Many of the most successful applications of parallel computing have been for various numerical algorithms. This is because these algorithms are computationally very intensive and also tend to be quite easy to parallelize. Many simulations of physical phenomena use numerical algorithms. Linear systems solvers are used for simulation fluid-flow (e.g. wind over a car or airplane), structural dynamics (e.g. the vibrational modes of the space shuttle), earthquakes, weather and much more. These linear systems solvers are typically based on various matrix routines. Fourier Transforms play an important role in both simulations and also in signal processing. Another numerical problem we will discuss is the calculation of the forces among n bodies. This is called the n-body problem and is used in simulating the formation of galaxies, the folding of proteins, and the interations within plasmas.

Matrix multiply

It is very easy to multiply matrices in parallel. For example, the following pseudocode

```
In parallel for i from 1 to n
    In parallel for j from 1 to n
       r[i,j] = sum(In parallel for k from 1 to n
                       a[i,k]*b[k,j])
```

will multiply two $n \times n$ matrices `a` and `b` and place the result in the matrix `r`. The work required by this code is $O(n^3)$ and the depth is the depth required by the `sum` ($O(\lg n)$). This gives a lot of freedom of deciding which loops should be parallelized. The problem with this triply nested loop is that it is very bad for cache behavior even on a sequential machine.

Matrix multiplication can also be parallelized using divide-and-conquer techniques. The idea is to recursively divide each matrices into four sub-matrices, multiply the sub-matrices, and add the results. For example we can write the matrix product $C = AB$ as:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

where a, b, c and d are blocks of A, e, f, g and h are blocks of B, r, s, t and u are blocks of the result C and are calculated as:

$$\begin{aligned} r &= ae + bf \\ s &= ag + bh \\ t &= ce + df \\ u &= cg + dh \end{aligned}$$

In general the four blocks of a matrix could be of different sizes. For example a could be a 1×1 matrix, and d could be an $(n-1) \times (n-1)$ matrix (b and c would be $(n-1) \times 1$

and $1 \times (n-1)$ matrices respectively). For a good divide and conquer algorithm, however, we pick the sizes so each submatrix has approximately half the rows and half the columns.

If the original matrices are $n \times n$, the divide-and-conquer technique will therefore solve the problem using 8 recursive calls on $(n/2) \times (n/2)$ matrices and then 4 matrix additions on $(n/2) \times (n/2)$ matrices. Since the matrices have $n^2/4$ elements, the four matrix additions require $O(n^2)$ work. The recurrence for work is therefore

$$W(n) = 8W(n/2) + O(n^2) = O(n^3)$$

and leads to the same work as the triply nested loop. This recursive algorithm is typically significantly faster than the triply nested loop on sequential machines since it has better cache behavior (this assumes the full matrix does not fit in the cache). In practice the recursion is stopped at some fixed size (e.g 16×16) and then the triply nested loop version is used since the overhead is less for small problems.

The algorithm also parallelizes very well. Since we can do the eight multiplications in parallel, and the addition of two matrices can be done in $O(1)$ steps, the depth is

$$S(n) = S(n/2) + O(1) = O(\log n).$$

The work required by this algorithm can be improved using Strassen's technique (see Cormen, Leiserson and Rivest). Strassen's technique saves one matrix multiplication over the above scheme, at the expense of more additions. The work complexity is

$$W(n) = 7W(n/2) + O(n^2) = O(n^{\log_2 7})$$

and the depth is the same as above.

Solving Linear Systems

For a matrix A and vector b, we want to solve for x in the equation:

$$Ax = b$$

For example, we might want to solve for x, y and z in the equations

$$\begin{aligned} 2x + 3y + 5z &= 10 \\ x + 2y + 4z &= 5 \\ 4x + 7y + 3z &= 15 \end{aligned}$$

This gives us a matrix

$$A = \begin{pmatrix} 2 & 3 & 5 \\ 1 & 2 & 4 \\ 4 & 7 & 3 \end{pmatrix}$$

and vector

$$B = \begin{pmatrix} 10 \\ 5 \\ 15 \end{pmatrix}$$

We can solve the equations using Gauss-Jordan elimination. This makes use of the properties

