ORNL/TM-2017/429

# PLATO: Parallel Load Assignment Tool
## A Parallel Workload Partitioner for Particle-based Methods

Gregory Muzyn
Sudip K Seal

**October 24, 2017**

**ORNL/TM-2017/429**

Computer Science and Mathematics Division

# PLATO: Parallel Load Assignment Tool
A Parallel Workload Partitioner for Particle-based Methods

Gregory Muzyn
Sudip K. Seal

Date Published: August, 2017

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

## CONTENTS

LIST OF FIGURES . . . . . iv
1. Introduction . . . . . v
2. Background . . . . . vi
3. Implementation . . . . . vi
    3.1 Communication . . . . . vi
    3.2 Defining Virtual Boundaries . . . . . vi
        3.2.1 Geometry-based approach . . . . . vii
        3.2.2 Load-based approach . . . . . vii
        3.2.3 Pre-fetching . . . . . viii
4. Tests and Results . . . . . ix
    4.1 Batching/Non-Batching Communication . . . . . ix
    4.2 Pre-fetching . . . . . ix
    4.3 Point Attributes . . . . . x
    4.4 Partitioning Method . . . . . xi
    4.5 Conclusions and Recommendations . . . . . xi

## LIST OF FIGURES

1 Geometric-based partitioning. . . . . vii
2 Load-based partitioning. . . . . vii
3 Geometric-based partitioning with pre- fetching value of 0.1. . . . . ix
4 Load-based partitioning with pre-fetching value of 0.1. . . . . ix
5 Comparison of runtime of batching and non- batching communication. . . . . x
6 Comparison of speedup of batching and non- batching communication. . . . . x
7 Comparison of runtime of different values of $\epsilon$. . . . . x
8 Comparison of speedup of different values of $\epsilon$. . . . . x
9 Comparison of runtime with different numbers of attributes. . . . . xi
10 Comparison of speedup with different numbers of attributes. . . . . xi
11 Comparison of runtime of different partitioning methods. . . . . xii
12 Comparison of speedup of different partitioning methods. . . . . xii

## ABSTRACT

A key first step in many large-scale high-performance scientific computing applications is the distribution of the computational domain across all participating processors of a parallel computing platform such as a shared memory machine, cluster or a tightly-coupled supercomputer. The primary goal of such a partitioning step is to ensure that the computational work performed by each process remains load-balanced throughout the execution of the parallel application. In this project, a general-purpose parallel partitioning tool, code-named PLATO, was designed and implemented. PLATO assumes that the workload is a set of points in an n-dimensional metric space. These points are abstractions of computational units that may represent point-like computational units such as celestial objects in astrophysics, atoms in materials sciences and molecules in biology/chemistry, or may represent more abstract computational units such as grid points in finite difference methods, grid elements in finite element methods or n-dimensional points in the feature space of a host of other applications.

PLATO supports several key and useful capabilities, namely: (a) multiple load partitioning strategies, (b) support for arbitrary user-defined processor topology, (c) pre-fetching data from neighboring processors based on user-defined parameters, and (d) support for arbitrary number and types of attributes associated with each point in the computational domain. PLATO is implemented in the C programming language using Message Passing Interface (MPI) for inter-processor communications and its parallel performance tested on up to 48 processors of a shared memory machine. This report includes a detailed performance study of PLATO using parallel speedup and iso-efficiency as the two main metrics of parallel performance. In addition, a user manual and DOXYGEN generated code documentation have been included for completeness.

## 1. Introduction

Data sets consisting of a large number of points often represent the computational domains of a broad class of large-scale spatial data analyses applications. Such analyses are commonly performed on parallel computers to decrease the analyses turnaround times. One of the key first steps for any large-scale parallel spatial data analysis algorithm is the efficient mapping of the n-dimensional point space to the rank space of the participating processors. Thus, a software capable of distributing the data into partitions that guarantees that the workload of subsequent computations on each processor remains balanced is very useful to numerous parallel spatial data analysis applications.

PLATO (Parallel Load Assignment Tool) is a general purpose, application-independent tool for efficiently distributing any point data set amongst a set of processors arranged in any topology. By design, PLATO supports an arbitrary number of attributes per point and an arbitrary processor topology, or the spatial mapping of processors into n-dimensional space. Pre-fetching from neighboring processors and different load assignment methods are also supported. In this report, we present a detailed performance analysis of PLATO across multiple use cases and discuss how the implementation incorporates user-defined parameters to operate as a general- purpose tool.

## 2. Background

Atom probe tomography (APT) is an example of an application that requires parallel computers to effectively analyze the voluminous atomic data that is gathered by atom probes. In APT, the position of atoms can be represented as abstract points in a three-dimensional geometric space. Materials scientists often analyze these massive APT data sets, consisting of several million to billions of atoms, to identify and track nano-scale clusters of atom species as the material sample undergoes thermodynamic changes. Cluster detection algorithms rely on computing spherical range queries to determine the number and types of atoms within a radial distance from each atom in the data set. For these algorithms to operate efficiently in parallel, the atoms need to be distributed evenly amongst the processors while maintaining spatial locality to minimize inter-processors communications. PLATO is designed to provide the very first parallel partitioning step if the point data set across the various processors.

## 3. Implementation

PLATO is implemented using the C programming language and Message Passing Interface (MPI) for inter-processor communication. Once the data is read into PLATO, it defines its partition boundaries based on the data and user through MPI. Upon completion, each process contains only those points whose positions fall within the processor's boundaries in the data space.

### 3.1 Communication

PLATO uses MPI to communicate by sending point data sending to receiving processes, each process responsible for computations on a single sub-division of the point data set. Each point's data is sent asynchronously using non-blocking point-to-point communication methods, which allows PLATO to compute the destination process of other points or receive points from other processes while waiting on the prior sent points to complete. In our case, this method is preferred over synchronous blocking methods as PLATO makes use of the time between asynchronous calls to perform other tasks.

Additionally, PLATO implements batched sends and receives, or sending/receiving a variable number of particles per MPI call. The default batch size PLATO is close to the highest possible and changes dynamically depending on the size of each communication. However, a user can override the value of the batch size, and while the lowest value of 1 is allowed (which would send a single particle with each MPI call), higher values generally result in lower runtimes. Due to the heterogeneous nature of different computing platforms, the user may wish to override the default value as the optimal batching amount may vary depending on the platform being used.

