Computational Mathematics, Modelling and Algorithms
Edited by J.C. Misra
Copyright © 2003, Narosa Publishing House, New Delhi, India

15. Fast Algorithms and Scientific Computation

Prabir Daripa

Department of Mathematics, Texas A&M University, College Station, TX-77843

Abstract Fast and accurate algorithms play an important role in scientific computation because they enable us to accomplish computations faster than otherwise it is possible. This also makes computation of high quality solutions of complex problems more feasible. However, significant progress in this direction is necessary in order to narrow the gap between today's scientific needs and system (hardware as well as software) performance. In recent years, one of the ways speed-up in computation has been achieved is by parallel computing, i.e., by simultaneous use of several processors in parallel. Efficient parallel computing requires efficient parallel algorithms. This chapter, among other things, discusses two parallel algorithms developed and implemented within last few years by the author and his colleague Leo Borges. This paper discusses only the theoretical aspects of these algorithms. More details and results of numerical implementation can be found in Borges and Daripa [6, 8].

Keywords Singular integral transform, fast algorithm, parallel processing, distributed memory, pipelining algorithm.

1. Introduction

Many problems of industrial or/and scientific interest may require to handle an excessive amount of data. Such problems usually presents large memory requirements and intensive floating point computations. The design of fast algorithms in itself 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. This basic philosophy behind parallel computing is addressed here through its application to singular integrals and partial differential equations.

Singular integrals play an important role in a wide variety of applied sciences including mathematical physics, computational sciences, and continuum mechanics and classical methods for partial differential equations (PDEs), just to mention a few. Their accurate and efficient evaluations help solve complex problems in many of these areas, and in many cases allow computations of high resolution solutions feasible.

In [15, 16, 18], Daripa and his colleagues laid the mathematical foundation of an algorithm for fast and accurate evaluation of certain singular integral transforms

in the complex plane. Such integrals arise in the integral methods of solving elliptic partial differential equations in the complex plane. By construction, the algorithm offers good parallelization opportunities and a lower computational complexity when compared with methods based on quadrature rules. Construction and implementation of such a parallel algorithm for singular integral transforms in complex plane has been described in detail by Borges and Daripa [6]. The basic principle behind these algorithms can be applied to solve partial differential equations in the real plane. This has been done for Poisson's equation in Borges and Daripa [8]. Both of these parallel algorithms only utilize a linear neighbour-to-neighbour communication path which makes the algorithms very suitable for any distributed memory architecture. The algorithms are highly parallelizable and our implementation of these algorithms is virtually architecture-independent. Numerical results and theoretical estimates (see [6, 8]) show good parallel scalability of the algorithms.

In this chapter, we present only theoretical aspects of both of these parallel algorithms and their complexity issues. For implementation and numerical results, we refer the reader to Borges and Daripa [6, 8]. Each algorithm is presented in a separate section and is self-contained. This has the advantage that one can read each of these sections without reading the other. For this convenience, I have allowed some amount of overlap between the sections on these algorithms. Section 2 below discusses the parallel algorithm for singular integral transforms in the complex plane. Section 3 provides the parallel algorithm for solving the Poisson equation on a disk. We conclude in section 4 where we suggest a list of problems related to the extension of these ideas and algorithms to arbitrary domains in two- and three-dimensions. Various combinations of some of these problems can be good candidates for new M.S./Ph.D. thesis problems.

2. A Parallel Algorithm for Singular Integral Transforms

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 [16]:

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

where h is a complex valued function of σ defined on $B(0; 1) = \{z: |z| < 1\}$, for a suitable finite positive integer m [18]. Daripa [16, 17] has used the Beltrami equation for quasiconformal mappings [16] and for inverse design of airfoils [14]. Singular integral operators arise in solving problems in partial differential equations [4, 5, 13, 15, 31, 32, 33], fluid mechanics [3, 14], and electrostatics [27] using integral equation methods.

The use of quadrature rules to evaluate (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 problem

sizes (also, quadrature methods deliver poor accuracy when employed to evaluate ertain singular integrals). Daripa [15, 16, 18] 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 $\mathcal{O}(N^2 \log_2 N)$ or equivalently $\mathcal{O}(\log_2 N)$ 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.

Below we present a parallel algorithm to solve the singular integral operator (1). The recursive relations of the original algorithm [15, 18] (see § 2.1 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 remaining of this section is organized as follows. Subsection 2.1 reviews from [18] the sequential algorithm while subsection 2.2 describes the parallel implementation, subsection 2.3 presents the analysis of the parallel algorithm, and subsection 2.4, presents our approach to analyze the scalability of the algorithm.

2.1 The Algorithm

The fast algorithm to evaluate the singular integral transform (1) was developed in [16, 17]. The method divides the interior of he 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 [17]:

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

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

where

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

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

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

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

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 $m \ge 2$

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

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 $0 = r_1 < r_2 < \ldots < r_M = 1$ be the radii defined on the discretization. The following corollaries of Theorem 2.1 are presented verbatim from [17]:

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

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

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

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

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

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

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 $n \le -m$ as *negative modes* and the ones with index values $n \ge 0$ 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 subsection, we overcome this problem by redefining the formal description of the above algorithm.

2.2 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 [24]. 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 subsection 2.1 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

Algorithm 2.1 Sequential Algorithm for the Singular Integral Transform

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

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 $T_m h(r_l e^{2\pi ik/N}) = \sum_{n=-N/2}^{N/2-m} S_{n,m}(r_l) e^{2\pi ikn/N}$, $k \in [1, N]$, for each radius r_l, $l \in [1, M]$.

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

available processors and M be a multiple of P. There are distinct strategies to solve multiple FFTs in parallel systems [9, 22]. Three approaches are summarized in [9]: (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 [9], their performance is very similar. Perhaps the major disadvantage of the truncated FFT verson 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 [11].

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 $r_l e^{2\pi ik/N}$ 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 $r_l e^{2\pi ik/N}$, $k \in [1, N]$ 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 [2]. 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 neighbour 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 $C_{n,m}^{i-1,j}$ with $i \in [2, l]$, if $n \leq -m$, or $C_{n,m}^{l,l+1}$ with $i \in [l, M-1]$ if $n \geq 0$. 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 meassage-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 $\widetilde{C}_{n,m}^{l-1,l}$ 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 transferred from one processor to another, the data dependency imposed by (10) and (11) indicates that at most two processors (one for $n \le -m$ and other for $n \ge 0$) 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 $\mathcal{O}(NM)$, 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

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

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

so that sums $\sum_{i=2}^{l} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i}$ and $\sum_{i=l}^{M-1} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i,i+1}$ 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 $n \le -m$, and $\alpha/\beta = r_l/r_{l+1}$, $l = 1, \ldots, M-1$, for $n \ge 0$. 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 $n \le -m$, or a fast underflow for large values of $n \ge 0$. We overcome this problem by making use of the stabilized recurrences

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

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

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

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 $n \le -m$ we obtain

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

which implies that equation (13) can be rewritten as

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

Similarly, for a fixed $l \in [1, M-1]$ and $n \ge 0$ 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

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

For the purpose of achieving an even distribution of computational load across

processors, it is helpful to split the computational work when performing recurrences (16) and (17). We define the following *partial sums* for each processor p_j, $j = 0, \ldots, P-1$. For the case $n \le -m$, let

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

and for $n \ge 0$, let

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

Since coefficients $C_{n,m}^{i-1,i}$ $(n \le -m)$ and $C_{n,m}^{i,i+1}$ $(n \ge 0)$ 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 $\hat{s}_j^+$, $j = 0, \ldots, P-1$, are defined by

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

and

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

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

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

and

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

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

$$\hat{s}_j^-(n) = \left(\frac{r_{(j+1)M/P+1}}{r_{jM/P+1}}\right)^n \left[\left(\frac{r_{jM/P+1}}{r_{(j-1)M/P+1}}\right)^n \hat{s}_{j-2}^-(n) + t_{j-1}^-\right] + t_j^-$$

$$= r_{(j+1)M/P+1}^{n} \left[\frac{\hat{s}_{j-2}^{-}(n)}{r_{(j-1)M/P+1}} + \sum_{i=(j-1)M/P+1}^{jM/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i} + \sum_{i=jM/P+1}^{(j+1)M/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i} \right]$$

$$= r_{(j+1)M/P+1}^{n} \left[\frac{t_0^{-}(n)}{r_{M/P+1}} + \sum_{i=M/P+1}^{(j+1)M/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i} \right]$$

$$= r_{(j+1)M/P+1}^{n} \sum_{i=2}^{(j+1)M/P} \left(\frac{1}{r_i}\right)^n C_{n,m}^{i-1,i}. \tag{28}$$

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 $n \leq -m$ 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:

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

For $n \geq 0$, local updates in processor pP_{-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:

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

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 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 paragraphs. 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 $p_{P/1}$ 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 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.

