Concurrent computing by sequential staging of tasks

by J. Gazdag
H.-H. Wang

Described is a new approach to parallel formulation of scientific problems on shared-memory multiprocessors such as the IBM ES/3090 system. The class of problems considered is characterized by repetitive operations applied over the computational domain D. In each such operation, some fields of interest are extrapolated or advanced by an amount of $\Delta\tau$. The integration variable τ may be time, distance, or iteration sequence number, depending on the problem under consideration. An extensively studied approach to parallel formulation of such computational problems is based on domain decomposition, which attempts to partition the domain of integration into many pieces, then construct the global solution from these local solutions. Thus, domain decomposition methods are confined to D alone at a single τ level. An inquiry into the possibilities of formulating parallel tasks in τ, or more significantly in the $D \times \tau$ domain, opens up new horizons and untapped opportunities. The aim of this paper is to detail an approach to exploit this τ domain parallelism that will be referred to as sequential staging of tasks (SST). Concurrency is realized by means of ordering the tasks sequentially and executing them in a partially overlapped or pipelined manner. The SST approach can yield remarkable speedup for jobs requiring intensive paging I/O, even when a single processor is available for executing multiple tasks. Noteworthy features of the SST method are demonstrated and highlighted by using results obtained from computer experiments performed with a numerical solution method of the Poisson equation and migration of seismic reflection data.

Over the past four decades the computer industry has experienced phenomenal growth. The performance of scientific computers has increased by at least five orders of magnitude. These improvements can be attributed to advancements in technology, improvements in machine organization, and the developments of reliable SIMD (single-instruction multiple data) extensions, such as the pipelined vector processors.

It is generally believed that single-processor performance is rapidly reaching its limits, and increase in performance by orders of magnitude can only come from further exploitation of the inherent parallelism in applications.[1] Consequently, over the past decade there has been increasing interest in parallel computing. In the context of this paper, the terms *parallel computing* or *concurrent computing* will signify the use of a number of processors working cooperatively on a single problem, e.g., a single FORTRAN job. Computing systems consisting of processors that are capable of working together by executing separate sets of instructions asynchronously are known as MIMD (multiple-instruction multiple data) architectures. It is not necessary that these processors be dedicated to the same problem or job, since one can show that load balancing on MIMD systems is simplified enormously when multiple parallel jobs are executed over multiple processors. MIMD architectures can be put into many different classes depending on one's objective.[2] The ideas discussed and the work reported in this paper have been inspired by and

implemented on the IBM ES/3090 system, a shared-memory multiprocessor with vector facilities (VF) on each of the processors.

To obtain near-optimum performance from a computer, algorithms may be tailored to the architecture of the computer under consideration, which is also true for the traditional von Neumann central processors. As a consequence of the complexity of algorithms, effective exploration of parallel vector architectures has proven more difficult than that of serial architectures. The degree of effort required to implement or enable a given program on any computer is a very important factor that can determine the acceptance, and eventually the commercial success, of the computer under consideration. For convenience, this will be referred to as *implementation complexity.*[1]

On current shared-memory MIMD systems, development of parallel computing methods is motivated almost exclusively by the desire to improve turnaround time of a single job. It has been observed that the size and complexity of a problem is determined by the turnaround the user is willing to tolerate. In turn, the user is only willing to accept a limited amount of implementation complexity in order to improve the turnaround time. Thus it appears that the perceived cost associated with the implementation complexity compared with the benefit of shorter turnaround time is a key factor in deciding for or against parallel implementation. The approach to concurrent computing discussed in this paper has been motivated by the desire to achieve a high degree of parallelism with relatively low implementation complexity. The problems under consideration include time-dependent partial differential equations and iterative methods for solving large systems of equations.

The method presented has an important characteristic of recognizing that in solving time-dependent simulation problems, the processors may be assigned tasks representing work at different time levels, and the work need not be divided within a single time level. Similarly an iterative method may assign processors tasks representing work for different iteration sequence numbers and need not divide the work within a single iteration. By defining a task as one or more time steps or iterations over the computational domain, one can initiate a set of tasks in a sequence displaced from each other. Each task represents all computing to be done at a given physical time or iteration sequence number. Since no more than one task is assigned to a processor at any given time, one can visualize the scheme as a number of processors sweeping over the computational domain, moving in unison, and processing successive time levels in a sequentially-ordered but partially overlapped or *pipelined* manner. This pipelined approach to concurrent processing of macro tasks will be referred to as *sequential staging of tasks* (SST).

