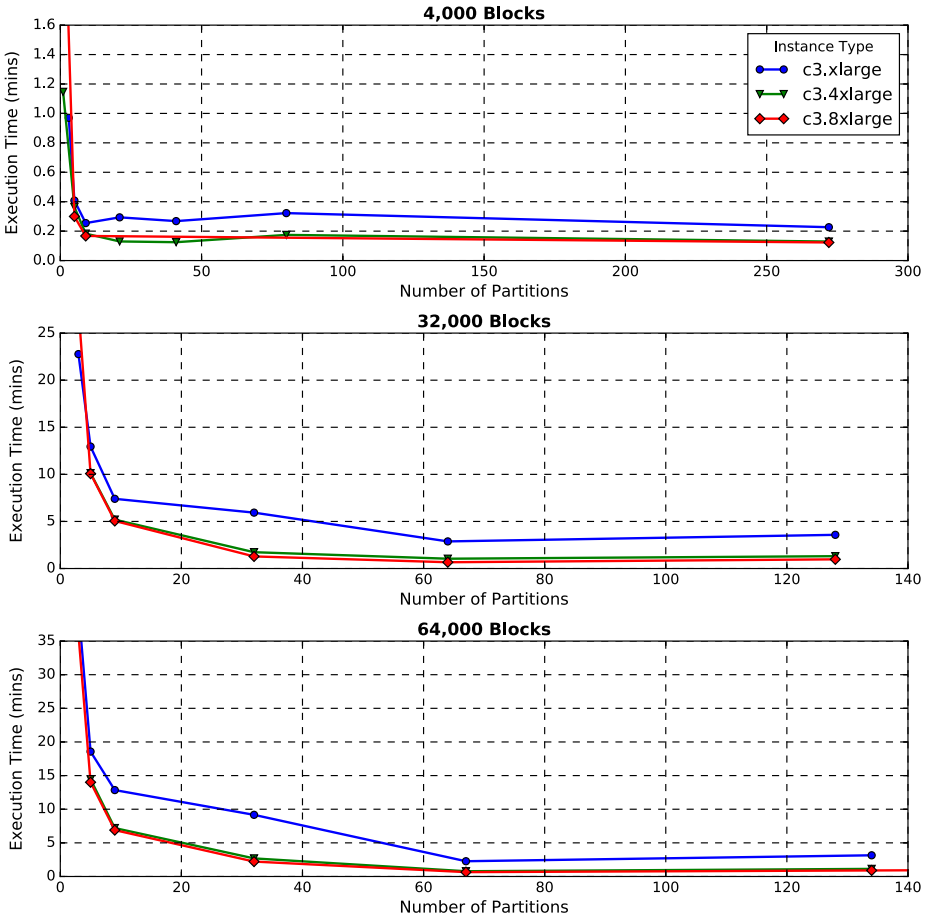


Fig. 9. Execution time vs. Number of initial partitions in the rock volume on a four-node cluster.