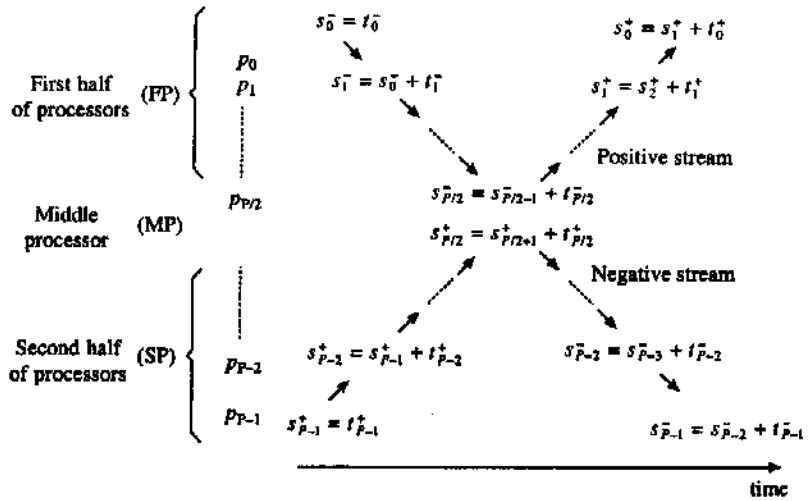

Fig. 2. Message distribution in the algorithm. Two streams of neighbor-to-neighbor messages cross communication channels simultaneously.

Load balance is a fundamental issue in parallel programming. Additionally, communication overhead is typically several orders of magnitude larger than hardware overhead [34]. 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 ($n \leq -m$), and the right side indicates computation work for positive modes ($n \geq 0$). 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

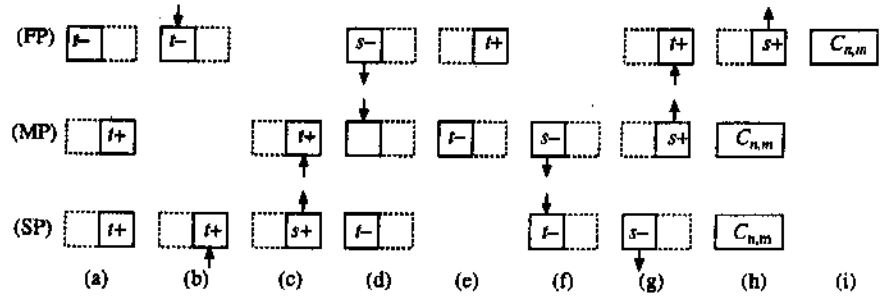

Fig. 3. Coordination scheme to minimize delays due to interprocessor communication. The middle processor (MP) plays a key role to forward the positive stream to the first half of processors (FP) and to forward the negative stream to the second half of processors (SP).

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 ($n \leq -m$) 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 ($n \geq 0$) 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 subsection, we analyze the late computations algorithm in detail and compare it with other approaches.

2.3 Analysis of the Parallel Algorithm

2.3.1 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 asociated 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 $C_{n,m}^{l,l+1}$ 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 $2((P-1)/2)(t_s + (N/2)t_w)$. In 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

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

To analyze the performance of the parallel algorithm, we must compare the above equation against the timing estimate for the sequential algorithm. In the later case, the algorithm starts performing M Fourier tansforms 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 tansforms take $(c_1/2)MN\log_2 N$ computations. Therefore, the sequential timing T_s is given by

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

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 speed-up S^{stream} for the algorithm

$$S^{\text{stream}} = \frac{T_s}{T_P^{\text{stream}}} = \frac{c_1 MN\log_2 N + \frac{2}{3} c_2 MN}{c_1 \frac{M}{P} N\log_2 N + c_2 \frac{M}{P} N + 2(P-1)\left(t_s + \frac{N}{2} t_w\right)} \tag{33}$$

$$= P \frac{c_1 MN\log_2 N + \frac{2}{3} c_2 MN}{c_1 MN\log_2 N + c_2 MN + 2P(P-1)\left(t_s + \frac{N}{2} t_w\right)}, \tag{34}$$

and the resulting efficiency

$$E^{\text{stream}} = \frac{S^{\text{stream}}}{P} = \frac{c_1 MN\log_2 N + \frac{2}{3} c_2\, MN}{c_1 MN\log_2 N + c_2 MN + 2P(P-1)\left(t_s + \frac{N}{2} t_w\right)}. \tag{35}$$

$$= \frac{1}{1 + \left(\frac{c_2}{3} MN + 2P(P-1)\left(t_s + \frac{N}{2} t_w\right)\right) \Big/ \left(c_1 MN\log_2 N + \frac{2}{3} c_2 MN\right)}. \tag{36}$$

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

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

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 $\frac{\partial T_P}{\partial P} = 0$. In the case of (37), we have

$$P_{opt}^{\text{asymp}} = \sqrt{\frac{c_1 MN \log_2 N}{2\left(t_s + \frac{N}{2} t_w\right)}}, \tag{38}$$

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.

2.3.2 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 $C_{n,m}^{l-1,j}$, $n \in [jN/P - N/2, (j+1)N/P - N/2]$. Figure 4(b) describes the communication pattern. Rows are divided into P blocks of size NM/P^2 so that the processor p_j 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 [26] is given by

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

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

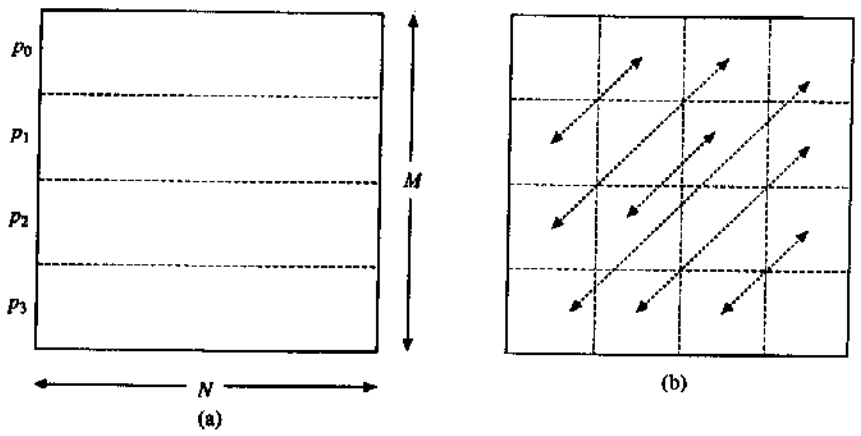

Fig. 4. Coordination pattern based on all-to-all personalized communication: (a) *M/P* Fourier transforms are evaluated locally; (b) each two processors exchange blocks of size MN/P^2.

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

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

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 $M >> P$ due to practical limitations on the number of available processors which makes $PN << 4MN/\sqrt{P}$. 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 subsection 2.2. For this purpose, we consider a parallel binary-exchange algorithm for FFT as described in [26]. In binary-exchange FFT, data is exchanged betwen all pairs of processors with labelling indexes 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 proce 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 inpose 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 subsection 2.2, the amount of communication due to the binary-exchange algorithm is given by [26]

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

which is equivalent to a communication delay with the same order $O(NM/\sqrt{P})$ 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.

2.4 Analysis for A Coarse-grained Data Distribution

The degree of parallelism indicates the extent to which a parallel program matches the parallel architecture. Speed up captures the performance gain when utilizing a parallel system [35]:

- *True speed-up* 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 speed-up* the sequential time to be used is determined by scaling down the parallel code to one processor.

Efficiency indicates the degree of speed-up 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 speed-ups and efficiencies in a coarse-grained data distribution, we define

- *Modified speed-ups* $S^{[20]}$ 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 speed-up, the modified speed-up $S^{[20]}$ adopts $20T_{20}$ as the running time for the sequential version of the algorithm. It basically corresponds to assuming optimal speed-ups 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. Speed-ups and efficiencies can be analyzed for up to 60 processors by using $P = 20$ as a reference configuration.

The implementation of the parallel algorithm described above and the numerical results of various case studies with this algorithm are presented and discussed in detail in Borges and Daripa [6, 7]. These are not being presented due to restrictions on the number of pages of this chapter.

Below we present sequential and parallel fast algorithms for Poisson equation on a disk. The following section is self-contained and can be studied on its own.

3. A Fast Parallel Algorithm for the Poisson Equation

The Poisson equation is one of the fundamental equations in mathematical physics which, for example, governs the spatial variation of a potential function for given source terms. The range of applications covers from magnetostatic problems to ocean modelling. Fast, accurate and reliable numerical solvers play a significant role in the development of applications for scientific problems. In this section, we present an efficient sequential and parallel algorithms for solving the Poisson equation on a disk using Green's function method.

