Parallel Fast Multipole Method for Particle Simulations

DN2265: Parallel Computations for Large Scale Problems

S. Nicholas Barkas

May 14, 2008

1 Introduction to the Fast Multipole Method

The computational complexity of multiplying a dense, square matrix by a vector is in general $O(n^2)$ where n is the number of rows and columns in the matrix, as well as the length of the vector. As n becomes large, this product quickly becomes computationally infeasible to compute in a reasonable amount of time. The fast multipole method decreases the number of necessary computations down to $O(\frac{1}{\epsilon}n\log n)$, where ϵ is the desired precision of the result. With FMM, we can compute the products of much larger matrices in an acceptable amount of time.

The basic idea of the fast multipole method is to replace a single $n \times n$ matrix B that has rank $k < n$ with the product of $n \times k$ matrix C and a $k \times n$ matrix W: $B = CW$. Substituting in this matrix product to the original multiplication of B with a vector x we get

$$Bx = (CW)x = C(Wx) \tag{1}$$

[1]. Computing Wx requires kn operations, and multiplying C with the product Wx requires another kn operations. As long as we have a low enough rank k for the matrix B satisfying $2k < n$, the total number of operations, $2kn$, will be less than n^2.

2 Charged Particle Simulations in 1D

One of the possible applications of the fast multipole method is in evaluating the potential field created by a large number of charged particles. If we have N particles with the ith particle's coordinates given by a vector p_i and its charge is q_i, the potential Φ_i generated by the charge is

$$\Phi_i = \sum_{j=1, j\neq i}^{N} \frac{q_j}{||p_i - p_j||} \tag{2}$$

[1]. In one dimension the norm $||p_i - p_j||$ can be just written as the absolute value $|p_i - p_j|$, and $\mathbf{\Phi}$ can be written as a matrix-vector product:

$$\mathbf{\Phi} = \begin{pmatrix} 0 & \frac{1}{|p_1-p_2|} & \cdots & \frac{1}{|p_1-p_N|} \\ \frac{1}{|p_2-p_1|} & 0 & \cdots & \frac{1}{|p_2-p_N|} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{1}{|p_N-p_1|} & \frac{1}{|p_N-p_2|} & \cdots & 0 \end{pmatrix} \begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ q_N \end{pmatrix}. \tag{3}$$

If the particles are uniformly distributed in order on the interval $[0, 1]$, the attractive/repellant force between any pair of points p_i and p_j will become weaker the greater the difference between i and j.

Because of this, we need only be most accurate in our matrix-vector multiplication for the values close to the diagonal in the matrix. The matrix can be divided up into a collection of smaller $k \times k$ sub-matrices A_{lmk} where l, m give the coordinates in A of the upper left corner of the sub-matrix. These sub-matrices are as large as possible while their vertical and horizontal distance from the diagonal is at least k. The layout of the sub-matrices within the complete matrix is shown in figure 1.

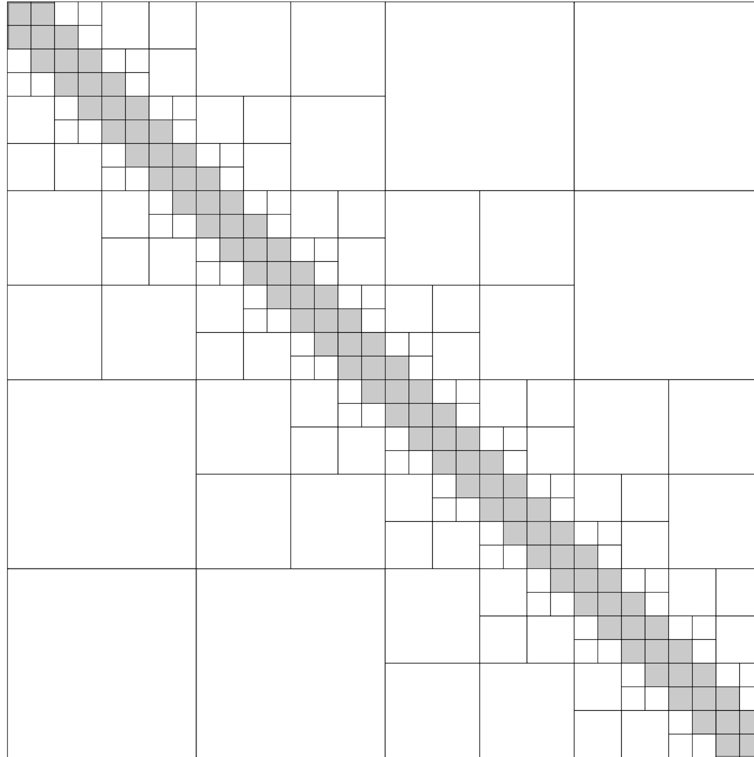

Figure 1: Layout of sub-matrices.