### 3.2 Defining Virtual Boundaries

Virtual boundaries are the boundaries along each coordinate dimension of the spatial sub-division mapped to each processor. Virtual boundaries are simply the maximum and minimum value of each dimension that belong to each partition or process. If a given particle's position is greater than the minimum and less than the maximum value for all dimensions of that partition, it is placed in that partition. It is important to note that a particle may, depending on the desired partitioning result and particle distribution, be placed into multiple partitions.

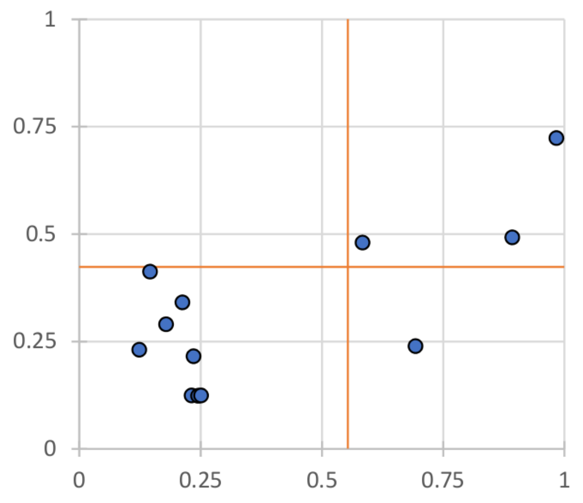

**Figure 1. Geometric-based partitioning.**

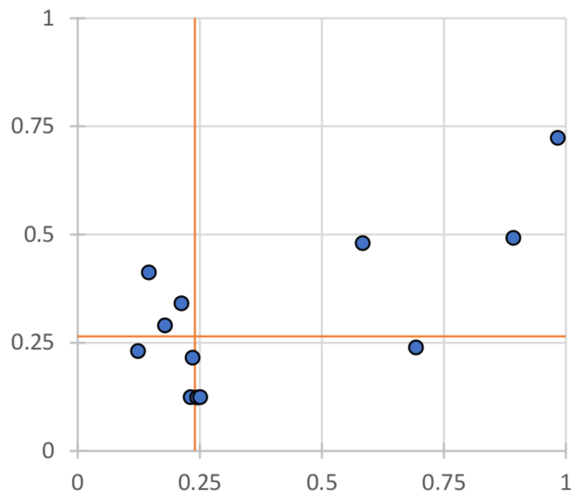

**Figure 2. Load-based partitioning.**

PLATO supports two different strategies to map the data space onto to the rank space of the processors.

### 3.2.1 Geometry-based approach

Geometry-based boundaries are often used when the distribution of points is not relevant to the computation or when the position of the points is of greater importance than the load on each processor. These boundaries are simple to compute; the span $s$ of each dimension is divided evenly by the number of processors along that dimension in the rank space as specified by the processor topology. The value of the upper and lower boundaries, $v_u$ and $v_\ell$, respectively, in the 1 dimensional case for a process of position $r$ is:

$$v_u = \left(\frac{s}{p} + M\right)(r + 1) + \epsilon, \qquad v_\ell = \left(\frac{s}{p} + M\right) r + \epsilon \tag{1}$$

where $p$ is the total number of processors and $m$ is the minimum value of all points. For the $d^{th}$ dimension, the upper and lower boundaries are computed using equation (1) $d$ times. Given 2-dimensional spatial data for 12 points, the resulting boundaries are represented in Fig. 1, where each processor requires $d$ upper and lower boundaries. In this case, the points are not very evenly distributed; the resulting partitions contain a maximum of 8 points and a minimum of 1 point each for a total of 12.

### 3.2.2 Load-based approach

When the points being partitioned have an uneven distribution, a simple geometric partitioning strategy can potentially have high variance in the number of points local to each partition. To solve this problem, we implemented the option to create the boundaries for each partition using the median of the set of all point positions. The median is recursively found on smaller sets of these positions to create the number of boundaries needed; in the one-dimensional case, a total of $p$ processors requires $p - 1$ medians to be computed. To compute the medians, we used a median of medians algorithm implemented in parallel. The algorithm does not find the exact median, but instead yields an accurate estimate. The algorithm is as follows for one-dimensional data:

1. Each process sorts its array of local position values using quicksort.
2. Each process reports the median of its local array, or the local median, to the root process, which inserts each value into a heap and broadcasts the median of the collected medians to all other processors. This is the global median.
3. Each process now does the following:
   (a) If size of local position values array = 0, stop.
   (b) If local median < global median, remove values less than the local median from consideration and go to step 2.
   (c) If global median > local median, remove values greater than the local median from consideration and go to step 2.
   (d) If global median = local median, stop.
4. Once all processes have stopped, the root process will find the global median for the last time and broadcast it to all other processors; this is the result of the algorithm.

Quicksort is used for each process to sort its local array because there are no guarantees on what the values will be, likely giving the average complexity of $n \log n$. A heap is used because by the root process to store the medians as heap insertion has a complexity of $\log n$. Thus, the resulting complexity of the approximate median finding algorithm for finding all necessary medians is:

$$O\left(\frac{n}{p}\log\frac{n}{p} + \log p\left[\log\frac{n}{p}\log\left(\log\frac{n}{p}\right) + \log p + comm\right]\right) \tag{2}$$

as the median must be computed $\log p$ times recursively. Figure 2 represents a 2-dimensional set of points partitioned with the Load-Balancing algorithm. By finding the approximate median of the X and Y values of the points, PLATO creates boundaries that result in an almost equal number of points belonging to each partition and process. In this case, each process now has 3 points, which is a more evenly distributed result than using geometric partitioning.

### 3.2.3 Pre-fetching

For both geometry-based and load-based strategies of data partitioning, a pre-fetching value represented by epsilon ($\epsilon$) can be specified. $\epsilon$ is a positive real number which is added to the upper boundary and subtracted from the lower boundary of each process. Figure 3 and Fig. 4 represent a pre-fetching value of 0.1 applied to the geometry-based and load-based boundary methods respectively, and show how pre-fetching increases the overall amount of communication and affects load-balancing. Points that are within the dashed line on Fig. 3 and Fig. 4 are sent to multiple processes. For example, the point at (0.584, 0.480) belongs to only one partition as shown in Fig. 1, but belongs to all four partitions after the pre-fetching value is applied to the boundaries as shown in Fig. 3.