A standard procedure to solve the Poisson equation using Green's function method requires evaluation of volume integrals which define contribution to the solution due to source terms. However, the complexity of this approach in two-dimension is $\mathcal{O}(N^4)$ for a N^2 net of grid points which makes the method prohibitive for large-scale problems. Here, we expand the potential in terms of Fourier series by deriving radius dependent Fourier coefficients. These Fourier coefficients can be obtained by recursive relations which only utilize one-dimensional integrals in the radial directions of the domain. Also, we show that these recursive relations make it possible to define high-order numerical integration schemes in the radial directions without taking additional grid points. Results are more accurate because the algorithm is based on exact analysis: the method presents high accuracy even for problems with sharp variations on inhomogeneous term. On single processor machine, the method has a theoretical computational complexity $\mathcal{O}(N^2 \log_2 N)$ or equivalently $\mathcal{O}(\log_2 N)$ per point which represents substantial savings in computational time when compared with the complexity $\mathcal{O}(N^2)$ for standard procedures.

The above basic philosophy mentioned has been applied in section 2 above in the context of developing fast algorithms for efficient evaluation of singular integrals (see also [18]) in the complex plane. The mathematical machinery behind this philosophy is applied in subsection 3.2 below for the presentation of a theorem (Theorem 2.1) which outlines the fast algorithm for solving the Poisson equation in the real plane. The proof of this theorem can be found in Borges and Daripa [8]. The theorem 3.1 follows the general format of the theorem 2.1.

We address the parallelization of the algorithm in some detail which is one of the main thrusts of this section. The resulting algorithm is very scalable due to

the fact that communication costs are independent of the number of annular regions taken for the domain discretization. It means that an increasing number of sample points in the radial direction does not increase overheads due to interprocessor coordination. Message lengths depend only on the number of Fourier coefficients in use. Communication is performed in a linear path configuration which allows overlapping of computational work simultaneously with data-exchanges. This overlapping guarantees that the algorithm is well suited for distributed and shared memory architectures. Here our numerical experiments show the good performance of the algorithm in a shared memory computer. Related work in section 2 (see also [6, 7]) shows the suitability for distributed memory. It makes the algorithm architecture-independent and portable. Moreover, the mathematical formulation of the parallel algorithm presents a high level of data locality, which results on an effective use of cache.

At this point, it is worth mentioning that there now exists a host of fast parallel Poisson solvers based on various principles including the use of FFT and Fast Multipole method [28, 10, 12, 30]. The fast solver of this section is based on the theorem 2.1 which is derived through exact analyses and properties of convolution integrals involving Green's function. *Thus, this solver is very accurate due to these exact analyses which is demonstrated on a host of problems in [8]. Accuracy can be further improved by incorporating some symmetry properties of a disk which we do not discuss here.*

Moreover, this solver is easy to implement and has a very low constant hidden behind the order estimate of the complexity of the algorithm. This gives this solver an advantage over many other solvers with similar complexity which usually have a high value of this hidden constant. Furthermore, this solver can be very optimal for solving certain classes of problems involving circular domains or overlapped circular domains. this solver can also be used in arbitrary domains via spectral domain embedding technique. This work is currently in progress.

In subsection 3.1 we start presenting the mathematical preliminaries of the algorithm and deriving the recursive relations. In subsection 3.2 we describe the sequential implementation and two variants of the integration scheme. Subsection 3.3 introduces the parallel implementation and its theoretical analysis.

3.1 Mathematical Preliminaries

In this subsection we introduce the mathematical formulation for a fast solver for Dirichlet problems. Also recursive relations are presented leading to an efficient numerical algorithm. Finally, the mathematical formulation is extended to Neumann problems.

3.1.1 The Dirichlet Problem and its Solution on a Disk

Consider the Dirichlet problem of the Poisson equation

$$\begin{cases} \Delta u = f & \text{in } B \\ u = g & \text{on } \partial B, \end{cases} \tag{43}$$

where $B = B(0, R) = \{x \in \mathbb{R}^2 : |x| < R\}$. Specifically, let v satisfy

$$\Delta v = f \quad \text{in } B, \tag{44}$$

and w be the solution of the homogeneous problem

$$\begin{cases} \Delta w = 0 & \text{in } B \\ w = g - v & \text{on } \partial B. \end{cases} \tag{45}$$

Thus, the solution of the Dirichlet problem (43) is given by

$$u = v + w. \tag{46}$$

A principal solution of equation (44) can be written as

$$v(x) = \int_B f(\eta)\, G(x, \eta)\, d\eta, \qquad x \in B, \tag{47}$$

where $G(x, \eta)$ is the free-space Green's function for the Laplacian given by

$$G(x, \eta) = \frac{1}{2\pi} \log | x - \eta |. \tag{48}$$

To derive a numerical method based on equation(47), the interior of the disk $B(0, R)$ is divided into a collection of annular regions. The use of Quadrature rules to evaluate (47) incurs in poor accuracy for the approximate solution. Moreover, the complexity of a quadrature method is $\mathcal{O}(N^4)$ for a N^2 net of grid points. For large problem sizes it represents prohibitive costs in computational time. Here we expand $v(\cdot)$ in terms of Fourier series by deriving radius dependent Fourier coefficients of $v(\cdot)$. These Fourier coefficients can be obtained by recursive relations which only utilize one-dimensional integrals in the radial direction. The fast algorithm is embedded in the following theorem:

Theorem 3.1 If $u(r, \alpha)$ is the solution of the Dirichlet problem (43) for $x = re^{i\alpha}$ and $f(re^{i\alpha}) = \sum_{n=-\infty}^{\infty} f_n(r) e^{in\alpha}$, then the nth Fourier coefficient $u_n(r)$ of $u(r, \cdot)$ can be written as

$$u_n(r) = v_n(r) + \left(\frac{r}{R}\right)^{|n|} (g_n - v_n(R)), \qquad 0 < r < R, \tag{49}$$

where g_n are the Fourier coefficients of g on ∂B, and

$$v_n(r) = \int_0^r p_n(r, \rho)\, d\rho + \int_r^R q_n(r, \rho)\, d\rho, \tag{50}$$

with

$$p_n(r, \rho) = \begin{cases} \rho \log r f_0(\rho), & n = 0, \\ \dfrac{-\rho}{2|n|} \left(\dfrac{\rho}{r}\right)^{|n|} f_n(\rho), & n \neq 0, \end{cases} \tag{51}$$

and

$$q_n(r, \rho) = \begin{cases} \rho \log \rho f_0(\rho), & n = 0, \\ \dfrac{-\rho}{2|n|} \left(\dfrac{r}{\rho}\right)^{|n|} f_n(\rho), & n \neq 0. \end{cases} \tag{52}$$

3.1.2 Recursive Relations of the Algorithm

Despite the fact that the above theorem presents the mathematical foundation of the algorithm, an efficient implementation can be devised by making use of recursive relations to perform the integrations in (50). Consider the disk $\overline{B(0,R)}$ discretized by $N \times M$ lattice points with N equidistant points in the angular direction and M distinct points in radial direction. Let $0 = r_1 < r_2 < \ldots < r_M = R$ be the radii defined on the discretization. Theorem 3.1 leads to the following corollaries:

Corollary 3.1 It follows from (50) and (52) that $v_n(0) = 0$ for $n \neq 0$.

Corollary 3.2 Let $0 = r_1 < r_2 < \ldots < r_M = R$, and

$$C_n^{i,j} = \int_{r_i}^{r_j} \frac{\rho}{2n} \left(\frac{r_j}{\rho}\right)^n f_n(\rho)\, d\rho, \quad n < 0, \tag{53}$$

$$D_n^{i,j} = -\int_{r_i}^{r_j} \frac{\rho}{2n} \left(\frac{r_i}{\rho}\right)^n f_n(\rho)\, d\rho, \quad n > 0. \tag{54}$$

If for $r_j > r_i$, we define

$$\begin{cases} v_n^-(r_1) = 0, & n < 0, \\ v_n^-(r_j) = \left(\dfrac{r_j}{r_i}\right)^n v_n^-(r_i) + C_n^{i,j}, & n < 0, \end{cases} \tag{55}$$

and

$$\begin{cases} v_n^+(r_M) = 0, & n > 0, \\ v_n^+(r_i) = \left(\dfrac{r_i}{r_j}\right)^n v_n^+(r_j) + D_n^{i,j}, & n > 0, \end{cases} \tag{56}$$

then for $i = 1, \ldots, M$, we have

$$v_n(r_i) = \begin{cases} v_n^-(r_i) + \overline{v_{-n}^+(r_i)}, & n < 0, \\ v_n^+(r_i) + \overline{v_{-n}^-(r_i)}, & n > 0. \end{cases} \tag{57}$$

