Parallel Datastore System for Parallel Java

by

Omonbek Salaev

A Capstone Project Final Report

Submitted to the faculty of

ROCHESTER INSTITUTE OF TECHNOLOGY

In partial fulfillment of the requirements for the degree of

Master of Computer Science

January 2010

Committee Chair – Professor A. Kaminsky

Committee Reader – Professor M. Kwon

Committee Observer – Professor M. Rege
Abstract

With the recent tremendous advances in processor and memory technology, disk and network Input/Output has become the bottleneck in high-performance computing for many applications [1]. While the development of parallel file systems, which distribute individual files across multiple disks and servers, has helped to decrease the performance gap, most of the existing parallel file systems have drawbacks, such as complexity of the systems, platform limitations and lack of easy-to-use API.

This project aims to explore developing of a flexible, easy-to-use parallel datastore system. The term “datastore” here refers to a general infrastructure capable of storing data.

The primary goal of this project will be to design a parallel datastore system (PDS) and to implement it as an extension of the current I/O context of Professor Kaminsky's Parallel Java API.
# Table of Contents

1. Literature Overview .................................................................................................................. 5
2. Parallel Java ................................................................................................................................. 6
   2.1. Input/Output in PJ .................................................................................................................. 6
   2.2. The Range class ..................................................................................................................... 10
3. Parallel Datastore System .......................................................................................................... 10
   3.1. MatrixFile ............................................................................................................................. 13
      3.1.1. MatrixFileReader ........................................................................................................ 14
      3.1.2. MatrixFileWriter ........................................................................................................ 15
      3.1.3. Matrix Segment ........................................................................................................... 15
   3.2. Subclasses of MatrixFile ...................................................................................................... 17
      3.2.1. DoubleMatrixFile ...................................................................................................... 17
      3.2.2. FloatMatrixFile ......................................................................................................... 19
      3.2.3. LongMatrixFile ......................................................................................................... 20
      3.2.4. IntMatrixFile ............................................................................................................ 21
      3.2.5. ShortMatrixFile ....................................................................................................... 21
      3.2.6. CharMatrixFile ....................................................................................................... 22
      3.2.7. ByteMatrixFile ....................................................................................................... 22
      3.2.8. BooleanMatrixFile .................................................................................................. 23
      3.2.9. ObjectMatrixFile<T> ............................................................................................ 24
   3.3. ArrayFile .............................................................................................................................. 26
      3.3.1. ArrayFileReader ...................................................................................................... 27
      3.3.2. ArrayFileWriter ....................................................................................................... 28
      3.3.3. Array Segment ......................................................................................................... 28
   3.4. Subclasses of ArrayFile ....................................................................................................... 29
      3.4.1. DoubleArrayFile ..................................................................................................... 29
      3.4.2. FloatArrayFile ......................................................................................................... 30
      3.4.3. LongArrayFile ......................................................................................................... 31
      3.4.4. IntArrayFile ............................................................................................................ 31
      3.4.5. ShortArrayFile ....................................................................................................... 32
      3.4.6. CharArrayFile ....................................................................................................... 32
      3.4.7._BYTEArrayFile ..................................................................................................... 33
      3.4.8. BooleanArrayFile .................................................................................................. 34
      3.4.9. ObjectArrayFile<T> ............................................................................................ 34
   3.5. MatrixFileManager ............................................................................................................... 35
   3.6. ArrayFileManager ............................................................................................................... 36
4. Tests and Benchmarks ............................................................................................................... 36
   4.1. Matrix Multiplication ......................................................................................................... 36
   4.2. K-Means Clustering .......................................................................................................... 40
   4.3. Floyd’s Algorithm .............................................................................................................. 42
   4.4. Common Design Patterns ................................................................................................. 44
5. Test Runs .................................................................................................................................... 46
   5.1. Results ................................................................................................................................ 47
      5.1.1 Matrix Multiplication ................................................................................................. 47
      5.1.2. K-Means Clustering, ............................................................................................... 51
5.1.3. Floyd’s Algorithm ................................................................. 54
6. Future Work ................................................................................ 57
7. Conclusions ................................................................................ 58
Appendix A. Developer’s guide ....................................................... 59
A.1. Matrix Files ............................................................................. 59
    A.1.1. MatrixFileWriter ................................................................. 60
    A.1.2. MatrixFileReader ............................................................ 62
    A.1.3. MatrixFileManager .......................................................... 64
        A.1.3.1. Using MatrixFileManager Interactively ......................... 66
A.2. Array Files ............................................................................... 68
    A.2.1. ArrayFileWriter ................................................................. 69
    A.2.2. ArrayFileReader .............................................................. 70
    A.2.3. ArrayFileManager ............................................................ 72
        A.2.3.1. Using ArrayFileManager Interactively ......................... 73
References ................................................................................... 75
1. Literature Overview

The topics of high-performance and parallel programming with Java are addressed in the following works.

In [3], the author presents parallel programming concepts and writing parallel programs using pure Java. The Parallel Java (PJ) library, developed by the author, provides facilities for creating parallel programs for SMP, cluster and hybrid SMP cluster parallel computers. This work includes detailed examples on how to use PJ library and its various classes, such as the parallel image classes that support parallel input files and parallel output files patterns. Some of the examples explain and demonstrate the usage of `DoubleMatrixFile` class which allows reading and writing of matrix sections from and to multiple files and can effectively be used in parallel programs. Because of its simplicity and ease of use, the `DoubleMatrixFile` class was used as a starting point when developing the Parallel Datastore System. The concepts presented in this class such as the file structure, matrix segments, reader and writer, and others were extended and improved upon to create a set of classes for working with matrices and arrays of various element types.

MPI Version 2, described in [8], includes a chapter on parallel I/O. MPI-IO provides a standard interface for parallel I/O. One of its main features is the notion of “file views”, which define what part of a file a parallel process can see. This allows multiple processes to perform I/O operations on different parts of a file at the same time. File views also support patterned data placement via user-defined data types. Currently the MPI standard is supported primarily in C++, although work is in progress to provide support for the standard in other programming languages. The idea of supporting data placement via user-defined data types presented in this work provided the insight for including generic classes in PDS to support matrices and arrays of user-defined types.

Authors of [9] present the design of `jExpand` (Java Expandable Parallel File System) - a new Java I/O library for high-performance Java computing in heterogeneous distributed environments. `jExpand` uses the ideas applied in Expand parallel file system, developed by the same authors. `jExpand` combines several NFS servers to provide a distributed partition where files are declustered. The experiments carried out by the authors show a high improvement in the bandwidth obtained to access files. As a conclusion, the authors claim that the usage of `jExpand` can improve the I/O performance for parallel applications. The idea of partitioning files on several servers, presented in this work, was considered in the earlier stages of our project, but wasn’t included in PDS because of its complexity.
2. Parallel Java

Parallel Java (PJ) is an API and middleware for parallel programming in 100% Java on shared memory multiprocessor (SMP) parallel computers, cluster parallel computers, and hybrid SMP cluster parallel computers [2]. PJ was developed by Professor Alan Kaminsky and Luke McOmber in the Department of Computer Science at the Rochester Institute of Technology.

PJ provides SMP parallel programming features (such as parallel thread teams and parallel code regions) which are inspired by OpenMP, cluster parallel programming features (such as communicators and message passing patterns) which are inspired by MPI, as well as hybrid parallel programming features which both OpenMP and MPI currently lack [4].

2.1. Input/Output in PJ

Currently the I/O context of the Parallel Java API has limited capability – class StreamFile represents a file that resides in the user's account in the job front-end process of a Parallel Java cluster parallel program, and class DoubleMatrixFile provides an object for reading or writing a double matrix from or to a file [2].

The DoubleMatrixFile class was used as a starting point when developing the Parallel Datastore System. Let's review this class in more detail.

DoubleMatrixFile, as the name suggests, represents a matrix consisting of doubles. The number of rows and the number of columns in the matrix are known. The actual data are stored in a two-dimensional array, which can be specified as a parameter to the constructor or to the setMatrix method of DoubleMatrixFile class. DoubleMatrixFile provides a reader and a writer, which deal with reading a matrix of doubles from an input stream, and writing a matrix of doubles to an output stream, correspondingly.

When written out to a file, a double matrix file consists of the following items (each primitive item is written in binary format, with each int written as four bytes and each double written as eight bytes, most significant byte first):

- Number of matrix rows, \( R \) (int). \( R \geq 0 \).
- Number of matrix columns, \( C \) (int). \( C \geq 0 \).
- Zero or more segments of matrix elements.

Each matrix segment contains the following items:

- The segment's lower row index, \( RL \) (int). \( RL \geq 0 \).
- The segment's lower column index, \( CL \) (int). \( CL \geq 0 \).
- Number of rows in the segment, \( M \) (int). \( M \geq 0 \). \( RL+M \leq R \).
- Number of columns in the segment, \( N \) (int). \( N \geq 0 \). \( CL+N \leq C \).
- The matrix elements in rows \( RL..RL+M-1 \) and columns \( CL..CL+N-1 \) (double). Stored in row major order [5].

All writing operations are provided by the Writer class, which is an inner class of DoubleMatrixFile. To get a Writer object, it is necessary to call the prepareToWrite()
method on the `DoubleMatrixClass`, passing a valid output stream as a parameter. The `Writer` class provides several methods for writing data to a file:

- **write**
  
  This method writes the elements in all rows and columns of the matrix to the output stream.

- **writeRowSlice**

  This method writes the elements contained in a “slice” of rows of the matrix to the output stream. A “slice” here refers to a consecutive range of rows (see Figure 1), defined by the parameter passed in.

- **writeColSlice**

  This method writes the elements contained in a “slice” of columns of the matrix to the output stream. A “slice” here refers to a consecutive range of columns (see Figure 2), defined by the parameter passed in.

- **writePatch**

  This method writes the elements contained in a “patch” of the matrix to the output stream. A “patch” here refers to the intersection of a consecutive range of rows with a consecutive range of columns (see Figure 3), both passed in as parameters.

