Matrix Representation

```
typedef struct cell_t {
    offset_t j;
    value_t val;
} cell_t;

typedef struct matrix_row_t {
    offset_t num_cells;
    cell_t* cells;
} matrix_row_t;

typedef struct mapped_row_t {
    offset_t i;
    matrix_row_t row;
} mapped_row_t;

typedef struct matrix_t {
    offset_t m;
    offset_t n;
    offset_t from_i;
    offset_t to_i;
    matrix_row_t* rows;

    // these are transient (not serialized)
    offset_t num_mapped_rows;
    mapped_row_t* mapped_rows;
} matrix_t;

void matrix_init( matrix_t* mat, const offset_t m, const offset_t n, const
   offset_t from_i, const offset_t to_i );
void matrix_free( matrix_t* mat );
```

```
offset_t matrix_num_rows( const matrix_t* mat );
matrix_row_t* matrix_row( const matrix_t* mat, const offset_t i );

void matrix_print( const matrix_t* mat );

void matrix_serialize( const matrix_t* mat, char** buf, offset_t* bufsize )
    ;
void matrix_deserialize( matrix_t* mat, const char* buf );

void matrix_read(matrix_t* mat, const char* file , const offset_t m, const
    offset_t n, MPI_Comm comm, const boolean_t is_vec);
void matrix_read_rows(matrix_t* mat, FILE* input , const boolean_t is_vec);

void matrix_map_row(matrix_t* mat, const matrix_row_t* row, const offset_t
    i);



void matrix_sync_mult_rows(const matrix_t* mat, matrix_t* vec , MPI_Comm
    comm);
void matrix_vector_mult(const matrix_t* mat, const matrix_t* vec , matrix_t*
     res);
```

Algorithm

1. Root process reads input file, sending row-wise partitions of the matrix and vector in p/n size partitions.
    a) Send size of buffer
    b) Send actual buffer
2. Each process receives and deserializes the matrix and vector partitions.
3. Each process then determines which columns it needs from remote processes and adds them to a an array, and increments a p-length array with each index corre-

sponding to the remote rank, and the value corresponding to how many (unique) vector rows are needed.

   a) If a any column has a value that is not within the row-wise partition range, it is needed remotely.

4. The processes do an All-to-all personalized (MPI_Allgather) of the p-length size array, and end up with p^2-length array to use as a communication schedule.

5. Each process iterates the p^2 schedule and participates in interactions it is involved with.

   a) Sender = i / p

      i. Receive list of vector rows to send

      ii. Send list of values for those rows back

   b) Receiver = i % p

      i. Send list of vector rows to send

      ii. Receive list of values for those rows

      iii. Map rows into our vector, so mult can proceed with no special attention paid.

   c) This makes each processor do an async send to all other processors before recving the data that they need.

6. Perform matrix-vector multiplication as we normally would

7. Synchronize result vector

Analysis

a: average # of columns per row

  a << n

b: average # of cols needed per processor = (p-1)a/p~omega(a)

b <= a

Computation:

$T_p = a\frac{n}{p}t_c + log(p)t_s + p(p-1)t_w + b(t_s + \frac{a^2}{p}t_w) + (a+b)\frac{n}{p}t_c$

The first term is the time to calculate needed columns. The second term is an all-to-all broadcast to exchange communication sizes. The third term is the time needed to exchange the needed columns. And the fourth term is the time to compute the matrix-vector product. Simplified:

$T_p = a\frac{n}{p}t_c + log(p)t_s + (p-1)pt_w + b(t_s + \frac{a^2}{p}t_w) + (a+b)\frac{n}{p}t_c$

$T_p = (2a+b)\frac{n}{p}t_c + (b + log(p))t_s + (p^2 - p + b\frac{a^2}{p})t_w$

The p-sized all-to-all broadcast of communication sizes creates the dominant p^2 term. This allows for communication to be performed without the chance of deadlock, however, since all processes have the same communication schedule.

Which gives:

$pT_p = (2a+b)nt_c + p(b + log(p))t_s + (p^3 - p^2 + ba^2)t_w$

$T_s = na$

First Term: $O(na)$ since a>>b.

Second Term: $O(plog(p))$ since asymptotically log(p) > b.

Third Term: $O(p^3)$ since a<<n. This is the dominating term.

Cost Optimality:$O(p^3) \sim O(na) \Rightarrow p = (na)^{1/3}$ and $n = \frac{p^3}{a}$.