Corollary 3.3 Let $0 = r_1 < r_2 < \ldots < r_M = R$, and add $n = 0$ to the definitions in Corollary 3.2 as

$$C_0^{i,j} = \int_{r_i}^{r_j} \rho f_0(\rho)\, d\rho \quad \text{and} \quad D_0^{i,j} = \int_{r_i}^{r_j} \rho \log \rho f_0(\rho)\, d\rho, \tag{58}$$

then given $l = 1, \ldots, M$ we have

$$v_n(r_l) = \begin{cases} \log r_l \displaystyle\sum_{i=2}^{l} C_0^{i-1,i} + \sum_{i=l}^{M-1} D_0^{i,i+1}, & \text{for } n = 0, \\ \displaystyle\sum_{i=2}^{l} \left(\frac{r_l}{r_i}\right)^n C_n^{i-1,i} + \sum_{i=l}^{M-1} \left(\frac{r_i}{r_l}\right)^n \overline{D_{-n}^{i,i+1}}, & \text{for } n < 0, \\ \displaystyle\sum_{i=l}^{M-1} \left(\frac{r_l}{r_i}\right)^n D_n^{i,i+1} + \sum_{i=2}^{l} \left(\frac{r_i}{r_l}\right)^n \overline{C_{-n}^{i-1,i}}, & \text{for } n > 0. \end{cases} \tag{59}$$

It is important to emphasize that M distinct points $r_1, \ldots, r_M$ need not to be equidistant. Therefore, the fast algorithm can be applied on domains that are nonuniform in the radial direction. This anisotropic grid refinement may at first seem unusual with elliptic problems. Even though it is true that isotropic grid refinement is more common with solving elliptic equations, there are exceptions to the rule, in particular with a hybrid method such as ours (Fourier in one direction and finite difference in the other direction). Since, Fourier methods are spectrally accurate, grid refinement along the circumferential direction beyond a certain optimal level may not always offer much advantage. This is well known because of the exponential decay rate of Fourier coefficients for a classical solution (c^∞ function). This fact has been exemplified in Borges and Daripa [8] (see Example 1 and Table 1 in subsection 5.1 of [8]) where we have shown that to get more accurate results one needs to increase the number of annular regions without increasing the number of Fourier coefficients participating in the calculation, i.e. anisotropic grid refinement with more grids in the radial direction than in the circumferential direction is more appropriate for that problem.

3.1.3 The Neumann Problem and its Solution on a Disk

The same results obtained for solving the Dirichlet problem can be generalized for the Neumann problem by expanding the derivative of the principal solution v in (47). Consider the Neumann problem

$$\begin{cases} \Delta u = f & \text{in } B \\ \dfrac{\partial u}{\partial \vec{n}} = \psi & \text{on } \partial B, \end{cases} \tag{60}$$

The analogous of Theorem 3.1 for the Neumann problem is given by

Theorem 3.2 If $u(r, \alpha)$ is the solution of the Neumann problem (60) for $x = re^{i\alpha}$ and $f(re^{i\alpha}) = \sum_{n=-\infty}^{\infty} f_n(r)e^{in\alpha}$, then the nth Fourier coefficient $u_n(r)$ of $u(r, \cdot)$ can be written as

$$\begin{cases} u_0(r) = v_0(r) + \varphi_0, & n = 0 \\ u_n(r) = v_n(r) + \left(\dfrac{r}{R}\right)^{|n|}\left(\dfrac{R}{|n|}\psi_n + v_n(R)\right), & n \neq 0, \end{cases} \tag{61}$$

where ψ_n are the Fourier coefficients of ψ on ∂B, v_n are defined as in Theorem 3.1, and φ_0 is the parameter which sets the additive constant for the solution.

3.2 The sequential Algorithm

An efficient implementation of the algorithm embedded in Theorem 3.1 is derived from Corollary 3.2. It defines recursive relations to obtain the Fourier coefficients v_n in (49) based on the sign of the index n of v_n. In the description of the algorithm, we address the coefficients with index values $n \leq 0$ as *negative modes* and the ones with index values $n \geq 0$ as *positive modes*. Equation (55) shows that negative modes are built up from the smallest radius r_1 towards the largest radius

r_M. Conversely, equation (56) constructs positive modes from r_M towards r_1. Figure 5 presents the resulting sequential algorithm for the Dirichlet problem. The counterpart algorithm for the Neumann problem similarly follows from Theorem 3.2 and Corollary 3.2.

Algorithm 3.1 Sequential Algorithm for the Poisson Equation on a Disk

Given M, N, the grid values $f(r_l e^{2\pi ik/N})$ and the boundary conditions $g(Re^{2\pi ik/N})$, $l \in [1, M]$, $k \in [1, N]$, the algorithm returns the values $u(r_l e^{2\pi ik/N})$, $l \in [1, M, k \in [1, N]$ of the solution for the Dirichlet problem (43).

1. Compute the Fourier coefficients $f_n(r_l)$, $n \in [-N/2, N/2]$, for M sets of data at $l \in [1, M]$, and the Fourier coefficients g_n on ∂B.
2. For $i \in [1, M-1]$, compute the radial one-dimensional integrals $C_n^{i,i+1}$, $n \in [-N/2, 0]$ as defined in (53) and (58); and compute $D_n^{i,i+1}$, $n \in [0, N/2]$ as defined in (54) and (58).
3. Compute coefficients $v_n^-(r_l)$ for each of the negative modes $n \in [-N/2, 0]$ as defined in (55) and (59):
   (a) Set $v_n^-(r_1) = 0$ for $n \in [-N/2, 0]$.
   (b) For $l = 2, \ldots, M$
   $$v_n^-(r_l) = \left(\frac{r_l}{r_{l-1}}\right)^n v_n^-(r_{l-1}) + C_n^{l-1,l}, \quad n \in [-N/2, 0].$$
4. Compute coefficients $v_n^+(r_l)$ for each of the positive modes $n \in [0, N/2]$ as defined in (56) and (59):
   (a) Set $v_n^+(r_M) = 0$ for $n \in [0, N/2]$.
   (b) For $l = M-1, \ldots, 1$
   $$v_n^+(r_l) = \left(\frac{r_l}{r_{l+1}}\right)^n v_n^+(r_{l+1}) + D_n^{l,l+1}, \quad n \in [0, N/2].$$
5. Combine coefficients v_n^+ and v_n^- as defined in (57) and (59):
   For $l = 1, \ldots, M$
   $$v_0(r_l) = \log r_l\, v_0^-(r_l) + v_0^+(r_l).$$
   $$v_n(r_l) = \overline{v_{-n}(r_l)} = v_n^-(r_l) + \overline{v_{-n}^+(r_l)}, \quad n \in [-N/2, -1].$$
6. Apply the boundary conditions as defined in (49):
   For $l = 2, \ldots, M$
   $$u_n(r_l) = v_n(r_l) + \left(\frac{r_l}{R}\right)^{|n|} (g_n - v_n(R)), \quad n \in [-N/2, N/2].$$
7. Compute $u(r_l e^{2\pi ik/N}) = \sum_{n=-N/2}^{N/2} u_n(r_l) e^{2\pi ikn/N}$, $k \in [1, N]$, for each radius r_l, $l \in [1, M]$.

Fig. 5. Description of the sequential algorithm for the Dirichlet problem.

Notice that Algorithm 3.1 requires the radial one-dimensional integrals $C_n^{i,i+1}$ and $D_n^{i,i+1}$ to be calculated between two successive points (indexed by i and $i+1$) on a given radial direction (defined by n). One possible numerical method to obtain these integrals would be to use the trapezoidal rule. However, the trapezoidal rule presents an error of quadratic order. One natural approach to increase the accuracy of the numerical integration would be to add auxiliary points between the actual points of the discretization of the domain to allow higher order integration methods to obtain $C_n^{i,i+1}$ and $D_n^{i,i+1}$. This approach presents two major

disadvantages: *1*. It substantially increases computational costs of the algorithm because the fast Fourier transforms in step 1 of Algorithm 3.1 must also be performed for all the new circles of extra points added for the numerical integration; 2. In practical problems the values for function f may only be available on a finite set of points, which constrains the data to a fixed discretization of the domain and no extra grid points can be added to increase the accuracy of the solver.