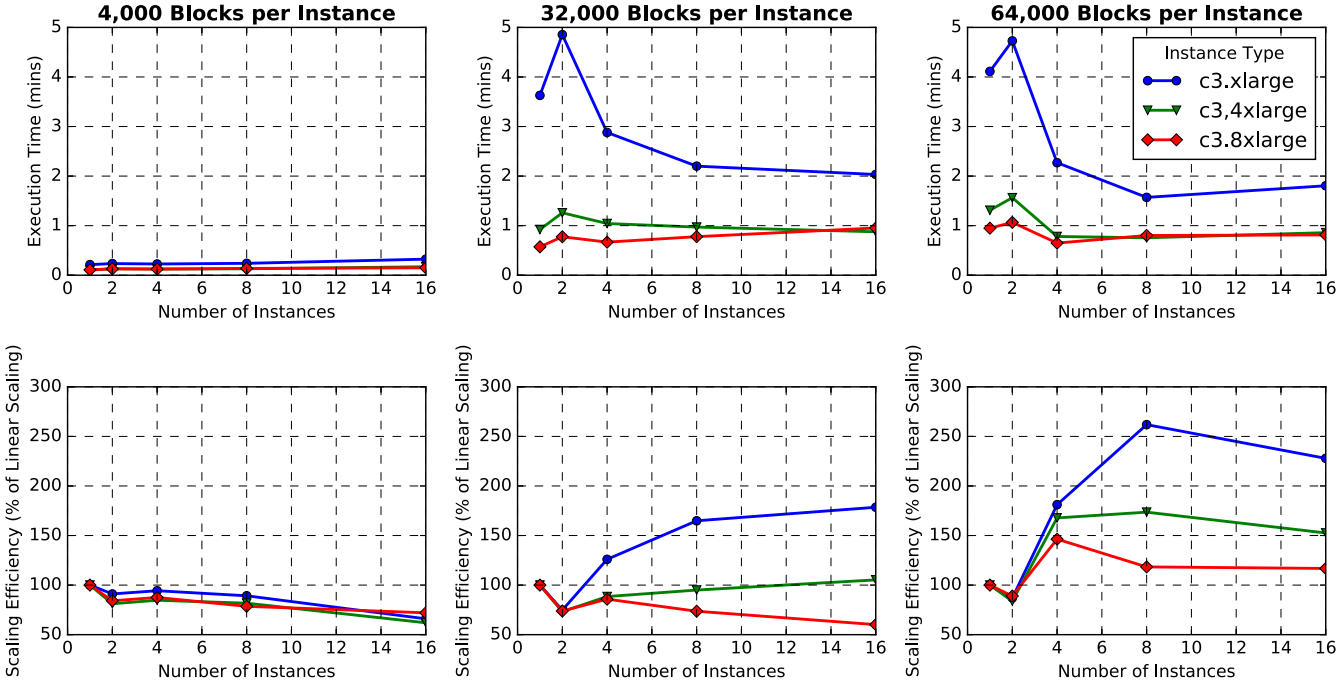


Fig. 10. Execution time and scaling efficiency as cluster size increases. All experiments shown here used 64 partitions for all cluster sizes.