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>97</td>
<td>19</td>
<td>42</td>
<td>50</td>
<td>91</td>
<td>74</td>
<td>94</td>
<td>87</td>
<td>80</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>95</td>
<td>42</td>
<td>93</td>
<td>81</td>
<td>62</td>
<td>72</td>
<td>12</td>
<td>11</td>
<td>93</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>89</td>
<td>42</td>
<td>10</td>
<td>36</td>
<td>77</td>
<td>81</td>
<td>25</td>
<td>47</td>
<td>77</td>
</tr>
<tr>
<td>3</td>
<td>91</td>
<td>74</td>
<td>97</td>
<td>53</td>
<td>49</td>
<td>59</td>
<td>98</td>
<td>32</td>
<td>85</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>56</td>
<td>71</td>
<td>66</td>
<td>41</td>
<td>69</td>
<td>95</td>
<td>5</td>
<td>7</td>
<td>70</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>29</td>
<td>62</td>
<td>68</td>
<td>50</td>
<td>19</td>
<td>95</td>
<td>92</td>
<td>73</td>
<td>19</td>
<td>71</td>
</tr>
<tr>
<td>6</td>
<td>82</td>
<td>93</td>
<td>91</td>
<td>72</td>
<td>53</td>
<td>49</td>
<td>33</td>
<td>18</td>
<td>68</td>
<td>59</td>
</tr>
<tr>
<td>7</td>
<td>19</td>
<td>18</td>
<td>66</td>
<td>53</td>
<td>84</td>
<td>54</td>
<td>67</td>
<td>22</td>
<td>53</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>95</td>
<td>76</td>
<td>6</td>
<td>57</td>
<td>83</td>
<td>14</td>
<td>56</td>
<td>82</td>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>46</td>
<td>18</td>
<td>66</td>
<td>30</td>
<td>93</td>
<td>77</td>
<td>69</td>
<td>8</td>
<td>33</td>
<td>92</td>
</tr>
<tr>
<td>10</td>
<td>63</td>
<td>96</td>
<td>62</td>
<td>37</td>
<td>91</td>
<td>34</td>
<td>43</td>
<td>18</td>
<td>97</td>
<td>50</td>
</tr>
</tbody>
</table>
```

Figure 1. Row Slice
Figure 2. Column Slice

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>26</td>
<td>16</td>
<td>30</td>
<td>21</td>
<td>72</td>
<td>57</td>
<td>92</td>
<td>68</td>
<td>41</td>
<td>57</td>
<td>88</td>
</tr>
<tr>
<td>1</td>
<td>85</td>
<td>34</td>
<td>50</td>
<td>79</td>
<td>39</td>
<td>37</td>
<td>60</td>
<td>11</td>
<td>58</td>
<td>84</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>57</td>
<td>22</td>
<td>62</td>
<td>98</td>
<td>91</td>
<td>97</td>
<td>95</td>
<td>60</td>
<td>69</td>
<td>34</td>
<td>67</td>
</tr>
<tr>
<td>3</td>
<td>43</td>
<td>73</td>
<td>92</td>
<td>43</td>
<td>12</td>
<td>72</td>
<td>95</td>
<td>58</td>
<td>70</td>
<td>9</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>67</td>
<td>69</td>
<td>9</td>
<td>20</td>
<td>38</td>
<td>54</td>
<td>27</td>
<td>56</td>
<td>84</td>
<td>88</td>
<td>72</td>
</tr>
<tr>
<td>5</td>
<td>76</td>
<td>72</td>
<td>96</td>
<td>30</td>
<td>36</td>
<td>72</td>
<td>85</td>
<td>20</td>
<td>17</td>
<td>75</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
<td>45</td>
<td>31</td>
<td>73</td>
<td>47</td>
<td>24</td>
<td>1</td>
<td>15</td>
<td>72</td>
<td>63</td>
<td>88</td>
</tr>
<tr>
<td>7</td>
<td>88</td>
<td>6</td>
<td>25</td>
<td>49</td>
<td>72</td>
<td>83</td>
<td>1</td>
<td>43</td>
<td>66</td>
<td>77</td>
<td>71</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>90</td>
<td>28</td>
<td>20</td>
<td>45</td>
<td>54</td>
<td>21</td>
<td>70</td>
<td>88</td>
<td>44</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>28</td>
<td>49</td>
<td>92</td>
<td>64</td>
<td>72</td>
<td>66</td>
<td>60</td>
<td>90</td>
<td>66</td>
<td>32</td>
<td>77</td>
</tr>
<tr>
<td>10</td>
<td>21</td>
<td>69</td>
<td>44</td>
<td>15</td>
<td>90</td>
<td>44</td>
<td>14</td>
<td>2</td>
<td>90</td>
<td>90</td>
<td>18</td>
</tr>
</tbody>
</table>

Figure 3. Patch

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>71</td>
<td>75</td>
<td>40</td>
<td>75</td>
<td>71</td>
<td>44</td>
<td>6</td>
<td>7</td>
<td>66</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>33</td>
<td>8</td>
<td>35</td>
<td>38</td>
<td>38</td>
<td>94</td>
<td>97</td>
<td>20</td>
<td>5</td>
<td>86</td>
<td>91</td>
</tr>
<tr>
<td>2</td>
<td>57</td>
<td>83</td>
<td>7</td>
<td>41</td>
<td>46</td>
<td>68</td>
<td>44</td>
<td>39</td>
<td>98</td>
<td>32</td>
<td>68</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>19</td>
<td>53</td>
<td>77</td>
<td>94</td>
<td>75</td>
<td>65</td>
<td>98</td>
<td>31</td>
<td>6</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>86</td>
<td>87</td>
<td>97</td>
<td>41</td>
<td>79</td>
<td>47</td>
<td>83</td>
<td>54</td>
<td>65</td>
<td>96</td>
<td>93</td>
</tr>
<tr>
<td>5</td>
<td>89</td>
<td>82</td>
<td>6</td>
<td>91</td>
<td>56</td>
<td>99</td>
<td>11</td>
<td>30</td>
<td>74</td>
<td>78</td>
<td>99</td>
</tr>
<tr>
<td>6</td>
<td>93</td>
<td>36</td>
<td>36</td>
<td>10</td>
<td>53</td>
<td>6</td>
<td>10</td>
<td>19</td>
<td>52</td>
<td>99</td>
<td>85</td>
</tr>
<tr>
<td>7</td>
<td>50</td>
<td>95</td>
<td>36</td>
<td>99</td>
<td>79</td>
<td>14</td>
<td>31</td>
<td>77</td>
<td>90</td>
<td>15</td>
<td>75</td>
</tr>
<tr>
<td>8</td>
<td>91</td>
<td>21</td>
<td>53</td>
<td>65</td>
<td>83</td>
<td>94</td>
<td>56</td>
<td>85</td>
<td>88</td>
<td>90</td>
<td>53</td>
</tr>
<tr>
<td>9</td>
<td>19</td>
<td>78</td>
<td>14</td>
<td>55</td>
<td>92</td>
<td>30</td>
<td>26</td>
<td>4</td>
<td>83</td>
<td>69</td>
<td>78</td>
</tr>
<tr>
<td>10</td>
<td>44</td>
<td>7</td>
<td>42</td>
<td>83</td>
<td>49</td>
<td>34</td>
<td>4</td>
<td>12</td>
<td>22</td>
<td>82</td>
<td>62</td>
</tr>
</tbody>
</table>
As a result of each write operation, a new segment is written out to the file. *DoubleMatrixFile* can be used in parallel programs, where each instance of the parallel program needs to write a subset of the output data. By using *DoubleMatrixFile*, each instance of the parallel program can write its own subset of data separately to a different file, as opposed to having one designated instance write all of the data to one file. This way, the overall running time of the parallel program can be significantly reduced. The different files containing separate subsets of data can be combined together into one file using the main method of the *DoubleMatrixFile* class.

All reading operations are provided by the *Reader* class, which is an inner class of *DoubleMatrixFile*. To get a *Reader* object, it is necessary to call the `prepareToRead()` method on the *DoubleMatrixClass*, passing a valid input stream as a parameter. The *Reader* class provides several methods for reading data from the input stream, organized into two groups. The first group of reader methods read all of the segments contained in the file:

- **read**
  This method reads all of the elements from all segments of the matrix from the input stream.

- **readRowSlice**
  This method reads the elements from all segments of the matrix from the input stream, but only stores elements contained in a “slice” of rows of the matrix. A “slice” here refers to a consecutive range of rows (see Figure 1), defined by the parameter passed in.

- **readColSlice**
  This method reads the elements from all segments of the matrix from the input stream, but only stores elements contained in a “slice” of columns of the matrix. A “slice” here refers to a consecutive range of columns (see Figure 2), defined by the parameter passed in.

- **readPatch**
  This method reads the elements from all segments of the matrix from the input stream, but only stores elements contained in a “patch” of the matrix to the output stream. A “patch” here refers to the intersection of a consecutive range of rows with a consecutive range of columns (see Figure 3), both passed in as parameters.

The second group of reader methods read only the next matrix element segment contained on the file:

- **readSegment**
  This method reads all of the elements from the next segment of the matrix from the input stream.

- **readSegmentRowSlice**
  This method reads the elements from the next segment of the matrix from the input stream, but only stores elements contained in a “slice” of rows of the matrix. A “slice” here refers to a consecutive range of rows (see Figure 1), defined by the parameter passed in.

- **readSegmentColSlice**
This method reads the elements from the next segment of the matrix from the input stream, but only stores elements contained in a “slice” of columns of the matrix. A “slice” here refers to a consecutive range of columns (see Figure 2), defined by the parameter passed in.

- **readSegmentPatch**
  This method reads the elements from the next segment of the matrix from the input stream, but only stores elements contained in a “patch” of the matrix to the output stream. A “patch” here refers to the intersection of a consecutive range of rows with a consecutive range of columns (see Figure 3), both passed in as parameters.

### 2.2. The Range class

PJ provides a utility class `Range`, which represents a range of integers. A range object has the following attributes: lower bound $L$, upper bound $U$, stride $S$, and length $N$. A range object represents the following set of integers: \( \{L, L+S, L+2*S, \ldots, L+(N-1)*S\} \), where \( U = L+(N-1)*S \) [5].

A range object can be specified using its lower bound, upper bound and stride, which is sufficient to calculate the length of the range. A range object can also be specified using only its lower bound and upper bounds – in this case the stride is set to 1, and the length can still be calculated.

This class is used by the `DoubleMatrixFile` to define slices and patches. All range objects used to specify slices and patches for `DoubleMatrixFile` have the stride of one, meaning all slices and patches consist of consecutive rows and columns.

The row slice in Figure 1 can therefore be defined using a range object with lower bound of 2 and upper bound of 5. Similarly, the column slice in Figure 2 can be defined using a range object with lower bound of 2 and upper bound of 4, and the patch in Figure 3 can be defined using one range object with lower bound of 5 and upper bound of 9 and another range object with lower bound of 2 and upper bound of 5.

### 3. Parallel Datastore System

The Parallel Datastore System (PDS) is implemented as a set of classes that are modeled after the `DoubleMatrixFile` class. This set of classes extends the functionality provided by the original `DoubleMatrixFile` to support other primitive and user-defined types, as well as further improves the concept of segments.

There are two groups of classes – one group for working with vectors (one-dimensional arrays) and another group for working with matrices (two-dimensional arrays). Class hierarchies of the two groups are shown in Figures 4 and 5.
Figure 4. PDS Matrix File Class Hierarchy
Figure 5. PDS Array File Class Hierarchy
3.1. MatrixFile

MatrixFile is an abstract class that defines attributes and methods applicable to all matrix files, regardless of matrix element type. This class also defines inner classes MatrixFileReader and MatrixFileWriter which incorporate the most general read and write operations which are not dependent on the type of the matrix element.

When written out to a file, a matrix file consists of the following items (each primitive item is written in binary format, most significant byte first):

- Matrix elements’ class name (string).
- Number of matrix rows, \( R \) (int). \( R \geq 0 \).
- Number of matrix columns, \( C \) (int). \( C \geq 0 \).
- Zero or more segments of matrix elements.

Each matrix segment contains the following items.

- The segment's lower row index, \( RL \) (int). \( RL \geq 0 \).
- The segment's lower column index, \( CL \) (int). \( CL \geq 0 \).
- Number of rows in the segment, \( M \) (int). \( M \geq 0 \). \( RL+M \leq R \).
- Number of columns in the segment, \( N \) (int). \( N \geq 0 \). \( CL+N \leq C \).
- The segment’s row stride, \( RS \) (int). \( RS \geq 1 \).
- The segment’s column stride, \( CS \) (int). \( CS \geq 1 \).
- The matrix elements in rows \( RL..RL+M-1 \) and columns \( CL..CL+N-1 \) (element type varies). Stored in row major order.

MatrixFile is extended by 9 subclasses: 8 classes that define matrix files for primitive types and one generic matrix file that can be used store elements of any non-primitive types.

Each of the 9 subclasses of MatrixFile contains inner classes Reader and Writer, which provide functionality for reading and writing elements of the matrix, correspondingly. Each inner class Reader extends the MatrixFileReader class and each inner class Writer extends the MatrixFileWriter class.

MatrixFile class has the following protected attributes:

- \texttt{elementClass}
  Describes the type (class) of matrix elements.
- \( R \)
  Number of rows in the matrix
- \( C \)
  Number of columns in the matrix

This class has the following methods:

- \texttt{getRowCount}
  Returns the number of rows in this matrix
- \texttt{getColCount}
  Returns the number of columns in this matrix
• **setRC**
  Accepts two integers as parameters and sets the number of rows and the number of columns to these values. Throws an exception if any of the parameters is negative
• **getBytesPerElement** - abstract
  Returns the number of bytes required to store a single matrix element
• **getElementClass** - abstract
  Returns the type (class) of the matrix elements
• **PrepareToRead** - abstract
  Returns a *MatrixFileReader* object that can be used to read matrix elements from an input stream passed in as parameter
• **PrepareToWrite** - abstract
  Returns a *MatrixFileWriter* object that can be used to write matrix elements to an output stream passed in as parameter

### 3.1.1. MatrixFileReader

*MatrixFileReader* is an abstract class that describes the generic functionality to read data from a matrix file. This class is extended by other classes that provide implementation specific to matrix file of certain types. An instance of this class is returned by the *prepareToRead* method of class *MatrixFile*, which accepts an *InputStream* object as its parameter.

This class provides implementation for some of the generic methods, whose behavior doesn’t depend on the type of the matrix elements:

- The constructor of this class accepts an *InputStream* as its parameter, creates a *DataInputStream* from this input stream and reads the meta information about the matrix: the number of rows and columns, and the type name of the matrix elements (e.g., “int” for an matrix file of integers).
- Method *getNextSegment* reads meta information about the next matrix segment from the input stream (matrix segments are described in more detail in a later section)
- Methods *getRowRange* and *getColRange* return the range of rows and columns in the next segment, respectively.
- Similar to the *Reader* class declared inside of the original *DoubleMatrixFile*, class *MatrixFileReader* provides several methods for reading data from the input stream, organized into two groups. The first group of reader methods read all of the segments contained in the file, and the second group of reader methods read only the next matrix element segment contained in the file (see section 2.1). All of these reader methods ultimately delegate the actual reading operation to the abstract method *readSegment* – this method is implemented in the subclasses of *MatrixFileReader* and the implementation is dependent on the type of the matrix elements.
- Method *close* closes the *DataInputStream* created in the constructor.
3.1.2. MatrixFileWriter

MatrixFileWriter is an abstract class that describes the generic functionality to write data to a matrix file. This class is extended by other classes that provide implementation specific to matrix file of certain types. An instance of this class is returned by the prepareToWrite method of class MatrixFile, which accepts an OutputStream object as its parameter.

This class provides implementation for some of the generic methods, whose behavior doesn’t depend on the type of the matrix elements:

- The constructor of this class accepts an OutputStream as its parameter, creates a DataOutputStream from this output stream and writes the meta information about the matrix: the number of rows and columns, and the type name of the matrix elements (e.g., “int” for an matrix file of integers).
- Similar to the Writer class declared inside of the original DoubleMatrixFile, class MatrixFileWriter provides several methods for writing data to the output stream (see section 2.1). All of these writer methods ultimately delegate the actual writing operation to method write – this method writes the meta information about the matrix segment and is called from the write method implemented in the subclasses of MatrixFileWriter before writing the actual data of the matrix segment.
- Method close closes the DataOutputStream created in the constructor.

3.1.3. Matrix Segment

Similar to the Reader and Writer classes in the original DoubleMatrixFile class, the new MatrixFileReader and MatrixFileWriter classes read and write segments of matrices. A segment can be read or written as a row slice, a column slice, or a patch. As described in section 2.2, a slice is defined by a single Range object, and a patch is defined by a pair of Range objects.

The original DoubleMatrixFile functionality required all Range objects used to define slices and patches to have a stride of 1. The new functionality provided by PDS drops this restriction – now the Range objects used to define slices and patches may have a stride greater than one. This implies that slices and patches such as those shown on Figures 6a-6c can be used.
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>51</td>
<td>7</td>
<td>53</td>
<td>8</td>
<td>96</td>
<td>89</td>
<td>95</td>
<td>5</td>
<td>85</td>
<td>66</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>19</td>
<td>76</td>
<td>45</td>
<td>55</td>
<td>77</td>
<td>56</td>
<td>89</td>
<td>75</td>
<td>71</td>
</tr>
<tr>
<td>2</td>
<td>70</td>
<td>19</td>
<td>27</td>
<td>78</td>
<td>92</td>
<td>68</td>
<td>26</td>
<td>29</td>
<td>73</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>80</td>
<td>4</td>
<td>78</td>
<td>26</td>
<td>53</td>
<td>73</td>
<td>64</td>
<td>73</td>
<td>42</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>81</td>
<td>40</td>
<td>86</td>
<td>12</td>
<td>37</td>
<td>57</td>
<td>81</td>
<td>48</td>
<td>38</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>67</td>
<td>29</td>
<td>91</td>
<td>67</td>
<td>26</td>
<td>67</td>
<td>71</td>
<td>96</td>
<td>63</td>
</tr>
<tr>
<td>6</td>
<td>75</td>
<td>47</td>
<td>98</td>
<td>1</td>
<td>78</td>
<td>98</td>
<td>27</td>
<td>94</td>
<td>45</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>60</td>
<td>34</td>
<td>74</td>
<td>59</td>
<td>76</td>
<td>17</td>
<td>17</td>
<td>85</td>
<td>82</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>98</td>
<td>68</td>
<td>86</td>
<td>31</td>
<td>43</td>
<td>94</td>
<td>82</td>
<td>26</td>
<td>34</td>
<td>98</td>
</tr>
<tr>
<td>9</td>
<td>96</td>
<td>82</td>
<td>84</td>
<td>47</td>
<td>56</td>
<td>81</td>
<td>40</td>
<td>66</td>
<td>2</td>
<td>55</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
<td>46</td>
<td>85</td>
<td>40</td>
<td>89</td>
<td>13</td>
<td>87</td>
<td>81</td>
<td>36</td>
<td>12</td>