Here, we increase the accuracy of the radial integrals by redefining steps 2, 3 and 4 of Algorithm 3.1 based on the more general recurrences presented in equations (55) and (56). Terms $C_n^{i,i+1}$ and $D_n^{i,i+1}$ are evaluated only using two consecutive points. In fact, for the case $n < 0$ one can apply the trapezoidal rule for (53) leading to

$$C_n^{i,i+1} = \frac{(\delta r)^2}{4n}\left(i\left(\frac{i}{i+1}\right)^{-n} f_n(r_i) + (i+1) f_n(r_{i+1})\right) \tag{62}$$

for a uniform discretization where $r_i = (i-1)\delta r$. It corresponds to the trapezoidal rule applied between circles r_i and r_{i+1}. A similar equation holds for $D_n^{i,i+1}$. By evaluating terms of the form $C_n^{i-1,i+1}$ and $D_n^{i-1,i+1}$, three consecutive points can be used in the radial direction. It allows the use of the Simpson's rule

$$C_n^{i-1,i+1} = \frac{(\delta r)^2}{6n}\left((i-1)\left(\frac{i-1}{i+1}\right)^{-n} f_n(r_{i-1}) + 4i\left(\frac{i}{i+1}\right)^{-n} f_n(r_i) + (i+1) f_n(r_{i+1})\right), \tag{63}$$

which increases the accuracy of the method. In the algorithm, it corresponds to redefining step 3 for $n < 0$ as

$$\begin{cases} v_n^-(r_1) = 0, \\ v_n^-(r_2) = C_n^{1,2}, \\ v_n^-(r_l) = \left(\dfrac{r_l}{r_{l-2}}\right)^n v_n^-(r_{l-2}) + C_n^{l-2,l}, \quad l = 3, \ldots, M, \end{cases}$$

and step 4 for $n > 0$ as

$$\begin{cases} v_n^+(r_M) = 0, \\ v_n^+(r_{M-1}) = D_n^{M-1,M}, \\ v_n^+(r_l) = \left(\dfrac{r_l}{r_{l+2}}\right)^n v_n^+(r_{l+2}) + D_n^{l,l+2}, \quad l = M-2, \ldots, 1. \end{cases}$$

It results on an integration scheme applied between three successive circles, say

r_{i-1}, r_i and r_{i+1}, with computational costs practically similar to the trapezoidal rule but with higher accuracy. The above Simpson's rule presents an error formula of fourth order in the domain of length $2\delta r$. For sufficiently smooth solutions, it allows cubic convergence in δr.

3.3 The Parallel Algorithm

Current resources in high performance computing can be divided into two major models: distributed and shared memory architectures. The design of a parallel and portable application must attempt to deliver a high user-level performance in both architectures. In this subsection, we present a parallel implementation suited for the distributed and shared models. Although we conduct our presentation using the message-passing model, this model can also be employed to describe interprocessor coordination: higher communication overhead corresponds to larger data dependency in the algorithm, which results on loss of data locality. Even though shared memory machines have support for coherence, good performance requires locality of reference because of the memory hierarchy. Synchronization and true sharing must be minimized [1]. Efficient parallelized codes synchronize infrequently and have little true sharing [38]. Therefore, a good parallelization requires no communication whenever possible. Using the data decomposition which allows lower communication cost also improves the data locality. The performance of the parallel algorithm on distributed memory systems have been addressed in section 2 (see also [6]). There a variant of the algorithm has been used for fast and accurate evaluation of singular integral transforms.

The recursive relations in Corollary 3.2 are very appropriate to a sequential algorithm. However, they may represent a bottleneck in a parallel implementation. In this subsection we use the results presented in Corollary 3.3 to devise an efficient parallel solver for the Poisson equation. Theoretical estimates for the performance of the parallel version of the algorithm are given below. We also show that this parallel solver has a better performance characteristics than an implementation based on Corollary 3.2. Finally, we compare our parallel algorithm with other Poisson solvers.

3.3.1 Parallel Implementation

The fast algorithm for the Poisson equation requires multiple fast Fourier transforms (FFT) to be performed. There are distinct strategies to solve multiple FFTs in parallel systems [9, 22]. In section 2 (see also [6]) we have shown that an improved implementation of parallel calls to sequential FFTs is the best choice for the fast algorithm. For the sake of a more clear explanation, let P be the number of available processors and M be a multiple of P. Data partitioning is defined by distributing the circles of the domain into P groups of consecutive circles so that each processor contains the grid points for M/P circles. 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

$l \in [\gamma(j) + 1, \gamma(j + 1)]$. Figure 6 exemplifies the data distribution for a system with three processors ($P = 3$).

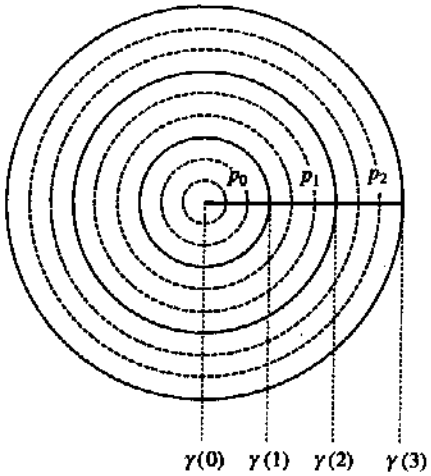

Fig. 6. Data distribution for the parallel version of the fast algorithm.

One optimized version of a sequential N-point FFT algorithm is available on each processor: multiple Fourier transforms of the same length are performed simultaneously. The M sequences of values assumed on the N grid points belonging to a circle are distributed between processors so that each one performs one unique call to obtain M/P FFT transforms. Overall, the FFT transforms contribute the most to the computational cost of the algorithm and the above data-locality allows the intensive floating point operations to be performed locally and concurrently. Thus, each FFT can be evaluated in place, without communication. Other strategies for solving the multiple FFTs required in the algorithm are discussed in section 2 (see also [6]).

Although Corollary 3.2 is formulated for the generic case $r_j > r_l$, the results in Corollary 3.3 only require consecutive radii (i.e., terms of the form $C_n^{l-1,l}$ and $D_n^{l,l+1}$, $l \in [\gamma(j) + 1, \gamma(j + 1)]$) in processor p_j. Therefore, the numerical integration for equations (53), (54) and (58) can be performed locally if one guarantees that all necessary data is available within the processor. Notice that p_j already evaluates the Fourier coefficients $f_n(r_l)$, $l \in [\gamma(j) + 1, \gamma(j + 1)]$. In the case of a numerical integration based on the trapezoidal rule (62) only the Fourier coefficients for $l = jM/P$ and $l = (j + 1)M/P + 1$ must be added to the set of known Fourier 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. Similarly, if the modified simpson's rule (63) is employed, processor p_j only needs to evaluates coefficients for radii r_l, $l \in [\gamma(j) - 1, \gamma(j + 1) + 2]$. The number of circles whose data overlap between any two neighbor processors remains fixed regardless of the total number of processors in use. Consequently, this strategy does not compromise the scalability of the algorithm.

Algorithm 3.1 was described based on the inherently sequential iterations from Corollary 3.2 which are more suitable for a sequential implementation. In the case of a parallel algorithm, an even distribution of computational load is obtained by splitting the computational work when performing recurrences (58) and (59) as described in Corollary 3.3. We evaluate iterative sums q_l, $l \in [\gamma(j), \gamma(j+1)]$, concurrently on all processors p_j, $j = 0, \ldots, P-1$, as follows. For the case $n \le 0$ let

$$\begin{cases} q^-_{\gamma(j)}(n) = 0, \\ q^-_l(n) = \left(\dfrac{r_{l+1}}{r_l}\right)^n (q^-_{l-1}(n) + C_n^{l-1,l}), \quad l = \gamma(j)+1, \ldots, \gamma(j+1), \end{cases} \tag{64}$$

where we have defined $r_{M+1} = 1$, and for the case $n \ge 0$ let

$$\begin{cases} q^+_{\gamma(j+1)+1}(n) = 0, \\ q^+_l(n) = \left(\dfrac{r_{l-1}}{r_l}\right)^n (q^+_{l+1}(n) + D_n^{l,l+1}), \quad l = \gamma(j+1), \ldots, \gamma(j)+1. \end{cases} \tag{65}$$

Since coefficients $C_n^{l-1,l}$ $(n \le 0)$ and $D_n^{l,l+1}$ $(n \ge 0)$ 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. In section 2 (see also [6]) we have shown that the above computations can be used to define the following *partial sums* for each processor p_j:

$$t_j^-(n) = q^-_{\gamma(j+1)}(n), \quad n \le 0,$$

$$t_j^+(n) = q^+_{\gamma(j)+1}(n), \quad n \ge 0.$$

Moreover, it follows from (64) and (65) that for $n \le 0$

$$\begin{cases} t_0^-(n) = r_{\gamma(1)+1}^n \displaystyle\sum_{i=2}^{\gamma(1)} \left(\frac{1}{r_i}\right)^n C_n^{i-1,i}, \\ t_j^-(n) = r_{\gamma(j+1)+1}^n \displaystyle\sum_{i=\gamma(j)+1}^{\gamma(j+1)} \left(\frac{1}{r_i}\right)^n C_n^{i-1,i}, \end{cases}$$

and for $n \ge 0$

$$\begin{cases} t_{P-1}^+(n) = r_{\gamma(P-1)}^n \displaystyle\sum_{i=\gamma(P-1)+1}^{M-1} \left(\frac{1}{r_i}\right)^n D_n^{i,i+1}, \\ t_j^+(n) = r_{\gamma(j)}^n \displaystyle\sum_{i=\gamma(j)+1}^{\gamma(j+1)} \left(\frac{1}{r_i}\right)^n D_n^{i,i+1}. \end{cases}$$

Although sums as described above may seem to produce either fast overflows or fast underflows for large absolute values of n, partial sums t_j^- and t_j^+ can be obtained by performing very stable computations (64) and (65) as described in

section 2 (see also [6]). Therefore the algorithm proceeds by performing the partial sums in parallel as represented in Figure 7.

