A parallel version of a fast algorithm for singular integral transforms

Leonardo Borges and Prabir Daripa *

Department of Mathematics, Texas A&M University, College Station, TX 77843-3368, USA
E-mail: {borges;daripa}@math.tamu.edu

Received 10 April 1999; revised 11 February 2000
Communicated by C. Brezinski

The mathematical foundation of an algorithm for fast and accurate evaluation of singular integral transforms was given by Daripa [9,10,12]. By construction, the algorithm offers good parallelization opportunities and a lower computational complexity when compared with methods based on quadrature rules. In this paper we develop a parallel version of the fast algorithm by redefining the inherently sequential recurrences present in the original sequential formulation. The parallel version only utilizes a linear neighbor-to-neighbor communication path, which makes the algorithm very suitable for any distributed memory architecture. Numerical results and theoretical estimates show good parallel scalability of the algorithm.

Keywords: singular integral transform, fast algorithm, parallel processing, distributed memory, pipelining algorithm

AMS subject classification: 65E05, 65R10, 65Y05, 65Y20

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 equations [10]:

$$T_m h(\sigma) = -\frac{1}{\pi} \iint_{B(0;1)} \frac{h(\zeta)}{(\zeta - \sigma)^m} \, \mathrm{d}\xi \, \mathrm{d}\eta, \quad \zeta = \xi + \mathrm{i}\eta, \tag{1}$$

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 [12]. Daripa [10,11] used the Beltrami equation for quasiconformal mappings [10] and for inverse design of airfoils [8]. Singular integral operators arise in solving problems in partial differential equations [3,4,7,9,17–19], fluid mechanics [2,8], and electrostatics [16] using integral equation methods.

* Corresponding author.

© J.C. Baltzer AG, Science Publishers

The use of quadrature rules to evaluate (1) presents two major disadvantages: First, the complexity of the method is O(N^4) for an N^2 net of grid points. In terms of computational time, it represents an impracticable approach for large problem sizes (also, quadrature methods deliver poor accuracy when employed to evaluate certain singular integrals). Daripa [9,10] and Daripa and Mashat [12] presented a fast and accurate algorithm for rapid evaluation of the singular integral (1). The algorithm is based on some recursive relations in Fourier space together with FFT (fast Fourier transform). The resulting method has theoretical computational complexity O($N^2 \log_2 N$) or equivalently O($\log_2 N$) per point, which represents substantial savings in computational time when compared with quadrature rules. Furthermore, results are more accurate because the algorithm is based on exact analyses.

Practical industrial problems may handle an excessive amount of data. This class of applications presents large memory requirements and intensive floating point computations. Consequently, the design of fast algorithms does not eliminate the need for improved computing resources. An immediate consequence is the demand for parallel computing. Distributed-memory multiprocessors provide the resources to deal with large-scale problems. Data can be partitioned along processors so that the storage constraints and the communication overhead are minimized. In this paper we present a parallel algorithm to solve the singular integral operator (1). The recursive relations of the original algorithm [9,12] (see section 2 below) are redefined in a way that message lengths depend only on the number of Fourier coefficients being evaluated, so that communication costs are independent of the number of annular regions in use. The implementation is based on having two simultaneous fluxes of data traversing processors in a linear path configuration. It allows overlapping of computational work simultaneously with data-exchanges, and having a minimal number of messages in the communication channels. The resulting algorithm is very scalable and independent of a particular distributed-memory configuration.

The remainder of the paper is organized as follows. In section 2, we review the sequential algorithm from [12]. In section 3, we describe the parallel implementation. Section 4 presents the analysis of the parallel algorithm. In section 5, we present our approach to analyze the scalability of the algorithm. In section 6, we present and discuss the numerical results, and finally we make our concluding remarks in section 7.

2. The algorithm

The fast algorithm to evaluate the singular integral transform (1) was developed in [10,11]. The method divides the interior of the unit disk $B(0;1)$ into a collection of annular regions. The integral and $h(\sigma)$ are expanded in terms of Fourier series with radius dependent Fourier coefficients. 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 radius dependent Fourier coefficients of

the singular integral (1). Then inverse Fourier transforms are applied on each circle to obtain the value of the singular integrals on all circles.

To review the mathematical foundation of the algorithm, we state the following theorem verbatim from [11]:

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

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

where

$$C_{n,m}(r) = \begin{cases} \dfrac{2(-1)^{m+1}}{r^{m-1}} \dbinom{-n-1}{m-1} \displaystyle\int_0^r \left(\frac{r}{\rho}\right)^{m+n-1} h_{m+n}(\rho)\,\mathrm{d}\rho, & n \leqslant -m, \\ 0, & -m < n < 0, \\ -\dfrac{2}{r^{m-1}} \dbinom{m+n-1}{m-1} \displaystyle\int_r^1 \left(\frac{r}{\rho}\right)^{m+n-1} h_{m+n}(\rho)\,\mathrm{d}\rho, & n \geqslant 0, \end{cases} \tag{3}$$

and $B_{n,m}(r)$ and $S_{0,m}(0)$ are defined as follows.

- Case 1. If $h(\sigma)$ is Hölder continuous in the unit disk with exponent γ, $0 < \gamma < 1$ and $m = 1$ or 2, then

$$S_{0,m}(0) = -2 \lim_{\varepsilon \to 0} \int_\varepsilon^1 \rho^{1-m} h_m(\rho)\,\mathrm{d}\rho, \tag{4}$$

$$B_{n,m}(r) = \begin{cases} 0, & m = 1, \\ h_{n+2}(r), & m = 2. \end{cases} \tag{5}$$

- Case 2. If $h(\sigma)$ is analytic in the unit disk and m is a finite positive integer, then

$$S_{0,m}(0) = -h_m(r = 1), \tag{6}$$

$$B_{n,1}(r) = 0, \tag{7}$$

and for $m \geqslant 2$

$$B_{n,m}(r) = \begin{cases} 0, & n < -1,\ n \neq -m, \\ (-1)^m\, r^{2-m}\, h_0(r), & n = -m, \\ \dbinom{m+n-1}{m-2} r^{2-m}\, h_{m+n}(r), & n \geqslant -1. \end{cases} \tag{8}$$

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 $0 = r_1 < r_2 <$

$\cdots < r_M = 1$ be the radii defined on the discretization. The following corollaries of theorem 2.1 are presented verbatim from [11]:

Corollary 2.1. It follows from (3) that $C_{n,m}(1) = 0$ for $n \geqslant 0$, and $C_{n,m}(0) = 0$ for $n \leqslant -m$. 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

$$C_{n,m}^{i,j} = \begin{cases} \dfrac{2(-1)^{m+1}}{r_j^{m-1}} \dbinom{-n-1}{m-1} \displaystyle\int_{r_i}^{r_j} \left(\frac{r_j}{\rho}\right)^{m+n-1} h_{m+n}(\rho)\,\mathrm{d}\rho, & n \leqslant -m, \\ \dfrac{2}{r_i^{m-1}} \dbinom{m+n-1}{m-1} \displaystyle\int_{r_i}^{r_j} \left(\frac{r_i}{\rho}\right)^{m+n-1} h_{m+n}(\rho)\,\mathrm{d}\rho, & n \geqslant 0, \end{cases} \tag{9}$$

then

$$C_{n,m}(r_j) = \left(\frac{r_j}{r_i}\right)^n C_{n,m}(r_i) + C_{n,m}^{i,j}, \quad n \leqslant -m, \tag{10}$$

$$C_{n,m}(r_i) = \left(\frac{r_i}{r_j}\right)^n C_{n,m}(r_j) - C_{n,m}^{i,j}, \quad n \geqslant 0. \tag{11}$$

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

$$C_{n,m}(r_l) = \begin{cases} \displaystyle\sum_{i=2}^{l} \left(\frac{r_l}{r_i}\right)^n C_{n,m}^{i-1,i} & \text{for } n \leqslant -m \text{ and } l = 2, \ldots, M, \\ -\displaystyle\sum_{i=l}^{M-1} \left(\frac{r_l}{r_i}\right)^n C_{n,m}^{i,i+1} & \text{for } n \geqslant 0 \text{ and } l = 1, \ldots, M-1. \end{cases} \tag{12}$$

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 $n \leqslant -m$ as *negative modes* and the ones with index values $n \geqslant 0$ as *positive modes*. Equation (10) shows that negative modes are built up from the smallest radius r_1 towards the largest radius r_M. Conversely, equation (11) constructs positive modes from r_M towards r_1. We summarize these concepts with a formal description of the algorithm in figure 1.