The organization of the paper is as follows. First, the relevant solution methods for partial differential equations are reviewed. Next, the basic principles of the SST method and programming considerations for its practical implementation are discussed, followed by numerical examples taken from electrostatics and exploration geophysics. The advantages of the SST method are then explained in the light of the simulation results. The paper concludes with a reflection and speculation on future trends from the end user's point of view.

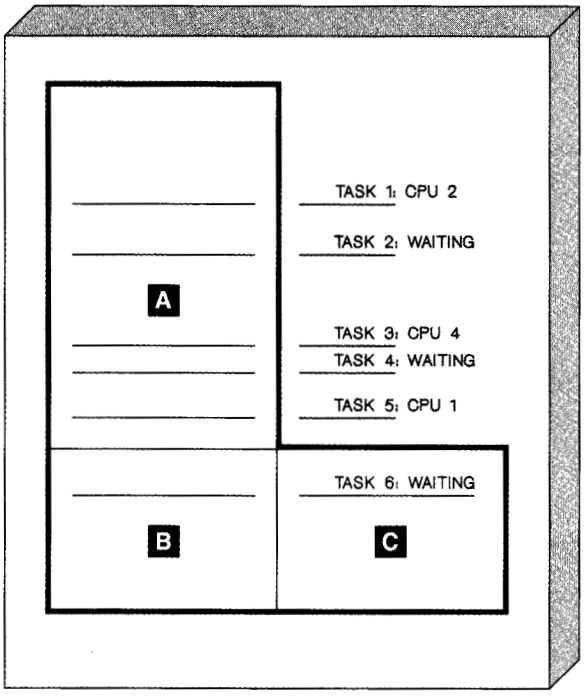
Domain decomposition for exploiting spatial parallelism

An important class of numerical solution methods calls for repeated application of algorithms over large multidimensional data arrays. A sweep over the data usually corresponds to a numerical integration whereby some physical fields are extrapolated with respect to a variable τ, the physical significance of which depends on the problem under consideration. Parallel formulation of such computational problems requires the creation of multiple tasks that can be executed concurrently in two distinct ways: (1) Partition the work at one τ level, i.e., within one sweep, into multiple tasks associated with subsets of the computational domain. (2) Define any task as the work to be done at one or more τ levels, thereby partitioning the total work with respect to τ.

The first approach is known as domain decomposition and is described below. The second corresponds to the SST method described in the latter part of this paper.

The most popular method of parallel programming is known as *domain decomposition.*[3,4] The computational domain D is partitioned into some n subdomains D_j, $j = 1, ..., n$. A computational task T defined over D is also partitioned into n subtasks T_j, $j = 1, ..., n$, corresponding to the n subdomains. Parallel (concurrent) processing is accomplished by assigning N_t subtasks to N_p available processors. The concept is illustrated on one of the simplest, but

Figure 1 Illustration of an ordered sequence of tasks propagating from the bottom to the top of the L-shaped domain. At the instant of this observation, three tasks are being actually processed.



widely used and frequently referenced, equations of mathematical physics, the Poisson equation. Under the simplifying assumption that the physical domain of interest is homogeneous and isotropic, the two-dimensional Poisson equation may be written as

$$\frac{\partial^2 \Phi(x,y)}{\partial x^2} + \frac{\partial^2 \Phi(x,y)}{\partial y^2} = \rho(x,y). \qquad (1)$$

For simple boundary conditions, e.g., a rectangular domain, direct solvers are available (for example, Hockney, p. 534).[2] On the other hand, even for slightly irregular boundaries such as the L-shaped domain D_L shown in Figure 1, no simple direct solvers are available. To take advantage of simple and well-tested programs, one is tempted to decompose the domain into two or more rectangular domains to which direct solvers can be applied readily. Since D_L is the union of three rectangular regions, A, B, C, one plausible approach is to work with two domains A and $B \cup C$, or alternatively, with domains $A \cup B$ and C. The procedure steps follow:

1. Obtain an approximate solution for Equation 1 on A.
2. Obtain an approximate solution for Equation 1 on $B \cup C$.
3. Compare the results on the boundary between A and B.
4. Make corrections in boundary conditions aimed at obtaining matching Φ values along this boundary.

Steps 1 to 4 are repeated until the solutions along the interfaces are in agreement within some prescribed tolerance. Two tasks, e.g., T_A and $T_{B\cup C}$, of finding an approximation to Equation 1 can be performed independently, and therefore may be assigned to two different processing units, P_1 and P_2. There are, however, serious problems with such a domain decomposition approach, some of which are listed below.