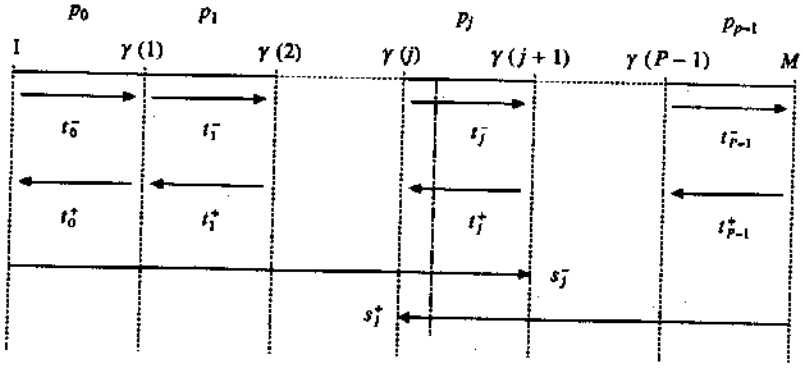

Fig. 7 Sums are evenly distributed across processors.

To combine partial sums t_j^- and t_j^+ evaluated on distinct processors, we define the *accumulated sums* $\hat{s}_j^-$ and $\hat{s}_j^+$, $j = 0, \ldots, P-1$. For $n \leq 0$ let

$$\begin{cases} \hat{s}_0^-(n) = t_0^-(n), \\ \hat{s}_j^-(n) = \left(\dfrac{r_{\gamma(j+1)+1}}{r_{\gamma(j)+1}}\right)^n \hat{s}_{j-1}^-(n) + t_j^-, \end{cases} \tag{66}$$

and for $n \geq 0$

$$\begin{cases} \hat{s}_{P-1}^+(n) = t_{P-1}^+(n), \\ \hat{s}_j^+(n) = \left(\dfrac{r_{\gamma(j)}}{r_{\gamma(j+1)}}\right)^n \hat{s}_{j+1}^+(n) + t_j^+. \end{cases} \tag{67}$$

Therefore we have a recursive method to accumulate partial sums t_j^- and t_j^+ computed in processors p_j. Accumulated sums $\hat{s}_j^-$ and $\hat{s}_j^+$ can now be used to calculate coefficients C_n and D_n locally on each processor. Given a fixed radius r_l, the associated data belongs to the processor p_j such that $l \in [\gamma(j) + 1, \gamma(j+1)]$. Computations in p_j only make use of accumulated sums from neighbor processors. For $n \leq 0$ local updates in processor p_0 are performed as described in Corollary 3.2. Local updates in processors p_j, $j = 1, \ldots, P-1$, use the accumulated sums $\hat{s}_{j-1}^-$ from the previous processor when obtaining terms v_n^- as defined in equation (55):

$$\begin{cases} v_n^-(r_{\gamma(j)+1}) = \hat{s}_{j-1}^-(n) + C_n^{\gamma(j),\gamma(j)+1} \\ v_n^-(r_l) = \left(\dfrac{r_l}{r_{l-1}}\right)^n v_n^-(r_{l-1}) + C_n^{l-1,l}. \end{cases} \tag{68}$$

For $n \geq 0$ local updates in processor p_{P-1} are also performed as described in Corollary 3.2. Local updates in processors p_j, $j = 0, \ldots, P-2$ use the accumulated sum $\hat{s}_{j+1}^+$ from the next processor to obtain terms v_n^+ from equation (56):

$$\begin{cases} v_n^+(r_{\gamma(j+1)}) = -\hat{s}_{j+1}^+(n) - D_n^{\gamma(j+1),\gamma(j+1)+1} \\ v_n^-(r_l) = \left(\dfrac{r_l}{r_{l+1}}\right)^n v_n^+(r_{l+1}) + D_n^{l,l+1}. \end{cases} \tag{69}$$

The advantage of using equations (68) and (69) over original recurrences in Corollary 3.2 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 next step in the algorithm consists of combining coefficients v_n^+ and v_n^- to obtain the component v_n of the solution as described in step 5 of Algorithm 3.1. Notice that for a fixed radius r_l, coefficients $v_n^-(r_l)$ and $v_{-n}^+(r_l)$, $n \in [-N/2, 0]$, are stored in the same processor. Therefore, computations in (59) can be performed locally and concurrently, without any communication. Specifically, processor p_j evaluates terms $v_n(r_l)$, $n \in [-N/2, N/2]$, where $l \in [\gamma(j) + 1, \gamma(j+1)]$. A final set of communications is employed to broadcast the values $v_n(R)$, $n \in [-N/2, N/2]$, from p_{P-1} to all other processors so that the Fourier coefficients u_n of the solution can be evaluated by using equation (49), similarly as represented in step 6 of Algorithm 3.1. This broadcast process is represented in Figure 8 by the second set of upward arrows starting from processor p_{P-1}.

The notation in equations (66) and (67) will be simplified to allow a clear exposition of the inter-processor communication present in our parallel implementation:

- Relation $s_j^- = s_{j-1}^- + t_j^-$ represents the updating process in recurrence (66), and
- Relation $s_j^+ = s_{j+1}^+ + t_j^+$ represents updating (67).

The parallel algorithm adopts the successful approach mentioned in section 2 and investigated in detail through implementation in [6, 7]. 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) as represented in Figure 8.

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 8. 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 first set of upward arrows in Figure 8. The resulting algorithm is composed by two simultaneous streams of neighbor-to-neighbor communication, each one with messages of

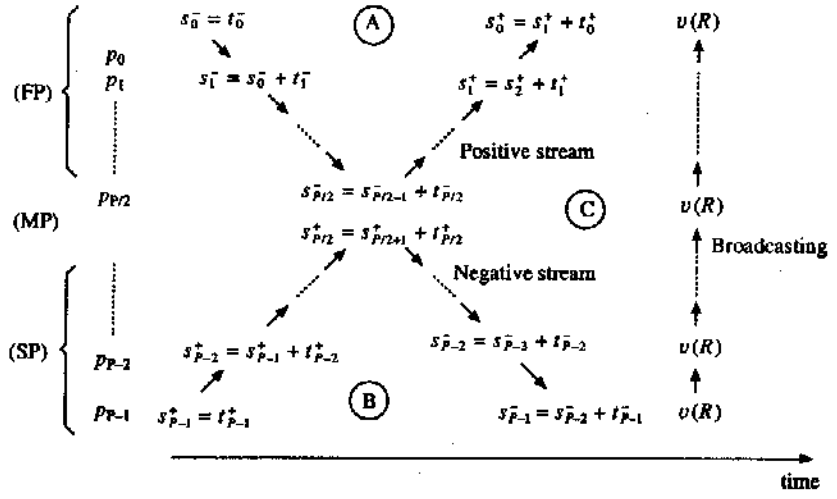

Fig. 8. Message distribution in the algorithm. Two streams of neighbor-to-neighbor messages cross communication channels simultaneously. Homogeneous and particular solution are combined after processor p_{P-1} broadcasts the boundary values of v.

length $N/2$. Note from Figure 8 that negative and positive streams arrive at the middle processor simultaneusly due to the symmetry of the communication structure. In section 2 (see also [6, 7]) 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 evaluating t_j^+ within region A in Figure 8. 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 section 2 (see also [6]) 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.

The parallel algorithm presented here is certainly based on decomposing the domain into full annular regions and hence, it has some analogy with domain decomposition method. But this analogy is superficial because domain decomposition methods by its very name have come to refer to methods which attempt to solve the same equations in every subdomain, wheres our algorithm *does not* attempt to solve the same equation in eacyh annular subdomain separately. Thus our algorithm is not a classical domain decomposition method. Interpreting otherwise would be misleading. In fact, decomposing a circular domain into full

annular domains and then attempting to solve the equation in each subdomain in the spirit of domain decomposition method would not be very appealing for a very large number of domains because the surface to volume area becomes very large. Our algorithm is not based on this principle in its entirety, even though there is some analogy which is unavoidable.

