from *SIAM News*, Volume 32, Number 1

Parallel Implementation of the Split-step Fourier Method For Solving Nonlinear Schrödinger Systems

By Scott Zoldi, Victor Ruban, Alexandre Zenchuk, and Sergey Burtsev

Large-scale simulations of the nonlinear Schrödinger equation (NLSE) are required in the solution of many problems in fiber optics—among them accurate modeling of wavelength division multiplexed transmission systems. The split-step Fourier (SSF) method, commonly used in the numerical solution of the NLSE, often proves too slow in serial versions, even on the fastest workstations. In this article, we present a paral-lelization of the SSF algorithm that is appropriate for multiprocessors.

APPLICATIONS ON ADVANCED ARCHITECTURE COMPUTERS
Greg Astfalk, Editor

Most commercial multiprocessors support both shared-memory and distributed-memory programming paradigms. As described here, we have implemented the SSF method under both of these parallel programming paradigms on a four-processor Silicon Graphics/Cray Research Origin 200 multiprocessor computer.

The NLSE,

$$iA_t + \sigma d \frac{1}{2} A_{xx} + A^* A^2 = G \quad (1)$$

is a nonlinear partial differential equation that describes wave packet propagation in a medium with cubic non-linearity. *A* is the amplitude of the electric field corresponding to the wave packet. The parameter σ specifies the fiber-anomalous group velocity dispersion (for $\sigma = 1$) or the normal group velocity dispersion (for $\sigma = -1$). The parameter *d* defines the normalized absolute value of the fiber's dispersion. The perturbation *G* is specified by details of the physical fiber being studied. Technologically, the most important application of the NLSE is in the field of nonlinear fiber optics [1, 6].

In the special case of $G = 0$, the NLSE is integrable [12] and can be solved analytically. More generally, when $G \neq 0$, the NLSE must be solved numerically. The SSF method is one of the most popular numerical methods for solving the perturbed NLSE [1].

In wavelength division multiplexed (WDM) transmission systems, many optical channels operating at their own frequencies share the same optical fiber. WDM technology is important in that it is one of the most effective ways to increase the transmission capacity of optical lines [1, 6, 7]. For accurate WDM modeling, a large number of Fourier harmonics must be included in the numerical simulation to cover the entire transmission frequency band.

In WDM systems, moreover, different channel pulses propagate at different velocities and, as a result, collide with each other. At the pulse collision, Stokes and anti-Stokes sidebands are generated. These high-frequency perturbations lead to signal deterioration [3, 8]. Another fundamental nonlinear effect, called four-wave mixing (FWM) [4], must be accurately simulated: The FWM components broaden the frequency domain, and even larger numbers of Fourier modes are required for accurate numerical simulation.

Dispersion management (concatenation of fiber links with variable dispersion characteristics) can be used to suppress FWM and make possible the practical realization of WDM transmission. In this case the dispersion coefficient *d* in the NLSE is no longer constant but rep-resents a rapidly varying piecewise constant function of the distance along the fiber. As a result, small step sizes along the fiber are required to resolve dispersion variations and the corresponding pulse dynamics. A final reason to include large numbers of Fourier modes in numerical simulations is to model the propagation of pseudorandom data streams over large distances. All these factors combine to make numerical simulation of the NLSE quite computa-tionally intensive.

Split-step Fourier Method

The SSF method is used to integrate many types of nonlinear partial differential equations. Because it is often more efficient than finite differences for the simulation of nonlinear Schrödinger (NLS) systems [11], the SSF method is the more commonly used.

An NLS system can be written in the form:

$$\frac{\partial A}{\partial t} = (L + N)A$$

where *L* and *N* are the linear and nonlinear parts of the equation. The solution over a short time interval τ can be written in the form

$$A(t+\tau, x) = \exp(\tau L)\exp(\tau N)A(t, x)$$

where the linear operator in the NLS system acting on a spatial field *B*(*t*,*x*) is written in Fourier space as

$$\exp(\tau L)B(t, x) = F^{-1}\exp\left(-ik^2\tau\right)F\,B(t, x)$$

where F denotes the Fourier transform (FT), F^{-1} denotes the inverse Fourier transform, and k is the spatial frequency.
We split the computation of A over time interval τ into four steps:

Step1(*nonlinearstep*): Compute $A_1 = \exp(\text{t } N)A(t, x)$ (by finite differences).

Step 2 (*forward FT*): Perform the forward FT on A_1: $A_2 = F\, A_1$.