Although steps 3 and 4 are very appropriate to a sequential algorithm, they may represent a bottleneck in a parallel implementation. In the next section, we overcome this problem by redefining the formal description of algorithm 2.1.

Algorithm 2.1 (Sequential algorithm).

Given $m \geqslant 1$, M, N and the grid values $h(r_l\,\mathrm{e}^{2\pi \mathrm{i}k/N})$, $l \in [1, M]$, $k \in [1, N]$, the algorithm returns the values of $T_m h(r_l\,\mathrm{e}^{2\pi \mathrm{i}k/N})$, $l \in [1, M]$, $k \in [1, N]$.

1. Compute the Fourier coefficients $h_n(r_l)$, $n \in [-N/2+m, N/2]$, for M sets of data at $l \in [1, M]$.
2. Compute the radial one-dimensional integrals $C_{n,m}^{i,i+1}$, $i \in [1, M-1]$, $n \in [-N/2, -m] \cup [0, N/2]$ as defined in (9).
3. Compute coefficients $C_{n,m}(r_l)$ for each of the negative modes $n \in [-N/2, -m]$ as defined in (10):
   (a) set $C_{n,m}(r_1) = 0$,
   (b) for $l = 2, \ldots, M$
$$C_{n,m}(r_l) = \left(\frac{r_l}{r_{l-1}}\right)^n C_{n,m}(r_{l-1}) + C_{n,m}^{l-1,l}.$$
4. Compute coefficients $C_{n,m}(r_l)$ for each of the positive modes $n \in [0, N/2]$ as defined in (11):
   (a) set $C_{n,m}(r_M) = 0$,
   (b) for $l = M-1, \ldots, 1$
$$C_{n,m}(r_l) = \left(\frac{r_l}{r_{l+1}}\right)^n C_{n,m}(r_{l+1}) - C_{n,m}^{l,l+1}.$$
5. If $m > 1$, set $C_{n,m}(r_l) = 0$, $l \in [1, M]$, for $n \in [-m, -1]$.
6. Compute the Fourier coefficients $S_{n,m}(r_l)$, $l \in [1, M]$, $n \in [-N/2, N/2-m]$, as defined in theorem 2.1.
7. Compute
$$T_m h\big(r_l\,\mathrm{e}^{2\pi \mathrm{i}k/N}\big) = \sum_{n=-N/2}^{N/2-m} S_{n,m}(r_l)\,\mathrm{e}^{2\pi \mathrm{i}kn/N}, \quad k \in [1, N],$$
   for each radius r_l, $l \in [1, M]$.

Figure 1. Sequential description of the fast algorithm for the evaluation of the singular integral transform (1).

3. Parallel implementation

The performance of a parallel system is largely determined by the degree of concurrency of its processors. The identification of intrinsic parallelism in the method leads to our choice for data partitioning [14]. The fast algorithm employs two groups of Fourier transforms (steps 1 and 7) which can be evaluated independently for each

fixed radius r_l. Consequently their computations can be performed in parallel. Since each FFT usually engages lengthy computations, the computational granularity of each processor will be large and therefore very well suited for MIMD architectures. Negative effects resulting from communication delays in a MIMD computer can be minimized by an efficient implementation. Mechanisms to reduce communication delays on message-passing architectures include: evenly distributed load balancing between processors, overlapping of communication and computations, reduced message lengths, and reduced frequency in exchanging messages. Often the above mechanisms are conflicting and, in practice, a tradeoff will define an efficient implementation. We address this issue in this section.

The fast algorithm in section 2 requires multiple Fourier transforms to be performed. Specifically, it computes M FFTs of length N in steps 1 and 7 of algorithm 2.1. 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 [5,13]. Three approaches are summarized in [5]:

(1) parallel calls to FFTs,

(2) parallel FFT with inner loop, and

(3) truncated parallel FFT.

In the first case, 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. For the second case, only one parallel FFT is implemented. In this case, the data manipulated by the algorithm is a set of N vectors, each vector of length M, such that each component of a vector belongs to a distinct M sequence. It corresponds to substituting single complex operations in the parallel FFT algorithm by an inner loop over M. In the third case, the bit-reversal is applied individually on each input sequence and then a unique sequence of length MN is obtained by concatenating all M sequences. A parallel FFT is applied but only for $\log_2 N$ stages. Therefore, the Fourier coefficients for a given M sequence can be extracted from the original place where it was concatenated. Since, both the parallel FFT with inner loop and the truncated parallel FFT approach present identical computational loads and synchronization overheads [5], their performance is very similar. Perhaps the major disadvantage of the truncated FFT version is the cumbersome programming overhead when M is not a power of 2. Parallel calls to sequential FFTs perform bit-reversal setup and sine–cosine calculations M/P times on each processor: it represents an overhead that may produce larger running times when comparing this strategy against parallel FFT with inner loop or truncated parallel FFT. Conversely, parallel calls to sequential FFTs presents no synchronization overhead because no interprocessor communication occurs. As a final remark, methods to maximize bandwidth utilization and minimize communication overhead for parallel FFTs may experience network congestion when aiming to overlap communication by computations [6].

We adopt an improved implementation of parallel calls to sequential FFTs by assigning grid points within a group of circles to each processor. 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 $r_l\,\mathrm{e}^{2\pi \mathrm{i}k/N}$ for a fixed radius r_l, $l \in [1, M]$. Data locality is preserved by performing Fourier transforms within a processor. Specifically, 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 $r_l\,\mathrm{e}^{2\pi \mathrm{i}k/N}$, $k \in [1, N]$ and $l \in [jM/P + 1, (j+1)M/P]$. Thus, each FFT can be evaluated in place without communication. This approach is free of network congestion. Moreover, several different forms of the FFT algorithm exist [1]. But all of them proceed in a similar way by recursively reordering the array of sample points. By having a group of data associated with $l \in [jM/P + 1, (j+1)M/P]$ in the same processor p_j, 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.

A straightforward formulation for the parallel algorithm might attempt to use two sets of communication. The first set is related with step 2 in algorithm 2.1 and encompasses communication between neighbor processors to exchange the boundary Fourier coefficients required in equation (9). The second set arises from the inherently sequential recurrences in steps 3 and 4, where a given coefficient $C_{n,m}(r_l)$ depends on all terms $C_{n,m}^{i-1,i}$ with $i \in [2, l]$, if $n \leqslant -m$, or $C_{n,m}^{i,i+1}$ with $i \in [l, M-1]$ if $n \geqslant 0$. Assume that there is no interprocessor communication. The only coefficients $C_{n,m}^{i,k}$ that can be computed on processors p_j are $C_{n,m}^{i,k}$; $i, k \in [jM/P + m + 1, (j+1)M/P - m]$. Consequently, a message-passing mechanism must be used to exchange coefficients $C_{n,m}^{i,j}$ across processors. A closer look into the algorithm reveals that a better parallel implementation can be formulated.

The mechanism of redundant computations, that is, computations that are performed on more than one processor, can be used to improve performance of parallel programs on distributed memory machines. The first set of communications can be totally eliminated. Since the algorithm employs equations (10) and (11) that only utilize consecutive radii, only terms of the form $C_{n,m}^{l-1,l}$ and $C_{n,m}^{l,l+1}$, $l \in [jM/P + 1, (j+1)M/P]$, are required in the processor p_j. Notice that p_j already evaluates the Fourier coefficients $h_n(r_l)$, $l \in [jM/P + 1, (j+1)M/P]$. 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, $l \in [jM/P, (j+1)M/P + 1]$, 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.