3.3.2 Complexity of the Parallel Algorithm

To analyze the overhead due to interprocessor coordination in the parallel algorithm we adopt a standard communication model for distributed memory computers. For the timing analysis we consider t_s as the message startup time and t_w the transfer time for a complex number. To normalize the model, we adopt constants c_1 as the computational cost for floating point operations in the FFT algorithm, and c_2 to represent operation counts for the other stages of the algorithm. To obtain the model, we analyze the timing for each stage of the algorithm:

- Each processor performs a set of M/P Fourier transforms in $(c_1/2)(M/P)N \log_2 N$ operations.
- Radial integrals $C_n^{l,l+1}$ and $D_n^{i-1,i}$ are obtained using $(c_2/4)(M/P)N$ operations for the trapezoidal rule (and $(c_2\ 2/3)(M/P)N$ for Simpson's rule).
- Each group of M/P partial sums t^+ and t^- takes $(c_2/4)(M/P)(N/2)$ operations on each processor.
- 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 (see Figure 8). The total time is $2(P-1)/2)(t_s + (N/2)t_w)$.
- The second group of M/P partial sums t^+ and t^- is performed in $(c_2/4)(M/P)(N/2)$ 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.
- Terms v^-, v^+ and v are computed in $(c_2/4)(M/P)N$ operations.
- Boundary conditions are broadcast in $(t_s + Nt_w) \log_2 P$ time units.
- Principal solution v and boundary conditions are combined in $(c_2/4)(M/P)N$ operations.
- $(c_1/2)(M/P)N \log_2 N$ operations are used to apply inverse Fourier transforms.

Therefore, the parallel timing T_P for the parallel fast algorithm is given by

$$T_P = \frac{MN}{P}(c_1 \log_2 N + c_2) + (2(P-1) + \log_2 P)\, t_s + N(P - 1 + \log_2 P)\, t_w. \tag{70}$$

To obtain an asymptotic estimate for the parallel timing, we drop the computational terms of lower order in (70) which leads to

$$T_P^{\text{asymp}} = c_1 \frac{MN}{P} \log_2 N + 2Pt_s + NPt_w. \tag{71}$$

The performance of the parallel algorithm can be observed by comparing the above equation against the timing estimate for the sequential algorithm. In the case of a sequential implementation, we have the following stages:

- M Fourier transfmorms are performed in $(c_1/2)MN \log_2 N$ operations.
- Radial integrals $C_n^{i,i+1}$ and $D_n^{i-1,i}$ are obtained after $(c_2/4)MN$ operations.
- Terms v^-, v^+ and v are computed in $(c_2/4)MN$ operations.
- Principal solution v and boundary conditions are combined in $(c_2/4)MN$ operations.
- M inverse Fourier transforms take $(c_1/2)MN \log_2 N$ computations.

Summarizing, the sequential timing T_s is given by

$$T_s = c_1 MN \log_2 N + \frac{3}{4} c_2 MN, \tag{72}$$

with asymptotic model

$$T_s^{\text{asymp}} = c_1 MN \log_2 N. \tag{73}$$

From equations (70) and (72) one can observe that most of the parallel overhead is attributed to the communication term in equation (70). 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. We use the asymptotic estimates to obtain the speed-up S for the parallel algorithm

$$S = \frac{T_s^{\text{asymp}}}{T_P^{\text{asymp}}} = \frac{c_1 MN \log_2 N}{c_1 \frac{MN}{P} \log_2 N + 2Pt_s + NPt_w} \tag{74}$$

$$= P \frac{c_1 MN \log_2 N}{c_1 MN \log_2 N + P^2 (2t_s + Nt_w)} \tag{75}$$

and the corresponding efficiency

$$E = \frac{S}{P} = \frac{1}{1 + P^2 (2t_s + Nt_w)/c_1 MN \log_2 N}, \tag{76}$$

which shows that the efficiency decays quadratically in the number of processors P.

Different problem sizes correspond to distinct levels of granularity, which implies that there is an optimal number of processors associated with each granularity. Since message lengths depend on N and computational work depends also on M, the theoretical model can be used to estimate the best performance for a given problem. The number of processors for which the asymptotic parallel running time T_P^{asymp} achieves its minimum is determined by $\frac{\partial T_P^{\text{asymp}}}{\partial P} = 0$. In the case of (71), we have

$$P_{\text{opt}}^{\text{asymp}} = \sqrt{\frac{c_1 MN \log_2 N}{2t_s + Nt_w}}, \tag{77}$$

which can be understood as an approximation for the optimal value of P which maximizes the efficiency (76) for given values of M and N.

3.3.3 Comparison with a Matrix Transposition-based Algorithm

Although the recursive relations in Corollary 3.2 are very appropriate to a sequential algorithm, these may introduce excessive communication on parallel implementation. The major difference is that if one attempts to evaluate recurrences (55) and (56), data must be reverted in all processors. In fact, steps 3 and 4 in Algorithm 3.1 show that each coefficient $v_n^-(r_l)$ depends on all terms $C_n^{l-1,i}$ with $i \in [2, l]$, and each coefficient $v_n^+(r_l)$ depends on all terms $D_n^{i,i+1}$ with $i \in [l, M-1]$. Consequently a message-passing mechanism must be used to exchange coefficients of the form $C_n^{i-1,i}$ and $D_n^{i,i+1}$ across processors. Figure 9 shows data being reverted in all processors for the case where $P = 4$. Initially each processor contains data for evaluating M/P Fourier transforms. It corresponds to each row on Figure 9(a). To calculate recurrences locally, 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 $C_n^{i-1,i}$ and $D_n^{i,i+1}$ with $n \in [jN/P - N/2, (j+1)N/P - N/2]$. Figure 9(b) describes the communication pattern. Rows are divided into P blocks of size NM/P^2 so that processor p_j 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 [26] is given by

$$T_{\text{comm}}^{\text{transpose}} = 2(\sqrt{P} - 1)\left(2t_s + \frac{MN}{P} t_w\right). \tag{78}$$

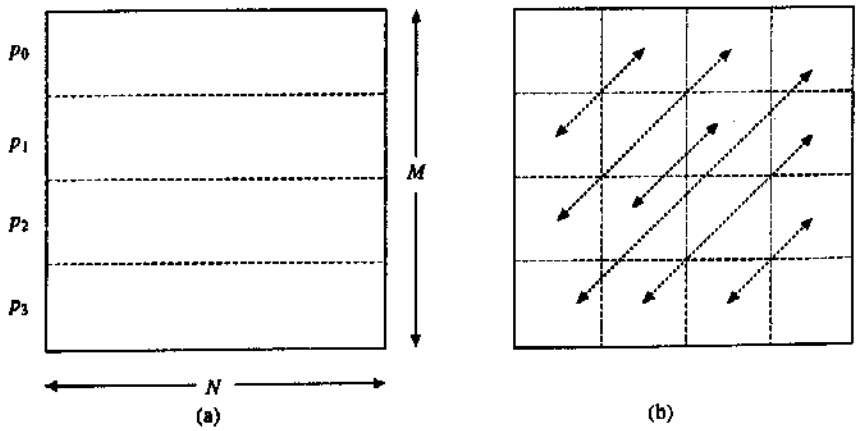

Fig. 9. Coordination pattern based on all-to-all personalized communication: (a) M/P Fourier transforms are evaluated locally; (b) each two processors exchange blocks of size MN/P^2.

Therefore, interprocessor communication introduces a delay of order $4MN/\sqrt{P}$. Comparatively, the stream-based algorithm generates a delay of order PN. In a large scale application, clearly $M >> P$ due to practical limitations on the number of available processors which makes $PN << 4MN/\sqrt{P}$. It implies that the stream-based algorithm must scale up better than the second approach because of a smaller communication overhead.

3.3.4 Comparison with Other Methods

Fourier Analysis Cyclic Reduction (FACR) solvers encompass a class of methods for the solution of Poisson's equation on regular grids [21, 40, 41]. In two-dimensional problems, one-dimensional FFTs are applied to decouple the equations into independent triangular systems. Cyclic reduction, Gaussian elimination (or another set of one-dimensional FFTs and inverse FFTs) are used to solve the linear systems. In the FACR(l) algorithm, l preliminary steps of block-cyclic reduction are performed to decrease the number or the length of the Fourier coefficients. The reduced system is solved by the FFt method and by l steps of block back-substitution. In particular, for $l = 0$ we have the basic FFT method, and $l = 1$ corresponds to a variant of the original FACR algorithm [21]. The basic idea of the FACR (l) method relies on switching to Fourier analysis in the middle of cyclic reduction to reduce the operation count when compared with either pure Fourier analysis or cyclic reduction. Formally, the optimal choice $l \sim \log_2 (\log_2 N)$ makes the asymptotic operation count for FACR(l) be $\mathcal{O}(N_2 \log_2 \log_2 N)$ in a $N \times N$ grid, which is an improvement over the estimate $\mathcal{O}(N^2 \log_2 N)$ associated with the basic FFT method (FACR(0)) and cyclic reduction.