- Load balancing: T_A and $T_{B\cup C}$ may require different amounts of computing, causing P_1 and P_2 to wait for each other.
- Matching Φ along the interface between subregions is additional overhead of programming and computing.
- The accommodation of a different number of processors would require recoding.

The concerns about load balancing, i.e., overall system utilization and implementation complexity, can increase dramatically when one graduates from textbook problems to those with realistic size and difficulty. The reluctance of the scientific programmer to accept the implementation complexity associated with this kind of approach to parallel computing is one important reason why some observers think that parallel computing is not driven by the user.[5] Neither the computer architect who designs complex parallel systems, nor the theoretical numerical analyst whose job ends with a proof that the solution exists, has demonstrated much appreciation for the implementation complexity that programmers must address to improve turnaround time.

Iterative methods. Fortunately, a significant portion of engineering and scientific problems is governed by linear partial differential equations (PDE), such as Equation 1, and may be solved numerically by one of the several well-tested iterative procedures. A brief overview of these procedures will be given with reference to Equation 1, a numerical approximation of which on a rectangular mesh can be expressed as a difference equation:

$$\phi_{k,l-1} + \phi_{k,l+1} + \phi_{k-1,l} + \phi_{k+1,l} - 4\,\phi_{k,l} = \rho_{k,l}. \qquad (2)$$

Iterative procedures are defined by starting with a guess $\phi^0_{k,l}$ at all mesh points. Improved values $\phi^n_{k,l}$ are calculated by using Equation 2. The superscript n signifies the results from the nth iteration. The process is repeated N times until $\phi^N_{k,l}$ converges to the solution of Equation 2 at all mesh points.

In the *Jacobi method*, new values $\phi^{n+1}_{k,l}$ are computed from old values $\phi^n_{k,l}$ of the last iteration. This can be stated more formally as:

$$\phi^{n+1}_{k,l} = \frac{\phi^n_{k,l-1} + \phi^n_{k,l+1} + \phi^n_{k-1,l} + \phi^n_{k+1,l} - \rho_{k,l}}{4}. \quad (3)$$

This equation can be evaluated for all mesh points independently from the others, which makes it ideally suited to implementation on parallel computers. Unfortunately, it has a slow convergence rate, and therefore it is useless for practical computations.

A more successful iterative method is the *SOR method* (successive over-relaxation). It is based on the following replacement algorithm:

$$\phi^{n+1}_{k,l} = \omega \frac{\phi^{n+1}_{k,l-1} + \phi^n_{k,l+1} + \phi^{n+1}_{k-1,l} + \phi^n_{k+1,l} - \rho_{k,l}}{4} + (1 - \omega)\phi^n_{k,l}, \quad (4)$$

where ω is a constant relaxation factor assuming values within the range of $\{1 \leq \omega \leq 2\}$. The convergence rate, which is a function of ω, is significantly better than that of Equation 3. This improvement is due to the fact that new values replace old ones as soon as they become available. For that very same reason, the algorithm has recurrences with respect to both indices k and l, and considering a single iteration, i.e., one sweep over the data arrays under consideration, the SOR method appears sequential and unsuitable for implementation on vector or parallel computers.

If the evaluation of Equation 4 is ordered according to the classical Red/Black ordering of the mesh points, then an SOR sweep can be substituted by two Jacobi-like sweeps of the mesh. The two sweeps correspond to computing the new values of $\phi^{n+1}_{k,l}$ at red and black mesh points. This procedure is also known as Odd/Even partitioning, since the mesh is partitioned into two groups, red or black, according to whether $k + l$ is odd or even, respectively. The SOR method based on Odd/Even ordering can be effectively implemented on vector or parallel computers. This strategy is limited to finite difference discretizations involving five points (in two dimensions). For higher-order finite difference discretizations or for equations involving mixed partial derivatives, one needs more than two colors to implement the SOR method on vector or parallel computers.[6]

Consider a method similar to Equation 4 but with the following changes. The sweeps are done in row order and intermediate $\tilde{\phi}$ values are computed simultaneously for all mesh points of the kth row from using the new values of the $(k - 1)$th row and the old value of the $(k + 1)$th row. This can be expressed for an interior mesh point as:

$$\tilde{\phi}_{k,l} = 0.25\,(\tilde{\phi}_{k,l-1} + \tilde{\phi}_{k,l+1} + \phi^{n+1}_{k-1,l} + \phi^n_{k+1,l} - \rho_{k,l}). \quad (5)$$