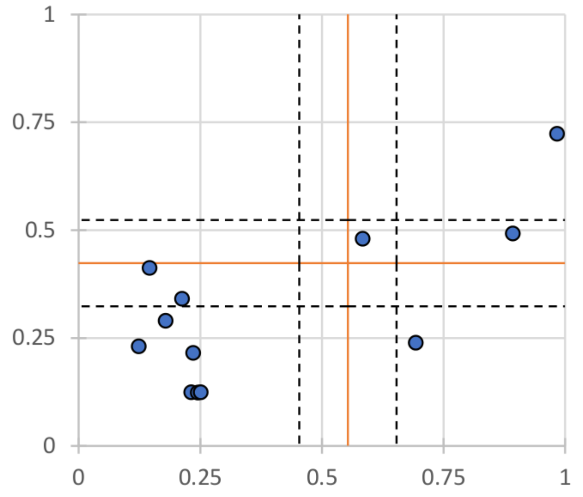

**Figure 3. Geometric-based partitioning with pre- fetching value of 0.1.**

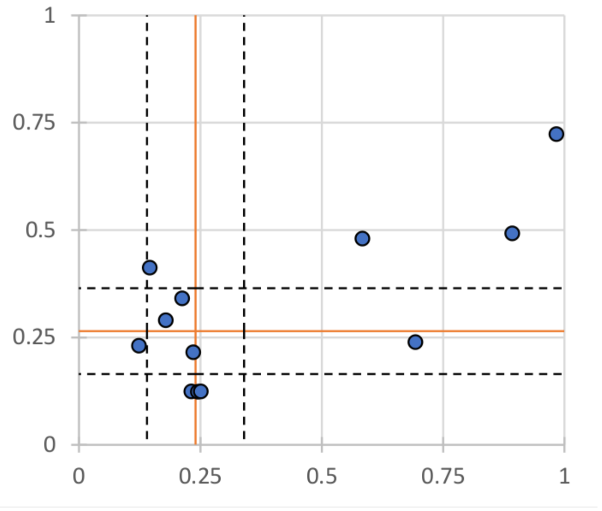

**Figure 4. Load-based partitioning with pre-fetching value of 0.1.**

## 4. Tests and Results

### 4.1 Batching/Non-Batching Communication

PLATO sends a user-specifiable number of particles per MPI call. At least, 1 point can be sent at a time, but, in general, a higher amount will reduce the total time taken as it will require less MPI calls but transfer the same total number of points and each MPI call adds a certain overhead to the overall runtime. Figure 5 shows the runtime and Fig. 6 shows the scaling for batching and non-batching communication, or, in our tests, sending 100 points and 1 point with each MPI call, respectively.

While a higher value will generally result in better performance as indicated by our results, there is a limit to how much data can be sent per MPI call, limited by the MPI implementation itself. If there are a large number of point attributes, it may not be possible to send 100 points per MPI call, in which case a lower value must be used. Additionally, the highest possible batching value may not always be the fastest; the optimal number of points to send at one time is highly dependent on the machine used, and user testing may be required to find the best value for a given machine.

### 4.2 Pre-fetching

The user-defined pre-fetching value  determines the amount of pre-fetching that PLATO computes. The result is that points can be sent from one processor to multiple others, increasing the total number of points that all processors have together on completion. Figure 7 and Fig. 8 show the runtime and speedup for different values of $\epsilon$.

Our results show that a large enough pre-fetching value can cause parallel speedup values to deteriorate, which we attribute to more processes receiving duplicates for higher values of $p$, resulting in redundant communication overhead. For lower values of $\epsilon$, however, the speedup generally has a higher value.

