A Parallel Solver for Singular Integrals

Leonardo Borges and Prabir Daripa
Department of Mathematics
Texas A&M University
College Station, TX-77843, U.S.A.

Abstract *A parallel version of a fast algorithm for singular integral transforms [6] is presented. The parallel version only utilizes a linear neighbor-to-neighbor communication path which makes the algorithm very scalable and suitable for any distributed memory architecture.*

Keywords: Fast Algorithm, Parallel Algorithm, Singular Integral Transforms, Distributed Memory, Partial Differential Equation

1 Introduction

Fast algorithms for the accurate evaluation of singular integral operators are of fundamental importance in solving elliptic partial differential equations using integral equation representations of their solutions. For example, the following singular integral transform arises in solving Beltrami equation [4].

$$T_m h(\sigma) = -\frac{1}{\pi} \iint\limits_{B(0;1)} \frac{h(\zeta)}{(\zeta - \sigma)^m} d\xi d\eta, \qquad (1)$$

$\zeta = \xi + i\eta$, where h is a complex valued function of σ defined on $B(0;1) = \{z\colon\ |z| < 1\}$, for a suitable finite positive integer m. Daripa [4, 5] used the Beltrami equation for quasiconformal mappings [4] and for inverse design of airfoils [2].

The use of quadrature rules to solve (1) presents two major disadvantages: First, the complexity of the method is $\mathcal{O}(N^4)$ for a N^2 net of grid points. In terms of computational time, it represents an impracticable approach for large problems sizes. Also, quadrature methods deliver poor accuracy when employed to evaluate certain singular integrals. Daripa [3, 6] presented a fast and accurate algorithm for rapid evaluation of the singular integral (1). The resulting method has theoretical computational complexity $\mathcal{O}(\log_2 N)$ per point which represents substantial savings in computational time when compared with the complexity $\mathcal{O}(N^2)$ of quadrature rules.

In this paper we summarize the parallel algorithm to solve the singular integral operator (1) presented in [**?**]. The implementation is based on having two simultaneous fluxes of data traversing processors in a linear configuration. It allows us overlap computational work simultaneously with data-exchanges, and minimize the number of messages in the communication channels. The algorithm is very scalable and independent of a particular computer architecture.

2 The Algorithm

The fast algorithm to evaluate the singular integral transform (1) was developed in [4, 5]. The method divides the interior of the unit disk $B(0;1)$ into a collection of annular regions so that the integral is expanded in terms Fourier series on each angular direction. The result is a collection of Fourier coefficients where each group of coefficients depends on the radius of the associated angular region. The good performance of the algorithm is due to the use of scaling one-dimensional integrals in the radial direction to produce the solution over the entire domain. Specifically, scaling factors are employed to define exact recursive relations

which evaluate the Fourier coefficients of (1). Then inverse Fourier transforms are applied on each angular region to obtain the solution of the integral. The mathematical foundation of the algorithm was presented in [5]:

Theorem 2.1 *If $T_m h(\sigma)$ exists in the unit disk as a Cauchy principal value, and $h(re^{i\alpha}) = \sum_{n=-\infty}^{\infty} h_n(r)e^{in\alpha}$, then the nth Fourier coefficient $S_{n,m}(r)$ of $T_m h(re^{i\alpha})$ can be written as*

$$S_{n,m}(r) = \begin{cases} C_{n,m}(r) + B_{n,m}(r), & r \neq 0, \\ 0, & r = 0 \text{ and } n \neq 0, \\ S_{0,m}(0), & r = 0 \text{ and } n = 0, \end{cases} \quad (2)$$

with $B_{n,m}(r)$ and $S_{0,m}(0)$ as in [5], and $C_{n,m}(r)$ defined by

$$\begin{cases} \frac{2(-1)^{m+1}}{r^{m-1}}\binom{-n-1}{m-1} \int_0^r \alpha_{n,m}(r)\, d\rho, & n \leq -m, \\ 0, & -m < n < 0, \\ -\frac{2}{r^{m-1}}\binom{m+n-1}{m-1} \int_r^1 \alpha_{n,m}(r)\, d\rho, & n \geq 0, \end{cases}$$

where

$$\alpha_{n,m}(r) = \left(\frac{r}{\rho}\right)^{m+n-1} h_{m+n}(\rho) \quad (3)$$