When applied to all mesh points along a row, Equation 5 forms a tridiagonal system of equations, the solution of which yields $\tilde{\phi}$. The new ϕ^{n+1} values are computed from the old ϕ^n values and those obtained from Equation 5 by implementing

$$\phi^{n+1}_{k,l} = \omega\tilde{\phi}_{k,l} + (1 - \omega)\phi^n_{k,l}. \quad (6)$$

Solving for new values at successive rows defines an iteration step of the *SLOR method* (successive line over-relaxation). The SLOR method, which reduces to repeated application of a tridiagonal solver (the possibility of parallel execution of which is not expected to be of great benefit and therefore is ignored), is essentially a sequential algorithm. Red/Black partitioning of lines can help to remove recurrence relations and render the modified SLOR method suitable for parallel computers.

Creative reordering of the sequence of computations can result in efficient parallel algorithms.

In summary, the most effective iterative algorithms do not always lend themselves conveniently to parallel architectures. Creative reordering of the sequence of computations can result in efficient parallel algorithms, although at some added implementation complexity. There is, of course, always a

chance for surprises, such as an I/O-bound (e.g., paging in a virtual-system environment) job becoming even more I/O bound if the standard SOR method is traded for the Odd/Even SOR. The turnaround time would roughly double regardless of how many processors are being used, since the two sweeps over ϕ on each iteration double the number of pages transferred. The next section discusses a strategy for adapting iterative schemes to shared-memory parallel computers based on the sequential staging of tasks. The SST method has a low implementation complexity, does not depend strongly on the equation or discretization being considered, allows speedups to be achieved on multiprocessor systems, and can even allow speedups of I/O-bound jobs in single-processor environments.

Sequential staging of tasks

A significant portion of the numerically intensive engineering and scientific computations deals with time-dependent partial differential equations and iterative methods applied to large systems of equations. Implementations of such problems have an important common characteristic. The same (or similar) set of algorithms is executed repeatedly by sweeping the data arrays in some organized fashion. In each such sweeping operation, some fields of interest are extrapolated or advanced by a $\Delta\tau$ amount. The integration variable τ may be time, distance, or iteration sequence number, depending on the problem under consideration. As it was seen in the previous section, domain decomposition methods are confined to the computational domain D alone at a single τ level. It is, therefore, natural to inquire into the possibility of defining tasks to be processed concurrently with respect to τ, or more precisely in the $D \times \tau$ domain. The extra dimension, τ, provides much greater freedom for organizing tasks conveniently. In the SST method a task corresponds to one or more iterations (sweeps) over D. Tasks are dispatched in a sequence and their execution is controlled to assure that each computes on a unique subdomain of D as well as at unique τ level.

Definition of tasks. A *task* is defined as a disjoint set of subroutines or the computing associated with the execution of these subroutines. Each task can be called independently of all other tasks, and by sharing data, different tasks can interact and work cooperatively. For the sake of precision, consider Equation 4 being applied iteratively to all interior points of the domain shown in Figure 1. Assume that computing is advanced from the left to the right boundary of each row. Such a horizontal sweep is represented by the following subroutine, HSWEEP.

```
      SUBROUTINE HSWEEP(PHI,RHO,KMAX,L)
...          ...
      DO 100 K = 2,KMAX(L)
      PHI(K,L) = 0.25 * OMEGA * (PHI(K,L-1)
     $ + PHI(K,L+1) + PHI(K-1,L)
     $ + PHI(K+1,L) - RHO(K,L))
     $ + (1.0 - OMEGA) * PHI(K,L)
100   CONTINUE
...          ...
```

In this particular example, a task can be defined as the computations associated with one or more complete iterations over the domain D. One such iteration requires N_{row} HSWEEP calls, where N_{row} is the number of rows to be processed. While the tasks are executed asynchronously, it is important that they maintain their relative positions in the sequence in which they were initiated. Moreover, each task must maintain some minimum distance from its preceding neighbor in order to prevent interference between them. This requires inter-task communication through data shared between tasks. This can be done by POST and WAIT events which are capabilities provided by Parallel FORTRAN.