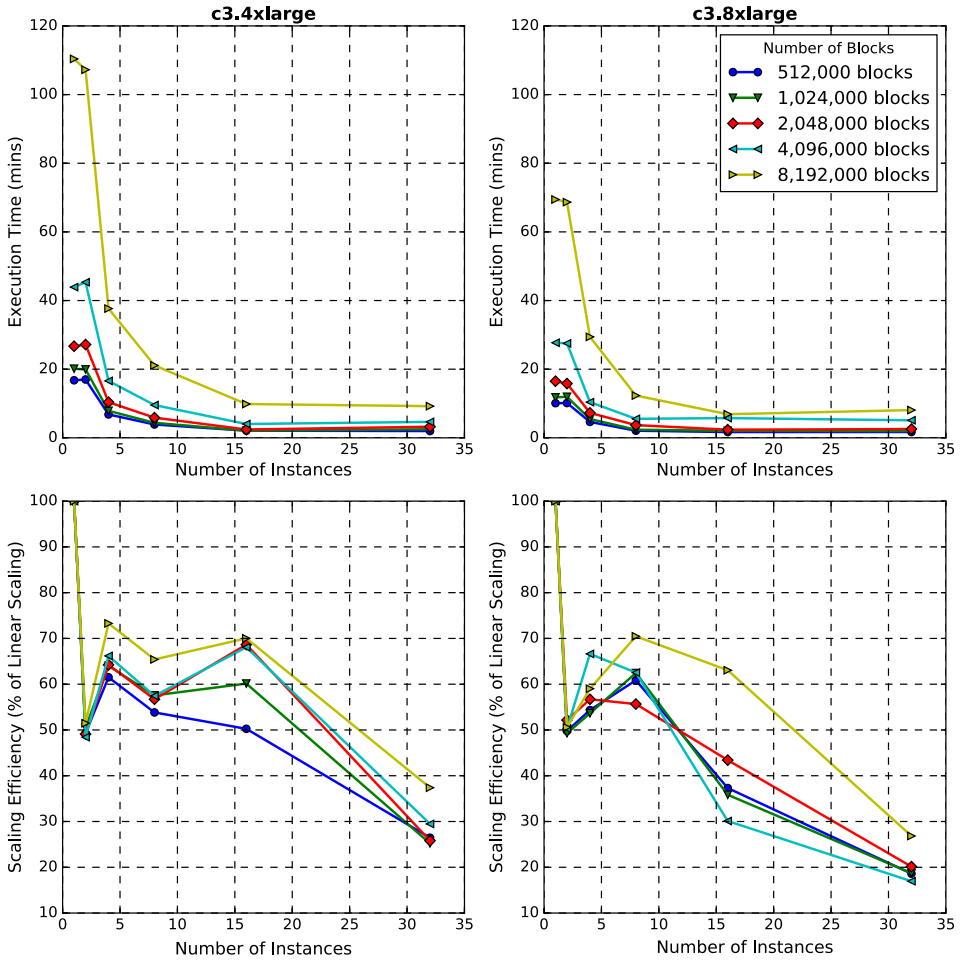


Fig. 11. Execution time and scaling efficiency as cluster size increases – 134 partitions.

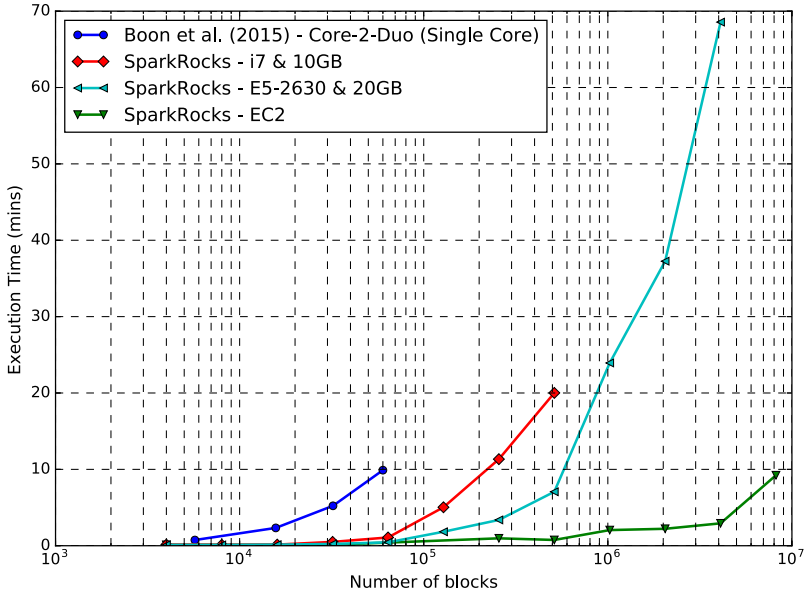


Fig. 12. Execution time vs. Problem size for `SparkRocks` and Boon et. al. (2015) [9]. Note: x-axis is in log scale.

this can be done without changing any of the actual rock slicing code. For the problem sizes typically seen in practice, the `SparkRocks` parallel implementation can generate full block systems in a matter of minutes. Historically, access to the kinds of computing clusters that can provide this level of performance has been prohibitively expensive for many. However, with the relatively recent

advent of Cloud Computing, users can forego provisioning their own clusters and access cloud resources instead; paying only for what they actually use. Thus, the computational resources are no longer a limitation and real-world scale problems are within reach during routine analysis.