Lemma 1.2.1 in [1] tells us that for a sub-matrix A_{lmk} that is separated from the diagonal like those in figure 1, there exists a $k \times k$ matrix B_{lmk} with rank $J + 1$ such that

$$||A_{lmk} - B_{lmk}||_\infty \leq k4^{-J}. \tag{4}$$

B_{lmk} can be written as a product of three matrices, $B_{lmk} = \bar{B}^T \bar{A} \bar{\Gamma}$, where $\bar{B}^T$ is a $k \times J + 1$ matrix, $\bar{\Gamma}$ is $J + 1 \times k$, and $\bar{A}$ is $J + 1 \times J + 1$. $\bar{B}$ and $\bar{\Gamma}$ have columns containing the $0, 1, ..., J$ Chebyshev polynomials evaluated at the k different coordinates for the particles contained in the sub-matrix. $\bar{B}$ covers the coordinates of the particles in the sub-matrix's horizontal span, and $\bar{\Gamma}$ covers those in the vertical span. $\bar{A}$ has entries α_{pq} for the expansion coefficients of the Chebyshev polynomials.

Multiplying B_{lmk} by a k length vector x can be done as three matrix-vector multiplications: $B_{lmk}x = \bar{B}^T(\bar{A}(\bar{\Gamma}x))$. The first product, $\bar{\Gamma}x$, takes $(J + 1)k$ scalar multiplications. The next, $\bar{A}(\bar{\Gamma}x)$, is only $(J + 1)^2$. The final product $\bar{B}^T(\bar{A}(\bar{\Gamma}x))$, again takes $k(J + 1)$. These three products together have an upper bound of $3(J + 1)k$ multiplications, so as long as $3(J + 1) < k$, less multiplications are necessary than doing a regular k^2 matrix vector multiplication.

J can be chosen depending on how accurate of a result we wish to receive. Because of (4), $||B_{lmk}x - A_{lmk}x||_\infty \leq k4^{-J}||x||_\infty$, so if we want a maximum error of $\epsilon||x||_\infty$, we should choose $J \approx \log_4 \frac{k}{\epsilon}$ [1]. We use the same J for all sub-matrices for simplicity's sake, so we just take $J \approx \log_4 \frac{N}{\epsilon}$, where N is the length of the entire vector of charges for the whole matrix.

For the parts of the matrix A that are close to the diagonal, the shaded sub-matrices in figure 1, it is most efficient to just directly multiply the diagonal bands with the charge vector and add this to all the products computed with the larger off-diagonal sub-matrices computed above. These sub-matrices are small enough that $3(J + 1) \geq k$.