Step3 (*linearstep*):Compute $A_3 = \exp(\text{t } L)A_2$.

Step 4 (*backward FT*): Perform the backward FT on A_3: $A(t + \text{t}) = F^{-1} A_3$.

For numerical approximation of this algorithm, the potential A is discretized in the form $A_l = A(lh)$ for $l = 0, \ldots, N - 1$, where h is the space-step and N is the total number of spatial mesh points.
This algorithm for the SSF method is the same for sequential and parallel code. Parallel implementation of the algorithm involves parallelizing each of the four steps.

Parallel Algorithm for the SSF Method

A prerequisite for parallel numerical algorithms is that sufficient independent computation be identified for each processor and that only small amounts of data be communicated between periods of independent computation. This can occasionally be done trivially through loop-level parallelism (for shared-memory implementations), or nontrivial-ly by storing independent data in each processor's local memory. For example, the nonlinear step in the SSF algorithm involves independent computation over portions of the spatial elements of A_l. Each of p processors will work on subarrays of the field A, e.g., the first processor updates A_0 to $A_{(N/p-1)}$, the second processor updates $A_{N/p}$ to $A_{2(N/p)-1}$, and so forth.
In the 1D FFT, elements of $(F\,A)_k$ cannot be computed in a straightforward parallel manner because all the elements of A_l are used in the construction of any element of $(F\,A)_k$. The problem of parallelizing the 1D FFT has been of great interest for vector [2, 10] and distributed-memory computers [5]. These algorithms are architecture-dependent and involve efficient methods for the data-rearrangement and transposition phases of the 1D FFT.
Communication issues are paramount in parallelizing the 1D FFT. In the past the classic butterfly communication patterns were exploited to lessen communication costs [5]. Rapid changes in parallel architectures, however, have resulted in multiprocessor systems with highly complex memory hierarchies and communication characteristics; previous algorithms are not directly applicable to many of the current multiprocessor systems. Shared-memory multiprocessors often have efficient interprocessor communication, and we therefore implement the parallel 1D FFT by writing A_l as a 2D array, in which we can identify independent serial 1D FFTs of rows and columns of this matrix. The rows and columns of the matrix A can be distributed to divide the computation among several processors. As a result of the efficient communication, the independent computation stages, and the absence of the transposition stage of the 1D FFT in SSF computations, this method exploits enough independent computation to result in significant speedups with a small number of processors.
The difficulty of parallelizing the SSF algorithm arises in Steps 2 and 4. The other two steps can be trivially evolved be-cause of the natural independence of the data in A and A_2. In Steps 2 and 4, there are nontrivial data dependences over the entire range $0 \le l \le N$ of $A_1(l)$ and $A_3(l)$, which involve forward and backward Fou-rier transforms (FFT and BFT, respectively). The discrete FFT is of the form

$$F(k) = \sum_{l=0}^{N-1} A(l) \exp\left(-\frac{2\pi i l k}{N}\right)$$

which requires all elements of $A(l)$. Parallel 1D FFT algorithms must deal with the memory hierarchy and communication issues in order to achieve good speedups. We achieve significant parallel speedup due to the elimination of the transposition stage in the 1D FFT for the SSF method by exploiting the independent computations achieved by performing many sequential 1D FFTs on small subarrays of $A(l)$.
Our method for parallelizing the SSF algorithm requires dividing the 1D array $A(l)$ into subarrays, which are processed via vendor-optimized sequential 1D FFT routines. We assume that the dimension of the 1D array A is the product of two in-tegers: $N = M_0 \times M_1$. A can then be written as a 2D $M_0 \times M_1$ matrix. As a result, we can reduce the expression for the Fourier transform of the array A to the form
where F is the Fourier transform of A and f is the result of an M_1-size Fourier transform of $A(M_0 n_1 + n_0)$ with fixed n_0:
The reduced expression, equation (2), demonstrates that the $N = M_0 \times M_1$ Fourier transform F is obtained by performing M_0-size Fourier transforms of $f(k_0, n_0)\exp\left(\frac{-2\pi i}{N} n_0 k_0\right)$ for fixed k_0. Therefore, the 1D array A is written as a 2D matrix a_{jk} of size $M_0 \times M_1$ with elements $(A(0), \ldots, A(M_0 - 1))$ in the first column, $(A(M_0), \ldots, A(2M_0 - 1))$ in the sec-ond column, and so forth. We use this matrix a_{jk} in our parallel FFT algorithm:

Step a: Perform independent M_1-size FFTs on the rows of a_{jk}.