6. Conclusion

We developed a parallel, scalable open source application, `SparkRocks`, which runs on Apache Spark [10] to allow fast, parallel block generation. Testing on different systems, ranging from multiprocessor workstations to Amazon EC2, shows that the parallel implementation offers orders of magnitude speed up for the solution of large problems. Moreover, the ability to take advantage of Cloud Computing greatly increases the scale of analyses that can be attempted. Real-world, large-scale block systems comprising millions of blocks can be generated in a matter of minutes. Cloud Computing makes this scale of analysis available to any user since Cloud resources can be rented as-needed, negating the need to maintain a local computing cluster. Users only pay for what they use and only use what they need.

The parallel implementation of the block cutting algorithm in the current version of `SparkRocks` generates a fractured rock mass with persistent joints; however, non-persistent joints are a common occurrence in natural rock. Future work should include a stochastic joint generator that can capture the variation in strike, dip, spacing and persistence of joint sets. While the intersection code currently implemented in `SparkRocks` is able to account for the non-persistence of joints, the code that generates the joint sets can be expanded to produce stochastic realizations such that natural variability in the rock mass can be considered.

With this in mind, `SparkRocks` is entirely open-source and modular. The code can be added to and modified as needed to address the needs of various types of analyses and applications. Though Apache Spark currently does not have the communication primitives necessary to assess the stability and displacement of the generated fractured rock mass over time, the output format can readily be modified to match the required inputs for other software packages with this capability. As more functionality is added to Apache Spark, it may become possible to incorporate the displacement and stability analyses into `SparkRocks`.

Acknowledgments

This research was supported in part by the National Science Foundation (NSF) grant CMMI-1363354 and the Edward G. Cahill and John R. Cahill Endowed Chair funds.

References

[1] Shi G. Discontinuous deformation analysis—a new model for the statics and dynamics of block systems Ph.D. thesis. Berkeley: University of California; 1988.
[2] Cundall PA, Strack ODL. A discrete numerical model for granular assemblies. Gotechnique 1979;29(1):47–65. http://dx.doi.org/10.1680/geot.1979.29.1.47.
[3] Warburton P. Applications of a new computer model for reconstructing blocky rock geometry – analysing single block stability and identifying keystones. In: Proceedings of the 5th international congress on rock mechanics. p. F225–30.
[4] Warburton P. A computer program for reconstructing blocky rock geometry and analyzing single block stability. Comput Geosci 1985;11(6):707–12. http://dx.doi.org/10.1016/0098-3004(85)90013-5.
[5] Heliot D. Generating a blocky rock mass. Int J Rock Mech Min Sci Geomech Abstr 1988;25(3):127–38. http://dx.doi.org/10.1016/0148-9062(88)92295-4.
[6] Ikegawa Y, Hudson J. A novel automatic identification system for three-dimensional multi-block systems. Eng Comput 1992;9(2):169–79. http://dx.doi.org/10.1108/eb023856.
[7] Lin D, Fairhurst C, Starfield A. Geometrical identification of three-dimensional rock block systems using topological techniques. Int J Rock Mech Min Sci Geomech Abstr 1987;24(6):331–8. http://dx.doi.org/10.1016/0148-9062(87)92254-6.
[8] Jing L. Block system construction for three-dimensional discrete element models of fractured rocks. Int J Rock Mech Min Sci 2000;37(4):645–59. http://dx.doi.org/10.1016/S1365-1609(00)00006-X.
[9] Boon C, Houlsby G, Utili S. A new rock slicing method based on linear programming. Comput Geotech 2015;65:12–29. http://dx.doi.org/10.1016/j.compgeo.2014.11.007.
[10] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauly M, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Presented as part of the 9th USENIX symposium on networked systems design and implementation (NSDI 12). San Jose, CA: USENIX; 2012. p. 15–28. URL <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>.
[11] Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, et al. Above the Clouds: A Berkeley View of Cloud Computing; Feb 2009. URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html>.
[12] Munjiza A. The combined finite-discrete element method. John Wiley & Sons, Ltd; 2004. http://dx.doi.org/10.1002/0470020180.