The second set of communication arises from the fact that recurrences (10) and (11) should be evaluated on the same processor. If terms $C_{n,m}^{i-1,i}$ and $C_{n,m}^{i,i+1}$ are not transfered from one processor to another, the data dependency imposed by (10) and (11)

indicates that at most two processors (one for $n \leqslant -m$ and other for $n \geqslant 0$) would be performing computations and keeping all remaining processors idle. It basically implies that the data partitioning scheme must be reverted to allow processor p_j to evaluate the coefficients $C_{n,m}(r_l)$, $l \in [1, M]$, for $n \in [jN/P - N/2, (j+1)N/P - N/2]$. When understanding data as an $N \times M$ matrix distributed in a row-wise partitioning, the above data-reversion operation (swap) corresponds to a matrix transposition problem, which may flood communication channels either with messages of length O(NM), or messages of the broadcast type. In both cases, it can easily result on large communication overhead for the algorithm.

However, corollary 2.3 leads to a more efficient parallelization strategy as shown below. To achieve this, we first rewrite the sums in equation (12) as

$$C_{n,m}(r_l) = r_l^n \sum_{i=2}^{l} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i} \qquad \text{for } n \leqslant -m \text{ and } l = 2, \ldots, M, \tag{13}$$

$$C_{n,m}(r_l) = -r_l^n \sum_{i=l}^{M-1} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i,i+1} \quad \text{for } n \geqslant 0 \text{ and } l = 1, \ldots, M-1, \tag{14}$$

so that sums

$$\sum_{i=2}^{l} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i} \quad \text{and} \quad \sum_{i=l}^{M-1} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i,i+1}$$

will be distributed across processors. Before we carry out computations with formulae (13) and (14), we should note that these new formulae are unstable for large values of n.

The above computations can be stabilized by performing more regular calculations as in the original recurrences (10) and (11). In both approaches, computations evaluate terms of the form

$$\left(\frac{\alpha}{\beta}\right)^n, \tag{15}$$

where $n \in [-N/2, N/2 - m]$ depends on the number N of Fourier coefficients. In the case of (10) and (11), we have $\alpha/\beta = r_l/r_{l-1},\ l = 2, \ldots, M$, for $n \leqslant -m$, and $\alpha/\beta = r_l/r_{l+1},\ l = 1, \ldots, M-1$, for $n \geqslant 0$. Since $r_{l-1} < r_l < r_{l+1}$, the algorithm in essence only evaluates increasing positive powers of values on the interval $(0, 1)$. Moreover, for the case of M equidistant points in the radial direction we have $r_l = (l-1)/(M-1),\ l = 1, \ldots, M$, which implies that those values belong to the interval $[0.5, 1)$. Unfortunately, in the case of (13) and (14) we have $\alpha/\beta = 1/r_i,\ r_i \in (0, 1]$, which may imply on either a fast overflow for large absolute values of $n \leqslant -m$, or a fast underflow for large values of $n \geqslant 0$. We overcome this problem by making use

of the stabilized recurrences

$$\begin{cases} q_1^-(n) = 0, \\ q_l^-(n) = \left(\dfrac{r_{l+1}}{r_l}\right)^n \left(q_{l-1}^-(n) + C_{n,m}^{l-1,l}\right), \quad l = 2, \ldots, M, \, \forall n \leqslant -m, \end{cases} \tag{16}$$

where we have defined $r_{M+1} = 1$, and

$$\begin{cases} q_M^+(n) = 0, \\ q_l^+(n) = \left(\dfrac{r_{l-1}}{r_l}\right)^n \left(q_{l+1}^+(n) + C_{n,m}^{l,l+1}\right), \quad l = M-1, \ldots, 1, \, \forall n \geqslant 0. \end{cases} \tag{17}$$

A first observation is that the above recurrences are stable as in the case of the original recurrences (10) and (11) because terms $(\alpha/\beta)^n$ are also increasing positive powers of values on the interval $(0,1)$. Secondly, recurrences (16) and (17) can be used to evaluate formulae (13) and (14). In fact, for a fixed $l \in [2, M]$ and $n \leqslant -m$ we obtain

$$\begin{aligned} q_l^-(n) &= \left(\frac{r_{l+1}}{r_l}\right)^n \left[q_{l-1}^-(n) + C_{n,m}^{l-1,l}\right] \\ &= \left(\frac{r_{l+1}}{r_l}\right)^n \left[\left(\frac{r_l}{r_{l-1}}\right)^n \left(q_{l-2}^-(n) + C_{n,m}^{l-2,l-1}\right) + C_{n,m}^{l-1,l}\right] \\ &= r_{l+1}^n \left[\frac{q_{l-2}^-(n)}{r_{l-1}^n} + \frac{C_{n,m}^{l-2,l-1}}{r_{l-1}^n} + \frac{C_{n,m}^{l-1,l}}{r_l^n}\right] \\ &= r_{l+1}^n \left[\frac{q_1^-(n)}{r_2^n} + \sum_{i=2}^{l} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i}\right] = r_{l+1}^n \sum_{i=2}^{l} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i}, \end{aligned} \tag{18}$$

which implies that equation (13) can be rewritten as

$$C_{n,m}(r_l) = \left(\frac{r_l}{r_{l+1}}\right)^n q_l^-(n) \quad \text{for } n \leqslant -m \text{ and } l = 2, \ldots, M. \tag{19}$$

Similarly, for a fixed $l \in [1, M-1]$ and $n \geqslant 0$ recurrence (17) evaluates

$$q_l^+(n) = r_{l-1}^n \sum_{i=l}^{M-1} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i,i+1}, \tag{20}$$

leading equation (14) to

$$C_{n,m}(r_l) = -\left(\frac{r_l}{r_{l-1}}\right)^n q_l^+(n) \quad \text{for } n \geqslant 0 \text{ and } l = 1, \ldots, M-1. \tag{21}$$

For the purpose of achieving an even distribution of computational load across processors, it is helpful to split the computational work when performing recur-

rences (16) and (17). We define the following *partial sums* for each processor p_j, $j = 0, \dots, P-1$. For the case $n \leqslant -m$, let

$$\begin{cases} t_0^-(n) = r_{M/P+1}^n \displaystyle\sum_{i=2}^{M/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i}, \\ t_j^-(n) = r_{(j+1)M/P+1}^n \displaystyle\sum_{i=jM/P+1}^{(j+1)M/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i}, \quad j = 1, \dots, P-1, \end{cases} \tag{22}$$

and for $n \geqslant 0$, let

$$\begin{cases} t_{P-1}^+(n) = r_{(P-1)M/P}^n \displaystyle\sum_{i=(P-1)M/P+1}^{M-1} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i,i+1}, \\ t_j^+(n) = r_{jM/P}^n \displaystyle\sum_{i=jM/P+1}^{(j+1)M/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i,i+1}, \quad j = 0, \dots, P-2. \end{cases} \tag{23}$$

Since coefficients $C_{n,m}^{i-1,i}$ ($n \leqslant -m$) and $C_{n,m}^{i,i+1}$ ($n \geqslant 0$) are already stored in the processor p_j when $i \in [jM/P+1, (j+1)M/P]$, partial sums t_j^- and t_j^+ can be computed locally in the processor p_j. Moreover, these computations are carried out using the same stable recurrences defined for q^- and q^+ in equations (16) and (17).