3 Serial FMM Algorithm

An algorithm for calculating the matrix-vector product for the problem described in the previous problem follows. This algorithm assumes that the matrix size $N = 2^n$, for some positive integer n, and that J is a positive integer.

```
Given vector x, function a(l, m, k) to get exact k x k sub-matrix A_{lmk},
function b(l, m, k) that calculates approximation B_{lmk}, matrix size N, and
J =~ log_4 N/eps, where eps is the desired error size.

Algorithm:
----------
# Zero-initialize result vector y
y = zeros(N, 1)

# Find smallest k value. Only compute product of B_{lmk} with section of
# vector if k > 3(J+1). Otherwise it is more efficient to just do a regular
# matrix-vector multiplication using A_{lmk}.
for (startk = 8; startk <= N/4; startk *= 2)
    if (startk > 3*(J+1))
        k = startk
        break
    endif
endfor

# Multiply sub-matrix approximations by corresponding portions of x
while (k <= N/4)
    # Lower triangular loop
    top = 2*k
    l = top
    m = 0
    while (l < N)
        y(l:l+k) += b(l, m, k) * x(m:m+k)
        l += k
        y(l:l+k) += b(l, m, k) * x(m:m+k)
        m += k
        y(l:l+k) += b(l, m, k) * x(m:m+k)
        l += k
        m += k
    endwhile

    # Upper triangular loop
    left = 2*k
    l = 0
    m = left
    while (m < N)
        y(l:l+k) += b(l, m, k) * x(m:m+k)
        m += k
        y(l:l+k) += b(l, m, k) * x(m:m+k)
        l += k
        y(l:l+k) += b(l, m, k) * x(m:m+k)
        m += k
        l += k
    endwhile

    # Double sub matrix size
    k *= 2
endwhile
```

```
# Now multiply the three diagonal bands of exact sub-matrices of the matrix by
# corresponding portions of x and add that to the result vector.
k = startk / 2
l = 0
m = 0
y(0:k) += a(0, 0, k) * x(0:k)
l += k;
while (l < N)
    # Sub-diagonal sub-matrix
    y(l:l+k) += a(l, m, k) * x(m:m+k)
    # Super-diagonal sub-matrix
    y(m:m+k) += a(m, l, k) * x(l:l+k)
    m += k
    # Main-diagonal sub-matrix
    y(l:l+k) += a(l, m, k) * x(m:m+k)
    l += k
endwhile
```

4 Parallelization of the Algorithm

A very simple way to parallelize this algorithm is to have individual processes compute only the products of all sub-matrices of a given size k, with another process responsible for multiplying the diagonal portions of the matrix directly. In very simplified pseudo-code:

```
k = startk * 2^rank

if (k <= N/4)
    y = upper_triangular_loop(k, x)
    y += lower_triangular_loop(k, x)
else if (k == N/2)
    y = multiply_diagonals(x)
else
    # Extra processes do nothing
    y = zeros(N, 1)
end

# Sum computed products across all processes
reduce(y, sum)
```

This is only a good parallelization strategy if the amount of work is evenly distributed among all processes. In figure 2 we see that the total number of operations involved in all the matrix-vector multiplications in a given level is indeed relatively close to being the same for each process. The biggest difference is an order of magnitude, and this only happens with very small ϵ for big matrices, because there are a great many multiplications of diagonal elements that must happen in this case. This disparity could potentially be decreased by splitting up the work of doing the diagonal multiplications among more than one process.

5 Experimental Results

The parallel version of the algorithm described above was implemented in C++ and run on lenngren with matrices of size $N = 2^{10}, 2^{15}$, and 2^{20}. Each process read in a vector of randomly generated charges for the particles from the same text file on shared storage, so the only MPI communication necessary was a call to `MPI_Allreduce()` at the end to sum each process' local product vector together.