</tr>
</tbody>
</table>

a) Strided row slice

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>73</td>
<td>42</td>
<td>33</td>
<td>65</td>
<td>50</td>
<td>90</td>
<td>79</td>
<td>85</td>
<td>68</td>
<td>37</td>
</tr>
<tr>
<td>1</td>
<td>59</td>
<td>65</td>
<td>22</td>
<td>88</td>
<td>11</td>
<td>11</td>
<td>98</td>
<td>90</td>
<td>39</td>
<td>29</td>
</tr>
<tr>
<td>2</td>
<td>67</td>
<td>17</td>
<td>3</td>
<td>19</td>
<td>21</td>
<td>33</td>
<td>29</td>
<td>29</td>
<td>82</td>
<td>75</td>
</tr>
<tr>
<td>3</td>
<td>90</td>
<td>59</td>
<td>47</td>
<td>76</td>
<td>11</td>
<td>84</td>
<td>91</td>
<td>33</td>
<td>55</td>
<td>74</td>
</tr>
<tr>
<td>4</td>
<td>42</td>
<td>22</td>
<td>72</td>
<td>16</td>
<td>12</td>
<td>82</td>
<td>90</td>
<td>71</td>
<td>25</td>
<td>68</td>
</tr>
<tr>
<td>5</td>
<td>96</td>
<td>98</td>
<td>3</td>
<td>28</td>
<td>96</td>
<td>74</td>
<td>44</td>
<td>29</td>
<td>86</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
<td>46</td>
<td>68</td>
<td>64</td>
<td>27</td>
<td>94</td>
<td>88</td>
<td>94</td>
<td>75</td>
<td>96</td>
</tr>
<tr>
<td>7</td>
<td>42</td>
<td>98</td>
<td>72</td>
<td>77</td>
<td>97</td>
<td>99</td>
<td>2</td>
<td>26</td>
<td>6</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>46</td>
<td>36</td>
<td>57</td>
<td>26</td>
<td>7</td>
<td>20</td>
<td>82</td>
<td>26</td>
<td>78</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td>85</td>
<td>67</td>
<td>17</td>
<td>46</td>
<td>52</td>
<td>48</td>
<td>7</td>
<td>64</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>96</td>
<td>64</td>
<td>51</td>
<td>15</td>
<td>34</td>
<td>20</td>
<td>10</td>
<td>51</td>
<td>85</td>
<td>91</td>
</tr>
</tbody>
</table>

b) Strided column slice
3.2. Subclasses of MatrixFile

There are nine subclasses of MatrixFile, which provide implementation specific to certain matrix element types. The following sections describe these subclasses.

3.2.1. DoubleMatrixFile

Class DoubleMatrixFile of PDS is different from the original DoubleMatrixFile of PJ library. The new DoubleMatrixFile inherits from the abstract class MatrixFile and provides implementation for matrix file whose elements are of type `double`.

This class provides the following attributes and methods (in addition to those inherited from class MatrixFile):

- `BYTES_PER_ELEMENT` is a constant with the value of 8. This constant represents the number of bytes required to store a single element of a matrix of doubles.
- `myMatrix` is an array of array of doubles. This object is used to store that matrix data.
- Two constructors: the default constructor initializes the attributes to pre-defined values, while the second constructor initializes the number of rows `R`, number of columns `C` and the array `myMatrix` to values specified as parameters.
- Method `getMatrix` returns the array `myMatrix`
- Method `setMatrix` sets the number of rows `R`, number of columns `C` and the array `myMatrix` to values specified as parameters.
- Method `getBytesPerElement` returns the value of constant `BYTES_PER_ELEMENT`
- Method `getElementClass` returns an instance of class `Class<? extends ?>` that corresponds to the primitive type `double`. In Java this return value is specified as "`double.class`".
- Method `prepareToRead` accepts an input stream and returns an instance of the inner class `DoubleMatrixFile.Reader`.
- Method `prepareToWrite` accepts an output stream and returns an instance of the inner class `DoubleMatrixFile.Writer`. An exception is thrown if this method is called before the array `myMatrix` is properly initialized.


Inner class `DoubleMatrixFile.Reader` inherits from class `MatrixFile.MatrixFileReader` and provides functionality of reading elements of type `double` from an input stream. This class provides implementation of method `readSegment` which accepts 6 integer parameters:
- `RL` – Segment’s lower row index.
- `RU` – Segment’s upper row index.
- `RS` – Segment’s row stride.
- `CL` – Segment’s lower column index.
- `CU` – Segment’s upper column index.
- `CS` – Segment’s column stride.

This method reads the next matrix segment from the input stream but only those elements that are members of the segment described by the 6 input parameters are stored in the `myMatrix` array. Reading of each element is performed by calling method `readDouble` of class `DataInputStream`, which reads 8 bytes from the input stream and converts those bytes into a single `double` number.

For example, if the next matrix segment contains a row slice `(10, 0) .. (19, 17)` and we invoke `readSegment(12, 18, 1, 3, 13, 1)`, then the following elements (indicated by dark-gray cells) will be stored in `myMatrix`:
Figure 7. Intersection of a row slice and a patch

Inner class `DoubleMatrixFile.Writer` inherits from class `MatrixFile.MatrixFileWriter` and provides functionality of writing elements of type `double` to an input stream. This class provides implementation of method `write` which accepts 6 integer parameters:

- **RL** – Segment’s lower row index.
- **RU** – Segment’s upper row index.
- **RS** – Segment’s row stride.
- **CL** – Segment’s lower column index.
- **CU** – Segment’s upper column index.
- **CS** – Segment’s column stride.

This method first invokes the `write` method of the base class, which writes the meta information about the matrix segment. Then this method writes those elements of array `myMatrix` that fall into the segment described by the 6 input parameters to the output stream. Writing of each element is performed by calling method `writeDouble` of class `DataOutputStream`, which writes each `double` number as 8 bytes to the output stream. Elements are written to the output stream in row-major order.

### 3.2.2. FloatMatrixFile

Class `FloatMatrixFile` inherits from the abstract class `MatrixFile` and provides implementation for matrix file whose elements are of type `float`.

This class provides the same attributes and methods as class `DoubleMatrixFile`, with the following differences:

- **BYTES_PER_ELEMENT** is a constant with the value of 4. This constant represents the number of bytes required to store a single element of type `float`. 
• `myMatrix` is an array of arrays of `floats`. This object is used to store that matrix data.
• Method `getElementClass` returns an instance of class `Class<?>` that corresponds to the primitive type `float`. In Java this return value is specified as “`float.class`”.
• Method `prepareToRead` accepts an input stream and returns an instance of the inner class `FloatMatrixFile.Reader`.
• Method `prepareToWrite` accepts an output stream and returns an instance of the inner class `FloatMatrixFile.Writer`. An exception is thrown if this method is called before the array `myMatrix` is properly initialized.

Class `FloatMatrixFile` defines two inner classes: `FloatMatrixFile.Reader` and `FloatMatrixFile.Writer`, which inherit from inner classes `MatrixFile.MatrixFileReader` and `MatrixFile.MatrixFileWriter`, respectively. The functionality of `FloatMatrixFile.Reader` and `FloatMatrixFile.Writer` is the same as the corresponding inner classes of `DoubleMatrixFile`, with the only difference being that these two classes read and write 4 bytes per element and convert those bytes to/from an element of type `float`.

### 3.2.3. LongMatrixFile

Class `LongMatrixFile` inherits from the abstract class `MatrixFile` and provides implementation for matrix file whose elements are of type `long`.

This class provides the same attributes and methods as class `DoubleMatrixFile`, with the following differences:
• `BYTES_PER_ELEMENT` is a constant with the value of 8. This constant represents the number of bytes required to store a single element of type `long`.
• `myMatrix` is an array of arrays of `longs`. This object is used to store that matrix data.
• Method `getElementClass` returns an instance of class `Class<?>` that corresponds to the primitive type `long`. In Java this return value is specified as “`long.class`”.
• Method `prepareToRead` accepts an input stream and returns an instance of the inner class `LongMatrixFile.Reader`.
• Method `prepareToWrite` accepts an output stream and returns an instance of the inner class `LongMatrixFile.Writer`. An exception is thrown if this method is called before the array `myMatrix` is properly initialized.

Class `LongMatrixFile` defines two inner classes: `LongMatrixFile.Reader` and `LongMatrixFile.Writer`, which inherit from inner classes `MatrixFile.MatrixFileReader` and `MatrixFile.MatrixFileWriter`, respectively. The functionality of `LongMatrixFile.Reader` and `LongMatrixFile.Writer` is the same as the corresponding inner classes of `DoubleMatrixFile`, with the only difference being that these two classes read/write 8 bytes and convert those bytes to/from an element of type `long`. 

20
3.2.4. IntMatrixFile

Class *IntMatrixFile* inherits from the abstract class *MatrixFile* and provides implementation for matrix file whose elements are of type *int*.

This class provides the same attributes and methods as class *DoubleMatrixFile*, with the following differences:

- *BYTES_PER_ELEMENT* is a constant with the value of 4. This constant represents the number of bytes required to store a single element of type *int*.
- *myMatrix* is an array of arrays of *ints*. This object is used to store that matrix data.
- Method *getElementClass* returns an instance of class *Class<?>* that corresponds to the primitive type *int*. In Java this return value is specified as “*int.class*”.
- Method *prepareToRead* accepts an input stream and returns an instance of the inner class *IntMatrixFile.Reader*.
- Method *prepareToWrite* accepts an output stream and returns an instance of the inner class *IntMatrixFile.Writer*. An exception is thrown if this method is called before the array *myMatrix* is properly initialized.

Class *IntMatrixFile* defines two inner classes: *IntMatrixFile.Reader* and *IntMatrixFile.Writer*, which inherit from inner classes *MatrixFile.MatrixFileReader* and *MatrixFile.MatrixFileWriter*, respectively. The functionality of *IntMatrixFile.Reader* and *IntMatrixFile.Writer* is the same as the corresponding inner classes of *DoubleMatrixFile*, with the only difference being that these two classes read/write 4 bytes and convert those bytes to/from an element of type *int*.

3.2.5. ShortMatrixFile

Class *ShortMatrixFile* inherits from the abstract class *MatrixFile* and provides implementation for matrix file whose elements are of type *short*.

This class provides the same attributes and methods as class *DoubleMatrixFile*, with the following differences:

- *BYTES_PER_ELEMENT* is a constant with the value of 2. This constant represents the number of bytes required to store a single element of type *short*.
- *myMatrix* is an array of arrays of *shorts*. This object is used to store that matrix data.
- Method *getElementClass* returns an instance of class *Class<?>* that corresponds to the primitive type *short*. In Java this return value is specified as “*short.class*”.
- Method *prepareToRead* accepts an input stream and returns an instance of the inner class *ShortMatrixFile.Reader*.
- Method *prepareToWrite* accepts an output stream and returns an instance of the inner class *ShortMatrixFile.Writer*. An exception is thrown if this method is called before the array *myMatrix* is properly initialized.
Class ShortMatrixFile defines two inner classes: ShortMatrixFile.Reader and ShortMatrixFile.Writer, which inherit from inner classes MatrixFile.MatrixFileReader and MatrixFile.MatrixFileWriter, respectively. The functionality of ShortMatrixFile.Reader and ShortMatrixFile.Writer is the same as the corresponding inner classes of DoubleMatrixFile, with the only difference being that these two classes read/write 2 bytes and convert those bytes to/from an element of type short.

3.2.6. CharMatrixFile

Class CharMatrixFile inherits from the abstract class MatrixFile and provides implementation for matrix file whose elements are of type char.

This class provides the same attributes and methods as class DoubleMatrixFile, with the following differences:

- `BYTES_PER_ELEMENT` is a constant with the value of 2. This constant represents the number of bytes required to store a single element of type char.
- `myMatrix` is an array of arrays of chars. This object is used to store that matrix data.
- Method `getElementClass` returns an instance of class `Class<?>` that corresponds to the primitive type char. In Java this return value is specified as “char.class”.
- Method `prepareToRead` accepts an input stream and returns an instance of the inner class CharMatrixFile.Reader.
- Method `prepareToWrite` accepts an output stream and returns an instance of the inner class CharMatrixFile.Writer. An exception is thrown if this method is called before the array `myMatrix` is properly initialized.

Class CharMatrixFile defines two inner classes: CharMatrixFile.Reader and CharMatrixFile.Writer, which inherit from inner classes MatrixFile.MatrixFileReader and MatrixFile.MatrixFileWriter, respectively. The functionality of CharMatrixFile.Reader and CharMatrixFile.Writer is the same as the corresponding inner classes of DoubleMatrixFile, with the only difference being that these two classes read/write 2 bytes and convert those bytes to/from an element of type char.

3.2.7. ByteMatrixFile

Class ByteMatrixFile inherits from the abstract class MatrixFile and provides implementation for matrix file whose elements are of type byte.

This class provides the same attributes and methods as class DoubleMatrixFile, with the following differences:

- `BYTES_PER_ELEMENT` is a constant with the value of 1. This constant represents the number of bytes required to store a single element of type byte.
- `myMatrix` is an array of arrays of bytes. This object is used to store that matrix data.
• Method `getElementClass` returns an instance of class `Class<?>` that corresponds to the primitive type `byte`. In Java this return value is specified as “byte.class”.
• Method `prepareToRead` accepts an input stream and returns an instance of the inner class `ByteMatrixFile.Reader`.
• Method `prepareToWrite` accepts an output stream and returns an instance of the inner class `ByteMatrixFile.Writer`. An exception is thrown if this method is called before the array `myMatrix` is properly initialized.

Class `ByteMatrixFile` defines two inner classes: `ByteMatrixFile.Reader` and `ByteMatrixFile.Writer`, which inherit from inner classes `MatrixFile.MatrixFileReader` and `MatrixFile.MatrixFileWriter`, respectively. The functionality of `ByteMatrixFile.Reader` and `ByteMatrixFile.Writer` is the same as the corresponding inner classes of `DoubleMatrixFile`, with the only difference being that these two classes read/write 1 byte and convert that byte to/from an element of type `byte`.

### 3.2.8. BooleanMatrixFile

Class `BooleanMatrixFile` inherits from the abstract class `MatrixFile` and provides implementation for matrix file whose elements are of type `boolean`.

This class provides the same attributes and methods as class `DoubleMatrixFile`, with the following differences:

• `BYTES_PER_ELEMENT` is a constant with the value of 1. This constant represents the number of bytes required to store a single element of type `boolean`.
• `myMatrix` is an array of arrays of `booleans`. This object is used to store that matrix data.
• Method `getElementClass` returns an instance of class `Class<?>` that corresponds to the primitive type `boolean`. In Java this return value is specified as “boolean.class”.
• Method `prepareToRead` accepts an input stream and returns an instance of the inner class `BooleanMatrixFile.Reader`.
• Method `prepareToWrite` accepts an output stream and returns an instance of the inner class `BooleanMatrixFile.Writer`. An exception is thrown if this method is called before the array `myMatrix` is properly initialized.

Class `BooleanMatrixFile` defines two inner classes: `BooleanMatrixFile.Reader` and `BooleanMatrixFile.Writer`, which inherit from inner classes `MatrixFile.MatrixFileReader` and `MatrixFile.MatrixFileWriter`, respectively. The functionality of `BooleanMatrixFile.Reader` and `BooleanMatrixFile.Writer` is the same as the corresponding inner classes of `DoubleMatrixFile`, with the only difference being that these two classes read/write 1 byte and convert that byte to/from an element of type `boolean`.
3.2.9. ObjectMatrixFile<T>

Class ObjectMatrixFile is a generic class that inherits from the abstract class MatrixFile and provides implementation for matrix file whose elements are of non-primitive types.

This class provides the following attributes and methods (in addition to those inherited from class MatrixFile):

- **myMatrix** is an array of arrays of generic type T. This object is used to store that matrix data.
- Two constructors: the default constructor initializes the attributes to pre-defined values, while the second constructor initializes the number of rows \( R \), number of columns \( C \) and the array myMatrix to values specified as parameters.
- Method `getMatrix` returns the array myMatrix
- Method `setMatrix` sets the number of rows \( R \), number of columns \( C \) and the array myMatrix to values specified as parameters.
- Method `getBytesPerElement` returns the value of -1 to indicate that the size in bytes of one matrix element may be different from the size of another element, i.e., the element size is not constant.
- Method `getElementClass` returns an instance of class `Class<?>` that corresponds to the actual type of the generic type T. For example, this method returns `Class<BigInteger>` when called on an instance of `ObjectMatrixFile<BigInteger>`.
- Method `prepareToRead` accepts an input stream and returns an instance of the inner class `ObjectMatrixFile.Reader`.
- Method `prepareToWrite` accepts an output stream and returns an instance of the inner class `ObjectMatrixFile.Writer`. An exception is thrown if this method is called before the array myMatrix is properly initialized.

Class ObjectMatrixFile defines two inner classes: `ObjectMatrixFile.Reader` and `ObjectMatrixFile.Writer`, which inherit from inner classes `MatrixFile.MatrixFileReader` and `MatrixFile.MatrixFileWriter`, respectively. The functionality of `ObjectMatrixFile.Reader` and `ObjectMatrixFile.Writer` is the same as the corresponding inner classes of `DoubleMatrixFile`, but the implementation is more complicated.

Because ObjectMatrixFile<T> is a generic class and works with almost any non-primitive data type, it wouldn’t be correct to assume that all elements of such matrix will have the same size. For example, different elements of a matrix of strings could have different lengths, and therefore different sizes in bytes.

Writing one matrix element to an output stream involves first writing the size of the element in bytes, and then writing the actual element data. The following code snippet demonstrates the procedure of writing a matrix element \( X \) to an output stream:

```java
//create a DataOutputStream.
//outputStream is passed as argument to method prepareToWrite():
DataOutputStream myDos = new DataOutputStream(outputStream);
```
Here, we create a new `ByteArrayOutputStream myBaos`, which will store the data written to it as a byte array. Then we create an `ObjectOutputStream oos` on top of `myBaos`. We can write any object that supports `java.io.Serializable` interface to `oos` – that object will end up in `myBaos` in the form of a byte array. After that, we can retrieve the actual array of bytes from the underlying output stream `myBaos`. This will provide us the size of the element being written. We then write the size of the element, followed by the element data stored in the byte array.

Reading of a matrix element `Y` is demonstrated by the following code snippet:

```java
//create a DataInputStream.
//inputStream is passed as argument to method prepareToRead():
DataInputStream myDis = new DataInputStream(inputStream);
//read the size of element:
int elementSize = myDis.readInt();
//allocate a byte array to store element data:
byte[] bytes = new byte[elementSize];
//read element data:
myDis.readFully(bytes);
//create a stream to read from the byte array:
ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
//create a stream to read objects from:
ObjectInputStream ois = new ObjectInputStream(bais);
//read the object and cast it to the generic type:
Y = (T) ois.readObject();
```
Here we first read the size of the element from the input stream and allocate a byte array big enough to store the element data and then read the element data into the byte array. Afterwards we create a stream to read from the byte array and on top of that stream we create another stream to read objects from. Finally we read an object and cast it to our generic type.

Classes `ObjectInputStream` and `ObjectOutputStream` require that objects to be read using method `readObject()` or written using method `writeObject()` support the `java.io.Serializable` interface [6]. Therefore, the only restriction for types to be used with `ObjectMatrixFile` is they must implement the `java.io.Serializable` interface.

Technically, it is unnecessary to write the size of each matrix element before writing the element itself, because when the element is later read using the `readObject()` method of class `ObjectInputStream`, that method will read the correct number of bytes and reconstruct the object. Writing the size of the matrix element before the actual element is useful when we need to skip one or more elements. For example, if we need to skip 100 matrix elements, for each of the 100 elements we can read the size of an element and then skip that many bytes. Whereas if we didn’t write the size of each element separately, we would need to call the `readObject()` method 100 times, and each time we called that method, it would cause memory allocation, object construction, and then memory de-allocation and eventual object destruction (because we don’t need these elements – we want to skip them). This could considerably increase the amount of time and memory spent on skipping matrix elements. Therefore it was decided to write the size of each matrix element separately before writing the element itself. This approach does have a disadvantage though – the output file will have bigger size, since four additional bytes are stored for each element.

3.3. ArrayFile

`ArrayFile` is an abstract class that defines attributes and methods applicable to all array files, regardless of array element type. This class also defines inner classes `ArrayFileReader` and `ArrayFileWriter` which incorporate the most general read and write operations which are not dependent on the type of the array element.

When written out to a file, an array file consists of the following items (each primitive item is written in binary format, most significant byte first):
- Array elements’ class name (string).
- Number of array elements, $N$ (int). $N \geq 0$.
- Zero or more segments of array elements.

Each array segment contains the following items:
- The segment's lower element index, $L$ (int). $L \geq 0$.
- Number of elements in the segment, $M$ (int). $M \geq 0$. $L+M \leq N$.
- The segment’s element stride, $S$ (int). $S \geq 0$.
- The array elements $L..L+M-1$ (element type varies).

`ArrayFile` is extended by 9 subclasses: 8 classes that define array files for primitive types and one generic array file that can be used store elements of any non-primitive types.
Each of the 9 subclasses of ArrayFile contains inner classes Reader and Writer, which provide functionality for reading and writing elements of the array, correspondingly. Each inner class Reader extends the ArrayFileReader class and each inner class Writer extends the ArrayFileWriter class.

ArrayFile class has the following protected attributes:
- `elementClass` Describes the type (class) of array elements.
- `N` Number of elements in the array

This class has the following methods:
- `getElementCount` Returns the number of elements in this array
- `setN` Accepts an integer as parameter and set the number of elements to this value. Throws an exception if the parameter is negative
- `getBytesPerElement` - abstract
  Returns the number of bytes required to store a single array element
- `getElementClass` - abstract
  Returns the type (class) of the array elements
- `PrepareToRead` - abstract
  Returns an ArrayFileReader object that can be used to read array elements from an input stream passed in as parameter
- `PrepareToWrite` - abstract
  Returns an ArrayFileWriter object that can be used to write array elements to an output stream passed in as parameter

### 3.3.1. ArrayFileReader

ArrayFileReader is an abstract class that describes the generic functionality to read data from an array file. This class is extended by other classes that provide implementation specific to array file of certain types. An instance of this class is returned by the `prepareToRead` method of class ArrayFile, which accepts an `InputStream` object as its parameter.

This class provides implementation for some of the generic methods, whose behavior doesn’t depend on the type of the array elements:
- The constructor of this class accepts an `InputStream` as its parameter, creates a `DataInputStream` from this input stream and reads the meta information about the array: the number of elements, and the type name of the array elements (e.g., “int” for an array file of integers).
- Method `getNextSegment` reads meta information about the next array segment from the input stream (array segments are described in more detail in a later section).
- Method `getRange` returns the range of elements in the next segment.
• Several methods for reading data from the input stream, organized into two groups. The first group of reader methods read all of the segments contained in the file, and the second group of reader methods read only the next array segment contained in the file. All of these reader methods ultimately delegate the actual reading operation to the abstract method `readSegment` – this method is implemented in the subclasses of `ArrayFileReader` and the implementation is dependent on the type of the array elements.

• Method `close` closes the `DataInputStream` created in the constructor.

### 3.3.2. ArrayFileWriter

`ArrayFileWriter` is an abstract class that describes the generic functionality to write data to an array file. This class is extended by other classes that provide implementation specific to array file of certain types. An instance of this class is returned by the `prepareToWrite` method of class `ArrayFile`, which accepts an `OutputStream` object as its parameter.

This class provides implementation for some of the generic methods, whose behavior doesn’t depend on the type of the array elements:

• The constructor of this class accepts an `OutputStream` as its parameter, creates a `DataOutputStream` from this output stream and writes the meta information about the array: the number of elements, and the type name of the array elements (e.g., “int” for an array file of integers).

• Several methods for writing data to the output stream. All of these writer methods ultimately delegate the actual writing operation to method `write` – this method writes the meta information about the array segment and is called from the `write` method implemented in the subclasses of `ArrayFileWriter` before writing the actual data of the array segment.

• Method `close` closes the `DataOutputStream` created in the constructor.

### 3.3.3. Array Segment

Classes `ArrayFileReader` and `ArrayFileWriter` read and write segments of arrays. A segment (also called array slice) is defined by a single `Range` object. `Range` object used to define an array slice may have a stride greater than one. This implies that array slices such as those shown on Figures 8a-8b can be used.

![Sequential Array Slice](image)

a) sequential array slice

![Strided Array Slice](image)

b) strided array slice

Figure 8. Possible array slices.
3.4. Subclasses of ArrayFile

There are nine subclasses of ArrayFile, which provide implementation specific to certain array element types. The following sections describe these subclasses.

3.4.1. DoubleArrayFile

Class DoubleArrayFile inherits from the abstract class ArrayFile and provides implementation for array file whose elements are of type double.

This class provides the following attributes and methods (in addition to those inherited from class ArrayFile):

- **BYTES_PER_ELEMENT** is a constant with the value of 8. This constant represents the number of bytes required to store a single element of a array of doubles
- **myArray** is an array of doubles. This object is used to store the array data.
- Two constructors: the default constructor initializes the attributes to predefined values, while the second constructor initializes the number of elements \( N \) and the array myArray to values specified as parameters.
- Method **getArray** returns the array myArray
- Method **setArray** sets the number of elements \( N \) and the array myArray to values specified as parameters.
- Method **getBytesPerElement** returns the value of constant BYTES_PER_ELEMENT
- Method **getElementClass** returns an instance of class Class<? that corresponds to the primitive type double. In Java this return value is specified as “double.class”.
- Method **prepareToRead** accepts an input stream and returns an instance of the inner class DoubleArrayFile.Reader.
- Method **prepareToWrite** accepts an output stream and returns an instance of the inner class DoubleArrayFile.Writer. An exception is thrown if this method is called before the array myArray is properly initialized.

Class DoubleArrayFile defines two inner classes: DoubleArrayFile.Reader and DoubleArrayFile.Writer.

Inner class DoubleArrayFile.Reader inherits from class ArrayFile.ArrayFileReader and provides functionality of reading elements of type double from an input stream. This class provides implementation of method readSegment which accepts 3 integer parameters:

- \( L \) – Segment’s lower element index.
- \( U \) – Segment’s upper element index.
- \( S \) – Segment’s element stride.

This method reads the next array segment from the input stream but only those elements that are members of the segment described by the 3 input parameters are stored in the myArray array. Reading of each element is performed by calling method
readDouble of class DataInputStream, which reads 8 bytes from the input stream and converts those bytes into a single double number.

For example, if the next array segment contains an array slice 0..17 and we invoke readSegment(3, 13, 1), then the following elements (indicated by dark-gray cells) will be stored in myArray:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

Figure 9. Reading an array slice

Inner class DoubleArrayFile.Writer inherits from class ArrayFile.ArrayFileWriter and provides functionality of writing elements of type double to an input stream. This class provides implementation of method write which accepts 3 integer parameters:

- \( L \) – Segment’s lower element index.
- \( U \) – Segment’s upper element index.
- \( S \) – Segment’s element stride.

This method first invokes the write method of the base class, which writes the meta information about the array segment. Then this method writes those elements of array myArray that fall into the segment described by the 3 input parameters to the output stream. Writing of each element is performed by calling method writeDouble of class DataOutputStream, which writes each double number as 8 bytes to the output stream.

### 3.4.2. FloatArrayFile

Class FloatArrayFile inherits from the abstract class ArrayFile and provides implementation for array file whose elements are of type float.

This class provides the same attributes and methods as class DoubleArrayFile, with the following differences:

- \( \text{BYTES\_PER\_ELEMENT} \) is a constant with the value of 4. This constant represents the number of bytes required to store a single element of type float.
- \( \text{myArray} \) is an array of floats. This object is used to store the array data.
- Method getElementClass returns an instance of class Class<?> that corresponds to the primitive type float. In Java this return value is specified as “float.class”.
- Method prepareToRead accepts an input stream and returns an instance of the inner class FloatArrayFile.Reader.
- Method prepareToWrite accepts an output stream and returns an instance of the inner class FloatArrayFile.Writer. An exception is thrown if this method is called before the array myArray is properly initialized.

Class FloatArrayFile defines two inner classes FloatArrayFile.Reader and FloatArrayFile.Writer, which inherit from inner classes ArrayFile.ArrayFileReader and ArrayFile.ArrayFileWriter, respectively. The functionality of FloatArrayFile.Reader and
FloatArrayFile.Writer is the same as the corresponding inner classes of DoubleArrayFile, with the only difference being that these two classes read and write 4 bytes per element and convert those bytes to/from an element of type float.

3.4.3. LongArrayFile

Class LongArrayFile inherits from the abstract class ArrayFile and provides implementation for array file whose elements are of type long.

This class provides the same attributes and methods as class DoubleArrayFile, with the following differences:

- `BYTES_PER_ELEMENT` is a constant with the value of 8. This constant represents the number of bytes required to store a single element of type long.
- `myArray` is an array of longs. This object is used to store the array data.
- Method `getElementClass` returns an instance of class `Class<?>` that corresponds to the primitive type long. In Java this return value is specified as “long.class”.
- Method `prepareToRead` accepts an input stream and returns an instance of the inner class LongArrayFile.Reader.
- Method `prepareToWrite` accepts an output stream and returns an instance of the inner class LongArrayFile.Writer. An exception is thrown if this method is called before the array `myArray` is properly initialized.

Class LongArrayFile defines two inner classes: LongArrayFile.Reader and LongArrayFile.Writer, which inherit from inner classes ArrayFile.ArrayFileReader and ArrayFile.ArrayFileWriter, respectively. The functionality of LongArrayFile.Reader and LongArrayFile.Writer is the same as the corresponding inner classes of DoubleArrayFile, with the only difference being that these two classes read/write 8 bytes and convert those bytes to/from an element of type long.

3.4.4. IntArrayFile

Class IntArrayFile inherits from the abstract class ArrayFile and provides implementation for array file whose elements are of type int.

This class provides the same attributes and methods as class DoubleArrayFile, with the following differences:

- `BYTES_PER_ELEMENT` is a constant with the value of 4. This constant represents the number of bytes required to store a single element of type int.
- `myArray` is an array of ints. This object is used to store the array data.
- Method `getElementClass` returns an instance of class `Class<?>` that corresponds to the primitive type int. In Java this return value is specified as “int.class”.
- Method `prepareToRead` accepts an input stream and returns an instance of the inner class IntArrayFile.Reader.
Method `prepareToWrite` accepts an output stream and returns an instance of the inner class `IntArrayFile.Writer`. An exception is thrown if this method is called before the array `myArray` is properly initialized.

Class `IntArrayFile` defines two inner classes: `IntArrayFile.Reader` and `IntArrayFile.Writer`, which inherit from inner classes `ArrayFile.ArrayFileReader` and `ArrayFile.ArrayFileWriter`, respectively. The functionality of `IntArrayFile.Reader` and `IntArrayFile.Writer` is the same as the corresponding inner classes of `DoubleArrayFile`, with the only difference being that these two classes read/write 4 bytes and convert those bytes to/from an element of type `int`.

### 3.4.5. ShortArrayFile

Class `ShortArrayFile` inherits from the abstract class `ArrayFile` and provides implementation for array file whose elements are of type `short`.

This class provides the same attributes and methods as class `DoubleArrayFile`, with the following differences:

- `BYTES_PER_ELEMENT` is a constant with the value of 2. This constant represents the number of bytes required to store a single element of type `short`.
- `myArray` is an array of `shorts`. This object is used to store the array data.
- Method `getElementClass` returns an instance of class `Class<?>` that corresponds to the primitive type `short`. In Java this return value is specified as "short.class".
- Method `prepareToRead` accepts an input stream and returns an instance of the inner class `ShortArrayFile.Reader`.
- Method `prepareToWrite` accepts an output stream and returns an instance of the inner class `ShortArrayFile.Writer`. An exception is thrown if this method is called before the array `myArray` is properly initialized.

Class `ShortArrayFile` defines two inner classes: `ShortArrayFile.Reader` and `ShortArrayFile.Writer`, which inherit from inner classes `ArrayFile.ArrayFileReader` and `ArrayFile.ArrayFileWriter`, respectively. The functionality of `ShortArrayFile.Reader` and `ShortArrayFile.Writer` is the same as the corresponding inner classes of `DoubleArrayFile`, with the only difference being that these two classes read/write 2 bytes and convert those bytes to/from an element of type `short`.

### 3.4.6. CharArrayFile

Class `CharArrayFile` inherits from the abstract class `ArrayFile` and provides implementation for array file whose elements are of type `char`.

This class provides the same attributes and methods as class `DoubleArrayFile`, with the following differences:

- `BYTES_PER_ELEMENT` is a constant with the value of 2. This constant represents the number of bytes required to store a single element of type `char`. 
• *myArray* is an array of *chars*. This object is used to store the array data.
• Method *getElementClass* returns an instance of class *Class<?>* that corresponds to the primitive type *char*. In Java this return value is specified as “char.class”.
• Method *prepareToRead* accepts an input stream and returns an instance of the inner class *CharArrayFile.Reader*.
• Method *prepareToWrite* accepts an output stream and returns an instance of the inner class *CharArrayFile.Writer*. An exception is thrown if this method is called before the array *myArray* is properly initialized.

Class *CharArrayFile* defines two inner classes: *CharArrayFile.Reader* and *CharArrayFile.Writer*, which inherit from inner classes *ArrayFile.ArrayFileReader* and *ArrayFile.ArrayFileWriter*, respectively. The functionality of *CharArrayFile.Reader* and *CharArrayFile.Writer* is the same as the corresponding inner classes of *DoubleArrayFile*, with the only difference being that these two classes read/write 2 bytes and convert those bytes to/from an element of type *char*.

### 3.4.7.ByteArrayFile

Class *ByteArrayFile* inherits from the abstract class *ArrayFile* and provides implementation for array file whose elements are of type *byte*.

This class provides the same attributes and methods as class *DoubleArrayFile*, with the following differences:
• *BYTES_PER_ELEMENT* is a constant with the value of 1. This constant represents the number of bytes required to store a single element of type *byte*.
• *myArray* is an array of *bytes*. This object is used to store the array data.
• Method *getElementClass* returns an instance of class *Class<?>* that corresponds to the primitive type *byte*. In Java this return value is specified as “byte.class”.
• Method *prepareToRead* accepts an input stream and returns an instance of the inner class *ByteArrayFile.Reader*.
• Method *prepareToWrite* accepts an output stream and returns an instance of the inner class *ByteArrayFile.Writer*. An exception is thrown if this method is called before the array *myArray* is properly initialized.

Class *ByteArrayFile* defines two inner classes: *ByteArrayFile.Reader* and *ByteArrayFile.Writer*, which inherit from inner classes *ArrayFile.ArrayFileReader* and *ArrayFile.ArrayFileWriter*, respectively. The functionality of *ByteArrayFile.Reader* and *ByteArrayFile.Writer* is the same as the corresponding inner classes of *DoubleArrayFile*, with the only difference being that these two classes read/write 1 byte and convert that byte to/from an element of type *byte*.
3.4.8. BooleanArrayFile

Class BooleanArrayFile inherits from the abstract class ArrayFile and provides implementation for array file whose elements are of type boolean.

This class provides the same attributes and methods as class DoubleArrayFile, with the following differences:

- `BYTES_PER_ELEMENT` is a constant with the value of 1. This constant represents the number of bytes required to store a single element of type boolean.
- `myArray` is an array of booleans. This object is used to store the array data.
- Method `getElementClass` returns an instance of class `Class<?>` that corresponds to the primitive type boolean. In Java this return value is specified as “boolean.class”.
- Method `prepareToRead` accepts an input stream and returns an instance of the inner class BooleanArrayFile.Reader.
- Method `prepareToWrite` accepts an output stream and returns an instance of the inner class BooleanArrayFile.Writer. An exception is thrown if this method is called before the array `myArray` is properly initialized.

Class BooleanArrayFile defines two inner classes: BooleanArrayFile.Reader and BooleanArrayFile.Writer, which inherit from inner classes ArrayFile.ArrayFileReader and ArrayFile.ArrayFileWriter, respectively. The functionality of BooleanArrayFile.Reader and BooleanArrayFile.Writer is the same as the corresponding inner classes of DoubleArrayFile, with the only difference being that these two classes read/write 1 byte and convert that byte to/from an element of type boolean.

3.4.9. ObjectArrayFile<T>

Class ObjectArrayFile<T> is a generic class that inherits from the abstract class ArrayFile and provides implementation for array file whose elements are of non-primitive types.

This class provides the following attributes and methods (in addition to those inherited from class ArrayFile):

- `myArray` is an array of generic type T. This object is used to store that array data.
- Two constructors: the default constructor initializes the attributes to predefined values, while the second constructor initializes the number of elements N and the array myArray to values specified as parameters.
- Method `getArray` returns the array myArray
- Method `setArray` sets the number of elements N and the array myArray to values specified as parameters.
- Method `getBytesPerElement` returns the value of -1 to indicate that the size in bytes of one array element may be different from the size of another element, i.e., the element size is not constant.
• Method `getElementClass` returns an instance of class `Class<?>` that corresponds to the actual type of the generic type `T`. For example, this method returns `Class<BigInteger>` when called on an instance of `ObjectArrayFile<BigInteger>`.

• Method `prepareToRead` accepts an input stream and returns an instance of the inner class `ObjectArrayFile.Reader`.

• Method `prepareToWrite` accepts an output stream and returns an instance of the inner class `ObjectArrayFile.Writer`. An exception is thrown if this method is called before the array `myArray` is properly initialized.

Class `ObjectArrayFile` defines two inner classes: `ObjectArrayFile.Reader` and `ObjectArrayFile.Writer`, which inherit from inner classes `ArrayFile.ArrayFileReader` and `ArrayFile.ArrayFileWriter`, respectively. The functionality of `ObjectArrayFile.Reader` and `ObjectArrayFile.Writer` is the same as the corresponding inner classes of `DoubleArrayFile`, but the implementation is more complicated.

Because `ObjectArrayFile<T>` is a generic class and works with almost any non-primitive data type, it wouldn’t be correct to assume that all elements of such array will have the same size. For example, different elements of an array of strings could have different lengths, and therefore different sizes in bytes.

Writing one array element to an output stream involves first writing the size of the element in bytes, and then writing the actual element data. Likewise, reading of an array element from an input stream involves first reading the size of the element in bytes, and then writing the actual element data.

The procedure of reading and writing an array element `X` to an output stream is exactly the same as reading and writing an element of an `ObjectMatrix<T>`, demonstrated by code snippets in Section 3.2.8 above.

Classes `ObjectInputStream` and `ObjectOutputStream` require that objects to be read using method `readObject()` or written using method `writeObject()` support the `java.io.Serializable` interface [6]. Therefore, the only restriction for types to be used with `ObjectArrayFile` is they must implement the `java.io.Serializable` interface.

The reasoning behind the decision of writing the size of each array element before writing the element itself is the same as described above in Section 3.2.8 – this allows for reduction of the running time and memory usage when skipping over array elements in the file.

### 3.5. MatrixFileManager

PDS includes a utility class that can be used to work with the different types of `MatrixFiles`. Class `MatrixFileManager` can be used to perform the following tasks on `MatrixFiles`:

• Split one `MatrixFile` into several `MatrixFiles`

• Join the several `MatrixFiles` into one `MatrixFile`

• Dump the contents of a `MatrixFile` in a text form
This class can be used interactively on the command prompt, or programmatically inside other programs. Because each MatrixFile includes the information on the element type, this class automatically uses the correct subclass of MatrixFile to perform its tasks. See the Developer’s guide section on the details on the usage of this utility.

3.6. ArrayFileManager

PDS includes a utility class that can be used to work with the different types of ArrayFiles. Class ArrayFileManager can be used to perform the following tasks on ArrayFiles:

- Split one ArrayFile into several ArrayFiles
- Join the several ArrayFiles into one ArrayFile
- Dump the contents of a ArrayFile in a text form

This class can be used interactively on the command prompt, or programmatically inside other programs. Because each ArrayFile includes the information on the element type, this class automatically uses the correct subclass of ArrayFile to perform its tasks. See the Developer’s guide section on the details on the usage of this utility.

4. Tests and Benchmarks

In order to assess the speed and correctness of PDS classes, three test programs have been developed.

4.1. Matrix Multiplication

If A is \(m \times n\) matrix and B is \(n \times p\) matrix, their product C is a \(m \times p\) matrix whose elements are defined as:

\[
C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j} \quad \text{where } 1 \leq i \leq m \text{ and } 1 \leq j \leq p
\]

There is one restriction on the dimension of the matrices A and B: the number of columns of matrix A should be equal to the number of rows of matrix B.

The most popular and simplest algorithm for calculating matrix product has the complexity \(O(n^3)\) and is implemented using three nested loops:

```java
for (int i = 0; i < m; i++) {
    for (int j = 0; j < p; j++) {
        c[i][j] = 0;
        for (int k = 0; k < n; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```
One thing about this algorithms that makes it easily parallelizable is that there are no sequential dependencies – to calculate element $C_{ij}$ we only need to know the elements in the $i$-th row of matrix $A$ and the elements in the $j$-th column of matrix $B$. We don’t need to know any element of matrix $C$ in order to be able to calculate any other element of $C$. Therefore, we can calculate any number of elements of matrix $C$ in parallel.