Step b: Multiply elements $a(j, k)$ by a factor $E_{jk} = \exp(-(2\text{pi}/N) \cdot j \cdot k)$.

Step c: Perform independent M_0-size FFTs on the columns of a_{jk}.

The result of Steps a through c is the $N = M_0 \times M_1$ 1D Fourier transform of A stored in rows: $(F(0), \ldots, F(M_1 - 1))$ in the first row, $(F(M_1), \ldots, F(2M_1 - 1))$ in the second row, and so on. Restoration of the proper ordering of A (the elements as originally stored in the matrix a_{jk}) requires a transposition of the matrix, which is the last step in a parallel FFT algorithm.

The transposition is *not* necessary in the SSF method. We avoid the transposition by defining a transposed linear operator array and multiplying a_{jk} by this operator. Steps a through c are then performed in reverse order, with the conjugate of the exponential term used in Step b.

The complete SSF parallel algorithm consists of the following steps:

```
Step A: nonlinear step
Step B: row-FFT
Step C: multiplication by $E$
Step D: column-FFT
Step E: linear step (transposed linear operator)
Step F: column-BFT
Step G: multiplication by $E^*$ (the complex conjugate of $E$)
Step H: row-BFT
```

The parallelism is due to the independent operations in steps A, C, E, and G and to the row and column subarray FFTs in steps B, D, F, and H. The row and column subarray FFTs of size M_1 and M_0 are performed independently with optimized serial 1D FFT routines. Working with subarray data, many processors can be used to divide the computational work; significant speedups are achieved if communication between processors is efficient. The smaller subarrays allow for better data locality in the primary and secondary processor caches.

Shared-memory Approach

Most of the parallel SSF algorithm can be implemented with **`doacross`** directives to distribute independent loop iterations over many processors. We im-plement the FFTs of size M_0 and M_1 by distributing the 1D subarray FFTs of rows and columns over the p processors. The performance can be significantly improved by keeping the same rows and columns local in a processor's secondary cache to alleviate true sharing of data from dynamic assignments of subarray FFTs by the **`doacross`** directive. Vendor-optimized sequential 1D FFT routines designed specifically for the architecture are used to perform the subarray FFTs.

It is "cache efficient" to perform all column operations, Steps C through G,in one pass. We copy a column into a local subarray S contained in the processor's cache and then, in order, perform the following steps: multiply by the exponents in Step C, perform the M_0-size FFT of S, multiply by the transposed linear operator $\exp(\tau L)$, invert the M_0-size FFT, multiply by the conjugate exponents, and, finally, store S in the same column of A.

Distributed-memory Approach

The Message Passing Interface (MPI) [9] is a tool for distributed parallelcomputing that has become an ad hoc standard. It is used on systems ranging from tightly coupled high-end parallel computers to weakly coupled distributed networks of workstations (NOW) [9]. In distributed parallel programming, different processors work on completely independent data and explicitly use send and receive library calls to communicate data between processors.

To implement the distributed parallel SSF algorithm for the NLS, we need to distribute the rows of array A among p processors. (The algorithm is shown below.) Steps 1 through 3 can then be executed without interprocessor communication.

It is necessary to redistribute the elements of A among the p processors, which incurs communication costs. Each processor must send a fraction $1/p$ of its data to each of the other processors. Each processor will then have the correct data for Steps 4 through 7, and column operations can be performed independently on all p processors. Finally, there is a second redistribution prior to Step 8.

To take T steps of the SSF algorithm, we use the following scheme (in which the step numbers are not related to those in the earlier algorithm):

```
distribute rows among processors
Step 1: nonlinear step
Step 2: row-FFT
Step 3: multiplication by a factor $E$

for $i = 1$ to $T - 1$ do
    data redistribution
    Step 4: column-FFT
    Step 5: linear step
    Step 6: column-BFT
    Step 7: multiplication by a factor $E^*$
    data redistribution
    Step 8: row-{BFT}
    Step 1: nonlinear step
    Step 2: row-FFT
    Step 3: multiplication by a factor $E$
end do
data redistribution
Step 4: column-FFT
Step 5: linear step
Step 6: column-BFT
Step 7: multiplication by a factor $E^*$
Step 8: row-BFT
```

The large performance cost in this algorithm is the redistribution of data between row and column operations. If the row and column computational stages result in significant speedup compared with the communication expense incurred in redistributing the matrix data, then the algorithm will be successful. The success of the algorithm is crucially dependent on fast communication between processors. This is the case for shared-memory multiprocessors, and less so for NOW-like systems.