If the *accumulated sums* $\hat{s}_j^-$ and $\hat{s}_j^+$, $j = 0, \dots, P-1$, are defined by

$$\begin{cases} \hat{s}_0^-(n) = t_0^-(n), & n \leqslant -m, \\ \hat{s}_j^-(n) = \left(\dfrac{r_{(j+1)M/P+1}}{r_{jM/P+1}}\right)^n \hat{s}_{j-1}^-(n) + t_j^-, & n \leqslant -m, \end{cases} \tag{24}$$

and

$$\begin{cases} \hat{s}_{P-1}^+(n) = t_{P-1}^+(n), & n \geqslant 0, \\ \hat{s}_j^+(n) = \left(\dfrac{r_{jM/P}}{r_{(j+1)M/P}}\right)^n \hat{s}_{j+1}^+(n) + t_j^+, & n \geqslant 0, \end{cases} \tag{25}$$

then we have a recursive method to accumulate partial sums t_j^- and t_j^+ computed in processors p_j, $j = 0, \dots, P-1$. The resulting formulas for $\hat{s}_j^-$ and $\hat{s}_j^+$ are given by

$$\hat{s}_j^-(n) = r_{(j+1)M/P+1}^n \sum_{i=2}^{(j+1)M/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i} \quad \text{for } n \leqslant -m, \tag{26}$$

and

$$\hat{s}_j^+(n) = r_{jM/P}^n \sum_{i=jM/P+1}^{M-1} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i,i+1} \quad \text{for } n \geqslant 0. \tag{27}$$

In fact, for the case of negative modes one can verify that

$$\begin{aligned}
\hat{s}_j^-(n) &= \left(\frac{r_{(j+1)M/P+1}}{r_{jM/P+1}}\right)^n \left[\left(\frac{r_{jM/P+1}}{r_{(j-1)M/P+1}}\right)^n \hat{s}_{j-2}^-(n) + t_{j-1}^-\right] + t_j^- \\
&= r_{(j+1)M/P+1}^n \left[\frac{\hat{s}_{j-2}^-(n)}{r_{(j-1)M/P+1}} + \sum_{i=(j-1)M/P+1}^{jM/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i} \right. \\
&\qquad \left. + \sum_{i=jM/P+1}^{(j+1)M/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i}\right] \\
&= r_{(j+1)M/P+1}^n \left[\frac{t_0^-(n)}{r_{M/P+1}} + \sum_{i=M/P+1}^{(j+1)M/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i}\right] \\
&= r_{(j+1)M/P+1}^n \sum_{i=2}^{(j+1)M/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i}. \qquad (28)
\end{aligned}$$

A similar proof holds for accumulated sums $\hat{s}_j^+$.

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 $l \in [jM/P+1, (j+1)M/P]$. Computations in p_j only make use of accumulated sums from neighbor processors. For $n \leqslant -m$ 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:

$$\begin{cases} C_{n,m}(r_{jM/P+1}) = \hat{s}_{j-1}^-(n) + C_{n,m}^{jM/P,jM/P+1}, \\ C_{n,m}(r_l) = \left(\dfrac{r_l}{r_{l-1}}\right)^n C_{n,m}(r_{l-1}) + C_{n,m}^{l-1,l}. \end{cases} \qquad (29)$$

For $n \geqslant 0$, 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:

$$\begin{cases} C_{n,m}(r_{(j+1)M/P}) = -\hat{s}_{j+1}^+(n) - C_{n,m}^{(j+1)M/P,(j+1)M/P+1}, \\ C_{n,m}(r_l) = \left(\dfrac{r_l}{r_{l+1}}\right)^n C_{n,m}(r_{l+1}) - C_{n,m}^{l,l+1}. \end{cases} \qquad (30)$$

The advantage of using equations (30) and (29) over original recurrences (10) and (11) 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 (10) and (11) is removed. It may be worth pointing out now that the data-dependency between

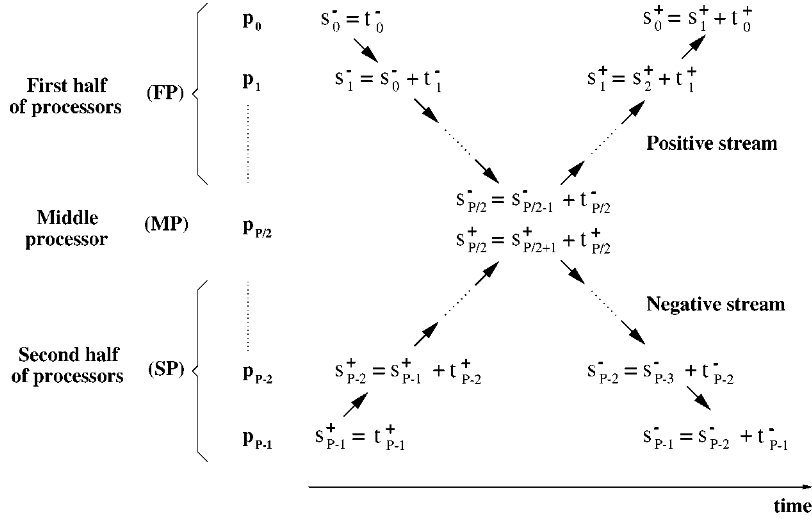


Figure 2. Message distribution in the algorithm. Two streams of neighbor-to-neighbor messages cross communication channels simultaneously.

processors appears only in equations (24) and (25). The only sequential component in this process is the message-passing mechanism to accumulate the partial sums, which will be explained in the next sections. The notation in equations (24) and (25) will be simplified to allow a clear exposition:

- relation $s_j^- = s_{j-1}^- + t_j^-$ represents the updating process in recurrence (24), and
- relation $s_j^+ = s_{j+1}^+ + t_j^+$ represents updating (25).

Figure 2 presents the general structure for the algorithm. Processors are divided into three groups: processor $p_{P/2}$ is defined as the *middle processor*, processors $p_0, \ldots, p_{P/2-1}$ are in the *first half*, and $p_{P/2+1}, \ldots, p_{P-1}$ are the *second half* processors. Due to the choice for data distribution, processors in the first half are more likely to obtain the accumulated sum s_j^- before the accumulated sum s_j^+. In fact, any processor in the first half has less terms in the accumulated sum s_j^- when compared against s_j^+. Additionally, the dependency is sequential. The accumulated sum s_j^- on a first half processor p_j depends on s_{j-1}^-, which in turn depends on s_{j-2}^-. It suggests the creation of 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. Processor p_1 updates the message to $s_1^- = s_0^- + t_1^-$ and sends it to processor p_2. Generically, processor p_j receives the message s_{j-1}^- from p_{j-1}, updates it as $s_j^- = s_{j-1}^- + t_j^-$, and sends the new message to processor p_{j+1}. It corresponds to the downward arrows in figure 2. In the same way, processors on the second half start computations for partial sums s_k^+. 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 resulting algorithm is composed by two simultaneous streams of neighbor-to-neighbor communication, each

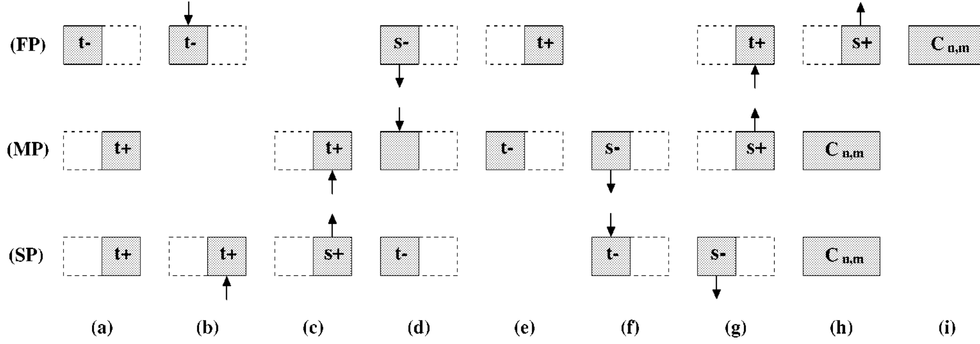


Figure 3. Coordination scheme to minimize delays due to interprocessor communication. The middle processor (MP) plays a key role to forward the positive stream to the first half of processors (FP) and to forward the negative stream to the second half of processors (SP).

one with messages of length N. In short, one pipe started on processor p_0 (negative stream), and a reverse pipe which starts on p_{P-1} (positive stream). The scheme is free of data-reversion and communication costs are lower than the same for a matrix transposition process.