The strength of the above theorem is evident when considering the unit disk $\overline{B(0;1)}$ discretized by $N \times M$ lattice points with N equidistant points in the angular direction and M equidistant points in the radial direction. Let $0 = r_1 < r_2 < \ldots < r_M = 1$ be the radii defined on the discretization. The following corollaries of Theorem 2.1 are extracted from [5]:

Corollary 2.1 *It follows from (3) that $C_{n,m}(1) = 0$ for $n \geq 0$, and $C_{n,m}(0) = 0$ for $n \leq -m$. We repeat from (3) that $C_{n,m}(r) = 0$ for $-m < n < 0$ for all values of r in the domain.*

Corollary 2.2 *If $r_j > r_i$ and*

$$C_{n,m}^{i,j} = \begin{cases} \frac{2(-1)^{m+1}}{r_j^{m-1}}\binom{-n-1}{m-1} \int_{r_i}^{r_j} \alpha_{n,m}(r_j)\, d\rho \\ \qquad\qquad \text{for } n \leq -m, \text{ and} \\ \frac{2}{r_i^{m-1}}\binom{m+n-1}{m-1} \int_{r_i}^{r_j} \alpha_{n,m}(r_i)\, d\rho \\ \qquad\qquad \text{for } n \geq 0, \end{cases}$$

then

$$C_{n,m}(r_j) = \left(\frac{r_j}{r_i}\right)^n C_{n,m}(r_i) + C_{n,m}^{i,j}, \quad n \leq -m,$$

$$C_{n,m}(r_i) = \left(\frac{r_i}{r_j}\right)^n C_{n,m}(r_j) - C_{n,m}^{i,j}, \quad n \geq 0.$$

Corollary 2.3 *Let $0 = r_1 < r_2 < \cdots < r_M = 1$, then*

$$C_{n,m}(r_l) = \sum_{i=2}^{l} \left(\frac{r_l}{r_i}\right)^n C_{n,m}^{i-1,i} \quad (4)$$

for $n \leq -m$ and $l = 2, \ldots, M$, and

$$C_{n,m}(r_l) = -\sum_{i=l}^{M-1} \left(\frac{r_l}{r_i}\right)^n C_{n,m}^{i,i+1} \quad (5)$$

for $n \geq 0$ and $l = 1, \ldots, M-1$.

Corollary 2.2 defines the recursive relations that are used in the calculation of the Fourier coefficients $S_{n,m}$ of the singular integrals in (1). It prescribes two recursive relations based on the sign of the index n of the Fourier coefficient $S_{n,m}$ being evaluated. We will address the coefficients (such as $C_{n,m}$) with index values $n \leq -m$ as *negative modes* and the ones with index values $n \geq 0$ as *positive modes*. Equation (4) shows that negative modes are built up from the smallest radius r_1 towards the largest radius r_M. Conversely, equation (5) constructs positive modes from r_M towards r_1.

3 Parallel Implementation

The fast algorithm in Section 2 requires multiple fast Fourier transforms (FFT) to be performed. For the sake of a more clear explanation, let P be the number of available processors and M be a multiple of P. There are