	$N = 2^{12}$ $T = 8000$	$N = 2^{14}$ $T = 2000$	$N = 2^{16}$ $T = 500$	$N = 2^{18}$ $T = 125$
$t_{(1)}$ (sec)	49.5	51.5	65.5	97.5
$t_{(2)}$ (sec)	29.5	30.5	33.5	61.0
$t_{(4)}$ (sec)	19.5	18.5	19.5	34.5
$SU = t_{(1)}/t_{(2)}$	1.7	1.7	2.0	1.6
$SU = t_{(1)}/t_{(4)}$	2.5	2.8	3.4	2.8

Table 1. *Results for a shared-memory implementation. N denotes array size, T the number of steps, $t_{(x)}$ the time on x processors, and SU the speedup.*

	$N = 2^{12}$ $T = 8000$	$N = 2^{14}$ $T = 2000$	$N = 2^{16}$ $T = 500$	$N = 2^{18}$ $T = 125$
$t_{(1)}$ (sec)	37.9	44.5	59.4	92.4
$t_{(2)}$ (sec)	24.7	25.4	34.9	65.9
$t_{(4)}$ (sec)	18.8	16.3	20.1	26.8
$SU = t_{(1)}/t_{(2)}$	1.5	1.8	1.7	1.4
$SU = t_{(1)}/t_{(4)}$	2.0	2.7	3.0	3.4

Table 2. *Results for distributed-memory and MPI implementations. N indicates array size, T the number of steps, $t_{(x)}$ the time on x processors, and SU the speedup.*

Results

We performed timings of the parallel SSF algorithm on the Silicon Graphics/Cray Research Origin 200 multiprocessor. The Origin 200 allows both shared- and distributed-memory parallel programming. It can be considered a generic multiprocessor, and it is efficient for fine-grained parallelism. The Origin 200 used in this study consisted of four MIPS R10000 64-bit processors with (peak) performance of 360 Mflops each. All timings are for the total wall-clock time for the code, including both initialization and execution.

For the following timings, $M_0 = M_1 = 2^K$, so that the entire 1D array is of size $N = 2^{2K}$. The one-processor implementation of our parallel SSF code was 10% to 20% faster than that of the serial SSF code using vendor-optimized 1D FFTs on the entire array of size $N = 2^{2K}$. The improvement is due to better cache utilization with the smaller subarrays; an entire subarray could be contained in the L1 cache. Additionally, the single-processor parallel SSF code does not do the transposition stage of the 1D FFT. All timings are compared with those for the one-processor parallel code at the same optimization level.

For shared-memory parallel implementations, we find that, for the range of $2^{12} < N < 2^{18}$, two-node SSF implementations have good speedup, with a maximum at $N = 2^{16}$ (See Table 1). With four processors and small array sizes the work per processor decreases by 25%, but contention is greater because of the sharing of pieces of data contained in the secondary caches of four different processors. The maximum speedup again occurs at $N = 2^{16}$, indicating that the ratio of computational speed gain to communication contention is optimal at this problem size.

Under the shared-memory programming model, subarrays are continually distributed among processors to divide the computational work. Data in a single subarray can be contained on one or more processors. The data contained in each processor's L2 cache is of size $O(N/p)$, where p is the number of processors. In the parallel code, the communication time for sending data between processors is proportional to $O(N/p)\tau_c$, where τ_c is the time required to transfer an L2 cache block between processors. Finally, the time required to perform a sequential 1D FFT on a subarray of size M is approximately $M \log(M)\tau_M$, where τ_M is the time for a floating-point operation.

For fixed p and with no contention, we can predict that the speedup will increase as the problem size N becomes larger. With contention, however, the speedup eventually decreases with larg-er N as a result of the sharing of subar-ray data between processors. These observations are in agreement with the qualitative trend seen in our empirical data on speedup for the shared-memory SSF method, where the maximum speedup was attained with a problem size of $N = 2^{16}$.

The preceding discussion of speedup must be reinterpreted for the distributed-memory SSF method owing to the implicit independent computational stages during which no data are communicated between processors. With the distributed-memory SSF method, because it uses communication stages to send data between processors, contention due to sharing of data between processors does not occur during the computation stages.

