

Dense Matrix Multiplication

- High-end simulation in the physical sciences = 7 numerical methods:
 - **Structured Grids (including adaptive block structured)**
 - **Unstructured Grids**
 - **Spectral Methods**
 - **Dense Linear Algebra**
 - **Sparse Linear Algebra**
 - **Particle Methods**
 - **Monte Carlo**

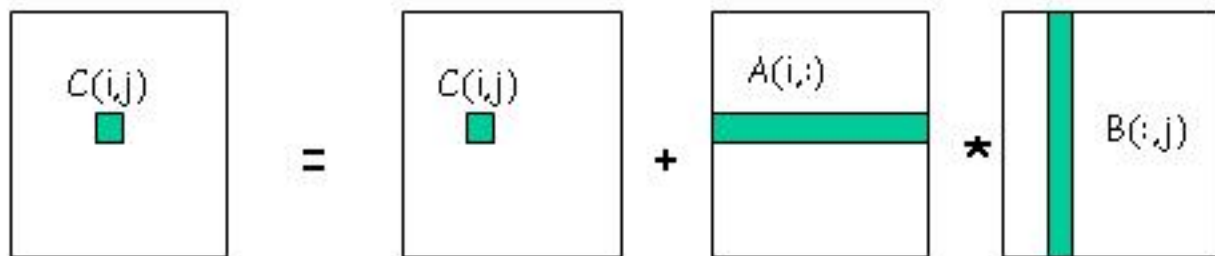
- A dwarf is a pattern of computation and communication
- Dwarfs are targets from algorithmic, software, and architecture standpoints

- Arises in some of the largest computations
- It achieves high machine efficiency
- There are important subcategories

- Implements $C = C + A * B$

```

for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    
```
- Algorithm has $2 * n^3 = O(n^3)$ Flop complexity



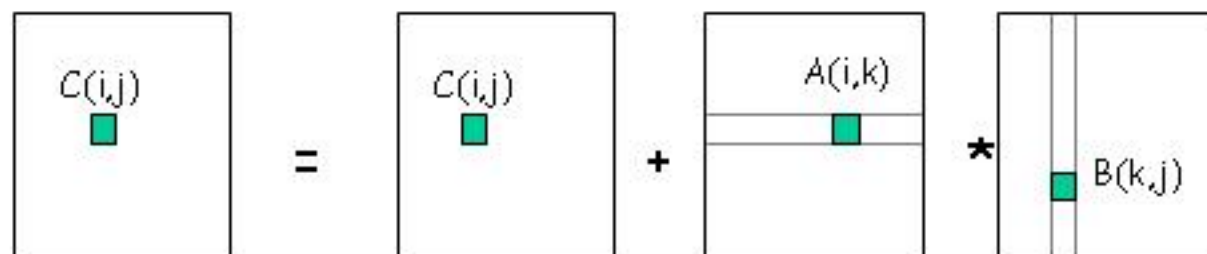


$O(N^3)$ performance would have constant cycles/flop
Performance looks like $O(N^{4.7})$

- Consider A, B, C to be $N \times N$ matrices of $b \times b$ sub-blocks where $b = n/N$ is called the block size

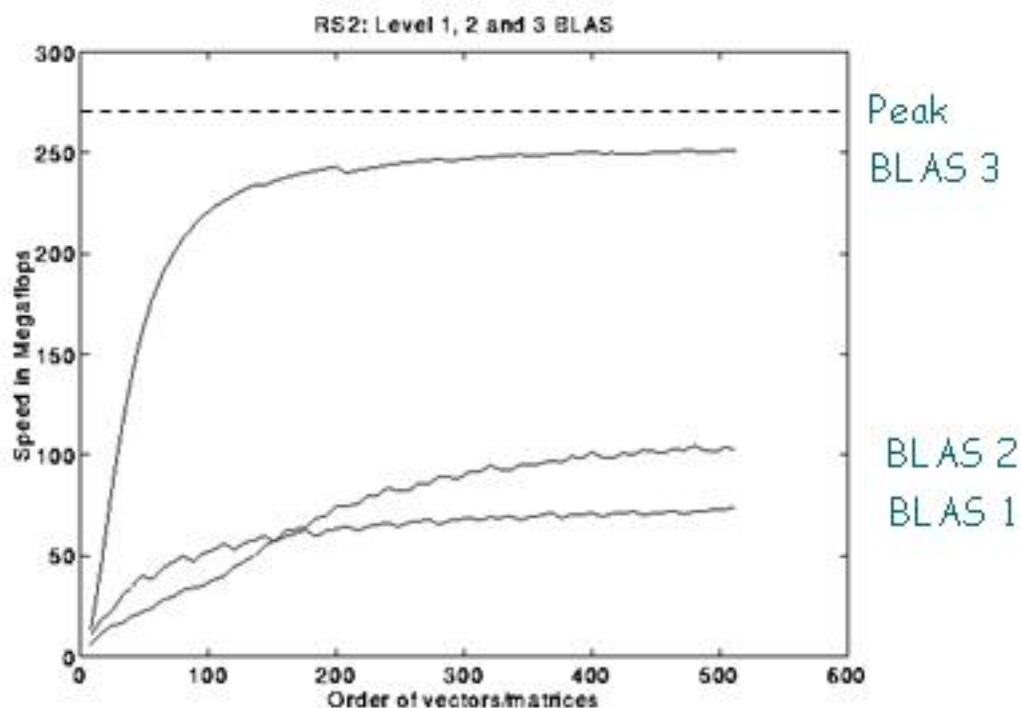
```

for i = 1 to N
  for j = 1 to N
    {read block C(i,j)}
    for k = 1 to N
      {read blocks A(i,k) and B(k,j)}
      C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
    {write block C(i,j)}
  