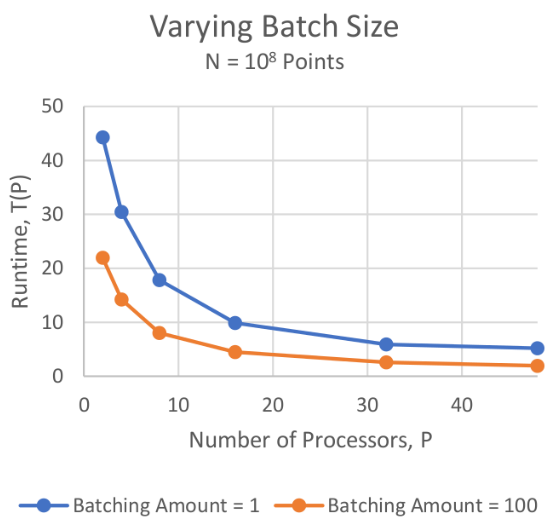

**Figure 5. Comparison of runtime of batching and non- batching communication.**

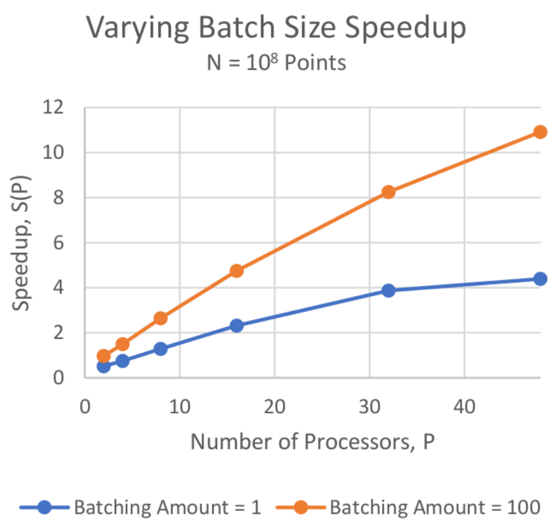

**Figure 6. Comparison of speedup of batching and non- batching communication.**

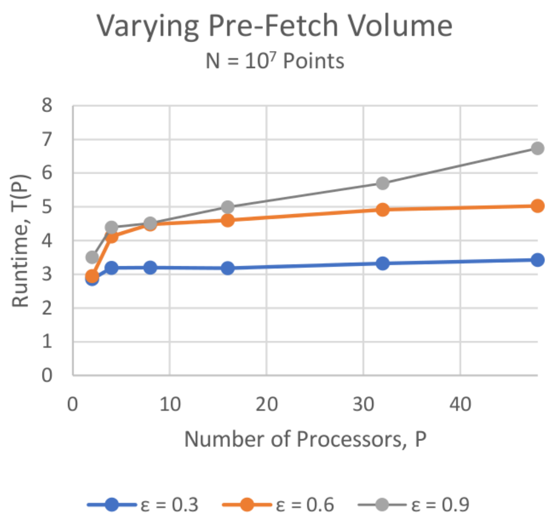

**Figure 7. Comparison of runtime of different values of $\epsilon$.**

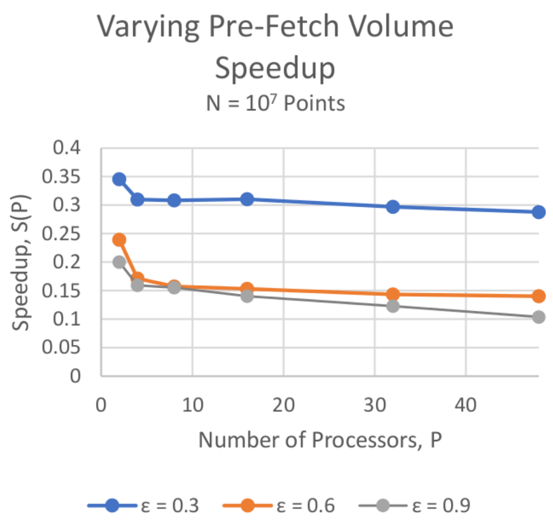

**Figure 8. Comparison of speedup of different values of $\epsilon$.**

## 4.3 Point Attributes

In any scientific application, each $n$-dimensional point in the input data set is associated with one or more attributes, such as, mass, temperature, a string tag, an integer tag, etc. When partitioning the input data, the attribute information for each point must also be communicated to the appropriate destination processors. PLATO supports a user-specifiable number of floating- point numbers, integers, characters, and/or strings (any combination thereof) associated with each point and sent along with the point's position as the data is partitioned. The resulting effect on performance is an increase in the overall time as a higher number of point attributes results in a higher communication cost.

The runtime and parallel speedup are represented in Fig. 9 and Fig. 10, respectively, and shows that, in our tests, a higher number of attributes results in a higher runtime.

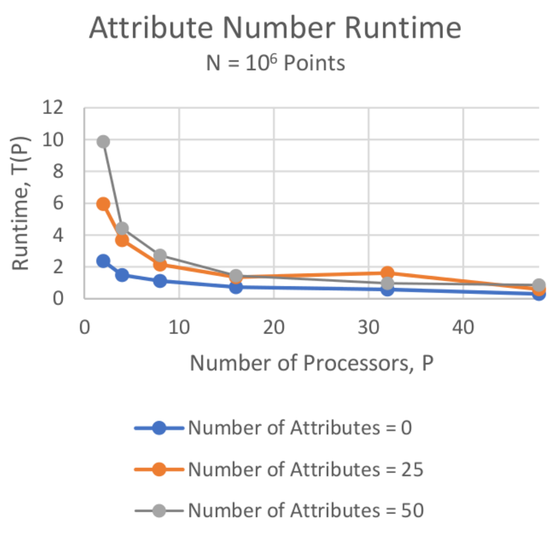

**Figure 9. Comparison of runtime with different numbers of attributes.**

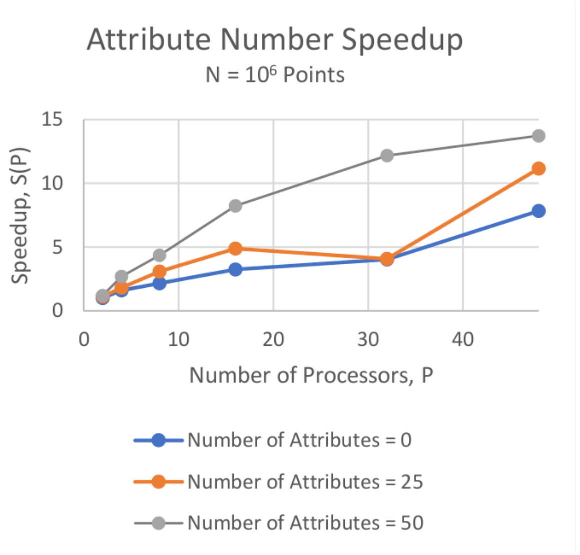

**Figure 10. Comparison of speedup with different numbers of attributes.**

## 4.4 Partitioning Method

The two different partitioning methods that PLATO implements are geometry-based and load-based partitioning. The geometric method requires less computation than the load-based method, as the load-based method implements a median-selection algorithm to determine the partition boundaries. However, depending on the distribution of the points, the geometric method is not guaranteed to be faster as it may result in a high number of points being sent to one process relative to the other processes, increasing the total time that a single process takes to receive points. In our testing, the points were evenly distributed, so the geometry-based method generally has the lower runtime as shown in Fig. 11.

## 4.5 Conclusions and Recommendations

Given our test results, we draw the following conclusions on performance and offer some recommendations for the users:

1. A larger batching amount will generally decrease PLATO runtime, but depending on the application, may not be possible. Thus, a default value is used which decreases the batching amount with a corresponding increase in the number of point attributes. The value itself can be user-specified,

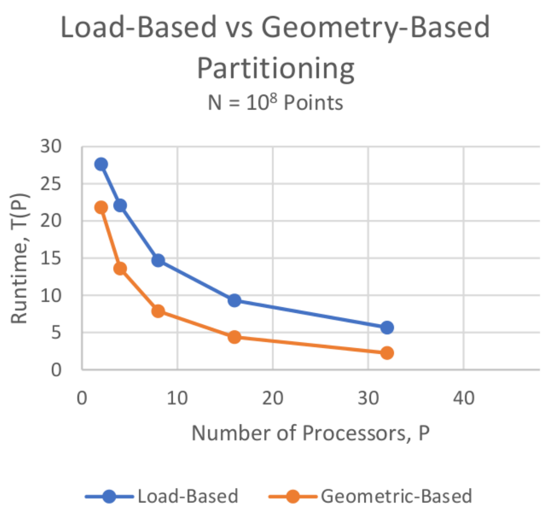

**Figure 11. Comparison of runtime of different partitioning methods.**

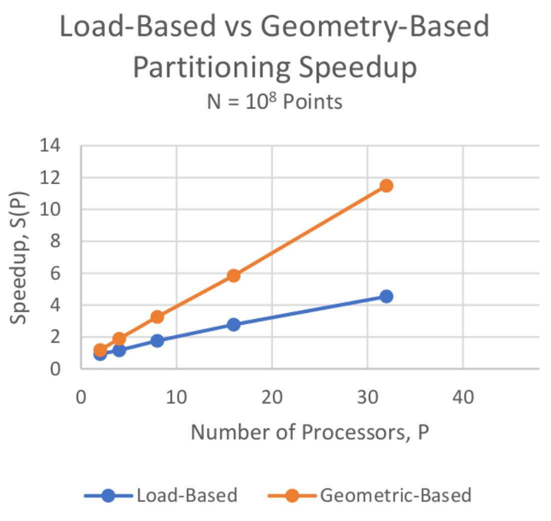

**Figure 12. Comparison of speedup of different partitioning methods.**

which may be preferred in cases where communication bandwidth is of concern, like in a distributed memory cluster.