distinct strategies to solve multiple FFTs in parallel systems [1, 7]. We adopt an improved implementation of parallel calls to sequential FFTs by assigning grid points within a group of circles to each processor. One sequential N-point FFT algorithm is available on each processor. For a total of P processors, the M sequences are distributed between processors so that each one performs M/P calls to the FFT routine. The FFT transforms present in the algorithm contribute the most to the computational cost of the algorithm. Also, each FFT calculation presents a high degree of data dependency between grid points $r_l e^{2\pi i k/N}$ for a fixed radius r_l, $l \in [1, M]$. Data locality is preserved by performing Fourier transforms within a processor. To obtain a more compact notation we define $\gamma(j) = jM/P$. Given P processors p_j, $j = 0, \ldots, P-1$, data is distributed so that processor p_j contains the data associated with the grid points $r_l e^{2\pi i k/N}$, $k \in [1, N]$ and $l \in [\gamma(j) + 1, \gamma(j+1)]$. Thus, each FFT can be evaluated in place, without communication. This approach is free of network congestion. Moreover, all M/P Fourier transforms can be performed simultaneously. In practice, it means that mechanisms like bit-reversal and calls to sines and cosines are computed only once on each processor. Other strategies for solving the multiple FFTs present in the algorithm are discussed in [**?**].

Redundant computations can be used to improve performance of the parallel algorithm. Since the algorithm employs equations from Corollaries 2.2 and 2.3 that only utilize consecutive radii, only terms of the form $C_{n,m}^{l-1,l}$ and $C_{n,m}^{l,l+1}$, $l \in [\gamma(j) + 1, \gamma(j+1)]$, are required in processor p_j. Notice that p_j already evaluates the Fourier coefficients $h_n(r_l)$, $l \in [\gamma(j)+1, \gamma(j+1)]$. In the case of a numerical integration based on the trapezoidal rule and $m = 1$, for example, only the Fourier coefficients for $l = jM/P$ and $l = (j+1)M/P + 1$ must be added to the set of known coefficients for processor p_j. That is, if the initial data is overlapped so that each processor evaluates coefficients for radii r_l, $l \in [\gamma(j), \gamma(j+1) + 1]$, there is no need for communication. The number of circles whose data overlap between any two neighbor processors remain fixed regardless of the total number of processors in use. Consequently, this strategy does not compromise the scalability of the algorithm.

An even distribution of computational load is obtained by splitting the computational work when performing recurrences (4) and (5). We define the following *partial sums* for each processor p_j, $j = 0, \ldots, P-1$. For the case $n \leq -m$ let the initial partial sum t_0^- obtained in processor p_0 be

$$t_0^-(n) = r_{\gamma(1)+1}^n \sum_{i=2}^{\gamma(1)} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i},$$

and the remaining partial sums t_j^- obtained in processors p_j, $j = 1, \ldots, P-1$, given by

$$t_j^-(n) = r_{\gamma(j+1)+1}^n \sum_{i=\gamma(j)+1}^{\gamma(j+1)} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i}.$$

Similarly, for the case $n \geq 0$ let the initial partial sum t_{P-1}^+ obtained in processor p_{P-1} be

$$t_{P-1}^+(n) = r_{\gamma(P-1)}^n \sum_{i=\gamma(P-1)+1}^{M-1} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i,i+1},$$

and the remaining partial sums t_j^+ obtained in processors p_j, $j = 0, \ldots, P-2$, given by

$$t_j^+(n) = r_{\gamma(j)}^n \sum_{i=\gamma(j)+1}^{\gamma(j+1)} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i,i+1}.$$

Since coefficients $C_{n,m}^{i-1,i}$ $(n \leq -m)$ and $C_{n,m}^{i,i+1}$ $(n \geq 0)$ are already stored in processor p_j when $i \in [\gamma(j)+1, \gamma(j+1)]$, partial sums t_j^- and t_j^+ can be computed locally in processor p_j. Although computations as defined above may look unstable, partial sums t_j^- and t_j^+ can be obtained by performing very stable computations as described in [**?**].