Distributed MPI timings are compared with those for a one-pro-cessor MPI code at the same optimization level. Because it did not have synchronization steps, the MPI one-processor code was faster than one-processor shared-mem-ory code. The parallel code was usually faster than the shared-memory parallel code, except for the $N = 2^{16}$ array size, for which the shared-mem-ory code did slightly better. We find that for distributed-memory parallel implementations of the SSF method over the range of $2^{12} < N < 2^{18}$, two-node implementations have good speedups, with a maximum at $N = 2^{14}$, beyond which the communication cost increases and the computation/communication ratio decreases (see Table 2). The communication cost for the MPI code is a result of "communication stages"; the less than perfect speedups are thus due to the volume of data communicated between processors in redistribution stages, and not to the constant sharing of small subarray data.

With four processors, the speedup increased with the working set size *N*. This is due both to the faster computation stages and to the reduced volume of data communicated between single processors in the redistribution stage. For small problem sizes, there is not enough work to make dividing the problem among four processors beneficial. The speedup in the distributed-memory SSF algorithm can be attributed to the independence of the data contained in a processor's local cache between data-rearrangement stages.

Conclusions

These results are encouraging in that the speedups achieved in multiprocessor SSF implementations are considerable. The speedups over sequential code with vendor-optimized full-array 1D FFTs are even greater. Because of the 10% to 20% speedup over optimized 1D sequential SSF algorithms, we recommend implementation of the parallel SSF algorithm even on sequential machines. The speedup reflects better exploitation of the L1 cache and data locality through the use of small subarrays and removal of the transposition stage of the 1D FFT in SSF.

For shared-memory implementations of the parallel SSF method, the maximum speedup is achieved through a balance of contention in communicating data contained in more than one processor and the computational performance gains achieved by means of small subarrays. For the distributed parallel SSF method, data locality is greater as data are distributed statically prior to the computational stages. This division of computation and communication stages is different from that for the shared-memory SSF method, which dynamically distributes subarray FFTs and shares data on more than one processor. Distributed SSF speedup is a function of the number of processors *p*, which reduces both the computation time and the communication volume between single processors. The speedup of the parallel SSF method is strongly dependent on reductions in both communication time and contention in the multiprocessor.

References

[1] G.P. Agrawal, *Nonlinear Fiber Optics*, 2nd edition, Academic Press, San Diego, 1995.

[2] A. Averbuch, E. Gabber, B. Gordissky, and Y. Medan, *A Parallel FFT on an MIMD machine*, Parallel Comput., 15 (1990), 61–74.

[3] N.S. Bergano and C.R. Davidson, *Wavelength division multiplexing in long-haul transmission systems,* J. Lightwave Tech., 14:6 (1996), 1299–1308.

[4] P.N. Butcher and D. Cotter, *The Elements of Nonlinear Optics*, Cambridge University Press, New York, 1990.

[5] A. Dubey, M. Zubair, and C.E. Grosch, *A general purpose subroutine for fast Fourier transform on a distributed memory parallel machine*, Parallel Comput., 20 (1994), 1697–1710.

[6] A. Hasegawa and Y. Kodama, *Solitons in Optical Communication*, Oxford University Press, New York, 1995.

[7] I.P. Kaminow and T.L. Koch, *Optical Fiber Telecommunications III*, Academic Press, San Diego, 1997.

[8] P.V. Mamyshev and L.F. Mollenauer, *Pseudo-phase-matched four-wave mixing in soliton wavelength-division multiplexing transmission,* Optics Lett., 21:6 (1996), 396–398.

[9] M. Snir, S. Otto, S.H. Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference,* MIT Press, London, 1996.

[10] P.N. Swartztrauber, *Multiprocessor FFTs*, Parallel Comput., 5 (1987), 197–210.

[11] T.R. Taha and M.J. Ablowitz, *Analytical and numerical aspects of certain nonlinear evolution equations, II, Numerical, nonlinear Schrödinger equation*, J. Comput. Phys., 55 (1984), 203–230.

[12] V.E. Zakharov and A.B. Shabat, *Exact theory of two-dimensional self-focusing and one-dimensional self-modulation of waves in nonlinear media,* Soviet Phys. JETP, 34 (1972), 62–69.

Scott Zoldi (zoldi@cnls.lanl.gov) is a post-doctoral research associate at the Los Alamos National Laboratory. V. Ruban (ruban@itp.ac.ru) and A. Zenchuk (zenchuk@itp.ac.ru) are graduate students at the L.D. Landau Institute for Theoretical Physics in Moscow, Russia. S. Burtsev (burtsev@cnls.lanl.gov) is a research scientist at Corning Inc.