2. A higher pre-fetching value will increase PLATO runtime. However, pre-fetching may be unavoidable for the subsequent computational analysis.
3. Increasing the number of point attributes adds a predictable runtime overhead to PLATO.
4. The load-based partitioning method will typically result in a longer PLATO runtime than geometric partitioning, but the resulting partitions will be more evenly distributed. This can be essential to ensuring load-balance in the subsequent computational analysis. Depending on the distribution of the points and the desired partitioning distribution, geometric partitioning can still result in an acceptable distribution.

## Acknowledgment

I am pleased to acknowledge Dr. Hao Lu (Computer Science and Mathematics Division, ORNL) for many useful discussions on related work.

# PLATO User Manual

**PLATO (Parallel Load Assignment Tool) User Manual**

A. Contents:

- Formatter.h, Formatter.c
- LoadBalancer.h, LoadBalancer.c
- ParticleSplitter.h, ParticleSplitter.c
- StartupFile
- Makefile

B. How to use PLATO:

1. Use the Makefile to compile PLATO. Makefile uses the mpicc command.

2. Configure the StartupFile. Each line of the StartupFile is explained here:

   a. FileTarget - Place the string of the file that contains your point data here. Example: "File Target = test.txt"

   b. Number of Points - This field determines the total number of points read by PLATO. Not affected by number of processors used. Example: "Number of Points = 100000"

   c. Number of Dimensions - This field should be equal to the number of dimensions in your points' positional data. Example: "Number of Dimensions = 3"

   d. Number of Attributes - This field should equal the total number of attributes you need to associate with each point. Example: "Number of Attributes = 3"

   e. Type of each Attribute - This field should contain a character to represent the type of each attribute in the order that they appear in your data file. Example: "Type of each Attribute ('c' or 'i' or 's' or 'f') = c i f". PLATO recognizes four different attribute types. Each type is stored internally as the C language type of the same name, with the exception of string:
      i. c = character
      ii. i = integer
      iii. s = string
      iv. f = float

   f. Processor Layout - This field should contain the processor topology you wish to partition your points by. Should have a number of integers equal to the number of dimensions you specify. Note that these numbers multiplied together should also be the number of processes you run PLATO with. For instance, the following example should be run with 4 processes. Example: Processor Layout (e.x. 3 3 2) = 2 2 1

g. Epsilon - Epsilon is identical to the pre-fetching value. Each process will expand its geometric boundaries by this amount in all directions. Example: Epsilon = 0.1

h. Geometric or Load-Based Partitioning - Geometric Partitioning creates boundaries by evenly divining up the span of your point data. Load-Based uses median finding to ensure equal partition size as output. Example: “Geometric or Load-Based Partitioning (G/L) = L”

i. Receive Array Size - Only use this field if your machine does not have enough memory to hold all data. Leave blank for the default value which will attempt to keep all points in memory. Example: “Receive Array Size = 10000”

3. Run PLATO with appropriate number of processors. PLATO does not take command line arguments.

4. Partitioned output files are placed in this directory and named PartitionedFile0, PartitionedFile1, etc. with the appropriate extension as specified.

# PLATO (Parallel Load Assignment Tool)

Generated by Doxygen 1.8.13

# Contents

**1 File Index** **1**

1.1 File List . . . . . 1

**2 File Documentation** **3**

2.1 Formatter.h File Reference . . . . . 3

2.1.1 Function Documentation . . . . . 3

2.1.1.1 CheckEOF() . . . . . 3

2.1.1.2 ConvertFile() . . . . . 4

2.1.1.3 OutputFile() . . . . . 4

2.2 LoadBalancer.h File Reference . . . . . 6

2.2.1 Detailed Description . . . . . 6

2.2.2 Function Documentation . . . . . 6

2.2.2.1 FindMedian() . . . . . 6

2.2.2.2 FindMedianRecursively() . . . . . 7

2.3 ParticleSplitter.c File Reference . . . . . 7

2.3.1 Detailed Description . . . . . 8

2.3.2 Variable Documentation . . . . . 8

2.3.2.1 MAX_PACKBUF_SIZE . . . . . 8

2.3.2.2 MAX_RECV_SIZE . . . . . 8

2.3.2.3 MAX_RECVBUF_SIZE . . . . . 8

2.3.2.4 MAX_STRING_SIZE . . . . . 8

2.3.2.5 PARTICLES_BETWEEN_PROBES . . . . . 8

2.3.2.6 SEND_SIZE . . . . . 8

2.4 ParticleSplitter.h File Reference . . . . . 9

2.4.1 Function Documentation . . . . . 9

2.4.1.1 GetMedians() . . . . . 9

2.4.1.2 ReadFile() . . . . . 10

2.4.1.3 ReadStartupFile() . . . . . 10

2.4.1.4 RecvParticle() . . . . . 11

2.4.1.5 SendParticle() . . . . . 12

2.4.1.6 UnpackParticle() . . . . . 13

2.4.1.7 WriteParticles() . . . . . 14

**Index** **15**

# Chapter 1

# File Index

## 1.1 File List

Here is a list of all documented files with brief descriptions:

Formatter.h . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
LoadBalancer.h . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
ParticleSplitter.c . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
ParticleSplitter.h . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

# Chapter 2

# File Documentation

## 2.1 Formatter.h File Reference

### Enumerations

- enum **attribute_type** { **FLOAT**, **INTEGER**, **CHARACTER**, **STRING** }

### Functions

- int ConvertFile (char ∗fileTarget, long long ∗fileSize, int numParticles, int numDimensions, int numAttributes, attribute_type ∗attribute_types)
- void OutputFile (float ∗array, int ∗ints, float ∗floats, char ∗chars, char ∗strings, int numDimensions, int numInts, int numFloats, int numChars, int numStrings, int numPoints, char fileType, int fileIndex)
- int CheckEOF (int numChars, FILE ∗fr)

### 2.1.1 Function Documentation

#### 2.1.1.1 CheckEOF()

```
int CheckEOF (
            int numChars,
            FILE * fr )
```

Checks numChars chars in stream for EOF. If EOF not found, returns characters to stream.