A cluster parallel program for matrix multiplication divides the calculation of the elements of matrix $C$ evenly among the available processors. Each processor is assigned a range of rows of matrix $A$ and a range of columns of matrix $B$ and calculates a corresponding subset of elements of matrix $C$. The rows of matrix $A$ and columns of matrix $B$ are assigned among the processors as evenly as possible. This results in balanced load to all processors. The following table shows the possible distribution of work among different number of processors:

<table>
<thead>
<tr>
<th># of processors $K$</th>
<th># of row ranges of $A$</th>
<th># of column ranges of $B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 10. Distribution of work in cluster parallel program for multiplication of matrices $A$ and $B$.

For example, if this cluster parallel program is used to calculate the product of matrix $A$ (100x300) and matrix $B$ (300x200) and is run on 4 processors, the processors would be allocated the following ranges:

<table>
<thead>
<tr>
<th>processor rank</th>
<th>row range of $A$</th>
<th>column range of $B$</th>
<th>patch of $C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0..49</td>
<td>0..99</td>
<td>0,0 .. 49,99</td>
</tr>
<tr>
<td>1</td>
<td>0.49</td>
<td>100..199</td>
<td>0,100 .. 49,199</td>
</tr>
<tr>
<td>2</td>
<td>50..99</td>
<td>0..99</td>
<td>50,0 .. 99,99</td>
</tr>
<tr>
<td>3</td>
<td>50..99</td>
<td>100..199</td>
<td>50,100 .. 99,199</td>
</tr>
</tbody>
</table>

Figure 11. Distribution of work in cluster parallel program running on 4 processors for multiplication of matrix $A$ (100x300) and matrix $B$ (300x200).

To decide which row range of $A$ and which column range of $B$ to read and to actually read the data, each processor runs the following code:
int world = Comm.world(); // the world communicator
int ws = world.size(); // world size (number of processors)
int rank = world.rank(); // this processor’s rank

int sq = (int)Math.sqrt(ws);
int dim1 = ws; // number of row ranges for matrix A
int dim2 = 1; // number of column ranges for matrix B
// find two factors of sq that are closest to each other:
while( sq > 1 ) {
    if ( ws % sq == 0 ) {
        dim1 = ws / sq;
        dim2 = sq;
        break;
    }
    sq--;
}

int[][] a = null; // matrix A
int[][] b = null; // matrix B
int omr=0, omc=0; // number of rows and columns of matrix C

IntMatrixFile imf1 = new IntMatrixFile();
// prepare to read matrix A:
MatrixFileReader reader1 = imf1.prepareToRead(
    new BufferedInputStream(new FileInputStream(args[0])));
int r = imf1.getRowCount(); // number of rows of A
int c = imf1.getColCount(); // number of columns of A
omr = r; // number of rows of C is the same as that of A

Range rangel1 = new Range(0,r-1).subrange(dim1, rank / dim2);
// read a row slice from matrix A:
reader1.readRowSlice(rangel1);
a = imf1.getMatrix(); // get the matrix
reader1.close();

IntMatrixFile imf2 = new IntMatrixFile();
// prepare to read matrix B:
MatrixFileReader reader2 = imf2.prepareToRead(
    new BufferedInputStream(new FileInputStream(args[1])));
r = imf2.getRowCount(); // number of rows of B

c = imf2.getColCount(); // number of columns of B

omc = c; // number of columns of C is that same as that of B

Range range2 = new Range(0, c-1).subrange(dim2, rank % dim2);

// read a column slice from matrix B:
reader2.readColSlice(range2);

b = imf2.getMatrix(); // get the matrix

reader2.close();

As the above code demonstrates, processors first calculate the number of row ranges to split the matrix A into and the number of column ranges to split the matrix B into. They do this by getting the world size ws (the number of processors executing the parallel program) and finding two factors (dim1 and dim2) of this number that are closest to each other. This is done to try to split matrices A and B into as equal number of slices as possible. Then each processor uses its rank to select which row slice of A and which column slice of B to read and uses the readRowSlice and readColSlice methods of class IntMatrixFile.Reader to read the data. Here, matrix A is split into dim1 equal row slices and processors with ranks 0..dim2-1 read the first row slice, processors dim2..2*dim2-1 read the next row slice, and so on. Matrix B is split into dim2 equal column slices and each processor reads the next available column slice, restarting at slice 0 if the number of processors exceeds the number of column slices.

When each processor finishes its work, it writes the calculated elements of matrix C in a separate file:

int[][] mx = new int[omr][omc]; // matrix C

// call the multiplication method:
multiply(a, b, mx, range1, range2);

IntMatrixFile imf3 = new IntMatrixFile(omr, omc, mx);

// prepare to write a part of matrix C
MatrixFileWriter writer = imf3.prepareToWrite(
    range1.lb()+"_"+range2.lb()+"-"+range1.ub()+"_"+range2.ub()
    +".piece"+rank)));

// write out the patch of C calculated by this processor:
writer.writePatch(range1, range2);
writer.close();

Each processor calculates a patch of elements of matrix C. This patch is defined by the “intersection” of the row slice of A and the column slice of B read by each
processor. Writing is performed by calling the `writePatch` method of class `IntMatrixFile.Writer`. The name of the file indicates which subset of elements of matrix \( C \) the file contains, e.g., `output.0_49-100_199.dat`. These separate files can then be joined together using the `MatrixManager` program.

### 4.2. K-Means Clustering

K-means clustering is a method for partitioning \( N \) observations into \( K \) clusters so that each observation belongs to the cluster with the nearest mean. This method is widely used in statistics and machine learning.

The input to the K-means clustering algorithm is a list of \( N \) data points, where each point consists of \( D \) coordinates. The algorithm finds \( K \) clusters in the data points and outputs the cluster centers.

The K-Means clustering algorithm is as follows:

1. Initialize the \( K \) cluster centers to be the first \( K \) data points
2. Initially, each data point is not assigned to any cluster
3. Repeat:
   4. Assign each data point to the cluster whose center is the closest using the Euclidean distance
   5. Recalculate the center point of each cluster
   6. Until none of the data points changed to a different cluster
   7. Output the coordinates of each cluster center

Here, the Euclidean distance between points \( A \) and \( B \) is calculated using the following formula:

\[
    d = \left( \sum_{i=1}^{D} (a_i - b_i)^2 \right)^{\frac{1}{2}}
\]

(1)

The general outline of the cluster parallel program is as follows:

The \( N \) points will be evenly divided among the \( Np \) processors of the cluster parallel computer. Thus each processor will be allocated a chunk of \( N/Np \) points. Each process will execute one or more rounds of computation, where each round consists of the following steps:

1. Assign each data point in the chunk to the closest cluster.
2. Perform an All-reduce operation using logical OR as reduction operator to determine if any data point changed to a different cluster. If no data point changed to a different cluster, then print out the results.
3. Perform an All-reduce operation using addition as reduction operator to propagate the cluster center data of each process to all other processes.
4. Recalculate the center points of each cluster.

Each process will have two \([K \times D+1]\) matrices of doubles. The first matrix (let’s denote this matrix as \( C \)) will contain the center point coordinates of each cluster. The second matrix of `doubles` (let’s denote this matrix as \( C_{nd} \)) will contain slightly different
data: the first $D$ columns of row $i$ of this matrix will contain the sum of the coordinates of all the points (in this processor’s chunk of points) that are assigned to cluster $i$. The last column of row $i$ will contain the number of points (in this processor’s chunk of points) that are assigned to cluster $i$. As each process calculates the closest cluster for each of the points in its chunk, it will subtract the coordinates of each point from the row in the matrix corresponding to the cluster that the point was previously assigned to and add the coordinates of the points to a row in the matrix corresponding to the new cluster to which the point is assigned.

Each process will also have a Boolean flag ‘changed’, which will be true, if at least 1 point changes clusters (i.e. is assigned from one cluster to another) during step 1 of a round.

After the 1st step of a round, matrix $C_{nd}$ of each process will contain the sum of coordinates of points clustered together (from the chunk of points assigned to each process).

The next step is to perform an all-Reduce operation using logical OR as the reduction operator and the Boolean flag ‘changed’ as data. As a result of this reduction, all the processes will know if any data point (across all the processes) changed to a different cluster. If there were no such points, then the program prints out the final coordinates of cluster center points and quits.

Then the contents of matrix $C_{nd}$ are copied to matrix $C$ and an all-Reduce operation is performed using addition as the reduction operator and matrix $C$ as the data buffer. After this step, the contents of matrix $C$ for all processes will be the same and will contain for each cluster the sum of coordinates of data points assigned to that cluster and the size of the cluster. Thus, step 4 of a round will consist of recalculating the center points of each cluster by just dividing each of the $D$ columns of matrix $C$ by the last column ($D+1$). Note that at this point matrix $C$ will contain exact coordinates of the center points of clusters (i.e. the sum of coordinates of all points that belong to common clusters, divided by the number of points in the cluster), whereas matrix $C_{nd}$ will contain only the sum of all the coordinates (not divided by the number of points in the clusters).

Once all the steps in a round are completed, the processes start the next round and keep computing in this manner until no data points change clusters.

We need to synchronize the parallel processes after step 1 because all the processes should complete calculating closest clusters for data points in their chunk before starting step 2. Since in step 2 we have an All-reduce operation, this operation acts as a barrier, as the operation is carried out only after all the processes complete step 1 and call the all-Reduce method. Thus we don’t need to take any additional actions to synchronize the parallel processes.

We don’t need to do any load-balancing because we divide the data points among all the parallel processes as evenly as possible, and the amount of computation performed by each process is roughly the same.

There are 3 arguments to the cluster parallel program: the number of clusters $K$, the name of the input file and the name of the output file. The input file is a $DoubleMatrixFile$ which is a $NxD$ matrix of doubles representing $N$ points with $D$ coordinates each. The input $DoubleMatrixFile$ consists of two segments – the first segments contains the first $K$ points needed by all processes, and the second segment contains the rest of the points. Each process will read the first segment completely, but
only a slice of rows from the second segment. The rows of the second segment are distributed among the available processes as evenly as possible:

```java
int world = Comm.world(); // the world communicator
int ws = world.size(); // world size (number of processors)
int rank = world.rank(); // this processor’s rank
int k = Integer.valueOf(args[0]); // number of clusters K
DoubleMatrixFile dmf = new DoubleMatrixFile();
MatrixFileReader reader = dmf.prepareToRead(
    new BufferedInputStream(new FileInputStream(args[1])));
// read the first K points from the first segment:
reader.readSegmentRowSlice(new Range(0, k-1));

int d = dmf.getColCount(); // number of coordinates D
int n = dmf.getRowCount(); // number of points N
Range chunk = new Range(0, n - 1).subrange(ws, rank);
// read the points for this process from the next segment:
reader.readSegmentRowSlice(chunk);
reader.close();
```

Since there are two segments in the input file, all processors first use method `readSegmentRowSlice` of class `DoubleMatrixFile.Reader` to read a slice of $K$ input points. Then each processor uses its rank to determine which of the row slices it should read and calls the `readSegmentRowSlice` method again to read a slice of input points to work with.

The algorithm stops when all of the points have been assigned to the correct clusters at which point all of the processes have the same coordinates of the cluster centers. When this happens, one of the processes will output the coordinates of the cluster centers to a text file.

### 4.3. Floyd’s Algorithm

Floyd’s algorithm is a well-known algorithm for solving the all-pairs shortest path problem. This problem usually identified as the problem of calculating the shortest path from every vertex of graph $G$ to every other vertex of $G$.

If the graph $G$ consists of $N$ vertices, the connectivity of the vertices can be represented by a $N \times N$ matrix $D$, where element $D_{r,c}$ represents the distance from vertex $r$ to vertex $c$. If these two vertices are not connected, then $D_{r,c}$ equals $\infty$. Matrix $D$ is called the distance matrix.

Floyd’s algorithm can be expressed using three nested loops [3]:

42
for i=1 to N do:
    for r=1 to N do:
        for c=1 to N do:
            $D_{r,c} = \min\{D_{r,c}, D_{r,i}+D_{i,c}\}$

It is easy to see that the complexity of Floyd’s algorithm is $O(n^3)$. Because of this complexity, a sequential program implementing Floyd’s algorithm may take a long time to complete for larger values of $N$. But we can significantly reduce the running time by parallelizing the algorithm.

In the cluster parallel program implementing Floyd’s algorithm, the rows of the distance matrix are distributed among the available processes of the cluster parallel computer as evenly as possible and each process performs calculations on its own row slice. This means, we parallelize the second loop in the algorithm outlined above.

The body of the inner-most loop calculates the minimum of $D_{r,c}$ and $D_{r,i}+D_{i,c}$. Therefore, each processes need to know the elements of the $i$-th row in order to be able to calculate this minimum.

We achieve this by having the process whose row slice includes the $i$-th row for the current value of $i$ broadcast the contents of the $i$-th row to all other processes. Therefore, we can summarize the cluster parallel program for Floyd’s algorithm as follows [3]:

for i=1 to N do:
    broadcast contents of the $i$-th row
    for r=1 to N do:
        for c=1 to N do:
            $D_{r,c} = \min\{D_{r,c}, D_{r,i}+D_{i,c}\}$

There are 2 arguments to this program: the name of the input file and the name of the output file. The input file is a DoubleMatrixFile that contains the $N \times N$ distance matrix of graph $G$. The rows of the distance matrix are distributed among the available processes as evenly as possible:

```java
int world = Comm.world(); // the world communicator
int size = world.size(); // world size (number of processors)
int rank = world.rank(); // this processor’s rank
// Prepare to read distance matrix from input file:
DoubleMatrixFile in = new DoubleMatrixFile();
MatrixFileReader reader = in.prepareToRead(
    new BufferedInputStream(new FileInputStream(args[0])));
double[][] d = in.getMatrix(); // the distance matrix
int n = d.length; // number of vertices N
```
// Divide distance matrix into equal row slices.
Range myrange = new Range(0, n - 1).subrange(size, rank);
int mylb = myrange.lb(); // range lower bound
int myub = myrange.ub(); // range upper bound

// Read just this process's row slice of the distance matrix.
reader.readRowSlice(myrange);
reader.close();

Each process uses its rank to determine which row slice to read and uses method
readRowSlice of class DoubleMatrixFile.Reader to read a row slice. Then each process
calculates the shortest distances for only those rows. Once the shortest distances have
been calculated, each process writes its row slice to a separate file, using method
writeRowSlice of class DoubleMatrixFile.Writer:

// Write distance matrix slice to a separate output file in each
// process:
DoubleMatrixFile out = new DoubleMatrixFile(n, n, d);
MatrixFileWriter writer = out.prepareToWrite(
    new BufferedOutputStream(new FileOutputStream(
        Files.fileForRank(outfile, rank))));
writer.writeRowSlice(myrange);
writer.close();

These separate files can then be joined together using the MatrixManager
program.
The algorithm and source code for this problem were taken from [3] and slightly
adjusted to work with the PDS API.