If the *accumulated sums* $\hat{s}_j^-$ and $\hat{s}_j^+$, $j = 0, \ldots, P-1$, are defined for $n \leq -m$ as

$$\begin{cases} \hat{s}_0^-(n) = t_0^-(n), \\ \\ \hat{s}_j^-(n) = \left(\frac{r_{\gamma(j+1)+1}}{r_{\gamma(j)+1}}\right)^n \hat{s}_{j-1}^-(n) + t_j^-, \end{cases} \tag{6}$$

and for $n \geq 0$ as

$$\begin{cases} \hat{s}^+_{P-1}(n) = t^+_{P-1}(n), \\ \\ \hat{s}^+_j(n) = \left(\frac{r_{\gamma(j)}}{r_{\gamma(j+1)}}\right)^n \hat{s}^+_{j+1}(n) + t^+_j, \end{cases} \tag{7}$$

then we have a recursive method to accumulate partial sums t^-_j and t^+_j computed in processors p_j, $j = 0, \ldots, P-1$. Accumulated sums $\hat{s}^-_j$ and $\hat{s}^+_j$ can now be used to calculate coefficients $C_{n,m}$ locally on each processor. Given a fixed radius r_l, the associated data belongs to processor p_j where $l \in [\gamma(j)+1, \gamma(j+1)]$. Computations in p_j only make use of accumulated sums from neighbor processors. For $n \leq -m$ local updates in processor p_0 are performed as described in Corollary 2.2. Local updates in processors p_j, $j = 1, \ldots, P-1$, use the accumulated sum $\hat{s}^-_{j-1}$ from the previous processor:

$$\begin{cases} C_{n,m}(r_{\gamma(j)+1}) = \hat{s}^-_{j-1}(n) + C^{\gamma(j),\gamma(j)+1}_{n,m} \\ \\ C_{n,m}(r_l) = \left(\frac{r_l}{r_{l-1}}\right)^n C_{n,m}(r_{l-1}) + C^{l-1,l}_{n,m}, \end{cases} \tag{8}$$

For $n \geq 0$ local updates in processor p_{P-1} are also performed as described in Corollary 2.2. Local updates in processors p_j, $j = 0, \ldots, P-2$ use the accumulated sum $\hat{s}^+_{j+1}$ from the next processor:

$$\begin{cases} C_{n,m}(r_{\gamma(j+1)}) = -\hat{s}^+_{j+1}(n) - C^{\gamma(j+1),\gamma(j+1)+1}_{n,m} \\ \\ C_{n,m}(r_l) = \left(\frac{r_l}{r_{l+1}}\right)^n C_{n,m}(r_{l+1}) - C^{l,l+1}_{n,m}. \end{cases} \tag{9}$$

The advantage of using equations (8) and (9) over original recurrences in Corollary 2.2 is that accumulated sums $\hat{s}^-_j$ and $\hat{s}^+_j$ are obtained using partial sums t^-_j and t^+_j. Since all partial sums can be computed locally (without message passing) and hence simultaneously, the sequential bottleneck of the original recurrences is removed. The only sequential component in this process is the message-passing mechanism to accumulate the partial sums. The notation in equations (6) and (7) will be simplified to allow a clear exposition:

- Relation $s^-_j = s^-_{j-1} + t^-_j$ represents the updating process in recurrence (6), and

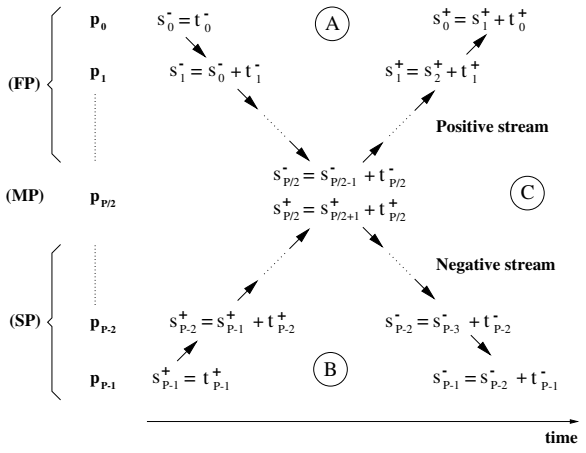


Figure 1: Message distribution in the algorithm. Two streams of neighbor-to-neighbor messages cross communication channels simultaneously.

- Relation $s^+_j = s^+_{j+1} + t^+_j$ represents updating (7).