Load balance is a fundamental issue in parallel programming. Additionally, communication overhead is typically several orders of magnitude larger than hardware overhead [20]. Coordination between processors must (1) attempt to have the local computational work performed simultaneously under the same time frame, and (2) avoid a message passing mechanism that delays local work. Thus, messages must arrive and leave the middle processor as early as possible so that idle times are minimized. As soon as one processor receives a message, it updates the information and forwards it to the next processor in the pipe. Figure 3 summarizes the strategy. The algorithm is divided into nine time frames (from (a) to (i)). The top row (FP) represents one processor belonging to the first half, the second row (MP) represents the middle processor, and the bottom row (SP) corresponds to one processor in the second half. Rectangles indicate the computational work performed by one processor: the left side represents computations for negative modes ($n \leqslant -m$), and the right side indicates computational work for positive modes ($n \geqslant 0$). Interprocessor communication is represented by an arrow. Upward arrows belong to the positive stream, and downward arrows form the negative stream. On the first time frame (a), all processors perform the same amount of work by evaluating FFT transforms and either the partial sum (`t+`) or the partial sum (`t-`). On frames (b), (c) and (d), negative and positive streams arrive at the middle processor (it corresponds to the intersection point at the center of figure 2). A processor p_j on the first half receives a message from p_{j-1}, and a processor p_k on the second half receives a message from p_{k+1} as indicated on (b). In frame (c), processor $p_{P/2+1}$ obtains the accumulated sum `s+` and sends it to the middle processor $p_{P/2}$. Similarly, processor $p_{P/2-1}$ updates the accumulated sum `s-` which is sent to $p_{P/2-1}$ in frame (d). The empty slots on (b) and (c) represent the delay due to interprocessor communication. On (b), the middle node is idle waiting

for the negative and positive streams to arrive. On this example, time frames for the processor on the top of the figure are shifted by one time slot in (c) because the middle node gives precedence to the incoming message from the positive stream. On frames (d) and (e), all processors evaluate their remaining partial sums. The middle processor updates the accumulated sums and sends `s-` to the second half of processors (f), and `s+` to the first half (g). The empty slots in frames (e) and (f) indicate the delay for the outgoing messages to arrive at processors p_0 and p_{P-1}. The last step is to have all processors obtaining terms $C_{n,m}$ and performing inverse FFT transforms in (h) and (i).

Figure 3 also suggests an improvement for the algorithm. Note that the last group of computations on each processor is composed by the calculation of the terms $C_{n,m}$, $n \in [-N/2, N/2]$, in (1) and the inverse Fourier transforms. For the first half processors, Fourier coefficients associated with negative modes ($n \leqslant -m$) only depend on the accumulated sums s^- which are evaluated in time frame (d). It indicates that these coefficients can be obtained earlier within the empty time frame (f). 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 ($n \geqslant 0$) can be evaluated in time frame (e). We distinguish these variants of the algorithm by defining

- the *late computations algorithm* as the original version presented in figure 3 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.

In the next section, we analyze the late computations algorithm in detail and compare it with other approaches.

4. Analysis of the parallel algorithm

4.1. Complexity of the stream-based algorithm

When designing the above coordination scheme, one can formulate a timing model for the stream-based algorithm. The parallel implementation presents a high degree of concurrence because major computations are distributed among distinct processors. However, interprocessor communication is always a source of parallel overhead. Different problem sizes correspond to distinct levels of granularity which implies that there is an optimal number of processors associated with each granularity. A complexity model plays a key role in the investigation of these characteristics. For the timing analysis, we consider t_{s} as the message startup time and t_w as the transfer

time for a complex number. To normalize the model, we adopt constants c_1 and c_2 to represent operation counts for distinct stages of the algorithm. The model follows the dependencies previously discussed in figure 3. Each processor performs a set of M/P Fourier transforms in $(c_1/2)(M/P)N\log_2 N$ operations, and computes the radial integrals $C^{i,i+1}_{n,m}$ using $(c_2/3)(M/P)N$ operations. To evaluate either M/P partial sums t^+ or M/P partial sums t^-, each processor takes $(c_2/3)(M/P)(N/2)$ operations. Positive and negative streams start from processors p_{P-1} and p_0, respectively, and each processor forwards (receive and send) a message of length $N/2$ towards the middle node. The total time is $2((P-1)/2)(t_s + (N/2)t_w)$. On the next stage, each processor performs either a partial sum t^+ or partial sum t^- at the cost of $(c_2/3)(M/P)(N/2)$ operations. Positive and negative streams restart from the middle node and arrive in p_0 and p_{P-1}, respectively, after $2((P-1)/2)(t_s + (N/2)t_w)$ time units for communication. Additionally, the coefficients $C_{n,m}$ are computed in $(c_2/3)(M/P)N$ operations. Finally, $(c_1/2)(M/P)N\log_2 N$ operations are used to apply inverse Fourier transforms. The parallel timing for our stream-based algorithm is given by

$$T_P^{\text{stream}} = c_1\frac{M}{P}N\log_2 N + c_2\frac{M}{P}N + 2(P-1)\left(t_{\text{s}} + \frac{N}{2}t_w\right). \tag{31}$$

To analyze the performance of the parallel algorithm, one must compare the above equation against the timing estimate for the sequential algorithm. In the later case, the algorithm starts performing M Fourier transforms in $(c_1/2)MN\log_2 N$ operations. Radial integrals are obtained after $(c_2/3)MN$ operations, and the timing for evaluating the Fourier coefficients is also $(c_2/3)MN$. Finally, M inverse Fourier transforms take $(c_1/2)MN\log_2 N$ computations. Therefore, the sequential timing T_{s} is given by

$$T_{\text{s}} = c_1 MN\log_2 N + \frac{2}{3}c_2 MN. \tag{32}$$

Clearly, most of the parallel overhead must be attributed to the communication term in equation (31). Although each processor performs an extra set of $(c_2/3)(M/P)N$ computations when obtaining the partial sums t^- and t^+, the overhead of the extra cost is still amortized as the number of processors P increases. An immediate consequence is that overheads are mainly due to increasing number of angular grid points N. No communication overhead is associated with the number of radial grid points M. This scenario is made clear when obtaining the speedup S^{stream} for the algorithm

$$S^{\text{stream}} = \frac{T_{\text{s}}}{T_P^{\text{stream}}} = \frac{c_1 MN\log_2 N + (2/3)c_2 MN}{c_1(M/P)N\log_2 N + c_2(M/P)N + 2(P-1)(t_{\text{s}} + (N/2)t_w)} \tag{33}$$

$$= P\frac{c_1 MN\log_2 N + (2/3)c_2 MN}{c_1 MN\log_2 N + c_2 MN + 2P(P-1)(t_{\text{s}} + (N/2)t_w)}, \tag{34}$$

and the resulting efficiency

$$
\begin{aligned}
&E^{\text{stream}} \\
&\quad = \frac{S^{\text{stream}}}{P} = \frac{c_1 MN \log_2 N + (2/3)c_2 MN}{c_1 MN \log_2 N + c_2 MN + 2P(P-1)(t_{\text{s}} + (N/2)t_w)} \qquad (35)\\
&\quad = \frac{1}{1 + ((c_2/3)MN + 2P(P-1)(t_{\text{s}} + (N/2)t_w))/(c_1 MN \log_2 N + (2/3)c_2 MN)}. \qquad (36)
\end{aligned}
$$

Efficiency measures the fraction of the total running time that a processor is devoting to perform computations of the algorithm, instead of being involved on interprocessor coordination stages. From the above equation, one can detect the sources of overhead which makes $E^{\text{stream}} < 1$. It shows that the efficiency decays quadratically in the number of processors P.

For the asymptotic analysis of the algorithm, we drop the computational terms of lower order in (31) since they represent a small amount of overhead when compared against the communication term in (35). The resulting asymptotic timing T_P^{asymp} for the parallel algorithm is given by

$$
T_P^{\text{asymp}} = c_1 \frac{M}{P} N \log_2 N + 2(P-1)\left(t_{\text{s}} + \frac{N}{2} t_w\right). \qquad (37)
$$

Since message lengths depend on N and computational work depends also on M, distinct problem sizes will present different performances. The number of processors for which the asymptotic parallel running time T_P^{asymp} achieves its minimum is determined by $\partial T_P / \partial P = 0$. In the case of (37), we have

$$
P_{\text{opt}}^{\text{asymp}} = \sqrt{\frac{c_1 MN \log_2 N}{2(t_{\text{s}} + (N/2)t_w)}}, \qquad (38)
$$

which can be understood as an approximation for the value of P which minimizes the numerator in (36) for given values of M and N.