4.4. Common Design Patterns

As one can easily note from the above 3 sections, all 3 test programs follow a
similar design pattern. All 3 programs process matrices by distributing slices (chunks of
rows or chunks of columns) of matrices evenly among the available processors. The
processors perform computations on only those slices of matrices that were allocated to
them. When all the processors finish their computations, the results are either written out
to separate output files or are combined and then written out to one output file. Here is
the higher-level description of this pattern:
1. Create an instance of class DoubleMatrixFile (or matrix file of other type).
2. Call the prepareToRead method of that object, passing a BufferedInputStream
   which wraps a FileInputStream connected to the input file, to obtain an
   instance of MatrixFileReader.
3. Using the number of available processors and this processor’s rank, determine which matrix slice(s) should be read by this processor and create a corresponding Range object.
4. Read the data from file by passing the Range object from the previous step to one of the reader methods on the instance of MatrixFileReader obtained in step 2, such as readRowSlice, readColSlice, readPatch, or similar methods for segments. Close the reader object.
5. Call the getMatrix method on the DoubleMatrixFile object from step 1 to get the matrix containing the data read from the input file.
6. Perform calculations on the data read by this processor.
7. Combine the results produced by all processors in step 6, if necessary.
8. Call the prepareToRead method on an instance of DoubleMatrixFile, passing in a BufferedOutputStream which wraps a FileOutputStream connected to the output file, to obtain an instance of MatrixFileWriter.
9. Write the results by calling one of the writer methods on the instance of MatrixFileWriter from the previous step, such as writeRowSlice, writeColSlice, writePatch, or similar methods for segments. Close the writer object.

Note that this pattern assumes that the user knows ahead of time how the matrices are partitioned. In other words, this pattern assumes static partitioning.

Some problems may not be able to use static partitioning and may instead require dynamic partitioning. This could be the case for the problems when the time required to process each matrix element is different and not known in advance. For this kind of problems, a master-worker paradigm can be used. In this paradigm, a master processor is responsible for reading data and passing it on to worker processors, which perform the calculations and pass the results back to the master process. Below is a description of a design pattern that follows the master-worker paradigm and uses PDS.

Master process:
1. Create an instance of class DoubleMatrixFile (or matrix file of other type).
2. Call the prepareToRead method of that object, passing a BufferedInputStream which wraps a FileInputStream connected to the input file, to obtain an instance of MatrixFileReader.
3. Call the getMatrix method on the DoubleMatrixFile object from step 1 to get the matrix containing the data read from the input file.
4. Wait for a request from a worker process.
5. Upon receiving a request for data from a worker process:
   a. Read the data from file by calling one of the reader methods on the instance of MatrixFileReader obtained in step 2, such as readRowSlice, readColSlice, readPatch, or similar methods for segments.
   b. Send the read data to the worker process.
   c. If there is more data to be read, go to step 4.
6. Upon receiving a set of results from a worker process:
   a. Store the results.
   b. Go to step 4.
7. Combine the results produced by all worker processors.
8. Call the `prepareToRead` method on an instance of `DoubleMatrixFile`, passing in a `BufferedOutputStream` which wraps a `FileOutputStream` connected to the output file, to obtain an instance of `MatrixFileWriter`.

9. Write the results by calling one of the writer methods on the instance of `MatrixFileWriter` from the previous step, such as `writeRowSlice`, `writeColSlice`, `writePatch`, or similar methods for segments. Close the writer object.

Worker process:
1. Send a request for data to the master process.
2. If data is received:
   a. Perform calculations on the data.
   b. Send the results to the master process.
   c. Go to step 1.

This pattern allows for dynamic partitioning and distribution of input data and can be used to ensure balanced load for all compute nodes.

Note that while the above patterns have been described for matrix files, they work the same with the array files.

5. Test Runs

Each test program was implemented in three variations:
1. Sequential program not using PDS
2. Cluster parallel program not using PDS
3. Cluster parallel program using PDS

In order to make the three variations comparable with each other, variations 1 and 2 performed the same or equivalent operations as variation 1, but without making any calls to the PDS API. For example, if variation 3 included this piece of code:

```java
IntMatrixFile imf = new IntMatrixFile();
MatrixFileReader reader = imf.prepareToRead(new BufferedInputStream(
    new FileInputStream("matrix.dat")));
int r = imf.getRowCount(); // get # of rows
int c = imf.getColCount(); // get # of columns
reader.read();
int[][] A = imf.getMatrix();
reader.close();
// use matrix A
```

then variations 1 and 2 achieved the same effect with the following code:

```java
DataInputStream dis = new DataInputStream(new BufferedInputStream(
```
As these two fragments of code demonstrate, by using the PDS API, the code is compact, elegant and easy to follow. The PDS classes encapsulate the details of reading and writing matrix elements, allowing the programmer to concentrate on the actual problem to be solved.

The test programs were run on a Hybrid SMP cluster parallel computer, which consists of the following [2]:

- Frontend computer -- tardis.cs.rit.edu -- UltraSPARC-IIe CPU, 650 MHz clock, 512 MB main memory
- 10 backend computers -- dr00 through dr09 -- each with two AMD Opteron 2218 dual-core CPUs, four processors, 2.6 GHz clock, 8 GB main memory
- 1-Gbps switched Ethernet backend interconnection network
- Aggregate 104 GHz clock, 80 GB main memory

Each variation of each test program was run a minimum of three times, and the smallest running time among the multiple runs was taken for comparison purposes.

5.1. Results

5.1.1 Matrix Multiplication

The matrix multiplication test run was run with two input data sets:

- Multiplication of a [1000x800] matrix with a [800x900] matrix, resulting in a [1000x900] matrix. Problem size for this input set is: $N=0.7 \, G$. 


• Multiplication of a [4000x4000] matrix with another [4000x4000] matrix, resulting in a [4000x4000] matrix. Problem size for this input set is: \( N=64G \).

Two variations of this test program were each run with the above input sets: variation 2 and variation 3 (see section 5 above).

In order to prevent the external factors such as the NFS load from influencing the results, all of the input data files were copied to the local disks of all of the cluster nodes before running this test program.

The following table summarizes the running times and other parameters of the test program:

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>T</th>
<th>Spdup</th>
<th>Effic</th>
<th>EDSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7G</td>
<td>seq</td>
<td>8719</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.7G</td>
<td>1</td>
<td>8610</td>
<td>1,013</td>
<td>1,013</td>
<td></td>
</tr>
<tr>
<td>0.7G</td>
<td>2</td>
<td>4255</td>
<td>2,049</td>
<td>1,025</td>
<td>-0,012</td>
</tr>
<tr>
<td>0.7G</td>
<td>4</td>
<td>2602</td>
<td>3,351</td>
<td>0,838</td>
<td>0,07</td>
</tr>
<tr>
<td>0.7G</td>
<td>8</td>
<td>1468</td>
<td>5,939</td>
<td>0,742</td>
<td>0,052</td>
</tr>
<tr>
<td>0.7G</td>
<td>10</td>
<td>1305</td>
<td>6,681</td>
<td>0,668</td>
<td>0,057</td>
</tr>
<tr>
<td>0.7G-PDS</td>
<td>seq</td>
<td>8719</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.7G-PDS</td>
<td>1</td>
<td>8396</td>
<td>1,038</td>
<td>1,038</td>
<td></td>
</tr>
<tr>
<td>0.7G-PDS</td>
<td>2</td>
<td>4266</td>
<td>2,044</td>
<td>1,022</td>
<td>0,016</td>
</tr>
<tr>
<td>0.7G-PDS</td>
<td>4</td>
<td>2586</td>
<td>3,372</td>
<td>0,843</td>
<td>0,077</td>
</tr>
<tr>
<td>0.7G-PDS</td>
<td>8</td>
<td>1505</td>
<td>5,793</td>
<td>0,724</td>
<td>0,062</td>
</tr>
<tr>
<td>0.7G-PDS</td>
<td>10</td>
<td>1199</td>
<td>7,272</td>
<td>0,727</td>
<td>0,048</td>
</tr>
<tr>
<td>64G</td>
<td>seq</td>
<td>1106687</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64G</td>
<td>1</td>
<td>949546</td>
<td>1,165</td>
<td>1,165</td>
<td></td>
</tr>
<tr>
<td>64G</td>
<td>2</td>
<td>489582</td>
<td>2,26</td>
<td>1,13</td>
<td>0,031</td>
</tr>
<tr>
<td>64G</td>
<td>4</td>
<td>243092</td>
<td>4,553</td>
<td>1,138</td>
<td>0,008</td>
</tr>
<tr>
<td>64G</td>
<td>8</td>
<td>127135</td>
<td>8,705</td>
<td>1,088</td>
<td>0,01</td>
</tr>
<tr>
<td>64G</td>
<td>10</td>
<td>99322</td>
<td>11,142</td>
<td>1,114</td>
<td>0,005</td>
</tr>
<tr>
<td>64G-PDS</td>
<td>seq</td>
<td>1106687</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64G-PDS</td>
<td>1</td>
<td>878225</td>
<td>1,26</td>
<td>1,26</td>
<td></td>
</tr>
<tr>
<td>64G-PDS</td>
<td>2</td>
<td>471507</td>
<td>2,347</td>
<td>1,174</td>
<td>0,074</td>
</tr>
<tr>
<td>64G-PDS</td>
<td>4</td>
<td>233593</td>
<td>4,738</td>
<td>1,184</td>
<td>0,021</td>
</tr>
<tr>
<td>64G-PDS</td>
<td>8</td>
<td>122009</td>
<td>9,071</td>
<td>1,134</td>
<td>0,016</td>
</tr>
<tr>
<td>64G-PDS</td>
<td>10</td>
<td>102612</td>
<td>10,785</td>
<td>1,079</td>
<td>0,019</td>
</tr>
</tbody>
</table>

Here, the columns are as follows:

\( N \) - problem size: 0.7G – 1st input data set, variation 2; 0.7G-PDS – 1st input data set, variation 3; 64G – 2nd input data set, variation 2; 64G-PDS – 2nd input data set, variation 3.

\( K \) – number of processors; “seq” refers to sequential program.

\( T \) – minimal running time from 3 runs.

\( Spdup \) – parallel program speedup.
Effic – parallel program efficiency.
EDSF – parallel program’s EDSF.

The following charts present the above data in graphical form:
As the above table and charts show, the parallel program demonstrates good speedup for the problem size $N=0.7G$ and somewhat above-ideal speedup for the problem size $N=64G$. This is due to the fact the for the smaller problem size, the amount
of time spent on performing Input/Output is comparable to the amount of time spent on performing actual calculations. In this case, the more processors participate in computation, the less is the amount of computing by each processor (each processor has to calculate smaller slice of rows), and thus the amount of time spent on actual calculations is dominated by the amount of time spent on performing I/O. Therefore, the speedup curve deviates more from the ideal speedup curve as the number of processors increases.

On the other hand, in the case with the problem size \(N=64G\), the amount of time spent on performing actual calculations is many times more than the amount of time spent on performing I/O. This fact, combined with sophisticated just-in-time (JIT) compiler optimizations used by the modern JVMs result in above-ideal speedup [3].

Another important point to notice here is the fact that for both of the input data sets, the running times for 2 variations are roughly the same. This means, that using the PDS API in the parallel program doesn’t affect the running time of the program negatively.

### 5.1.2. K-Means Clustering.

The K-Means Clustering test run was run with two input data sets:

- **Test1:** \(D = 2\) dimensions, \(N = 180,000\) data points, and \(K = 100\) clusters
- **Test3:** \(D = 6\) dimensions, \(N = 600,000\) data points, and \(K = 55\) clusters

Both of these datasets were taken from [7].

Two variations of this test program were each run with the above input sets: variation 2 and variation 3 (see section 5 above).

In order to prevent the external factors such as the NFS load from influencing the results, all of the input data files were copied to the local disks of all of the cluster nodes before running this test program.

The following table summarizes the running times and other parameters of the test program:

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>T</th>
<th>Spdup</th>
<th>Effic</th>
<th>EDSF</th>
<th>Devi</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1</td>
<td>seq</td>
<td>119364</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test1</td>
<td>1</td>
<td>122578</td>
<td>0,974</td>
<td>0,974</td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>test1</td>
<td>2</td>
<td>71402</td>
<td>1,672</td>
<td>0,836</td>
<td>0,165</td>
<td>19%</td>
</tr>
<tr>
<td>test1</td>
<td>3</td>
<td>66395</td>
<td>1,798</td>
<td>0,599</td>
<td>0,312</td>
<td>3%</td>
</tr>
<tr>
<td>test1</td>
<td>4</td>
<td>50256</td>
<td>2,375</td>
<td>0,594</td>
<td>0,213</td>
<td>2%</td>
</tr>
<tr>
<td>test1</td>
<td>6</td>
<td>33552</td>
<td>3,558</td>
<td>0,593</td>
<td>0,128</td>
<td>12%</td>
</tr>
<tr>
<td>test1</td>
<td>8</td>
<td>23374</td>
<td>5,107</td>
<td>0,638</td>
<td>0,075</td>
<td>0%</td>
</tr>
<tr>
<td>test1</td>
<td>10</td>
<td>11128</td>
<td>10,726</td>
<td>1,073</td>
<td>-0,01</td>
<td>1%</td>
</tr>
<tr>
<td>test1-PDS</td>
<td>seq</td>
<td>119364</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test1-PDS</td>
<td>1</td>
<td>121650</td>
<td>0,981</td>
<td>0,981</td>
<td>22%</td>
<td></td>
</tr>
<tr>
<td>test1-PDS</td>
<td>2</td>
<td>70768</td>
<td>1,687</td>
<td>0,843</td>
<td>0,163</td>
<td>5%</td>
</tr>
<tr>
<td>test1-PDS</td>
<td>3</td>
<td>67591</td>
<td>1,766</td>
<td>0,589</td>
<td>0,333</td>
<td>3%</td>
</tr>
<tr>
<td>test1-PDS</td>
<td>4</td>
<td>49815</td>
<td>2,396</td>
<td>0,599</td>
<td>0,213</td>
<td>3%</td>
</tr>
<tr>
<td>test1-PDS</td>
<td>6</td>
<td>33575</td>
<td>3,555</td>
<td>0,593</td>
<td>0,131</td>
<td>4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>test1-PDS</td>
<td>8</td>
<td>22727</td>
<td>5,252</td>
<td>0,657</td>
<td>0,071</td>
<td>12%</td>
</tr>
<tr>
<td>test1-PDS</td>
<td>10</td>
<td>11603</td>
<td>10,287</td>
<td>1,029</td>
<td>-0,005</td>
<td>1%</td>
</tr>
<tr>
<td>test3</td>
<td>seq</td>
<td>319327</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test3</td>
<td>1</td>
<td>311595</td>
<td>1,025</td>
<td>1,025</td>
<td></td>
<td>10%</td>
</tr>
<tr>
<td>test3</td>
<td>2</td>
<td>185976</td>
<td>1,717</td>
<td>0,859</td>
<td>0,194</td>
<td>12%</td>
</tr>
<tr>
<td>test3</td>
<td>3</td>
<td>115595</td>
<td>2,762</td>
<td>0,921</td>
<td>0,056</td>
<td>21%</td>
</tr>
<tr>
<td>test3</td>
<td>4</td>
<td>107607</td>
<td>2,968</td>
<td>0,742</td>
<td>0,127</td>
<td>2%</td>
</tr>
<tr>
<td>test3</td>
<td>6</td>
<td>69374</td>
<td>4,603</td>
<td>0,767</td>
<td>0,067</td>
<td>2%</td>
</tr>
<tr>
<td>test3</td>
<td>8</td>
<td>53247</td>
<td>5,997</td>
<td>0,75</td>
<td>0,052</td>
<td>5%</td>
</tr>
<tr>
<td>test3-PDS</td>
<td>seq</td>
<td>319327</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test3-PDS</td>
<td>1</td>
<td>311950</td>
<td>1,024</td>
<td>1,024</td>
<td></td>
<td>9%</td>
</tr>
<tr>
<td>test3-PDS</td>
<td>2</td>
<td>185151</td>
<td>1,725</td>
<td>0,862</td>
<td>0,187</td>
<td>13%</td>
</tr>
<tr>
<td>test3-PDS</td>
<td>3</td>
<td>115098</td>
<td>2,774</td>
<td>0,925</td>
<td>0,053</td>
<td>23%</td>
</tr>
<tr>
<td>test3-PDS</td>
<td>4</td>
<td>108308</td>
<td>2,948</td>
<td>0,737</td>
<td>0,13</td>
<td>1%</td>
</tr>
<tr>
<td>test3-PDS</td>
<td>6</td>
<td>70862</td>
<td>4,506</td>
<td>0,751</td>
<td>0,073</td>
<td>3%</td>
</tr>
<tr>
<td>test3-PDS</td>
<td>8</td>
<td>53433</td>
<td>5,976</td>
<td>0,747</td>
<td>0,053</td>
<td>4%</td>
</tr>
<tr>
<td>test3-PDS</td>
<td>10</td>
<td>44218</td>
<td>7,222</td>
<td>0,722</td>
<td>0,046</td>
<td>1%</td>
</tr>
</tbody>
</table>

For the explanation of the table columns, see Section 5.1.1 above.
The following charts present the above data in graphical form:
As the above table and charts show, the parallel program demonstrates speedups for both input data sets, although the speedups are far from ideal. This is due to the fact that this test program performs message passing after every round of the algorithm. But the general trend is that the running times decrease, as the number of processors increase.

Another important point to notice here is the fact that for both of the input data sets, the running times for 2 variations are very close to each other. This means, that using the PDS API in the parallel program doesn’t affect the running time of the program negatively.

**5.1.3. Floyd’s Algorithm**

The test program implementing Floyd’s algorithm was run with the one input data set: a [4000x4000] distance matrix for a graph consisting of 4000 vertices. Problem size for this input set is: \(N=64G\). This dataset was taken from [3].