Figure 1 presents the general structure for the algorithm. Processors are divided into three groups: processor $p_{P/2}$ is defined as the *middle processor* (MP), processors $p_0, \ldots, p_{P/2-1}$ are the *first half* processors (FP), and $p_{P/2+1}, \ldots, p_{P-1}$ are in the *second half* (SP). We define a *negative stream* (negative pipe): A message started from processor p_0 containing the values $s^-_0 = t^-_0$ and passed to the neighbor p_1. Generically, processor p_j receives the message s^-_{j-1} from p_{j-1}, updates the accumulated sum $s^-_j = s^-_{j-1} + t^-_j$, and sends the new message s^-_j to processor p_{j+1}. It corresponds to the downward arrows in Figure 1. In the same way, processors on the second half start computations for partial sums s^+. A *positive stream* starts from processor p_{P-1}: processor p_j receives s^+_{j+1} from p_{j+1} and sends the updated message $s^+_j = s^+_{j+1} + t^+_j$ to p_{j-1}. The positive stream is formed by the upward arrows in Figure 1. The resulting algorithm is composed by two simultaneous streams of neighbor-to-neighbor communication, each one with messages of length $N/2$.

Note from Figure 1 that negative and positive streams arrive at the middle processor simultaneously due to the symmetry of the communication structure. In [**?**] we describe an efficient interprocessor coordination scheme which leads to having local computational work performed simultaneously with the message passing mechanism. In short, it consists on having messages arriving and leaving the middle processor as early as possible so that idle times are minimized: Any processor p_j in the first half (FP) obtains the accumulated sum s_j^- and immediately sends it to the next neighbor processor p_{j+1}. Computations for partial sums t_j^+ only start after the negative stream have been sent. It correspond to evaluate t_j^+ within region A in Figure 1. Similarly, any processor p_j in the second half (SP) performs all the computations and message-passing work for the positive stream prior to the computation of partial sums t_j^- in region B. This mechanism minimizes delays due to interprocessor communication. In fact, in [**?**] we compare this approach against other parallelization strategies by presenting complexity models for distinct parallel implementations. The analysis shows the high degree of scalability of the algorithm.

Two variants of the parallel algorithm can be devised. As defined above, calculations of coefficients $C_{n,m}$ in equations (8) and (9) are locally performed on each processor after positive and negative streams are completed. It means that all $C_{n,m}$ are computed within region C in Figure 1. For the first half processors, Fourier coefficients associated with negative modes ($n \leq -m$) only depend on the accumulated sums s^-, which are already available in region A. It indicates that these coefficients can be obtained earlier than in region C. The tradeoff here is that lengthy computations for the Fourier coefficients may delay the positive stream and, consequently, delay all the next processors waiting for a message from the positive stream. Thus, the best choice depends on the problem size given by N and M, and also the number of processors P. The same idea applies for processors on the second half: Fourier coefficients associated with positive modes ($n \geq 0$) can be evaluated within region B. We distinguish these variants of the algorithm by defining

- the *late computations algorithm* as the original version presented here where each processor evaluates all the Fourier coefficients after all the neighbor-to-neighbor communications have been completed; and
- the *early computations algorithm* as the version in which half of the Fourier coefficients are evaluated right after one of the streams have crossed the processor.

4 Numerical Results

Equation (1) was solved for $m = 1$. Problem configurations where $N = 512, 1024, 2048$, and $M = 600, 1200, 2400$. Parallel experiments where carried out on an Intel Paragon computer using up to 60 processors. Two experiments were performed to compare the late and early computations versions and to observe the scalability of the algorithm. For a fixed number $N = 512$ of angular grid points three distinct numbers of radial grid points were employed: $M = 600, 1200$ and 2400. Figure 2(a) presents actual running times for both late and early computations algorithms when increasing the number of processors from $P = 20$ to $P = 60$. As it was expected, larger levels of granularity, i.e. larger problems sizes, imply more computational work performed locally on each processor and, consequently, better performance for the algorithm. Similarly, three distinct numbers of angular grid points ($N = 512, 1024$ and 2048) were adopted on a discretization with a fixed number of radial grid points $M = 600$, as shown in Figure 2(b). When comparing both versions of the algorithm, one can notice the influence of problem sizes and number of processors on the performance of the early computations version. For a relatively smaller problem size, the strategy of evaluating terms $C_{n,m}$ earlier only incurs on delays for communication. As a consequence, the problem of size $N = 512$