A parallel implementation of the FACR(l) solver must take into account the effect of the choice of l on the degree of parallelism of the algorithm [41]. At $l = 0$, the method performs a set of independent sine transforms and solves a set of independent tridiagonal systems, which makes the choice $l = 0$ ideally suited for parallel computations. The parallel implementation of the matrix decomposition Poisson solver (MD-Poisson solver) presented in [37] follows this concept: a block-pentadiagonal system is solved on a ring of P processors using Gaussian elimination without pivoting, so that only neighbor-to-neighbor communication is required. The complexity of the method on a ring of P processors is $\mathcal{O}(N^2/P \log_2 N)$ if one disregards communication overhead [37]. For $l > 0$, the degree of parallelism of the FACR(l) algorithm decreases at each additional stage of cyclic reduction. For example, in [25] a parallel variant of the FACR(l) algorithm exploits the numerical properties of the tridiagonal systems generated in the method. Factorization is applied based on the convergence properties of these systems. However this approach can lead to severe load-imbalance on a distributed memory architecture because convergence rates may be different for each system resulting in idle processors. Cyclic allocation must be used to diminish load-imbalance. Moreover, it is also known from [25] that any two-dimensional data partitioning would produce communication overhead due to data transposition.

The previous observations show that our parallel Poisson solver is competitive with other current techniques. Typically, the best parallel solvers are defined using an one-dimensional processor array configuration because of the unbalanced

communication requirements for the operations performed along the different coordinates of the grid.

4. Conclusion

We have presented two parallel algorithms. Both of these parallel algorithms were derived from their sequential analogues which are presented in the above two sections. Basic idea that has been exploited in constructing these fast sequential algorithms is some recursive relations in Fourier space. We noticed these while carrying out some analysis and thought they could be profitably used for speeding up computations. Our analysis of complexity of these algorithms and numerical results presented elsewhere do bear out these facts. This idea has not been exploited much and it may be useful for constructing similar fast algorithms for other singular operators and partial differential equations. In particular, it will be worthwhile to see the feasibility of the application of this idea to Helmholtz equations (oscillatory and monotonic).

It is worthwhile to extend these from disk to sphere in three-dimensions. This should be possible. Later these algorithms should be useful for developing fast algorithms in arbitrary domains, perhaps by making using of domain embedding techniques. We have done studies in this direction which will be reported in the future.

The accuracy of the sequential algorithms presented in sections 2 and 3 may not be optimal, and there is room for further improvement, in particular with the Poisson equation if one uses some of the symmetry properties in a clever fashion and modify these algorithms suitably. We will report on this in a future publication. Lastly, it will be worthwhile to apply these algorithms to practical problems involving these singular integrals and partial differential equations. Let me close by saying that I will be glad to interact with anyone who may be interested in pursuing these ideas.

Acknowledgements

Most of the material presented here (in particular sections 2 and 3) is a condensed version of two of my papers with my colleague Leo Borges [6, 8]. This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. TARP-97010366-030.

References

1. J. Anderson, S. Amarasingle, and M.S. Lam, *Data and computation transformations for multiprocessors*, in Proc. 5th Symposium on Principles and Practice of Parallel Programming, ACM SIGPLAN, July 1995.
2. F. Argüello, M. Amor and E. Zapata, *FFTs on mesh connected computers*, Parallel Comput., 22 (1996), 19–38.
3. L. Bers, *Mathematical aspects of subcritical and transonic gas dynamics*, John Wiley, New York, 1958.

4. 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.
5. ————, *On linear and nonlinear elliptic boundary value problems in the plane*, in Convegno Internazionale Suelle Equaziono Cremeonese, Roma, 1955, pp. 141–167.
6. L. Borges and P. Daripa, *A parallel version of a fast algorithm for singular integral transforms*, Num. Algor., 23 (2000), pp. 71–96.
7. ————, *A parallel solver for singular integrals*, in Proceedings of PDPTA'99—International Conference on Parallel and Distributed Processing Techniques and Applications, vol. III, Las Vegas, Nevada, June 28–July 1, 1999, pp. 1495–1501.
8. L. Borges and P. Daripa, *A fast parallel algorithm for the Poisson equation on a disk*, J. Comp. Phys., 169, (2001), pp. 151–192 (with L. Borges).
9. W. Briggs, L. Hart, R. Sweet, and A. O'Gallagher, *Multiprocessor FFT methods*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. 27–42.
10. W. Briggs and T. Turnbull, *Fast Poisson solvers for mimd computers*, Parallel Comput., 6 (1988), pp. 265–274.
11. C. Calvin, *Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication*, Parallel Comput., 22 (1996), pp. 1255–1279.
12. T.F. Chan and D.C. Resasco, *A domain-decomposed fast Poisson solver on a rectangle*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. S14-S26.
13. R. Courant and D. Hilbert, *Methods of Mathematical Physics*, vol. II, John Wiley, New York, 1961.
14. P. Daripa, *On applications of a complex variable method in compressible flows*, J. Comput. Phys., 88 (1990), pp. 337–361.
15. ————, *A fast algorithm to solve nonhomogeneous Cauchy-Riemann equations in the complex plane*, SIAM J. Sci. Stat. Comput., 6 (1992), pp. 1418–1432.
16. ————, *A fast algorithm to solve the Beltrami equation with applications to quasiconformal mappings*, J. Comput. Phys., 106 (1993), pp. 355–365.
17. 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.
18. ————, *Singular integral transforms and fast numerical algorithms*, Num. Algor., 18 (1998), pp. 133–157.
19. M.D. Greenberg, *Application of Green's Functions in Science and Engineering*, Prentice-Hall, 1971.
20. Hewlett-Packard, *HP 9000 V-Class Server Architecture*, second edition ed., March 1998.
21. R.W. Hockney, *A fast direct solution of Poisson equation using Fourier analysis*, J. Assoc. Comput. Mach., 8 (1965), pp. 95–113.
22. R. Hockney and C. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms*, Adam Hilger, Bristol, 1981.
23. E. Houstis, R. Lynch and J. Rice, *Evaluation of numerical methods for elliptic partial differential equations*, J. Comput. Phys., 27 (1978), pp. 323–350.
24. K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, New York, NY, 1993.
25. L.S. Johnsson and N.P. Pitsianis, *Parallel computation load balance in parallel FACR*, in High Performance Algorithms for Structured Matrix Problems, P. Arbenz, M. Paprzycki, A. Sameh and V. Sarin, eds., Nova Science Publishers, Inc., 1998.

26. V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings, Redwood City, CA, 1994.
27. J. Lawrynowicz, *Quasiconformal mappings in the plane*, in Lect. Notes in Mathematics, Springer-Verlag, New York, 1983.
28. J.Y. Lee and K. Jeong, *A parallel Poisson solver using the fast multipole method on networks of workstations*, Comput. Math. Appl., 36 (1998), pp. 47–61.
29. C.V. Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.
30. A. McKenney, L. Greengard and A. Mayo, *A fast Poisson solver for complex geometries*, J. Comput. Phys., 118 (1995), pp. 348–355.
31. C. Morrey, *On the solutions of quasi-linear elliptic differential equations*, Trans. Amer. Math. Soc., 43 (1938), pp. 126–166.
32. A. Mshimba and W. Tutschke, *Functional Analytic Methods in Complex Analysis and Applications to Partial Differential Equations*, World Scientific Publishing Co., Pte. Ltd., Singapore, 1990.
33. L. Nirenberg, *On nonlinear elliptic differential equations and Holder continuity*, Comm. Pure Appl. Math., 6 (1953), pp. 103–156.
34. P. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann, San Francisco, CA, 1997.
35. D. Patterson and J. Hennessy, *Computer Organization and Design: the hardwaresolftware interface*, Morgan Kaufmann, San Francisco, CA, 1994.
36. J. Rice, E. Houstis and R. Dyksen, *A population of linear, second order, elliptic partial differential equations on rectangular domains. I, II.*, Math. Comput., 36 (1981), pp. 475–484.
37. A. Sameh, *A fast Poisson solver for multiprocessors*, in Elliptic Problem Solvers II, G. Birkhoff and A. Schoenstadt, eds., Academic Press, Orlando, 1984, pp. 175–186.
38. J. Singh, W. Weber and A. Gupta, *Splash: Stanford parallel applications for shared-memory*, Comput. Arch. News, 20 (1992), pp. 5–44.
39. G. Sköllermo, *A Fourier method for the numerical solution of Poisson' equation*, Math. Comput., 29 (1975), pp. 697–711.
40. P.N. Swarztrauber, *The methods of cyclic reduction, Fourier analysis and the FACR algorithm for the discrete solution of Poisson's equation on a rectangle*, SIAM Review, 19 (1977), pp. 491–501.
41. C. Temperton, *On the FACR(l) algorithm for the discrete Poisson equation*, J. Comput. Phys., 34 (1980), pp. 315–329.