Two variations of this test program were each run with the above input set: variation 2 and variation 3 (see section 5 above).

In order to prevent the external factors such as the NFS load from influencing the results, all of the input data files were copied to the local disks of all of the cluster nodes before running this test program.

The following table summarizes the running times and other parameters of the test program:

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>T</th>
<th>Spdup</th>
<th>Effic</th>
<th>EDSF</th>
<th>Devi</th>
</tr>
</thead>
<tbody>
<tr>
<td>64G</td>
<td>seq</td>
<td>614362</td>
<td></td>
<td></td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>64G</td>
<td>1</td>
<td>587473</td>
<td>1,046</td>
<td>1,046</td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>64G</td>
<td>2</td>
<td>295097</td>
<td>2,082</td>
<td>1,041</td>
<td>0,005</td>
<td>29%</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>--------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td>64G</td>
<td>4</td>
<td>149873</td>
<td>4,099</td>
<td>1,025</td>
<td>0,007</td>
<td>16%</td>
</tr>
<tr>
<td>64G</td>
<td>6</td>
<td>100846</td>
<td>6,092</td>
<td>1,015</td>
<td>0,006</td>
<td>14%</td>
</tr>
<tr>
<td>64G</td>
<td>8</td>
<td>75500</td>
<td>8,137</td>
<td>1,017</td>
<td>0,004</td>
<td>0%</td>
</tr>
<tr>
<td>64G</td>
<td>10</td>
<td>63645</td>
<td>9,653</td>
<td>0,965</td>
<td>0,009</td>
<td>1%</td>
</tr>
<tr>
<td>64G-PDS</td>
<td>seq</td>
<td>614362</td>
<td></td>
<td></td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>64G-PDS</td>
<td>1</td>
<td>564362</td>
<td>1,089</td>
<td>1,089</td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>64G-PDS</td>
<td>2</td>
<td>276787</td>
<td>2,22</td>
<td>1,11</td>
<td>-0,019</td>
<td>4%</td>
</tr>
<tr>
<td>64G-PDS</td>
<td>4</td>
<td>143133</td>
<td>4,292</td>
<td>1,073</td>
<td>0,005</td>
<td>2%</td>
</tr>
<tr>
<td>64G-PDS</td>
<td>6</td>
<td>97553</td>
<td>6,298</td>
<td>1,05</td>
<td>0,007</td>
<td>1%</td>
</tr>
<tr>
<td>64G-PDS</td>
<td>8</td>
<td>72371</td>
<td>8,489</td>
<td>1,061</td>
<td>0,004</td>
<td>2%</td>
</tr>
<tr>
<td>64G-PDS</td>
<td>10</td>
<td>61410</td>
<td>10,004</td>
<td>1</td>
<td>0,01</td>
<td>0%</td>
</tr>
</tbody>
</table>

See Section 5.1.1 above for the explanation of the table columns. The following charts present the above data in graphical form:
As the above table and charts show, the parallel program demonstrates slightly above-ideal speedup for the given problem size. This is due to the fact that the amount of time spent on performing actual calculations is many times more than the amount of time spent on performing I/O. This fact, combined with sophisticated just-in-time (JIT) compiler optimizations used by the modern JVMs result in above-ideal speedup [3]. Correspondingly, the running time decreases as the number of processors increases.

Another important point to notice here is the fact that for both of the input data sets, the running times for 2 variations are roughly the same, with the variation using the PDS API demonstrating slightly better results. This means, that using the PDS API in the parallel program doesn’t affect the running time of the program negatively.

6. Future Work

While PDS in its current shape is a fully functional and useful addition to the PJ library, there is still room for improvement.

One of the missing features of PDS at this time is the ability to automatically copy the necessary input files to the local disks of the cluster nodes. Thus one of the possible improvements is to integrate PDS with PJ job scheduler so that when a parallel program is scheduled to run on one or more cluster nodes, the corresponding input files are automatically partitioned and distributed to the local disks of those nodes. Currently PJ job scheduler is responsible for allocating enough compute nodes for a parallel program to run on and for launching the parallel jobs simultaneously on those nodes. If PDS is integrated with PJ job scheduler, before launching the parallel jobs the job scheduler could “tell” the PDS which compute nodes have been allocated for a parallel program and
“ask” PDS to partition and copy necessary input files to local disks of compute nodes. Moreover, when a parallel program is run again with the same parameters, the PJ job scheduler together with PDS could detect this and make sure the same compute nodes that were allocated to that parallel program the last time it was executed are allocated again, thus saving the hassle of distributing the input files to the local disks. These features were considered at the beginning stages of this project but it was decided not to include them because of time constraints.

Another topic for future investigation is enabling PDS to work with the existing parallel file systems, so that the users have the choice of using PDS API to work with the files that are stored on an existing parallel file system. The PDS in that case would act as a layer on top of the existing parallel file system that provides a simple Java API for working with that parallel file system.

7. Conclusions

This project resulted in the development of an extension to the I/O section of the Parallel Java library.

The new PDS API incorporates a set of classes to work with matrices and arrays of different element types.

At the top of class hierarchy are abstract classes *MatrixFile* and *ArrayFile*, which define the interface for working with matrix files and array files of different types. Separate sub-classes exist for the 8 primitive types and one generic data type.

The new API encapsulates the details of low-level I/O operations and allows the programmer concentrate on the actual problem at hand. This also results in cleaner and more elegant code.

Manager programs were also developed for splitting, joining and dumping the contents of matrix and array files.

Three test programs were developed and run on a cluster parallel computer to assess the impact of using the new API on the running time of parallel programs. As the results of these test programs demonstrated, using the new API doesn’t affect the running time of parallel programs negatively at all.

Taking all this into consideration, the new PDS API can be used by programmers in writing parallel programs.

While working on this project, I learned about the concept of a parallel file system and some of the existing parallel file systems. I gained an in-depth knowledge of inner workings of the Parallel Java library. I learned how an API is designed so that it’s flexible and easy to use. I also learned what to pay attention to when writing applications to test an API. Overall I feel this project has been a very interesting and useful experience and I am confident that the knowledge I gained while working on this project will prove useful in my future career.
Appendix A. Developer’s guide.

A.1. Matrix Files

Class *MatrixFile* is an abstract file that defines the interface for an object for reading and writing a matrix to or from a file. There are 9 concrete (non-abstract) subclasses of *MatrixFile* which provide objects for reading and writing matrices of different types to or from files. Usage examples below are provided for class *IntMatrixFile*, but are very similar for other subclasses of *MatrixFile* as well.

Class *IntMatrixFile* provides an object for reading or writing a matrix of integers from or to a file. The matrix containing the actual data to read into or write from is a separate object, which can be specified as an argument to the constructor or the *setMatrix()* method:

```java
int[][] matrix = new int[10][20]; // 10 rows, 20 columns
IntMatrixFile imf = new IntMatrixFile(10, 20, matrix);
```

or:

```java
int[][] matrix = new int[10][20]; // 10 rows, 20 columns
IntMatrixFile imf = new IntMatrixFile();
imf.setMatrix(10, 20, matrix);
```

To write the matrix to a file, call the *prepareToWrite()* method, supplying the output stream to write as the parameter. For better results, it is recommended that the parameter passed to this method is an instance of class *BufferedOutputStream*. The return type of this method is *MatrixFile.MatrixFileWriter*, and this method returns an instance of class *IntMatrixFile.Writer* which is a subclass of *MatrixFile.MatrixFileWriter*. To write the matrix or segments of the matrix to the output stream, call the methods of this writer object. When finished writing, close the writer:

```java
int[][] matrix = new int[10][20]; // 10 rows, 20 columns
// fill in the matrix...
IntMatrixFile imf = new IntMatrixFile(10, 20, matrix);
try {
    FileOutputStream fos = new FileOutputStream("matrix10x20.dat");
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    MatrixFile.MatrixFileWriter wr = imf.prepareToWrite(bos);
    wr.write();
    wr.close();
}
catch (Exception e) {
```
To read the data from a file into the matrix, call the \texttt{prepareToRead()} method, supplying the input stream to read from as the parameter. For better results, it is recommended that the parameter passed to this method is an instance of class \texttt{BufferedInputStream}. The return type of this method is \texttt{MatrixFile.MatrixFileReader}, and this method returns an instance of class \texttt{IntMatrixFile.Reader} which is a subclass of \texttt{MatrixFile.MatrixFileReader}. To read the matrix or segments of the matrix from the input stream, call the methods of this reader object. When finished writing, close the reader:

```java
int[][] matrix = new int[10][20]; // 10 rows, 20 columns
IntMatrixFile imf = new IntMatrixFile(10, 20, matrix);
try {
    FileInputStream fis = new FileInputStream("matrix10x20.dat");
    BufferedInputStream bis = new BufferedInputStream(fis);
    MatrixFile.MatrixFileReader rdr = imf.prepareToRead(bis);
    rdr.read();
    rdr.close();
    matrix = imf.getMatrix();
} catch (Exception e) {
    // handle any exceptions
}
```

As the above code demonstrates, to get the matrix from the \texttt{IntMatrixFile} object, call the \texttt{getMethod()} method.

The following getter methods can be used to retrieve information about the matrix:

- \texttt{getRowCount()} – returns the number of rows of the matrix
- \texttt{getColCount()} – returns the number of columns of the matrix
- \texttt{getBytesPerElement()} – returns the size of one matrix element in bytes

\textbf{A.1.1. MatrixFileWriter}

As mentioned above, calling the \texttt{prepareToWrite()} method on a \texttt{IntMatrixFile} object returns an instance of class \texttt{IntMatrixFile.Writer} which is a subclass of the abstract class \texttt{MatrixFile.MatrixFileWriter}. The return type of this method is actually \texttt{MatrixFile.MatrixFileWriter}. This writer object provides the following methods for writing the whole matrix or segments of the matrix to the output stream:

- \texttt{write()} - writes the elements in all rows and columns of the matrix to the output stream.
writeRowSlice(Range r) - writes the elements contained in a “slice” of rows of the matrix to the output stream. A “slice” here refers to a consecutive range of rows, defined by the parameter passed in.

writeColSlice(Range r) - writes the elements contained in a “slice” of columns of the matrix to the output stream. A “slice” here refers to a consecutive range of columns, defined by the parameter passed in.

writePatch(Range rr, Range cr) - writes the elements contained in a “patch” of the matrix to the output stream. A “patch” here refers to the intersection of a consecutive range of rows with a consecutive range of columns, defined by the parameters passed in.

```java
int[][] matrix = new int[40][50]; //40 rows, 50 columns
IntMatrixFile imf = new IntMatrixFile(40, 50, matrix);
Range rowSlice = new Range(10, 15); // rows 10 through 15
Range colSlice = new Range(0, 50, 3); // columns 0, 3, 6, 9, ..., 48
Range patchRows = new Range(2, 20, 2); //rows 2,4,6,...,20
Range patchCols = new Range(1, 3); //columns 1 through 3
try {
    FileOutputStream fos = new FileOutputStream("matrix40x50.dat");
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    MatrixFile.MatrixFileWriter wr = imf.prepareToWrite(bos);

    //write elements (10,0) through (15, 49):
    // (10,0), (10,1), (10,2), (10,3), ..., (10,49),
    // (11,0), (11,1), (11,2), (11,3), ..., (11,49),
    // (12,0), (12,1), (12,2), (12,3), ..., (12,49),
    // ...………………………………
    // (15,0), (15,1), (15,2), (15,3), ..., (15,49),
    wr.writeRowSlice(rowSlice);

    //write all elements in columns 0, 3, 6, 9, ..., 48:
    // (0,0), (0,3), (0,6), (0,9), ..., (0,48),
    // (1,0), (1,3), (1,6), (1,9), ..., (1,48),
    // (2,0), (2,3), (2,6), (2,9), ..., (1,48),
    // ...………………………………
    // (39,0), (39,3), (39,6), (39,9), ..., (39,48),
    wr.writeColSlice(colSlice);

    //write elements (2,1), (2,2), (2,3),
    // (4,1), (4,2), (4,3),
```
Each write operation writes a new segment to the output stream, therefore, in the above code, 4 segments are written.

### A.1.2. MatrixFileReader

As mentioned above, calling the `prepareToRead()` method on a `IntMatrixFile` object returns an instance of class `IntMatrixFile.Reader` which is a subclass of the abstract class `MatrixFile.MatrixFileReader`. The return type of this method is actually `MatrixFile.MatrixFileReader`. This reader object provides several methods for reading data from the input stream, organized into two groups. The first group of reader methods read all of the segments contained in the file:

- **read()** - reads all of the elements from all segments of the matrix from the input stream.
- **readRowSlice(Range rowSlice)** - reads the elements from all segments of the matrix from the input stream, but only stores elements contained in a “slice” of rows of the matrix. A “slice” here refers to a consecutive range of rows, defined by the parameter passed in.
- **readColSlice(Range colSlice)** - reads the elements from all segments of the matrix from the input stream, but only stores elements contained in a “slice” of columns of the matrix. A “slice” here refers to a consecutive range of columns, defined by the parameter passed in.
- **readPatch(Range patchRows, Range patchCols)** - reads the elements from all segments of the matrix from the input stream, but only stores elements contained in a “patch” of the matrix to the output stream. A “patch” here refers to the intersection of a consecutive range of rows with a consecutive range of columns, both passed in as parameters.
The second group of reader methods read only the next matrix element segment contained on the file:

- `readSegment()` - reads all of the elements from the next segment of the matrix from the input stream.
- `readSegmentRowSlice(Range rowSlice)` - reads the elements from the next segment of the matrix from the input stream, but only stores elements contained in a “slice” of rows of the matrix, defined by the parameter passed in.
- `readSegmentColSlice(Range colSlice)` - reads the elements from the next segment of the matrix from the input stream, but only stores elements contained in a “slice” of columns of the matrix, defined by the parameter passed in.
- `readSegmentPatch(Range patchRows, Range patchCols)` - reads the elements from the next segment of the matrix from the input stream, but only stores elements contained in a “patch” of the matrix to the output stream, passed in as parameters

The following example assumes reading from the file written to in code segment in Section A.1.1 above:

```java
int[][] matrix = new int[40][50]; // 40 rows, 50 columns
IntMatrixFile imf = new IntMatrixFile(40, 50, matrix);
Range rowSlice = new Range(8, 12); // rows 8 through 12
Range colSlice = new Range(9, 40, 3); // columns 9, 12, 15, ..., 39
Range patchRows = new Range(2, 10, 2); // rows 2,4,6,...,10
Range patchCols = new Range(2, 3); // columns 2 through 3
try {
    FileInputStream fis = new FileInputStream("matrix40x50.dat");
    BufferedInputStream bis = new BufferedInputStream(fis);
    MatrixFile.MatrixFileReader rdr = imf.prepareToRead(bis);

    // read elements (10,0) through (12, 49):
    // (10,0), (10,1), (10,2), (10,3), ..., (10,49),
    // (11,0), (11,1), (11,2), (11,3), ..., (11,49),
    // (12,0), (12,1), (12,2), (12,3), ..., (12,49)
    rdr.readSegmentRowSlice(rowSlice);

    // read all elements in columns 9, 12, 15, ..., 39:
    // (0,9), (0,12), (0,15), (0,18), ..., (0,39),
    // (1,9), (1,12), (1,15), (1,18), ..., (1,39),
    // (2,9), (2,12), (2,15), (2,18), ..., (1,39),
    // ...........................................
```
// (39,9), (39,12), (39,15), (39,18), ..., (39,39),
rdr.readSegmentColSlice(colSlice);

// read elements  (2,2), (2,3),
//                (4,2), (4,3),
//                (6,2), (6,3),
//                ....................
//                (10,2),(10,3),
rdr.readSegmentPatch(patchRows, patchCols);

// read all matrix elements, (0,0) through (39,49):
rdr.read();

// close the reader:
rdr.close();
}
catch (Exception e) {
    // handle any exceptions
}

A.1.3. MatrixFileManager

Class MatrixFileManager is a utility class that provides several static methods to join several matrix files into one, split one matrix file into several matrix files, and dump the elements of a matrix file to an output stream in text form. This class can be used both programmatically and interactively.

Below is a description of the methods that can be used programmatically:

- getMatrixFileByElementClass(Class<?> elementClass) – creates and returns an instance of one of the subclasses of MatrixFile, based on the element class passed in as a parameter.
- getMatrixFileByElementClassName(String elementClassName) – creates and returns an instance of one of the subclasses of MatrixFile, based on the element class name passed in as a parameter.

// returns an instance of DoubleMatrixFile:
MatrixFile mf =
    MatrixFileManager.getMatrixFileByElementClass(double.class);
// returns an instance of ObjectMatrixFile<String>:
MatrixFile mf =
    MatrixFileManager.getMatrixFileByElementClassName("java.lang.String");
• `dump(String inFileName, String outFileName)` – dumps the contents of the matrix stored in file `inFileName` to text file named `outFileName`.