4.2. Comparison with other approaches

Estimate (31) can also be used to compare the performance of the parallel algorithm against an implementation based on matrix transposition. As stated earlier, this approach aims to evaluate recurrences (10) and (11) within a processor. Consequently, data must be reverted in all processors as exemplified on figure 4 for the case where $P = 4$. Initially, each processor contains data for evaluating M/P Fourier transforms. It corresponds to each row on figure 4(a). To calculate recurrences sequentially, each processor must exchange distinct data of size NM/P^2 with all $P-1$ remaining processors. At the end of the communication cycle, processor p_j contains all the terms $C_{n,m}^{i-1,i}$, $n \in [jN/P - N/2, (j+1)N/P - N/2]$. Figure 4(b) describes the communication pattern. Rows are divided into P blocks of size NM/P^2 so that the processor p_j

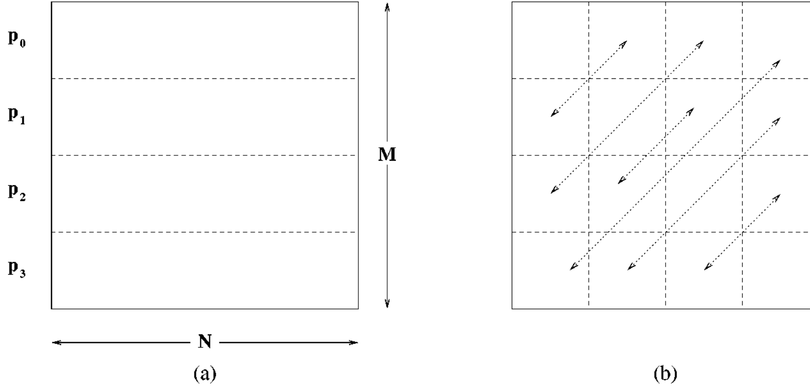


Figure 4. Coordination pattern based on all-to-all personalized communication: (a) M/P Fourier transforms are evaluated locally; (b) each two processors exchange blocks of size MN/P^2.

exchanges distinct data-blocks with different processors. The data-transfer pattern involves an all-to-all personalized communication as in a parallel matrix transposition procedure. For a mesh architecture, the estimated communication timing [15] is given by

$$T_{\text{comm}}^{\text{transpose}} = 2\big(\sqrt{P} - 1\big)\left(2t_{\text{s}} + \frac{MN}{P}\,t_w\right), \tag{39}$$

and the total parallel timing $T_P^{\text{transpose}}$ is obtained by adding the timing for M/P Fourier transforms, the timing to apply the recurrences, the same $T_{\text{comm}}^{\text{transpose}}$ to revert back data into the original ordering, and the timing for M/P inverse Fourier transforms. The basic difference in the computational timing when comparing with the case of positive and negative streams approach is that there is no need for the extra set of partial sums with cost $(c_2/3)(M/P)N$. The final estimate for the matrix transposition based algorithm is then given by

$$T_P^{\text{transpose}} = c_1\,\frac{M}{P}\,N\log_2 N + \frac{2}{3}\,c_2\frac{M}{P}\,N + 4\big(\sqrt{P} - 1\big)\left(2t_{\text{s}} + \frac{MN}{P}\,t_w\right), \tag{40}$$

which shows the different degree of scalability between both algorithms. In fact, for the case of the stream-based algorithm, interprocessor communication introduces a delay of order PN depending on the problem size as it can be derived from the coefficients in t_w in estimate (31). Under the same principle, an algorithm based on matrix transposition generates a delay of order $4MN/\sqrt{P}$. In a large scale application, clearly $M \gg P$ due to practical limitations on the number of available processors which makes $PN \ll 4MN/\sqrt{P}$. It implies that the stream-based algorithm must scale up better than the second approach because of a smaller communication overhead.

Theoretical estimates can also be used to compare the proposed algorithm against an implementation based on parallel FFT coding as discussed in section 3. For this purpose, we consider a parallel binary-exchange algorithm for FFT as described in [15].

In binary-exchange FFT, data is exchanged between all pairs of processors with labeling indices differing in one bit position. Although interprocessor communication takes place only during the first $\log_2 P$ iterations of the parallel FFT algorithm, the communication pattern is prone to produce large overheads. For a mesh architecture with $\sqrt{P}$ rows and $\sqrt{P}$ columns, the distance between processors which need to communicate grows from one to $\sqrt{P}/2$ links. In practice, it means that links between processors will be shared by multiple messages. It results from the fact that fast Fourier algorithms impose a large interdependency between the elements of the input data. Since a mesh architecture does not present the same degree of interprocessor connectivity as in a hypercube, for example, contention for the communication channels may occur. Considering the parallel FFT with inner loop described in section 3, the amount of communication due to the binary-exchange algorithm is given by [15]

$$T_{\mathrm{comm}}^{\mathrm{binary}} = (\log_2 P)t_{\mathrm{s}} + 4\frac{NM}{\sqrt{P}}\,t_w, \tag{41}$$

which is equivalent to a communication delay with the same order $\mathrm{O}(NM/\sqrt{P}\,)$ as in the case of the communication timing (39) for the matrix transposition approach. Consequently, the previous analysis for the matrix transposition approach also applies here, and the stream-based algorithm presents better parallel scalability than the parallel binary-exchange approach.

5. Analysis for a coarse-grained data distribution

The degree of parallelism indicates the extent to which a parallel program matches the parallel architecture. Speedup captures the performance gain when utilizing a parallel system [21]:

- *True speedup* is defined as the ratio of the time required to solve a problem on a single processor, using the best-known sequential algorithm, to the time taken to solve the same problem using P identical processors.
- For the *relative speedup* the sequential time to be used is determined by scaling down the parallel code to one processor.

Efficiency indicates the degree of speedup achieved by the system. The lowest efficiency $E = 1/P$ is equivalent to leave $P - 1$ processors idle and have the algorithm executed sequentially on a single processor. The maximum efficiency $E = 1$ is obtained when all processors devote the entire execution time to perform computations of the algorithm, with no delays due to interprocessor coordination or communication. 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 and efficiencies in a coarse-grained data distribution, we define

- *Modified speedups* $S^{[20]}$ and *modified efficiencies* $E^{[20]}$ which are calculated by comparing performance gains over the parallel algorithm running on a starting configuration with 20 processors. Specifically we have

$$S^{[20]} = \frac{20 \cdot T_{20}}{T_p} \quad \text{and} \quad E^{[20]} = \frac{S^{[20]}}{P}, \tag{42}$$

  where T_P is the parallel running time obtained using P processors.

Comparing with the actual definition for relative speedup, the modified speedup $S^{[20]}$ adopts $20T_{20}$ as the running time for the sequential version of the algorithm. It basically corresponds to assuming optimal speedups and efficiencies when using 20 processors, that is, $S = 20$ and $E = 1$ for $P = 20$. Although the actual efficiency for 20 processors is smaller than 1, the analysis allows us to observe the performance of the algorithm for a large number of processors without having strong constraints on problem sizes: values for M and N which could be used on a single processor represent an extreme low level of granularity for an increasing number of processors. Speedups and efficiencies can be analyzed for up to 60 processors by using $P = 20$ as a reference configuration. We present and discuss our numerical results in the next section.

6. Numerical results

For performance evaluation purposes, equation (1) was solved for $m = 1$ on a discretization of the unit disk $\overline{B(0;1)}$ with $N \times M$ lattice points composed by N equidistant points in the angular direction, and M equidistant points in the radial direction. Problem configurations where $N = 512, 1024, 2048$, and $M = 600, 1200, 2400$. Parallel experiments were carried out on an Intel Paragon computer using up to 60 processors.