The staging process. After the first task is initiated, execution proceeds at a rate determined by the task being scheduled for processing to one of the available central processing units (CPUs). The second task may be started immediately after the first one but will have to wait until the first task will have completed at least two HSWEEP calls. To avoid interference among tasks, at any stage of processing the ith task must have completed two more horizontal sweeps than task $i + 1$. The hypothetical snapshot of rows being associated with the dispatched tasks is shown in Figure 1. Some of the tasks are assigned to processing units as shown, while others are waiting for their turn. While the tasks are staged sequentially and displaced with respect to each other, there is considerable overlap among them as illustrated in Figure 2, which resembles a timing diagram of pipelined vector processors. There are, however, a few subtle differences:

- A vector unit exploits spatial parallelism; the SST method exploits temporal parallelism.
- In a vector unit identical (SIMD) subtasks are executed synchronously; in the SST method different (MIMD) subtasks are executed asynchronously.

- In vector processing, the operations are shifted in time to coordinate the arrival of subtasks at functionally different pipeline stages; in the SST method, the operations are displaced in space to prevent interference among functionally identical processing units.

The above indicates that symmetry exists between the vector concept and the SST approach.

Parallel formulation of tasks with respect to the τ variable and their sequential staging is characterized by simplicity that has noteworthy practical consequences. Since each task is defined by a serial program that requires no further debugging and the tasks are executed in a well-defined sequence, the necessity for extensive analysis for data dependencies is avoided. Concurrency is achieved by means of the sequential arrangement of tasks that minimize implementation complexity. In many applications, tasks corresponding to unique τ levels could be partitioned into many smaller subtasks based on the ideas of domain decomposition. Such attempts to achieve a higher degree of parallelization would increase implementation complexity and, in view of the very few processors of present-day shared-memory multiprocessors, would not be of much practical value in the near future.

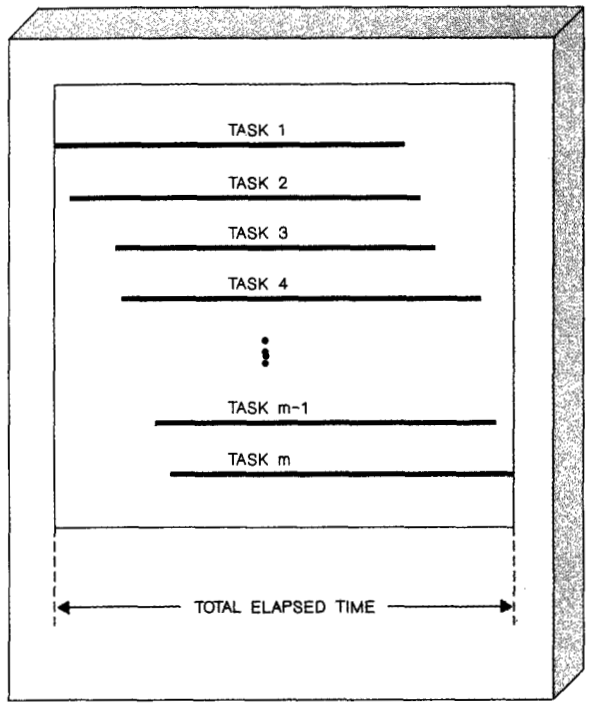
Parallel FORTRAN overview

Implementation and execution of parallel programs requires a means of identifying parallel pieces of work and assigning them to available processors. IBM Parallel FORTRAN[7] (PF) is a facility to specify the parallelism in an application, and only its execution environment and design philosophy will be highlighted. Parallel FORTRAN provides the following items:

- Extensions to the compiler for automatically generating parallel code
- Extensions to the language for explicitly programming in parallel
- Extensions to the library for synchronizing parallel execution through locks and events

From the FORTRAN programmer's point of view, the Parallel FORTRAN environment, as shown in Figure 3, consists of multiple tasks ready for execution, multiple FORTRAN processors associated with the program, and multiple real processors available for doing the work.

Figure 2 Timing diagram of sequentially staged tasks. Their processing is asynchronous. Their order in the sequence, with specified buffer space between consecutive tasks, is maintained by the language extension of the IBM Parallel FORTRAN.