• `dump(MatrixFile mf, OutputStream os)` – dumps the contents of the matrix file `mf` to the output stream `os`.

```java
//create and populate matrix:
int[][] matrix = new int[20][30];
IntMatrixFile imf = new IntMatrixFile(20, 30, matrix);
//dump the matrix file to text file ‘matrix20x30.dat’:
FileOutputStream fos = new FileOutputStream("matrix20x30.dat");
MatrixFileManager.dump(imf, fos);
fos.close();
```

• `join(String[] inFileNames, String outFileName)` – creates a new matrix by joining the matrices stored in different files and writes the new matrix to a new matrix file.

```java
String[] inFileNames = new String[] { "num_0,0-9,19.dat", "num_10,0-19,19.dat", "num_20,0-29,29.dat"};
String outFileName = "num_0,0-29,29.dat";
MatrixFileManager.join(inFileNames, outFileName);
```

• `split(MatrixFile mf, int numberOfPieces, int rowStride, int colStride, String outFileNamePrefix)` – splits the `MatrixFile` into several pieces and writes each piece into separate file. Parameters `numberOfPieces`, `rowStride` and `colStride` control the split pattern:
  o `rowStride=0, colStride=0`: matrix is split evenly into `numberOfPieces` row slices with the stride of 1 and each slice is written to a different file.
  o `rowStride=0, colStride=1`: matrix is split evenly into `numberOfPieces` column slices, where all slices have a stride equal to `numberOfPieces` and slice `i` begins at column `i`, `0 ≤ i < numberOfPieces`. Each slice is written to a different file.
  o `rowStride=0, colStride>1`: matrix is split evenly into `X` column slices where each column slice’s width equals to `colStride` (`X = ceil[number of columns of matrix / colStride]`). These slices are written to `numberOfPieces` different files: slices `0, X, 2X,...` are written to the first file, slices `1, X+1, 2X+1,...` are written to the second file, etc.
  o `rowStride=1, colStride=0`: matrix is split into `numberOfPieces` even row slices, where all slices have a stride equal to `numberOfPieces` and slice `i` begins at row `i`, `0 ≤ i < numberOfPieces`.
  o `rowStride>1, colStride=0`: matrix is split evenly into `X` row slices where each row slice’s width equals to `rowStride` (`X = ceil[number of rows of matrix / colStride]`). These slices are written to `numberOfPieces` different files: slices
are written to the first file, slices \( I, X+I, 2X+I, \ldots \) are written to the second file, etc.

- \( rowStride > 0, colStride > 0 \): matrix is split evenly into \( X \) row slices where each slice’s width equals to \( rowStride \) \( (X = \lceil \text{number of rows of matrix} / \text{colStride} \rceil) \) and into \( Y \) column slices where each column slice’s width equals to \( colStride \) \( (Y = \lceil \text{number of columns of matrix} / \text{colStride} \rceil) \). The intersections of these row slices and column slices create \( Z \) patches \( (Z = X \times Y) \). These patches are written to \( \text{numberOfPieces} \) different files: patches \( 0, X, 2X, \ldots \) are written to the first file, patches \( I, X+I, 2X+I, \ldots \) are written to the second file, etc.

```java
//create and populate matrix:
int[][] matrix = new int[20][30];
IntMatrixFile imf = new IntMatrixFile(20, 30, matrix);
//matrix is split into two [10x30] matrices which are written to files
//split_matrix.piece0 and split_matrix.piece1 :
MatrixFileManager.split(imf, 2, 0, 0, “split_matrix.piece”);

//matrix is split into three [20x10] matrices which are written to files
//split_matrix.piece0, split_matrix.piece1 and
//split_matrix.piece2 :
MatrixFileManager.split(imf, 3, 0, 10, “split_matrix.piece”);

//matrix is split into four [10x15] matrices which are written to files
//split_matrix.piece0, split_matrix.piece1, split_matrix.piece2 and
//split_matrix.piece3 :
MatrixFileManager.split(imf, 4, 10, 15, “split_matrix.piece”);

A.1.3.1. Using MatrixFileManager Interactively

Class MatrixFileManager can also be used interactively to perform the same tasks as described above.

The command to execute this class is:

```
java MatrixFileManager <command> <command-option> [ <command-option> ]
```

where \( <command> \) is one of the following:

- \( \text{dump} \)
- \( \text{join} \)
- \( \text{split} \)

and \( <command-option> \) is one of the following:

- \( -f <\text{input file name(s)}> \)
One or more the input file name. For the **dump** and **split** command this option should supply only one file name which contains the matrix file to be dumped or split, correspondingly. For the **join** command this option can specify several file names which contain matrix files to be joined

- **-o <output file name>**
  This option specifies the name of the output file. Used by all three commands to write the results. For the **dump** command, specifying “--” as the output file name causes the matrix file be dumped onto the standard output. For the **split** command, this option specifies the output file name prefix.

- **-n <number of pieces>**
  Supplied to the **split** command to specify the number of matrix files to split the original matrix file into.

- **-r <row stride>**
  Supplied to the **split** command to specify the row stride of the pieces.

- **-c <column stride>**
  Supplied to the **split** command to specify the column stride of the pieces.

The program prints an error message if the command name is invalid or if any of the required command options are missing.

**Dump the contents of file matrix10x20.dat to standard output:**

```
java MatrixFileManager dump -f matrix10x20.dat  -o --
```

**Join three matrix files into one:**

```
java MatrixFileManager join -f num_0,0-9,19.dat num_10,0-19,19.dat
num_20,0-29,29.dat  -o num_0,0-29,29.dat
```

**Split the [20x30] matrix contained in file matrix_20x30.dat into two [10x30] matrices and write the pieces to files split_matrix.piece0 and split_matrix.piece1:**

```
java MatrixFileManager split -f matrix_20x30.dat  -o split_matrix.piece
-n 2
```

**Split the [20x30] matrix contained in file matrix_20x30.dat into three [20x10] matrices and write the pieces to files split_matrix.piece0, split_matrix.piece1 and split_matrix.piece2:**

```
java MatrixFileManager split -f matrix_20x30.dat  -o split_matrix.piece
-n 3 -c 10
```

**Split the [20x30] matrix contained in file matrix_20x30.dat into four [10x15] matrices and write the pieces to files split_matrix.piece0, split_matrix.piece1, split_matrix.piece2 and split_matrix.piece3:**

```
A.2. Array Files

Class *ArrayFile* is an abstract file that defines the interface for an object for reading and writing an array to or from a file. There are 9 concrete (non-abstract) subclasses of *ArrayFile* which provide objects for reading and writing arrays of different types to or from files. Usage examples below are provided for class *IntArrayFile*, but are very similar for other subclasses of *ArrayFile* as well.

Class *IntArrayFile* provides an object for reading or writing an array of integers from or to a file. The array containing the actual data to read into or write from is a separate object, which can be specified as an argument to the constructor or the *setArray()* method:

```
int[] array = new int[20]; //20 elements
IntArrayFile iaf = new IntArrayFile(20, array);
```

or:

```
int[] array = new int[20]; //20 elements
IntArrayFile iaf = new IntArrayFile();
iaf.setArray(20, array);
```

To write the array to a file, call the *prepareToWrite()* method, supplying the output stream to write as the parameter. For better results, it is recommended that the parameter passed to this method is an instance of class *BufferedOutputStream*. The return type of this method is *ArrayFile.ArrayFileWriter*, and this method returns an instance of class *IntArrayFile.Writer* which is a subclass of *ArrayFile.ArrayFileWriter*. To write the array or segments of the array to the output stream, call the methods of this writer object. When finished writing, close the writer:

```
int[] array = new int[20]; //20 elements
//fill in the array...
IntArrayFile iaf = new IntArrayFile(20, array);
try {
    FileOutputStream fos = new FileOutputStream("array20.dat");
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    ArrayFile.ArrayFileWriter wr = iaf.prepareToWrite(bos);
    wr.write();
    wr.close();
}
```
catch (Exception e) {
    //handle any exceptions
}

To read the data from a file into the array, call the `prepareToRead()` method, supplying the input stream to read from as the parameter. For better results, it is recommended that the parameter passed to this method is an instance of class `BufferedInputStream`. The return type of this method is `ArrayFile.ArrayFileReader`, and this method returns an instance of class `IntArrayFile.Reader` which is a subclass of `ArrayFile.ArrayFileReader`. To read the array or segments of the array from the input stream, call the methods of this reader object. When finished writing, close the reader:

```java
int[] array = new int[20]; //20 elements
IntArrayFile iaf = new IntArrayFile(20, array);
try {
    FileInputStream fis = new FileInputStream("array20.dat");
    BufferedInputStream bis = new BufferedInputStream(fis);
    ArrayFile.ArrayFileReader rdr = iaf.prepareToRead(bis);
    rdr.read();
    rdr.close();
    array = iaf.getArray();
} catch (Exception e) {
    //handle any exceptions
}
```

As the above code demonstrates, to get the array from the `IntArrayFile` object, call the `getMethod()` method. The following getter methods can be used to retrieve information about the array:

- `getElementCount()` – returns the number of elements of the array
- `getBytesPerElement()` – returns the size of one array element in bytes

### A.2.1. ArrayFileWriter

As mentioned above, calling the `prepareToWrite()` method on a `IntArrayFile` object returns an instance of class `IntArrayFile.Writer` which is a subclass of the abstract class `ArrayFile.ArrayFileWriter`. The return type of this method is actually `ArrayFile.ArrayFileWriter`. This writer object provides the following methods for writing the whole array or segments of the array to the output stream:

- `write()` - writes all elements of the array to the output stream.
• writeSlice(Range r) - writes the elements contained in a “slice” of the array to the output stream. A “slice” here refers to a consecutive range of elements, defined by the parameter passed in.

```java
int[] array = new int[40]; // 40 elements
IntArrayFile iaf = new IntArrayFile(40, array);
Range slice1 = new Range(10, 15); // elements 10 through 15
Range slice2 = new Range(0, 40, 3); // elements 0, 3, 6, 9, …, 39
try {
    FileOutputStream fos = new FileOutputStream("array40.dat");
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    ArrayFile.ArrayFileWriter wr = iaf.prepareToWrite(bos);

    // write elements 10 through 15:
    wr.writeSlice(slice1);

    // write elements 0, 3, 6, 9, …, 39:
    wr.writeSlice(slice2);

    // write all array elements, 0 through 39:
    wr.write();

    // close the writer:
    wr.close();
} catch (Exception e) {
    // handle any exceptions
}
```

Each write operation writes a new segment to the output stream, therefore, in the above code, 3 segments are written.

### A.2.2. ArrayFileReader

As mentioned above, calling the prepareToRead() method on a IntArrayFile object returns an instance of class IntArrayFile.Reader which is a subclass of the abstract class ArrayFile.ArrayFileReader. The return type of this method is actually ArrayFile.ArrayFileReader. This reader object provides several methods for reading data from the input stream, organized into two groups. The first group of reader methods read all of the segments contained in the file:

- read() - reads all of the elements from all segments of the array from the input stream.
- **readSlice(Range slice)** - reads the elements from all segments of the array from the input stream, but only stores elements contained in a “slice” of the array. A “slice” here refers to a consecutive range of elements, defined by the parameter passed in.

The second group of reader methods read only the next array element segment contained in the file:
- **readSegment()** - reads all of the elements from the next segment of the array from the input stream.
- **readSegmentSlice(Range slice)** - reads the elements from the next segment of the array from the input stream, but only stores elements contained in a “slice” of the array, defined by the parameter passed in.

The following example assumes reading from the file written to in code segment in Section A.2.1 above:

```java
int[] array = new int[40]; // 40 elements
IntArrayFile iaf = new IntArrayFile(40, array);
Range slice1 = new Range(8, 15); // elements 8 through 15
Range slice2 = new Range(9, 40, 3); // elements 9, 12, 15, …, 39
try {
    FileInputStream fis = new FileInputStream("array40.dat");
    BufferedInputStream bis = new BufferedInputStream(fis);
    ArrayFile.ArrayFileReader rdr = iaf.prepareToRead(bis);

    // read elements 8 through 15:
    rdr.readSegmentSlice(slice1);

    // read elements 9, 12, 15, …, 39:
    rdr.readSegmentSlice(slice2);

    // read all array elements, 0 through 39:
    rdr.read();

    // close the reader:
    rdr.close();
} catch (Exception e) {
    // handle any exceptions
}
```
A.2.3. ArrayFileManager

Class *ArrayFileManager* is a utility class that provides several static methods to join several array files into one, split one array file into several array files, and dump the elements of an array file to an output stream in text form. This class can be used both programmatically and interactively.

Below is a description of the methods that can be used programmatically:

- `getArrayFileByElementClass(Class<?> elementClass)` – creates an returns an instance of one of the subclasses of `ArrayFile`, based on the element class passed in as a parameter.
- `getArrayFileByElementClassName(String elementClassName)` – creates an returns an instance of one of the subclasses of `ArrayFile`, based on the element class name passed in as a parameter.
- `dump(String inFileName, String outFileName)` – dumps the contents of the array stored in file `inFileName` to text file named `outFileName`.
- `dump(ArrayFile af, OutputStream os)` – dumps the contents of the array file `af` to the output stream `os`.
- `join(String[] inFileNames, String outFileName)` – creates a new array by joining the arrays stored in different files and writes the new array to a new array file.

```java
//returns an instance of DoubleArrayFile:
ArrayFile af =
    ArrayFileManager.getArrayFileByElementClass(double.class);

//returns an instance of ObjectArrayFile<String>:
ArrayFile af = ArrayFileManager.getArrayFileByElementClassName("java.lang.String");

• dump(String inFileName, String outFileName) – dumps the contents of the array stored in file `inFileName` to text file named `outFileName`.
• dump(ArrayFile af, OutputStream os) – dumps the contents of the array file `af` to the output stream `os`.

//create and populate array:
int[] array = new int[40]; //40 elements
IntArrayFile iaf = new IntArrayFile(40, array);
//dump the array file to text file ‘array40.dat’:
FileOutputStream fos = new FileOutputStream("array40.dat");
ArrayFileManager.dump(iaf, fos);
fos.close();

• join(String[] inFileNames, String outFileName) – creates a new array by joining the arrays stored in different files and writes the new array to a new array file.

String outFileName = "num_0-29.dat";
ArrayFileManager.join(inFileNames, outFileName);
```
• `split(ArrayFile af, int numberOfPieces, int stride, String outFileNamePrefix)` — splits the `ArrayFile` into several pieces and writes each piece into separate file. Parameters `numberOfPieces` and `stride` control the split pattern:
  - `stride=0`: array is split evenly into `numberOfPieces` slices with the stride of 1 and each slice is written to a different file.
  - `stride=1`: array is split into `numberOfPieces` even slices, where all slices have a stride equal to `numberOfPieces` and slice `i` begins at element `i`, `0 ≤ i < numberOfPieces`. Each slice is written to a different file.
  - `stride>1`: array is split evenly into `X` slices where each slice’s width equals to `stride` (`X = \text{ceil}[(\text{number of element in array} / \text{stride})]`). These slices are written to `numberOfPieces` different files: slices 0, `X`, `2X`,... are written to the first file, slices 1, `X+1`, `2X+1`,... are written to the second file, etc.

```java
//create and populate array:
int[] array = new int[40]; //40 elements
IntArrayFile iaf = new IntArrayFile(40, array);

//array is split into two 20-element arrays which are written to files
//split_array.piece0 and split_array.piece1:
ArrayFileManager.split(iaf, 2, 0, "split_array.piece");

//array is split into four 10-element arrays which are written to files
//split_array.piece0, split_array.piece1, split_array.piece2 and
//split_array.piece3:
ArrayFileManager.split(iaf, 4, 10, "split_array.piece");
```

### A.2.3.1. Using ArrayFileManager Interactively

Class `ArrayFileManager` can also be used interactively to perform the same tasks as described above.

The command to execute this class is:

```
java ArrayFileManager <command> <command-option> [<command-option>...] 
```

where `<command>` is one of the following:

- `dump`
- `join`
- `split`

and `<command-option>` is one of the following:

- `-f <input file name(s)>`

One or more the input file name. For the `dump` and `split` command this option should supply only one file name which contains the array file to be dumped.
or split, correspondingly. For the join command this option can specify several file names which contain array files to be joined

- -o <output file name>

This option specifies the name of the output file. Used by all three commands to write the results. For the dump command, specifying “--“ as the output file name causes the array file be dumped onto the standard output. For the split command, this option specifies the output file name prefix.

- -n <number of pieces>

Supplied to the split command to specify the number of array files to split the original array file into.

- -s <stride>

Supplied to the split command to specify the stride of the pieces.

The program prints an error message if the command name is invalid or if any of the required command options are missing.

Dump the contents of file array20.dat to standard output:

```java
java ArrayFileManager dump -f array20.dat  -o --
```

Join three array files into one:

```java
java ArrayFileManager join -f num_0-19.dat num_10-19.dat
  num_20-29.dat  -o num_0-29.dat
```

Split the 40-element array contained in file array_40.dat into two 20-element arrays and write the pieces to files split_array.piece0 and split_array.piece1:

```java
java ArrayFileManager split -f array_40.dat  -o split_array.piece -n 2
```

Split the 40-element array contained in file array_40.dat into four 10-element arrays and write the pieces to files split_array.piece0, split_array.piece1, split_array.piece2 and split_array.piece3:

```java
java ArrayFileManager split -f array_40.dat  -o split_array.piece -n 4
  -s 10
```
References


http://www.cs.rit.edu/~ark/pj.shtml


http://java.sun.com/javase/6/docs/api/

http://www.cs.rit.edu/~ark/736/project01/project01.shtml