**Returns**

1 if EOF is found, 0 if otherwise

**Parameters**

| | |
|---|---|
| *numChars* | Number of characters from fr to read |
| *fr* | File to read chars from |

#### 2.1.1.2 ConvertFile()

```
int ConvertFile (
            char * fileTarget,
            long long * fileSize,
            int numParticles,
            int numDimensions,
            int numAttributes,
            attribute_type * attribute_types )
```

Converts fileTarget to an internal binary file.

**Returns**

1 on success, 0 on error

**Parameters**

| | |
|---|---|
| *fileTarget* | String of target file. Assigned the string of the converted file upon returning |
| *fileSize* | Assigned the size of the .ibin file after creation |
| *numParticles* | Number of points to read from fileTarget |
| *numDimensions* | Number of dimensions of the position of each point |
| *numAttributes* | Number of attributes for each point in fileTarget |
| *attribute_types* | Array of attribute_type indexed on order of each attribute in fileTarget |

#### 2.1.1.3 OutputFile()

```
void OutputFile (
            float * array,
            int * ints,
            float * floats,
            char * chars,
            char * strings,
            int numDimensions,
            int numInts,
            int numFloats,
            int numChars,
            int numStrings,
            int numPoints,
            char fileType,
            int fileIndex )
```

Places partitioned points into file of specified type.

**Parameters**

| | |
|---|---|
| *array* | Array of positional data of points |
| *numDimensions* | Number of dimensions of the position of each point |
| *numPoints* | Number of points to output to file (same as number of points in array) |
| *fileType* | Specifies type of output file. 'T' = text file, 'B' = binary file, 'C' = csv file |
| *fileIndex* | Number of file, not necessarily equal to but related to processor rank |

## 2.2 LoadBalancer.h File Reference

### Functions

- float FindMedian (float ∗array, int targetDimension, int numPoints, int numDimensions, int rank, int size, float lowerBound, float upperBound, MPI_Comm ∗cartComm)
- float FindMedianRecursively (float ∗tmpArray, float ∗medians, int numPoints, int rank, int size, MPI_Comm ∗cartComm)

### 2.2.1 Detailed Description

Finds median of positional data

### 2.2.2 Function Documentation

#### 2.2.2.1 FindMedian()

```
float FindMedian (
            float * array,
            int targetDimension,
            int numParticles,
            int numDimensions,
            int rank,
            int size,
            float lowerBound,
            float upperBound,
            MPI_Comm * cartComm )
```

Populates temporary array from array to sort and calls recursive method

Returns the median of the entire set of particles (of all processes/partitions)

**Parameters**

| | |
|---|---|
| *array* | Array of point positional data |
| *targetDimension* | Specifies which dimension to find the median of. Can be 0 - (n-1) for an n-dimensional space |

**Parameters**

| | |
|---|---|
| *numParticles* | Total number of points in array |
| *numDimensions* | Number of dimensions of the physical space |
| *rank* | Rank of calling process |
| *size* | Total number of processes |
| *lowerBound* | Lower bound of all points considered. Points with value below lowerBound in the targetDimension will not be placed into tmpArray nor included in median finding, but will remain in array |
| *upperBound* | Upper bound of all points considered. Points with value above upperBound in the targetDimension will not be placed into tmpArray nor included in median finding, but will remain in array |
| *cartComm* | Communicator passed to MPI functions |

#### 2.2.2.2 FindMedianRecursively()

```
float FindMedianRecursively (
            float * tmpArray,
            float * medians,
            int numParticles,
            int rank,
            int size,
            MPI_Comm * cartComm )
```

Finds median of the set of all combined tmpArray among processes

Recursive method used by FindMedian()

**Parameters**

| | |
|---|---|
| *tmpArray* | Local process' positional data |
| *medians* | Populated with all process' medians to find median of medians. Relavent only at root |
| *numParticles* | Number of points that have data in tmpArray |
| *rank* | Rank of calling process |
| *size* | Total number of processes |
| *cartComm* | Communicator passed to MPI functions |

## 2.3 ParticleSplitter.c File Reference

### Variables

- const int MAX_PACKBUF_SIZE = 1000
- const int MAX_RECVBUF_SIZE = 1000
- int MAX_RECV_SIZE = -1
- const int MAX_STRING_SIZE = 50
- const int PARTICLES_BETWEEN_PROBES = 20
- int SEND_SIZE = 100

### 2.3.1 Detailed Description

Reads information from StartupFile in order to read the particle information from the Target File. Uses Formatter.c to convert the Target File to an internal binary (.ibin) file that is read in parallel. Uses LoadBalancer.c to find medians if Load-balancing is specified by the user in the StartupFile.

### 2.3.2 Variable Documentation

#### 2.3.2.1 MAX_PACKBUF_SIZE

```
const int MAX_PACKBUF_SIZE = 1000
```

Maximum allowed size of buffer of `MPI_Packed` particles pointed to `MPI_Isend`. Will use `MPI_Waitall` if reached.

#### 2.3.2.2 MAX_RECV_SIZE

```
int MAX_RECV_SIZE = -1
```

Maximum allowed size of unpacked particle data to remain in memory. Will write to temporary file if reached.

#### 2.3.2.3 MAX_RECVBUF_SIZE

```
const int MAX_RECVBUF_SIZE = 1000
```

Maximum allowed size of buffer of `MPI_Packed` particles pointed to `MPI_Irecv`. Will use `MPI_Waitall` if reached.

#### 2.3.2.4 MAX_STRING_SIZE

```
const int MAX_STRING_SIZE = 50
```

Maximum allowed size of strings.

#### 2.3.2.5 PARTICLES_BETWEEN_PROBES

```
const int PARTICLES_BETWEEN_PROBES = 20
```

Number of particles checked to send before `MPI_Iprobe` is called.