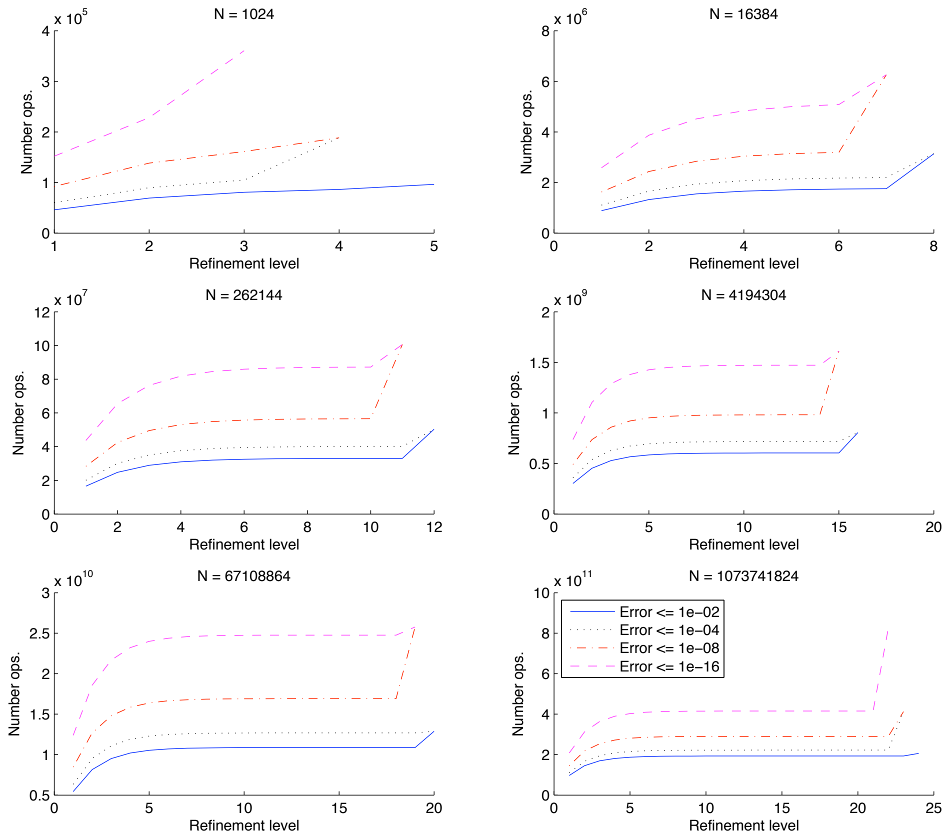

Figure 2: Number of operations per level of refinement (i.e. different k sizes) for various matrix sizes N and maximum error values ϵ.

The results from initial runs were not terribly encouraging. The first process, which handled the multiplications of the smallest sub-matrices, took nearly 300 times as long as the process that dealt with the largest $(N/4 \times N/4)$ sub-matrices. In the calculations used to generate the plots in figure 2, the time taken to calculate Chebyshev polynomials for the approximation matrices B_{lmk} was neglected, and it is likely that this time becomes significant when there are many small sub-matrices near the diagonal that need to be approximated.

Through a bit of experimentation, the initial code was modified to get a more even distribution of work among processes. First of all, the initial starting k was originally chosen to be the smallest power of 2 that is larger than $3(J + 1)$. This value is now multiplied by 8 so that there will be a wider band of sub-matrices on the diagonal that are directly multiplied with the charge vector. This provides us with fewer small matrices that need the Chebyshev calculations than were required in the original program. Additionally, if there are five or more processes being used, the work to calculate the products of the smallest off-diagonal sub-matrices is split between the first two processes (one handles the band in the upper triangle, the other the band in the lower triangle of the matrix) instead of all being done by one. Third, the work to calculate the products of the diagonal matrix entries, since there are many more of those now that the starting k is eight times bigger, is divided among the last three processes that also are handling multiplying the biggest sub-matrix bands (largest k values). One process deals with the main diagonal exact sub-matrix multiplications, and two more processes for each of the super-diagonal and sub-diagonal bands.

This strategy turned out to lead to a much better distribution of work, as we see in figure 3 and tables 1 and 2. These results were obtained by running the improved code for matrices with size $N = 2^{15}$ and 2^{20}, again with ϵ ranging from 10^{-2} to 10^{-8}. For the smaller matrix the product was also computed with a single process using the standard, "brute force" $O(N^2)$ matrix-vector multiplication method to ensure

that the FMM code is giving accurate results.

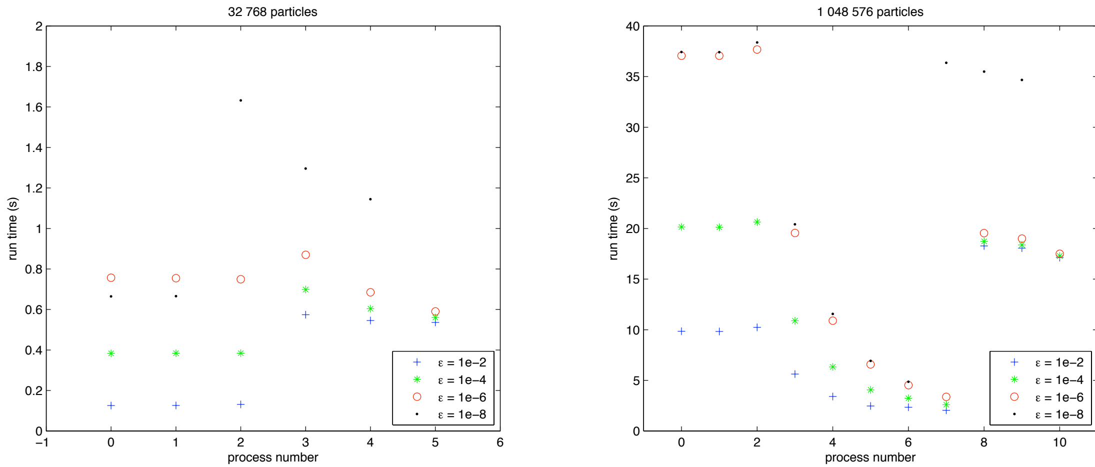

Figure 3: Run times for each process using various different max error values ϵ, for two different matrix sizes.