```



- **Optimized implementations by Industry and Vendors**
- **History**
 - **BLAS1 (1970s):**
 - **Vector operations: dot product, saxpy ($y = \alpha * x + y$), etc**
 - **BLAS2 (mid 1980s)**
 - **Matrix-vector operations: matrix vector multiply, etc**
 - **Somewhat faster than BLAS1**
 - **BLAS3 (late 1980s)**
 - **Matrix-matrix operations: matrix matrix multiply, etc**
 - **BLAS3 is potentially much faster than BLAS2**
- **Good algorithms used BLAS3 when possible (LAPACK)**

Peak speed = 266 Mflops



BLAS 3 (n-by-n matrix matrix multiply) vs
 BLAS 2 (n-by-n matrix vector multiply) vs
 BLAS 1 (saxpy of n vectors)

- **Performance models are useful for high level algorithms**
 - **Helps in developing a blocked algorithm**
 - **Models have not proven very useful for block size selection**
 - **too complicated to be useful**
 - See work by Sid Chatterjee for detailed model
 - **too simple to be accurate**
 - Multiple multidimensional arrays, virtual memory, etc.
- **Some systems use search**
 - **Atlas - being incorporated into Matlab**
 - **BeBOP - <http://www.cs.berkeley.edu/~richie/bebop>**

- **Traditional algorithm (with or without tiling) has $O(n^3)$ flops**
- **Strassen algorithm has asymptotically lower flops: $O(n^{2.81})$**
 - Current world's record is $O(n^{2.376})$ (Coppersmith & Winograd)
- **For 2×2 matrix multiply, it takes 8 multiplies, 7 adds**
 - Strassen does it with 7 multiplies and 18 adds

$$\text{Let } M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$\text{Let } p_1 = (a_{12} - a_{22}) * (b_{21} + b_{22})$$