#### 2.3.2.6 SEND_SIZE

```
int SEND_SIZE = 100
```

Number of points to batch send with each `MPI_Isend` and `MPI_Irecv` call.

## 2.4 ParticleSplitter.h File Reference

### Functions

- void ReadStartupFile (char ∗target, int ∗numPoints, int ∗numDimensions, int ∗numAttributes, attribute_type ∗attributeTypes, int ∗layout, float ∗epsilon, int ∗numInts, int ∗numFloats, int ∗numChars, int ∗numStrings, int ∗geometric)
- void ReadFile (char ∗target, int fileSize, int numInts, int numFloats, int numChars, int numStrings, float ∗array, int ∗ints, float ∗floats, char ∗chars, char ∗strings, int rank, int size, int numPoints, int ∗numPointsRead, int numDimensions, int numAttributes, attribute_type ∗attributeTypes)
- void SendParticle (int amount, int ∗particleIndeces, void ∗packBuffer, int rank, int numDimensions, int num←Ints, int numFloats, int numChars, int numStrings, int ∗ints, float ∗floats, char ∗chars, char ∗strings, float ∗array, float ∗recvArray, int ∗recvInts, float ∗recvFloats, char ∗recvChars, char ∗recvStrings, MPI_Request ∗sendRequests, int ∗numSendRequests, int ∗numLocalPoints, int ∗recvPos, char ∗tmpFileName, int send←Rank, MPI_Comm ∗cartComm)
- void RecvParticle (int source, int tag, void ∗recvBuffer, int total, int ∗arrayPos, MPI_Comm ∗cartComm, M←PI_Request ∗recvRequests, int ∗numRecvRequests)
- void WriteParticles (int ∗recvPos, float ∗recvArray, int ∗recvInts, float ∗recvFloats, char ∗recvChars, char ∗recvStrings, int numDimensions, int numInts, int numFloats, int numChars, int numStrings, char ∗tmpFile←Name)
- void UnpackParticle (int recvSize, void ∗buffer, int total, int numDimensions, int numInts, int numFloats, int numChars, int numStrings, int ∗recvPos, float ∗recvArray, int ∗recvInts, float ∗recvFloats, char ∗recvChars, char ∗recvStrings, char ∗tmpFileName, MPI_Comm ∗cartComm, int ∗numLocalPoints)
- void GetMedians (int height, int index, float ∗medians, float ∗array, int numParticlesRead, int targetDimension, int numDimensions, int rank, int size, float min, float max, MPI_Comm ∗cartComm)

### 2.4.1 Function Documentation

#### 2.4.1.1 GetMedians()

```
void GetMedians (
            int height,
            int index,
            float * medians,
            float * array,
            int numParticlesRead,
            int targetDimension,
            int numDimensions,
            int rank,
            int size,
            float min,
            float max,
            MPI_Comm * cartComm )
```

Recursive median finder

#### 2.4.1.2 ReadFile()

```
void ReadFile (
            char * target,
            int fileSize,
            int numInts,
            int numFloats,
            int numChars,
            int numStrings,
            float * array,
            int * ints,
            float * floats,
            char * chars,
            char * strings,
            int rank,
            int size,
            int numPoints,
            int * numPointsRead,
            int numDimensions,
            int numAttributes,
            attribute_type * attributeTypes )
```

Reads .ibin file in local directory

**Parameters**

| | |
|---|---|
| *target* | String of targetFile |
| *fileSize* | FileSize in bytes |
| *numInts* | Number of integer attributes per point |
| *numFloats* | Number of float attributes per point |
| *numChars* | Number of character attributes per point |
| *numStrings* | Number of string attributes per point |
| *array* | Array of positional data, populated from file on return |
| *ints* | Array of integer attributes, populated from file on return |
| *floats* | Array of float attributes, populated from file on return |
| *chars* | Array of character attributes, populated from file on return |
| *strings* | Array of string attributes, populated from file on return |
| *rank* | Rank of calling process |
| *size* | Total number of processes to read from file |
| *numPoints* | Total number of points |
| *numPointsRead* | Number of points read from file, populated on return |
| *numDimensions* | Number of dimension of positional data |
| *numAttributes* | Total number of attributes per point |
| *attributeTypes* | Array of attribute types in order as shown in file, populated by ReadStartupFile() |

#### 2.4.1.3 ReadStartupFile()

```
void ReadStartupFile (
            char * target,
            int * numPoints,
            int * numDimensions,
            int * numAttributes,
            attribute_type * attributeTypes,
            int * layout,
            float * epsilon,
            int * numInts,
            int * numFloats,
            int * numChars,
            int * numStrings,
            int * geometric )
```

Reads StartupFile in local directory

**Parameters**

| | |
|---|---|
| *target* | string of StartupFile |
| *numPoints* | Maximum number of points to read |
| *numDimensions* | Number of dimensions in points' space |
| *numAttributes* | Number of total attributes associated with each point |
| *attributeTypes* | Array of attribute_types populated in order as they appear in StartupFile (and by targetFile by requirement) |
| *layout* | Processor Layout |
| *epsilon* | Pre-fetching value |
| *numInts* | Equal to number of integer attributes per point on return |
| *numFloats* | Equal to number of float attributes per point on return |
| *numChars* | Equal to number of character attributes per point on return |
| *numStrings* | Equal to number of string attributes per point on return |
| *geometric* | Equal to 1 if using geometric partitioning or 0 if load-based on return |

#### 2.4.1.4 RecvParticle()

```
void RecvParticle (
            int source,
            int tag,
            void * recvBuffer,
            int total,
            int * arrayPos,
            MPI_Comm * cartComm,
            MPI_Request * recvRequests,
            int * numRecvRequests )
```

Receives at most `SEND_SIZE MPI_Packed` particles from a single source

**Parameters**

| | |
|---|---|
| *source* | Rank of sending process |
| *tag* | Tag of message to be received |
| *recvBuffer* | Buffer of `MPI_Packed` pointed to `MPI_Irecv()` |
| *total* | Maximum possible size of each message |

**Parameters**

| | |
|---|---|
| *arrayPos* | Number of points currently received |
| *cartComm* | Communicator passed to MPI functions |
| *recvRequests* | Array of requests created by `MPI_Irecv()` |
| *numRecvRequests* | Total number of active recvRequests |

#### 2.4.1.5 SendParticle()

```
void SendParticle (
            int amount,
            int * particleIndeces,
            void * packBuffer,
            int rank,
            int numDimensions,
            int numInts,
            int numFloats,
            int numChars,
            int numStrings,
            int * ints,
            float * floats,
            char * chars,
            char * strings,
            float * array,
            float * recvArray,
            int * recvInts,
            float * recvFloats,
            char * recvChars,
            char * recvStrings,
            MPI_Request * sendRequests,
            int * numSendRequests,
            int * numLocalPoints,
            int * recvPos,
            char * tmpFileName,
            int sendRank,
            MPI_Comm * cartComm )
```

Packs and sends `amount` particles. If `sendRank = rank` then `MPI_Isend()` will not be called.