1. multiplying a row by a constant does not effect the solution
2. adding or subtracting one row from another does not effect the solution

The idea is to loop over the rows and use the i^{th} row to eliminate all the values in the i^{th} column. To make the implementation easier we can just append the vector b to the end of our matrix A. For our example, this would give:

$$\begin{pmatrix} 2 & 3 & 5 & 10 \\ 1 & 2 & 4 & 5 \\ 4 & 7 & 3 & 15 \end{pmatrix}$$

With this representation we can implement Gauss-Jordan elimination with the following pseudocode:

```
Loop from i = 1 to n                        % each row %
  in parallel for j from 1 to n             % each row again %
     scale = a[j,i]/a[i,i]                  % scale factor for the row %
     if i /= j then                         % don't eliminate the row itself %
       in parallel for k from 1 to n+1      % each column %
          a[j,k] = a[j,k] - scale * a[i,k]
```

When the algorithm is done, `a` will be diagonal and we can just scale to get the final answer. For example, for the above matrix:

After step 1

$$\begin{pmatrix} 2 & 3 & 5 & 10 \\ 0 & .5 & 1.5 & 0 \\ 0 & 1 & -7 & -5 \end{pmatrix}$$

After step 2

$$\begin{pmatrix} 2 & 0 & -4 & 10 \\ 0 & .5 & 1.5 & 0 \\ 0 & 0 & -10 & -5 \end{pmatrix}$$

After step 3

$$\begin{pmatrix} 2 & 0 & 0 & 12 \\ 0 & .5 & 0 & -.75 \\ 0 & 0 & -10 & -5 \end{pmatrix}$$

So the solutions are $x = 12/2 = 6$, $y = -.75/.5 = -1.5$ and $z = -5/-10 = .5$.

Since it is a triply nested loop each with n iterations the total work required by this algorithm is $O(n^3)$. Since the two inner loops are fully parallel, and the outer loop is sequential, the depth of the algorithm is $O(n)$.

A problem with Gauss-Jordan elimination is that if there are any 0s on the diagonal, the algorithm will not work since it will try to divide a number by zero. Small numbers along the diagonal can also be a problem when using finite precision arithmetic (e.g. floating point numbers). To deal with this we can use partial pivoting. The basic idea is to always pick the remaining row that has the largest value in the current column we want to eliminate. This row can be found in $O(\log n)$ depth and $O(n)$ work. Gauss-Jordan elimination with partial pivoting therefore runs in $O(n \lg n)$ depth and $O(n^3)$ work.

It is also possible to use Gauss-Jordan elimination to solve multiple simultaneous right-hand sides, and to invert a matrix. The basic idea for inverting a matrix A is that we place an identity matrix adjacent to A, for example

$$\begin{pmatrix} 2 & 3 & 5 & 1 & 0 & 0 \\ 1 & 2 & 4 & 0 & 1 & 0 \\ 4 & 7 & 3 & 0 & 0 & 0 \end{pmatrix}$$

and then run the elimination algorithm on the combined matrix. When we are done we scale the rows so the first n columns form an identity matrix, and then the remaining columns are the answer we want.

Sparse Matrices

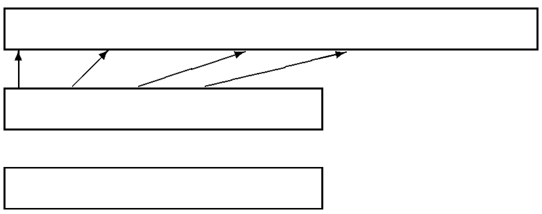
Sparse matrices are represented as adjacency lists. For each non-zero element we store its column index and its value. For example, the matrix:

$$\begin{pmatrix} 0 & 0 & 0 & .6 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & .2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ .5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .7 \end{pmatrix}$$

is represented as:

$$\begin{array}{llll} \text{Row number:} & \quad 0 & \quad 1 & \quad 2 \\ \begin{array}{l}\text{Column number:} \\ \text{Value:}\end{array} & \begin{pmatrix} 3 & 6 \\ .6 & 2 \end{pmatrix} & \begin{pmatrix} 1 & 2 \\ .2 & 3 \end{pmatrix} & \begin{pmatrix} 0 & 8 & 9 \\ .5 & 1 & .7 \end{pmatrix} \end{array}$$

In a serial implementation this could be stored as an array of linked lists. In a parallel implementation we don't want to use linked lists; instead we could use a flattened array of elements and an array of pointers to the subarrays. This is a standard method of representing sparse matrices in e.g., Fortran:



Flatten into single array

Array of pointers to subarrays of rows

Array of lengths of subarrays

This is basically the adjacency matrix representation for weighted graphs.

