ELEMENTS SPECIFICATION

# SPLITTER

The working assumption for now is that a precomposed matrix in MTX format will be preloaded into a splitting system, which will reside on a server. This is reasonable since matrix splitting is considered to be an unparallelizable task.

The sequence of tasks will be as follows:

1. MTX file handle is opened
2. Matrix dimensions are read
   1. If matrix is not square, file is rejected – appropriate error message is produced
3. Matrix is conceptually divided into parts of configurable heights
4. File handles for the parts are opened
5. MTX is read and entries are passed on to appropriate files
6. MTX file handle is closed
7. Part file handles are closed
8. **FUTURE** Files are encrypted
9. Files are compressed
10. Files are recorded as parts in the database
11. 0-Wave is recorded composed of the parts
12. Splitter exits

# PROBLEM FRAGMENTATION

# DEFINITIONS

The process of progressing a matrix from the introductory state to solved state is called a **solution**.

A **solution** is composed of (usually more than one) **waves.**

A **wave** is composed of **tasks** which reference **packages**.

**Packages** are recorded separately, identified by **solution** and **wave**.

# CLIENT ACTIONS

The client which was tasked with completing a **task** will request the appropriate **packages** and perform the tasks using whatever computational powers are available.

# HOW WAVES ARE CREATED

The initial wave is created by the input mechanism, called a splitter. Whenever the controller detects that all the tasks in a particular wave have been completed, it checks several conditions.

First of those conditions is whether that was a simple solution wave or a cross-check wave (which will be elaborated upon in section regarding tasks). If it was a simple solution wave, a cross-check wave is created by placing tasks in the database marked for being sent out and processed. If it was a cross-check wave, two scenarios can happen: if the cross-check tasks have returned an unanimous zero operations, then nothing will be done and the solution will be marked complete. Otherwise, packages returned from crosscheck will be advanced into the next simple solution wave if need be.

*If need be* means here that – if in the last simple solution wave a package returned after 0 operations and was not affected by cross-check, there is no need to perform simple solution on it again, and it should not be placed in the pool for simple solution.

# SERVER ACTIONS

# TASKS

There currently are two kinds of tasks.

The first kind of task is a simple solution task: the input for that is an .mcx (Matrix Combined Format) file from the package. This file will be loaded into appropriate memory structures. The original Parallel Gauss solution will be performed on it. The result will be another package (for crosscheck) and the resultant map vector. The package should be uploaded to the server directly after compression, the map vector might be passed through differently.

The second type of task is the crosscheck. The crosscheck happens for *n*th package with regard to (0..n-1) previous packages. Maps in previous packages are scanned to figure out whether there are conflicting rows in earlier packages. If there are, the appropriate packages will be downloaded, uncompressed, the appropriate rows will be assembled and Gaussian elimination will be performed.

It has already been previously proved that such an elimination does not require locking parts of the matrix, since all the rows that are in fact conflicted are removed against a row that is never affected (since there are none before it to be considered canonical). The amount of data that has to be downloaded in a best-case scenario is also relatively small.