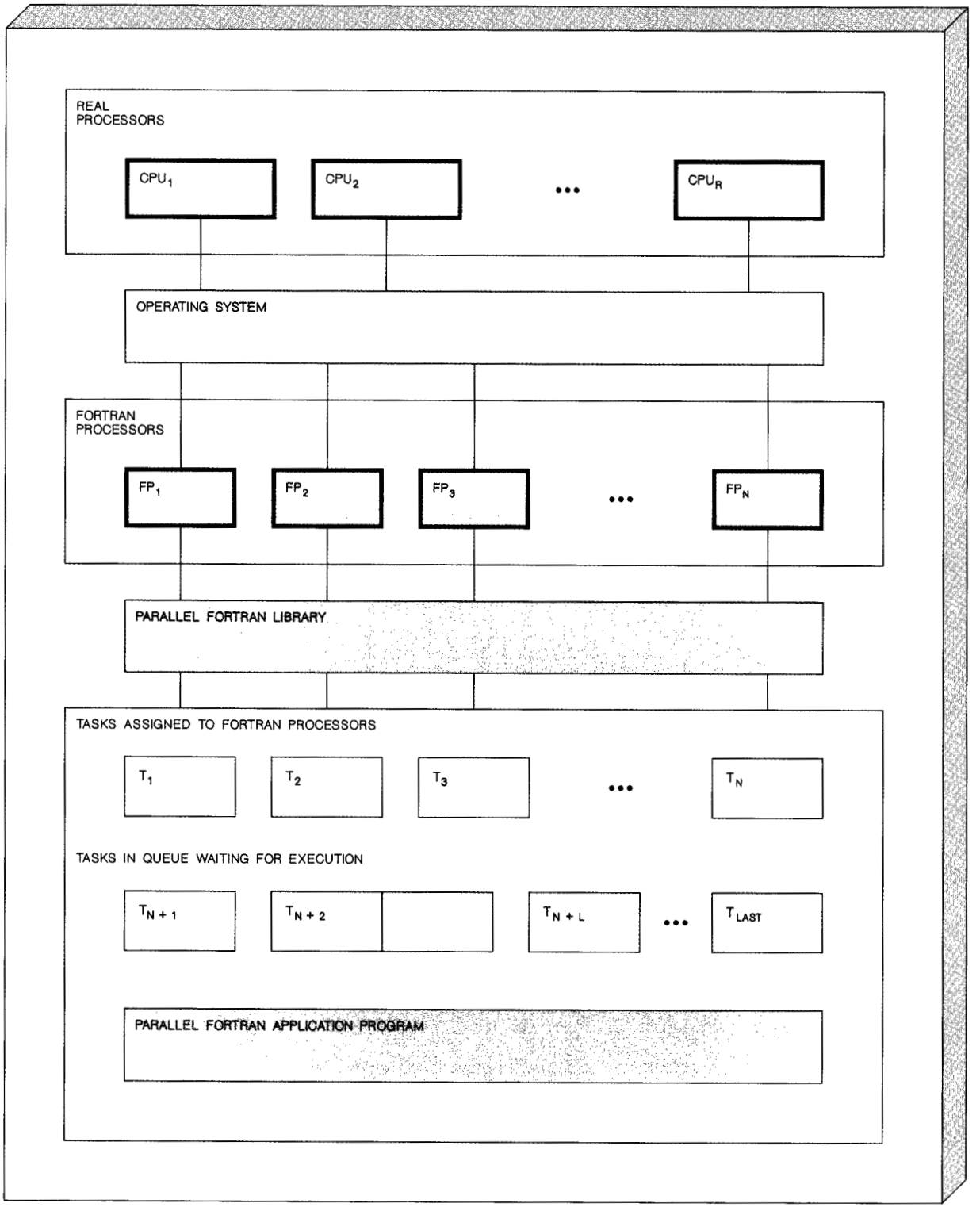


The programmer controls only the first two of these items, whereas the number of real processors available depends on the machine configuration and other jobs being executed on the system. The Parallel FORTRAN language and the compiler are used to identify parallel tasks, and a run-time option is used to specify the number of FORTRAN processors. The computing in this execution environment (Figure 3) is managed as follows.

- Tasks to be executed in parallel are identified within the FORTRAN application program either automatically by the compiler or manually by the programmer.
- FORTRAN library programs assign these tasks to the FORTRAN processors.
- The operating system schedules the FORTRAN processors for execution on available real processors.

Each task is placed in a queue as it is encountered during execution. As a FORTRAN processor completes execution of a task, the FORTRAN library selects the next available task from this queue to be executed

Figure 3 Parallel FORTRAN execution environment



on the processor. Such a transition from one task to another one is accomplished with relatively little effort, since the operating system job scheduler is not invoked.

There is no one-to-one or many-to-one correspondence between FORTRAN processors and real processors. This was an extremely important design decision, some advantages of which have already been described in detail.[7,8] The user's ability to specify the number of tasks, N_{task}, and the number of FORTRAN processors, N_{proc}, enable the user to exploit the potential of an MIMD system. The user's control of N_{task} and N_{proc} are important to a practical and commercially sound parallel computing environment.