**Parameters**

| | |
|---|---|
| *amount* | Number of points to send with `MPI_Isend()` call |
| *particleIndeces* | Indeces of array to send |
| *packBuffer* | Buffer of `MPI_Packed` points used by `MPI_Isend()` |
| *rank* | Rank of calling process |
| *numDimensions* | Number of dimension of positional data |
| *numInts* | Number of integer attributes per point |
| *numFloats* | Number of float attributes per point |
| *numChars* | Number of character attributes per point |
| *numStrings* | Number of string attributes per point |
| *ints* | Array of integer attributes |

**Parameters**

| | |
|---|---|
| *floats* | Array of float attributes |
| *chars* | Array of character attributes |
| *strings* | Array of string attributes |
| *array* | Array of positional data |
| *recvArray* | Array of partitioned positional data |
| *recvInts* | Array of partitioned integer attributes |
| *recvFloats* | Array of partitioned float attributes |
| *recvChars* | Array of partitioned character attributes |
| *recvStrings* | Array of partitioned string attributes |
| *sendRequests* | Array of requests created by `MPI_Isend()` |
| *numSendRequests* | Total number of active sendRequests |
| *numLocalPoints* | Number of points in local arrays |
| *recvPos* | Number of points currently received |
| *tmpFileName* | Name of temporary file. Written to only if receive arrays overflow |
| *sendRank* | Rank of process to send points to |
| *cartComm* | Communicator passed to MPI functions |

### 2.4.1.6 UnpackParticle()

```
void UnpackParticle (
             int recvSize,
             void * buffer,
             int total,
             int numDimensions,
             int numInts,
             int numFloats,
             int numChars,
             int numStrings,
             int * recvPos,
             float * recvArray,
             int * recvInts,
             float * recvFloats,
             char * recvChars,
             char * recvStrings,
             char * tmpFileName,
             MPI_Comm * cartComm,
             int * numLocalPoints )
```

Unpacks particles and places in receive arrays

**Parameters**

| | |
|---|---|
| *buffer* | Buffer of received `MPI_Packed` points |
| *total* | Total size of each `MPI_Packed` object |
| *numDimensions* | Number of dimension of positional data |
| *numInts* | Number of integer attributes per point |
| *numFloats* | Number of float attributes per point |

**Parameters**

| | |
|---|---|
| *numChars* | Number of character attributes per point |
| *numStrings* | Number of string attributes per point |
| *recvPos* | Number of points currently received |
| *recvArray* | Array of partitioned positional data |
| *recvInts* | Array of partitioned integer attributes |
| *recvFloats* | Array of partitioned float attributes |
| *recvChars* | Array of partitioned character attributes |
| *recvStrings* | Array of partitioned string attributes |
| *tmpFileName* | Name of temporary file used if receive arrays overflow |
| *cartComm* | Communicator passed to MPI functions |
| *numLocalPoints* | Total number of received points |

#### 2.4.1.7 WriteParticles()

```
void WriteParticles (
             int * recvPos,
             float * recvArray,
             int * recvInts,
             float * recvFloats,
             char * recvChars,
             char * recvStrings,
             int numDimensions,
             int numInts,
             int numFloats,
             int numChars,
             int numStrings,
             char * tmpFileName )
```

Writes particles in receive arrays to temporary file

**Parameters**

| | |
|---|---|
| *recvPos* | Number of points currently received |
| *recvArray* | Array of partitioned positional data |
| *recvInts* | Array of partitioned integer attributes |
| *recvFloats* | Array of partitioned float attributes |
| *recvChars* | Array of partitioned character attributes |
| *recvStrings* | Array of partitioned string attributes |
| *numDimensions* | Number of dimension of positional data |
| *numInts* | Number of integer attributes per point |
| *numFloats* | Number of float attributes per point |
| *numChars* | Number of character attributes per point |
| *numStrings* | Number of string attributes per point |
| *tmpFileName* | Name of temporary file used if receive arrays overflow |

# Index

CheckEOF
    Formatter.h, 3
ConvertFile
    Formatter.h, 4

FindMedian
    LoadBalancer.h, 6
FindMedianRecursively
    LoadBalancer.h, 7
Formatter.h, 3
    CheckEOF, 3
    ConvertFile, 4
    OutputFile, 4

GetMedians
    ParticleSplitter.h, 9

LoadBalancer.h, 6
    FindMedian, 6
    FindMedianRecursively, 7

MAX_PACKBUF_SIZE
    ParticleSplitter.c, 8
MAX_RECV_SIZE
    ParticleSplitter.c, 8
MAX_RECVBUF_SIZE
    ParticleSplitter.c, 8
MAX_STRING_SIZE
    ParticleSplitter.c, 8

OutputFile
    Formatter.h, 4

PARTICLES_BETWEEN_PROBES
    ParticleSplitter.c, 8
ParticleSplitter.c, 7
    MAX_PACKBUF_SIZE, 8
    MAX_RECV_SIZE, 8
    MAX_RECVBUF_SIZE, 8
    MAX_STRING_SIZE, 8
    PARTICLES_BETWEEN_PROBES, 8
    SEND_SIZE, 8
ParticleSplitter.h, 9
    GetMedians, 9
    ReadFile, 9
    ReadStartupFile, 10
    RecvParticle, 11
    SendParticle, 12
    UnpackParticle, 13
    WriteParticles, 14

ReadFile
    ParticleSplitter.h, 9
ReadStartupFile
    ParticleSplitter.h, 10
RecvParticle
    ParticleSplitter.h, 11

SEND_SIZE
    ParticleSplitter.c, 8
SendParticle
    ParticleSplitter.h, 12

UnpackParticle
    ParticleSplitter.h, 13

WriteParticles
    ParticleSplitter.h, 14