Matrix Multiply:

The table below shows the performance of the matrix multiply program through three different iterations. The MPI Revised version was used for the Karp-Flatt calculations, and all of the values are speedup shown as a floating-point value.
1.2 = 120% speedup.

Karp-Flatt Metric calculation was done as follows:

$$e = \frac{\left(\frac{1}{\psi} - \frac{1}{P}\right)}{\left(1 - \frac{1}{P}\right)}$$

Core	2	4	8	16
OpenMP	2.16	3.78	7.69	12.58
MPI Initial	1.11	1.70	1.75	3.10
MPI Revised	1.88	3.18	4.78	8.05
e	0.0638	0.0859	0.0962	0.0658

Iso-Efficiency Analysis of MPI Revised:

General Equations:

$$T_O = \ Total\ Overhead$$
$$T_1 = Single\ Processor\ Time$$
$$= W * \ t_C = (Units\ of\ Work) * (Time\ per\ unit\ of\ work)$$
$$T_P = Time\ Parallel$$
$$PT_P = \ T_1 + \ T_O = Total\ Time$$
$$S = \frac{T_1}{T_P} = \ \frac{PT_P}{T_1 + \ T_O} = Speedup$$

Analysis:

Assumptions:

Matrix A has n x n elements.
Matrix B has n x n elements.
Matrix C = A*B and has n x n elements.

Basic A * B = C:

```
for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
                for (int off = 0; off < n; off++) {
                        C[row][col] += a[row][off] * b[off][col];
                }
        }
}
```

$A * B^T = C$:

If B is stored as the transpose, then all of the traversal of arrays is done through a row, which makes all communication of B easier, and speeds up the program in general (better cache performance). The highlighted change is the only other thing necessary to make sure that the results are still accurate.

```
for (int row = 0; row  < n; row++) {
        for (int col = 0; col < n; col++) {
                for (int off = 0; off < n; off++) {
                        C[row][col] += a[row][off] * b[col][off];
                }
        }
}
```

Assumption: A and B are pregenerated.

$$T_P = Time\ to\ Receive\ Data + Time\ to\ Multiply\ Matrix\ Segment + Time\ to\ Share\ Matrix * Sub\ Matricies + Time\ to\ Return\ Results$$

$$T_P = (\text{Items to Receive}) * (\text{Cores Receiving}) * (\text{Comm Time}) + (\text{n multiplications} + (\text{n} - 1)\text{additions}) * \left(\frac{n^2}{\text{P}}\ \text{entries}\right) + (\text{Items to Share}) * 2 * (\text{Cores Sharing}) * (\text{Comm Time}) + (\text{Number of Communications}) * (\text{Data Returned}) * (\text{Time Return})$$

$$T_p = \frac{n^2}{P} * P * t_C + (n * t_{mult} + (n-1) * t_{add}) * \left(\frac{n^2}{P}\right) + \frac{n}{P} * 2 * P * t_C + \log_2 P * \frac{n}{P} * t_C$$

$$T_P = (n^2 + 2 * n + \frac{n}{P} * \log_2 P) * t_c + (n * t_{mult} + (n-1) * t_{add}) * \frac{n^2}{P}$$

$$PT_P = T_0 + T_1$$

$$T_O = (n^2 + 2 * n + \frac{n}{P} * \log_2 P) * t_c * P$$

$$T_1 = (n * t_{mult} + (n-1) * t_{add}) * n^2$$

$$\theta(P * \log_2 P) = Iso - Efficiency$$

When determining the final iso-efficiency of this algorithm, the main concern was how cost would scale as the number of processors increased. When looking at the actual algorithm time, the computations were divided in such a way that the cost of multiplying the actual matrices was split evenly, so that cost does not increase due to parallelization. The cost that does increase is the cost of actually sharing the data between cores, which is described by the final reduction necessary. The initial distribution is linear, so even though more communications would be needed as the number of cores increased, the time of each communication would decrease.

Binary Tree Search:

Cores	2	4	8	16
OpenMP	1.49	1.66	2.09	2.69
MPI	1.92	3.56	5.34	7.64
e (on MPI)	0.041	0.041	0.071	0.0729

When designing the binary tree search, the major important factors in the speedup found in the MPI version were the following assumptions:

1. Identifying nodes is more important than returning nodes
2. The goal is a total count

With the assumption that the nodes content is more important than the actual node object, it becomes possible to convert the tree to a flat file that every thread can access at the same time. The tree was printed recursively using the following format:

(node value (left node) (right node))

Which allows for two things. The first is easy parsing of the flat file to completely recreate the tree, and the second is a flat file that multiple threads can process at once. Each thread was given a starting point in the file and had the responsibility of scanning from its start to end point (start of the next thread) and regenerate the nodes. This regeneration did not recreate the structure (as that was not the goal) but did allow for processing of the node values, which made counting the number of hits quite easy. Some additional logic was needed to make sure that nodes whose values spanned thread lines were recreated properly and only recreated once, but that is an edge case that has little impact on total performance.

$$T_P = Time\ to\ Scan\ Subset\ of\ Files + Time\ to\ Return\ Results$$
$$T_P = \frac{file\ size}{P} * t_{parse} + \log_2 P$$
$$P * T_P = T_O + T_1$$
$$(file\ size) * t_{parse} + P * \log_2 P = T_O + T_1$$
$$T_O = P * \log_2 P$$
$$T_1 = (file\ size) * t_{parse}$$
$$\theta(P * \log_2 P) = Iso - Efficiency$$

As shown above, the Iso-Efficiency of the parallelized binary tree search depends solely on the number of threads used, and is based on the cost of the reduction at the end of the search.