Numerical experiments

Numerical experiments were conducted on problems taken from two different disciplines to verify the concept of the SST method and to evaluate the practical feasibility of exploiting temporal parallelism. The first example deals with the Poisson equation and its solution by the SLOR method. The second example is taken from exploration geophysics. It addresses the problem of seismic migration, an inverse problem aimed at imaging the cross section of reflectivity of the subterrain from measurements made at the surface of the earth.

Poisson equation. The numerical solution of the three-dimensional Poisson equation

$$\frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2} + \frac{\partial^2\Phi}{\partial z^2} = \rho \tag{7}$$

by an SLOR method is similar to that of the two-dimensional one. In this case, Equations 5 and 6 become

$$\tilde{\phi}_{k,l,m} = 1/6(\tilde{\phi}_{k,l-1,m} + \tilde{\phi}_{k,l+1,m} + \phi^{n+1}_{k-1,l,m} + \phi^{n}_{k+1,l,m} + \phi^{n+1}_{k,l,m-1}\ \phi^{n}_{k,l,m+1} - \rho_{k,l,m}), \tag{8}$$

and

$$\phi^{n+1}_{k,l,m} = \omega\tilde{\phi}_{k,l,m} + (1 - \omega)\phi^{n}_{k,l,m}. \tag{9}$$

The computational domain is a regular parallelepiped of dimensions N_x, N_y, N_z. Computations begin with the first interior row (along x) within the first interior (x,y) plane at $z = \Delta z$ and proceed plane-by-plane to the last interior plane at $z = (N_z - 1)\Delta z$. The relaxation of each row involves the solution of a tridiagonal system of order $N_x - 2$. Each experiment consists of 24 complete iterations over the volume under consideration. Since the computations are the same no matter how many tasks are used, the error after these 24 iterations is the same for all cases. In the SST formulation, the 24 iterations are partitioned into N_{task} tasks. Successive iterations are assigned to successive tasks. Thus for example, in the case $N_{task} = 8$, the third task T_3, performs the 3rd, 11th, and 19th iterations. The Parallel FORTRAN library and language extensions used in implementing the SLOR method for Equation 7 are shown in Appendix A. Results will be discussed in the next section.

Migration of seismic data. Migration calls for the numerical solution of partial differential equations, which govern the propagation of the recorded signals from the surface to the reflector locations, in reverse time. These methods, generally referred to as wave-equations migration, consist of two steps: (1) *wave extrapolation* and (2) *imaging*. Downward extrapolation results in a wave field that is an approximation to the one that would have been recorded if both sources and recorders had been located at depth z. Thus, events appearing at $t = 0$ are at their correct lateral position, and the extrapolated zero-offset data at $t = 0$ are taken as being the correctly migrated data at the current depth. These data are then mapped onto the depth section at z, the depth of extrapolation. This mapping process is also referred to as imaging.

Let $p = p(x, z, t)$ be the zero-offset seismic data, where x is the horizontal distance, z is depth, and t is the two-way travel time. The downward extrapolation of zero-offset data is governed by the one-way wave equation[9]

$$\frac{\partial P}{\partial z} = \frac{2i\omega}{v}\left[1 - \left(\frac{k_x v}{2\omega}\right)^2\right]^{1/2} P, \tag{10}$$

where P is the Fourier Transform of p, v is the velocity, k_x is the wave number with respect to x, and ω is the temporal frequency. Equation 10 is expressed in the wave-number-frequency domain (k_x, ω) and does not have an explicit representation in the midpoint-time domain (x, t).

For practical reasons, the discussion of which is beyond the scope of this paper, the square root expression is often substituted by its approximate equivalent. For example, a rational approximation of Equation 10 is calculated by truncated continued fractions and splitting, which results in two extrapolators[10]

Table 1 Performance of the SLOR method on the IBM 3090 600S computer with 256 megabytes of available processor storage (problem size = 80Mbytes; $N_x = 400$, $N_y = 400$, $N_z = 125$)