$$p_2 = (a_{11} + a_{22}) * (b_{11} + b_{22})$$

$$p_3 = (a_{11} - a_{21}) * (b_{11} + b_{12})$$

$$p_4 = (a_{11} + a_{12}) * b_{22}$$

$$p_5 = a_{11} * (b_{12} - b_{22})$$

$$p_6 = a_{22} * (b_{21} - b_{11})$$

$$p_7 = (a_{21} + a_{22}) * b_{11}$$

$$\text{Then } m_{11} = p_1 + p_2 - p_4 + p_6$$

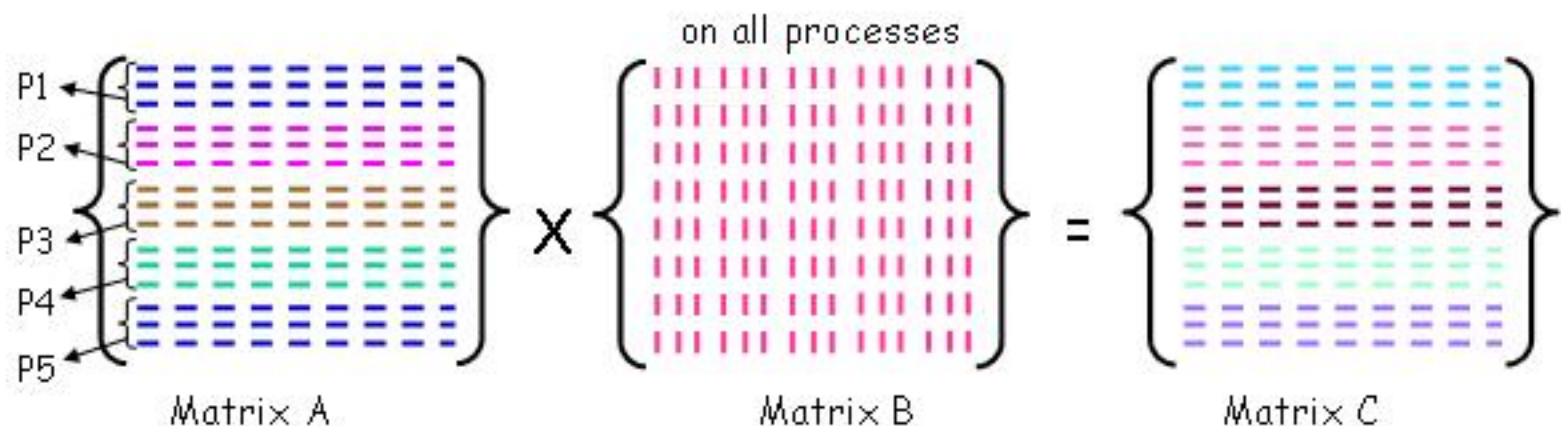
$$m_{12} = p_4 + p_5$$

$$m_{21} = p_6 + p_7$$

$$m_{22} = p_2 - p_3 + p_5 - p_7$$

Extends to $n \times n$ by divide&conquer

- **The master process is the controller process that**
 - **Distributes Matrix A (row-wise) to each worker process**
 - **Communicates complete Matrix B to each worker process**
- **Each worker multiplies assigned rows with matrix B**
- **Master process collects resultant Matrix C from each worker process**



Note: Not an efficient algorithm!

```

#define NRA 50 /* Rows in Matrix A */
#define NCA 40 /* Columns in Matrix A */
#define NCB 30 /* Columns in Matrix B */
#include "mpi.h"
#include <stdio.h>

int main(int argc, char **argv)
{
    int numtasks, taskid; /* No. of tasks and task identifier */
    int source, dest      /* Task id of message source and destination*/
    double a[NRA][NCA], b[NCA][NCB], c[NRA][NCB]; /* Matrix A, B, C */
    int rows, averow, extra, offset, numworkers, i, j, k, rc; /* Miscellaneous */
    MPI_Status status;

    rc = MPI_Init(&argc, &argv);
    rc |= MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    rc |= MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    if (rc != 0) printf("\nError initializing MPI or Task ID\n");
    else printf("\nTask ID = %d\n", taskid);
    numworkers = numtasks - 1;

```

```

if(taskid == 0)
{
    printf("Number of worker tasks = %d\n",numworkers);
    for (i=0; i<NRA; i++) /* Generate data for Matrix A & B */
        for (j=0; j<NCA; j++)a[i][j]= i+j;
    for (i=0; i<NCA; i++)
        for (j=0; j<NCB; j++)b[i][j]= i*j;

    averow = NRA/numworkers;
    extra = NRA%numworkers;
    offset = 0;
    for (dest=1; dest<=numworkers; dest++)
    {
        rows = (dest <= extra) ? averow+1 : averow;
        printf("\nSending %d rows to task %d\n",rows,dest);
        MPI_Send(&offset,      1,      MPI_INT,   dest,1,MPI_COMM_WORLD);
        MPI_Send(&rows,       1,      MPI_INT,   dest,1,MPI_COMM_WORLD);
        MPI_Send(&a[offset][0],rows*NCA,MPI_DOUBLE,dest,1,MPI_COMM_WORLD);
        MPI_Send(&b,          NCA*NCB, MPI_DOUBLE,dest,1,MPI_COMM_WORLD);
        offset = offset + rows;
    }
}

```

```

for (i=1; i<=numworkers; i++) /* Wait for results from workers */
{
    source = i;
    MPI_Recv(&offset,1,MPI_INT,source,2,MPI_COMM_WORLD,&status);
    MPI_Recv(&rows, 1,MPI_INT,source,2,MPI_COMM_WORLD,&status);
    MPI_Recv(&c[offset][0],rows*NCB,MPI_DOUBLE,source,2,
            MPI_COMM_WORLD, &status);
}

printf("Here is the result matrix\n"); /* Print Results */
for (i=0; i<NRA; i++)
{
    printf("\n");
    for (j=0; j<NCB; j++)
        printf("%6.2f ", c[i][j]);
}
printf ("\n");
}

```

```

if (taskid > 0) /* Worker Tasks */
{
    MPI_Recv(&offset, 1,          MPI_INT,      0, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows,  1,          MPI_INT,      0, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(&a,     rows*NCA, MPI_DOUBLE,    0, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(&b,     NCA*NCB, MPI_DOUBLE,    0, 1, MPI_COMM_WORLD, &status);
    for (k=0; k<NCB; k++)
        for (i=0; i<rows; i++)
            {
                c[i][k] = 0.0;
                for (j=0; j<NCA; j++)
                    c[i][k] = c[i][k] + a[i][j] * b[j][k];
            }
    MPI_Send(&offset, 1,          MPI_INT,      0, 2, MPI_COMM_WORLD);
    MPI_Send(&rows,  1,          MPI_INT,      0, 2, MPI_COMM_WORLD);
    MPI_Send(&c,     rows*NCB, MPI_DOUBLE,    0, 2, MPI_COMM_WORLD);
}

MPI_Finalize();
} /* End main */

```