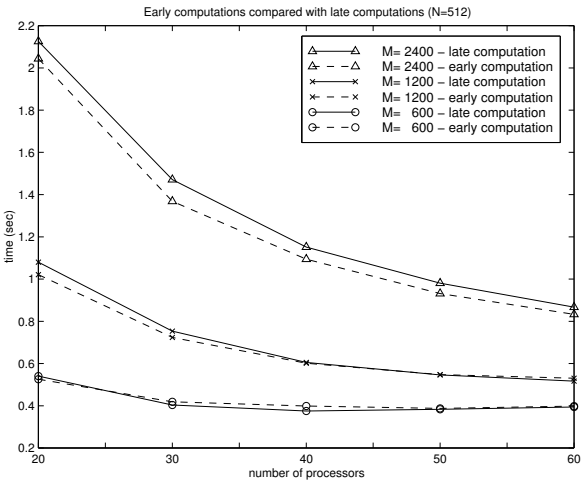
and $M = 600$ presents a better performance for the late computations algorithm. In the case of a large amount of data per processor, early computations outperform late computations. A tradeoff between both approaches can be observed for $N = 512$ and $M = 1200$ in Figure 2(a). For a higher level of computational granularity on each processor, i.e. larger pieces of input data per processor, early computations deliver results faster. However, as the number of processors increases, the late computations algorithm presents the best results. It shows that the choice between early or late computations depend the problem size and the number of processors available.

In practice, performance critically depends on the data-mapping and interprocessor coordination process adopted for a coarse-grain parallel architecture. By limiting the amount of data based on memory constraints imposed by a single-processor version of the algorithm, one cannot perform numerical experiments to validate a timing model for coarse-grain data distribution when using large values of P. To allow the usage of large problem sizes to observe speedups in a coarse-grained data distribution, we define *modified speedups* $S^{[20]}$ which are calculated by comparing performance gains over the parallel algorithm running on a starting configuration with 20 processors. Specifically we have

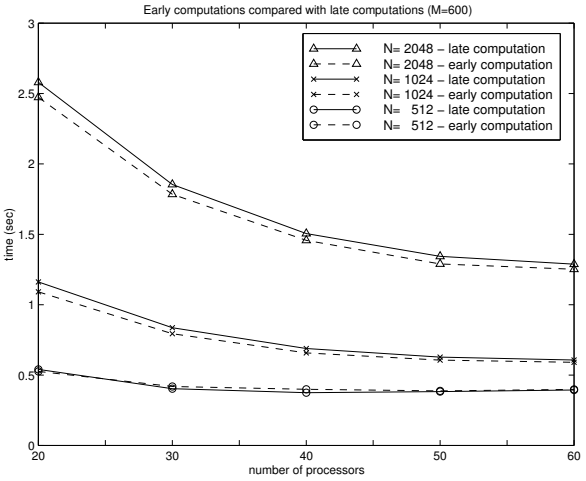
$$S^{[20]} = \frac{20 \cdot T_{20}}{T_p},$$

where T_P is the parallel running time obtained using P processors. Comparing with the actual definition for relative speedup, the analysis allows us to observe the performance of the algorithm for a large number of processors without having strong constraints on problem sizes.