ϵ	Min run time	Max run time	Slowest process time multiple	Communication time	Norm of diff. between FMM and brute force
1e-2	0.126	0.574	4.56	3.43e-03	6.213e-05
1e-4	0.383	0.698	1.82	4.27e-03	6.102e-08
1e-6	0.591	0.870	1.47	4.26e-03	3.779e-08
1e-8	0.664	1.632	2.46	4.28e-03	3.754e-08

Table 1: Run times per process for 2^{15} particles. ϵ is the maximum error bound given to the FMM code, slowest process time multiple is the number of times longer the slowest process takes than the fastest, and the error norm compared to brute force is the norm of the difference of the product vectors calculated with FMM and a "brute force" $O(n^2)$ matrix-vector multiplication.

ϵ	Min run time	Max run time	Slowest process time multiple	Communication time
1e-2	2.041	18.278	8.96	0.109
1e-4	2.600	20.639	7.94	0.109
1e-6	3.375	37.676	11.16	0.108
1e-8	4.864	38.368	7.89	0.107

Table 2: Run times per process for 2^{20} particles. No error norms with brute force are included here, as it would take a very long time to multiply such a large matrix-vector product directly.

6 Conclusion

The primary benefit of this parallelization strategy is that it is very easy to implement once we have code that can do the serial algorithm. Also, MPI communication time is kept very small, as we see in tables 1 and 2, because the only time processes need to communicate with one another is during the global summation of product vectors at the end.

Table 3 shows the speedup and parallel efficiency of the final program for both matrix sizes tried. Efficiency is a bit better overall for the smaller N value used.

There are some drawbacks to this parallel algorithm, though. It is likely that it will not work equally well for all possible matrix sizes, given the differences in efficiency just on the two sizes tried. The size of the problem and desired accuracy dictate exactly the number of processes to use, so if one has too many or too few computers the algorithm cannot be run with optimal efficiency. This code also can

$N = 2^{15}$		
ϵ	Speedup	Efficiency
1e-2	3.53	0.59
1e-4	4.29	0.72
1e-6	5.04	0.84
1e-8	3.30	0.66

$N = 2^{20}$		
ϵ	Speedup	Efficiency
1e-2	5.40	0.49
1e-4	6.86	0.62
1e-6	5.63	0.51
1e-8	6.85	0.68

Table 3: Speedup and parallel efficiency with different error values for both matrices.

only handle the very specific case where the number of particles is a power of two, they are uniformly distributed, and they are distributed only in one dimension. The fast multipole method is applicable to more general problems in higher dimensions, but a completely different strategy for splitting up the matrix into different $k \times k$ pieces that can be multiplied in better than k^2 time is necessary.

Despite these drawbacks, the simplicity of the algorithm makes it potentially useful. Also, even though the parallel efficiency of the algorithm in terms of the performance of a serial FMM code is not all that close to the optimal efficiency of 1, it is substantially faster than a "brute force" matrix- vector multiplication. The time for a single process to multiply the matrix with $N = 2^{15}$ using the $O(N^2)$ algorithm was about 32.75 seconds. If this workload was spread among six processors with perfect efficiency, the brute force algorithm would still take four times as long as the slowest performing FMM calculation (where $\epsilon = 10^{-8}$) on those same six processes. The difference becomes much more dramatic as we increase the number of particles to 2^{20}. Each time the number of particles is doubled, brute force run time will be multiplied by a factor of four. So, we can estimate that the brute force run time would take $32.75 \cdot 4^5$ seconds for $N = 2^{20}$, which is over nine hours. Even if this work was evenly divided among the ten processes used in the $\epsilon = 10^{-8}$ run of the same problem using FMM, the nearly one hour run time is a lot worse than the ≈ 40 seconds that the parallel FMM code took.

References

[1] M. Hanke, *Lecture notes: Advanced numerical methods*, 2005.