To observe the scalability of the algorithm, two experiments were performed. For a fixed number $N = 512$ of angular grid points, three distinct numbers of radial grid points were employed: $M = 600, 1200$ and 2400. Tables 1–3 present actual running times when increasing the number of processors from $P = 20$ to $P = 60$. 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 tables 1, 4 and 5. The first observation derived from tables 1–5 is that saving in running times are more prominent as the dimension of the problem increases. For the case where $N = 512$ is fixed and $M = 600, 1200$ and 2400, total times decrease faster for $M = 2400$ (table 3) than for $M = 1200$ (table 2), which in turn decreases faster than for $M = 600$ (table 1). 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. Similar behavior is

Table 1
Actual running times, modified speedups $S^{[20]}$ and efficiencies $E^{[20]}$ based on the timing for 20 processors when applied for a problem of size $N = 512$ and $M = 600$.

Number of processors	Time (s)	Modified speedup $S^{[20]}$	Modified efficiency $E^{[20]}$
20	0.5402	20.0000	1.0000
30	0.4035	26.7764	0.8925
40	0.3750	28.8131	0.7203
50	0.3830	28.2117	0.5642
60	0.3942	27.4094	0.4568

Table 2
Actual running times, modified speedups $S^{[20]}$ and efficiencies $E^{[20]}$ based on the timing for 20 processors when applied for a problem of size $M = 1200$ and $N = 512$.

Number of processors	Time (s)	Modified speedup $S^{[20]}$	Modified efficiency $E^{[20]}$
20	1.0802	20.0000	1.0000
30	0.7535	28.6733	0.9558
40	0.6053	35.6946	0.8924
50	0.5459	39.5730	0.7915
60	0.5167	41.8090	0.6968

Table 3
Actual running times, modified speedups $S^{[20]}$ and efficiencies $E^{[20]}$ based on the timing for 20 processors when applied for a problem of size $M = 2400$ and $N = 512$.

Number of processors	Time (s)	Modified speedup $S^{[20]}$	Modified efficiency $E^{[20]}$
20	2.1257	20.0000	1.0000
30	1.4709	28.9039	0.9635
40	1.1514	36.9250	0.9231
50	0.9808	43.3475	0.8670
60	0.8671	49.0284	0.8171

observed for $M = 600$ fixed and $N = 512, 1024$ and 2048. The larger granularity for the case $N = 2048$ and $M = 600$ in table 5 provides a better scalability for increasing number of processors when compared against problem sizes $N = 1024$ and $M = 600$ in table 4, and $N = 512$ and $M = 600$ in table 1.

Tables 1–5 also describe the scalable performance of the algorithm. Since memory requirements for the testing problems exceed the capacity of a single processor, we performed our analysis based on running times for 20 processors as described in the previous section. Tables 1–5 contain modified speedups $S^{[20]}$ and efficiencies $E^{[20]}$

Table 4
Actual running times, modified speedups $S^{[20]}$ and efficiencies $E^{[20]}$ based on the timing for 20 processors when applied for a problem of size $M = 600$ and $N = 1024$.

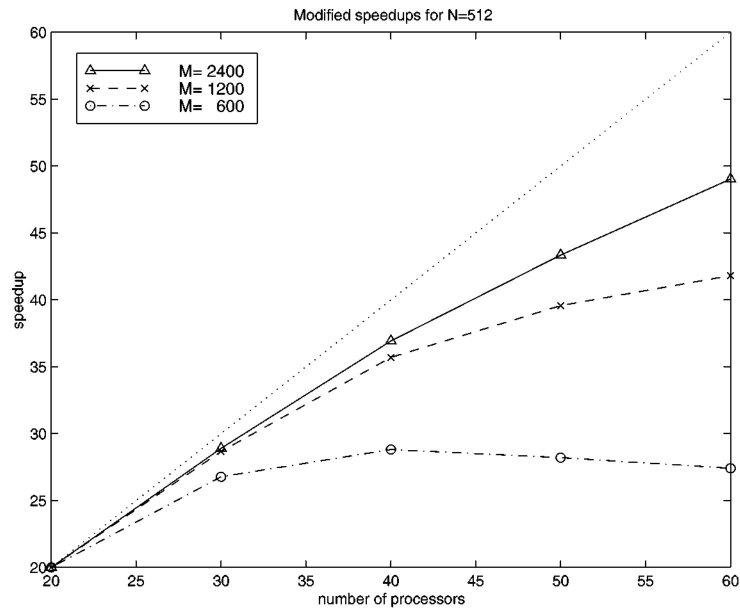
Number of processors	Time (s)	Modified speedup $S^{[20]}$	Modified efficiency $E^{[20]}$
20	1.1615	20.0000	1.0000
30	0.8365	27.7706	0.9257
40	0.6885	33.7395	0.8435
50	0.6277	37.0103	0.7402
60	0.6069	38.2774	0.6380

Table 5
Actual running times, modified speedups $S^{[20]}$ and efficiencies $E^{[20]}$ based on the timing for 20 processors when applied for a problem of size $M = 600$ and $N = 2048$.

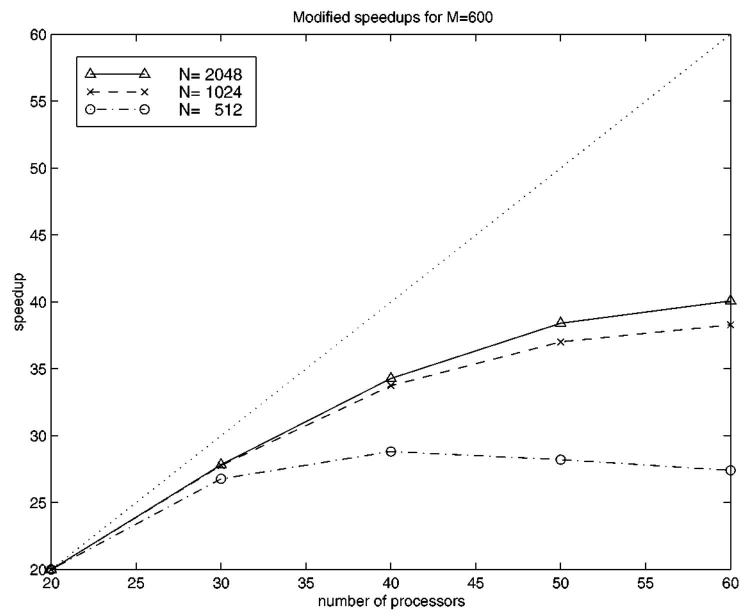
Number of processors	Time (s)	Modified speedup $S^{[20]}$	Modified efficiency $E^{[20]}$
20	2.5798	20.0000	1.0000
30	1.8541	27.8278	0.9276
40	1.5054	34.2744	0.8569
50	1.3435	38.4044	0.7681
60	1.2877	40.0679	0.6678

for all problem configurations. Recall from estimate (31) 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. When comparing efficiencies for problems of the same order of magnitude but with different values of N, larger efficiencies occur in the case of smaller values for N. Indeed, efficiencies are higher for $M = 1200$ and $N = 512$ (table 2) than their counterparts for $M = 600$ and $N = 1024$ (table 4). This contrast is even more visible when comparing the efficiencies for $M = 2400$ and $N = 512$ in table 3 against the efficiencies for $M = 600$ and $N = 2048$ in table 5. Figure 5 present plots for all speedups $S^{[20]}$. In the first case 5(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 that more processors would deliver even larger speedups. In the case of figure 5(b), problems of size $N = 1024$ and 2048 present increasing message lengths and are almost at the highest value for speedup, that is, adding a few more processors to the system will not provide any substantial savings in running times.