Sparse Matrix Transposition

Given a flattened array representation in row-major order the transpose is equivalent to storing the array in column-major order.

To transpose, sort the elements by the column index (keeping track of which row each element came from), and then replace the old column indices with the old row indices. A radix sort would typically be used, since we know in advance the range of the column indices.

Fast Fourier Transform

The Fourier transform is an analytical tool often used in science and engineering. The continuous Fourier transform of a function $f(t)$ is given by

$$F(w) = \int_{-\infty}^{\infty} f(t) e^{2\pi i w t} dt$$

where $i = \sqrt{-1}$ and $e^{2\pi i w t} = \cos(2\pi w t) + i \sin(2\pi w t)$. Conceptually, the Fourier transform decomposes the signal $f(t)$ into its components at each frequency w.

An approximation to the Fourier transform suitable for computation by digital computers is the so-called discrete Fourier transform (DFT):

$$F_k = \sum_{j=0}^{n-1} f_j e^{2\pi i j k/n}$$

DFTs are extensively used in signal processing. One common application is filtering, e.g. of signals above a certain frequency. The domain of the function transformed is often time (e.g., in sound), but can instead be space (e.g., in image filtering) or some other variable.

DFTs are also used in their own right (not as an approximation to the continuous transform) for polynomial multiplication. The straightforward algorithm to multiply two polynomials of degree n in coefficient form takes $O(n^2)$ time. The DFT of the coefficients of each polynomial gives a point-wise representation of the same polynomial. The point-wise representations allow multiplication in $O(n)$ time. The result is converted back to coefficient form by the inverse discrete Fourier transform. Using a technique called FFT (Fast Fourier Transform), the DFT and its inverse can be computed in $O(n \log n)$ time, giving a total time of $O(n \log n)$ for polynomial multiplication.

The fast Fourier transform is a divide-and-conquer algorithm that computes the discrete Fourier transform. The main idea is to separate the even and odd components and solve recursively two subproblem only half as large, as follows

$$\begin{aligned} F_k &= \sum_{j=0}^{n-1} f_j e^{2\pi i j k/n} \\ &= \sum_{j=0}^{n/2-1} f_{2j} e^{2\pi i k(2j)/n} + \sum_{j=0}^{n/2-1} f_{2j+1} e^{2\pi i k(2j+1)/n} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{j=0}^{n/2-1} f_{2j} e^{2\pi ijk/(n/2)} + w^k \sum_{j=0}^{n/2-1} f_{2j+1} e^{2\pi ijk/(n/2)} \\
&= F_k^E + w^k F_k^O
\end{aligned}
$$

where $w = e^{2\pi i/n}$ and F^E and F^O correspond to the DFT computed only on the even or odd elements of the input function, respectively. Note that $F_k = F_{k+n}$, so $F_k^E = F_{k+n/2}^E$ and $F_k^O = F_{k+n/2}^O$. This means that the two recursive calls are each of half the size, since we only need to determine elements up to $F_{n/2}^E$ and $F_{n/2}^O$ and can then just copy them for $n/2 \leq k < n$.

The recurrences for the work and step complexities of the FFT are therefore:

$$
\begin{aligned}
W(n) &= 2W(n/2) + O(n) = O(n \log n) \\
S(n) &= S(n/2) + O(1) = O(\log n)
\end{aligned}
$$

Note that to implement real algorithms with these complexities we either have to precalculate all the w^k, or be somewhat careful about calculating them on the fly.

The FFT can also be implemented in parallel using a fixed circuit of depth $\log n$, each stage consisting of $n/2$ *butterflies*. Each butterfly is a circuit with two inputs and two outputs that performs one multiplication, one addition and one subtraction.