Figures 3(a) and 3(b) describe the scalable performance of the late computations algorithm. They present modified speedups $S^{[20]}$ for all problem configurations. Recall that the performance of the parallel algorithm is mainly determined by the number of processors and the communication overhead which also depends on N. Although both configurations with either M or N fixed present running times for problems of same order $N \times M$, one can notice that the algorithm is more sensitive to changes in N due to larger messages. In Figure 3(a), message lengths are constant with $N = 512$ and only the problem of size $M = 600$ cannot scale up to 60 processors. For $M = 1200$ and 2400, both curves indicate

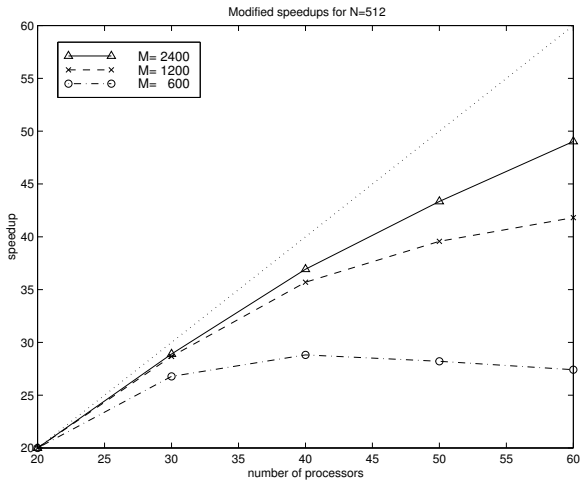


(a)

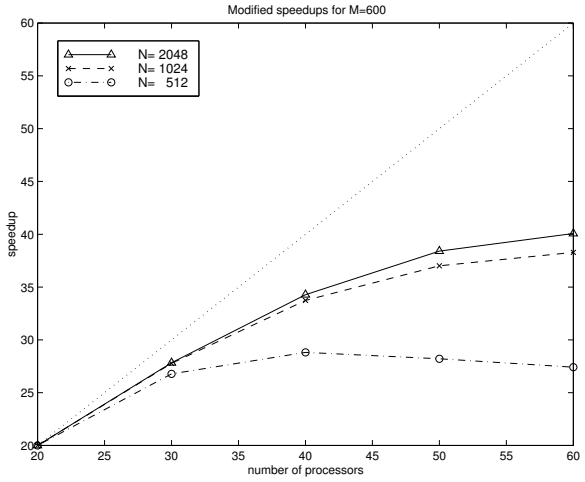


(b)

Figure 2: Comparison between early and late computations for the coefficients $C_{n,m}$ of the singular integral (1): (a) timings for a fixed number of angular points $N = 512$; (b) timings for a fixed number of radial points $M = 600$.



(a)



(b)

Figure 3: Modified speedups $S^{[20]}$ for $20, 30, 40, 50$ and 60 processors: (a) for $N = 512$ fixed; (b) for $M = 600$ fixed.

that more processors would deliver even larger speedups. In the case of Figure 3(b), problems of size $N = 1024$ and 2048 present increasing message lengths and are almost at the highest value for speedup, that is, after adding a few more processors to the system, no more significant savings on running times would be observed.

Acknowledgments

This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. TARP-97010366-030.

References

[1] W. Briggs, L. Hart, R. Sweet, and A. O'Gallagher, *Multiprocessor FFT methods*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. 27–42.

[2] P. Daripa, *On applications of a complex variable method in compressible flows*, J. Comput. Phys., 88 (1990), pp. 337–361.

[3] ——, *A fast algorithm to solve nonhomogeneous Cauchy-Riemann equations in the complex plane*, SIAM J. Sci. Stat. Comput., 6 (1992), pp. 1418–1432.

[4] ——, *A fast algorithm to solve the Beltrami equation with applications to quasiconformal mappings*, J. Comput. Phys., 106 (1993), pp. 355–365.

[5] P. Daripa and D. Mashat, *An efficient and novel numerical method for quasiconformal mappings of doubly connected domains*, Num. Algor., 18 (1998), pp. 159–175.

[6] ——, *Singular integral transforms and fast numerical algorithms*, Num. Algor., 18 (1998), pp. 133–157.

[7] R. Hockney and C. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms*, Adam Hilger, Bristol, 1981.