The above variations on speedups are closely related with the optimal number of processors obtained in estimate (38). Specifically, the condition $\partial T_P/\partial P = 0$ implies that $\partial S/\partial P = 0$ so that speedups achieve largest values for the optimal

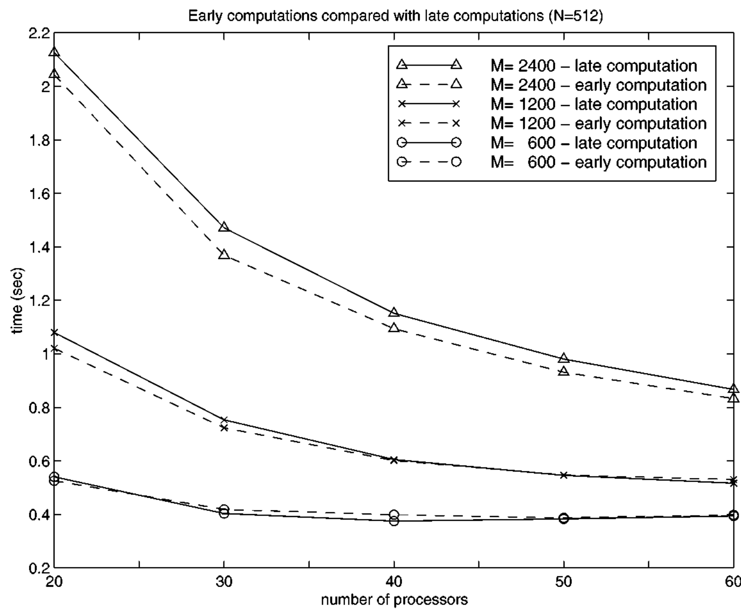


(a)

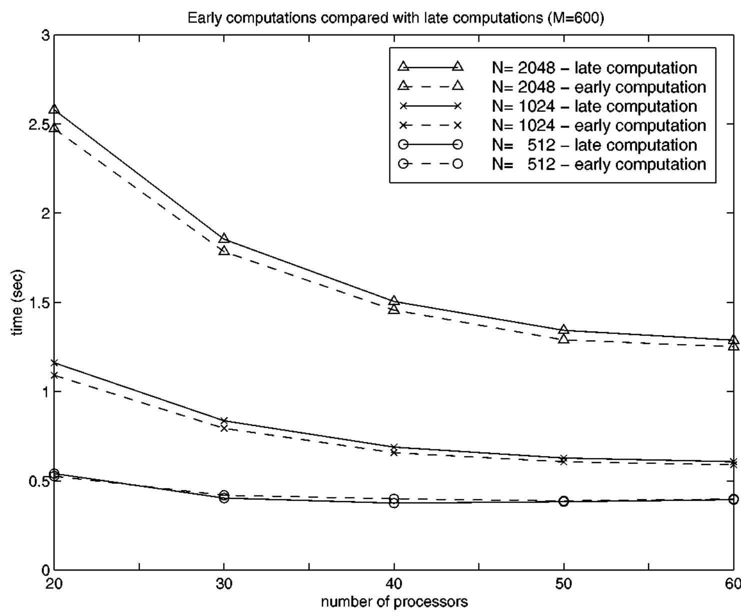


(b)

Figure 5. Modified speedups $S^{[20]}$ for 20, 30, 40, 50 and 60 processors: (a) for variable number of radial grid points $M = 600, 1200$ and 2400, with $N = 512$ fixed; (b) for variable number of angular grid points $N = 512, 1024$ and 2048, with $M = 600$ fixed.



(a)



(b)

Figure 6. 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 $N = 512$ with distinct number of radial points $M = 600, 1200$ and 2400; (b) timings for a fixed number of radial points $M = 600$ and with distinct number of angular points $N = 512, 1024$ and 2048.

Table 6
Estimated optimal number of processors P_{opt} for distinct problem sizes.

M	N	P_{opt}
600	512	59
1200	512	83
2400	512	118
600	1024	62
600	2048	66

number of processors. Table 6 presents the optimal number of processors obtained via estimate (38). Computational cost c_1 was obtained based on the running time for the smallest problem configuration $N = 512$ and $M = 600$ using 20 processors which is found in table 1. Corroborating the numerical results, the smallest problem ($M = 600$ and $N = 512$) cannot scale up far from 60 processors. For the case of $N = 512$ fixed, the largest problem would scale up to more than 100 processors. Conversely, for the case of $M = 600$ fixed, even the largest problem has an optimal number of processors close to $P = 60$. It means that no substantial savings in running time can be expected by adding more processors to the system.

Recall that section 3 presents two variants of the parallel algorithm. In the late computations algorithm, each processor evaluates terms $C_{n,m}$ after completion of all accumulated sums. In the early computations algorithm, half of the terms $C_{n,m}$ are evaluated right after the first accumulated sum is completed. To compare both versions of the algorithm, we use the same problem sizes for the early computations algorithm. Figure 6 contains running times for both versions. Running times for the late computations algorithm correspond to the timings present in tables 1–5. Overall 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 6(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 provides the best results. It shows that the choice between early or late computations depends on the problem size and the number of processors available.

7. Conclusions

Recently, progress has been made in the accurate and efficient evaluation of the singular integral operator (1) based on some recursive relations in Fourier space [10,11].

In this paper, we reviewed the fast numerical algorithm and developed its parallelization. By reformulating the inherently sequential recurrences present in the original algorithm, we were able to obtain a reduced amount of communication, and even message lengths depending only on the number of Fourier coefficients being evaluated. Moreover, we have shown that the new approach can be defined in a way that is numerically stable as in the original formulation of the fast algorithm. Additionally, two interprocessor coordination strategies were presented based on early and late evaluation of half of the Fourier coefficients of the singular integral operator. A timing model for the algorithm was established to compare the scalability of the parallel algorithm against a matrix inversion-based implementation. Numerical results were presented to corroborate theoretical estimates.

The implementation is very scalable in a parallel distributed environment and is virtually independent of the computer architecture. It only utilizes a linear neighbor-to-neighbor communication path which makes the algorithm very suitable for any architecture where a topology of the type ring or array of processors can be embedded.

Acknowledgement

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

References

[1] F. Argüello, M. Amor and E. Zapata, FFTs on mesh connected computers, Parallel Comput. 22 (1996) 19–38.

[2] L. Bers, *Mathematical Aspects of Subcritical and Transonic Gas Dynamics* (Wiley, New York, 1958).

[3] L. Bers and L. Nirenberg, On a representation theorem for linear elliptic systems with discontinuous coefficients and its applications, in: *Convegno Internazionale Suelle Equaziono Cremeonese*, Roma (1955) pp. 111–140.

[4] L. Bers and L. Nirenberg, On linear and nonlinear elliptic boundary value problems in the plane, in: *Convegno Internazionale Suelle Equaziono Cremeonese*, Roma (1955) pp. 141–167.

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

[6] C. Calvin, Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication, Parallel Comput. 22 (1996) 1255–1279.

[7] R. Courant and D. Hilbert, *Methods of Mathematical Physics*, Vol. II (Wiley, New York, 1961).

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

[9] P. Daripa, A fast algorithm to solve nonhomogeneous Cauchy–Riemann equations in the complex plane, SIAM J. Sci. Statist. Comput. 6 (1992) 1418–1432.

[10] P. Daripa, A fast algorithm to solve the Beltrami equation with applications to quasiconformal mappings, J. Comput. Phys. 106 (1993) 355–365.

[11] P. Daripa and D. Mashat, An efficient and novel numerical method for quasiconformal mappings of doubly connected domains, Numer. Algorithms 18 (1998) 159–175.

[12] P. Daripa and D. Mashat, Singular integral transforms and fast numerical algorithms, Numer. Algorithms 18 (1998) 133–157.
[13] R. Hockney and C. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms* (Adam Hilger, Bristol, 1981).
[14] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability* (McGraw-Hill, New York, 1993).
[15] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing* (Benjamin/Cummings, Redwood City, CA, 1994).
[16] J. Lawrynowicz, Quasiconformal mappings in the plane, in: *Lecture Notes in Mathematics*, Vol. 978 (Springer, New York, 1983).
[17] C. Morrey, On the solutions of quasi-linear elliptic differential equations, Trans. Amer. Math. Soc. 43 (1938) 126–166.
[18] A. Mshimba and W. Tutschke, *Functional Analytic Methods in Complex Analysis and Applications to Partial Differential Equations* (World Scientific, Singapore, 1990).
[19] L. Nirenberg, On nonlinear elliptic differential equations and Holder continuity, Comm. Pure Appl. Math. 6 (1953) 103–156.
[20] P. Pacheco, *Parallel Programming with MPI* (Morgan Kaufmann, San Francisco, CA, 1997).
[21] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface* (Morgan Kaufmann, San Francisco, CA, 1994).