FORTRAN Processors	CPU Time (s)	Elapsed Time (s)	Speedup Ratio
Case 1. Parallel tasks initiated: $N_{task} = 4$			
Serial run	243	248	1.00
$N_{proc} = 1$	245	254	0.976
$N_{proc} = 2$	246	129	1.92
$N_{proc} = 3$	246	88	2.82
$N_{proc} = 4$	246	68	3.65
$N_{proc} = 5$	246	68	3.65
$N_{proc} = 6$	246	68	3.65
$N_{proc} = 12$	246	68	3.65
Case 2. Parallel tasks initiated: $N_{task} = 8$			
Serial run	243	248	1.00
$N_{proc} = 1$	244	252	0.984
$N_{proc} = 2$	245	129	1.92
$N_{proc} = 3$	245	88	2.82
$N_{proc} = 4$	245	68	3.65
$N_{proc} = 5$	245	56	4.43
$N_{proc} = 6$	245	48	5.17
$N_{proc} = 12$	245	50	4.96
Case 3. Parallel tasks initiated: $N_{task} = 12$			
Serial run	243	248	1.00
$N_{proc} = 1$	247	255	0.973
$N_{proc} = 2$	247	131	1.89
$N_{proc} = 3$	248	89	2.89
$N_{proc} = 4$	248	69	3.59
$N_{proc} = 5$	248	56	4.43
$N_{proc} = 6$	248	48	5.17
$N_{proc} = 12$	248	49	5.06

$$\frac{\partial P}{\partial z} = \left(\frac{2i\omega}{v}\right) P, \tag{11}$$

which is known as the thin lens term, and

$$\left[1 + \frac{v^2}{16\omega^2}\frac{\partial^2}{\partial x^2}\right]\frac{\partial P(x,\omega,z)}{\partial z} = \left(\frac{iv}{4\omega}\right)\frac{\partial^2 P(x,\omega,z)}{\partial x^2}, \tag{12}$$

which is the Fresnel diffraction term. Advancing to greater depths is achieved by applying Equations 11 and 12 alternately in small Δz steps. Equation 11 represents a simple phase shift, whereas Equation 12 is implemented by solving a complex tridiagonal system of equations. To construct a depth section of dimensions $N_x \cdot N_z$, from a $N_x \cdot N_t$, time section, Equations 11 and 12 are solved $N_z \cdot N_\omega$ times, where N_z and N_ω are the number of lines imaged in the migrated section and the number of frequencies used in the computations, respectively.

Discussion of results

Over 60 numerical experiments were performed on the two sets of problems described in the previous section. The timing and speedup results are tabulated in six tables, shown later. From the computational and data-handling points of view, the jobs fall into two classes:

1. Compute-intensive jobs, such as those in Tables 1, 3, and 5, where the processor storage is greater than the data requirements of the job
2. I/O-intensive jobs, such as those in Tables 2, 4, and 6, where the available processor storage is smaller than the data requirements of the job

Parallel runs can be characterized by the parameters illustrated in Figure 3:

1. N_{task}, the number of tasks specified by the programmer
2. N_{proc}, the number of FORTRAN processors
3. The number of real processors available for doing the work

Parallel FORTRAN allows the programmer to initiate more parallel tasks than there are FORTRAN processors. The extra tasks reside in a queue waiting to be executed by one of the FORTRAN processors. The programmer can also specify N_{proc}, which may differ from the number of real processors, which will execute the FORTRAN processors. Every experiment reported in this paper was executed on a stand-alone IBM ES/3090 Model 600S computer having six real processors.

Table 1 shows timing and speedup results corresponding to 24 SLOR iterations as per Equations 8 and 9. The *serial run* refers to the execution of the serial code without introducing multiple tasks for concurrent execution. When there are only 4 tasks defined, such as in the first case, the degree of parallelism can only be four at any time of the execution. Indeed, the maximum speedup is attained with 4 (or more) FORTRAN processors. When the job is partitioned into 8 or 12 tasks, the speedup peaks at $N_{proc} = 6$, which is the number of available real processors. In all three cases, 6 FORTRAN processors perform as well or better than 12. This is not surprising since there are only 6 real processors available and the computing load among tasks is well bal-

anced. The extra FORTRAN processors represent, in this case, additional overhead with no added benefit to computational efficiency. All speedup figures are given with reference to the serial run. The $N_{proc} = 1$ run is slightly slower than the serial run. This is due to the overhead associated with the parallel constructs and running multiple tasks while only one FORTRAN processor is being used. However, considering that the parallel code can be executed on any number of processors between 1 and 6 without any further alteration, this small performance reduction is a modest price to pay.

The 22 jobs represented in Table 1 were repeated with one and only one change: the available processor storage was set to 64 megabytes. The results are tabulated in Table 2. The serial job became I/O intensive due to paging, as evidenced by the elapsed time being 25 times that of the serial job in Table 1. In general, jobs with intensive paging I/O do not benefit markedly from parallel execution on multiple processors. This is particularly true for domain decomposition techniques, where the matching of domain boundaries can increase rather than decrease computing, paging, and I/O. However, the present (SST) approach to exploit temporal parallelism allows for the reduction of paging I/O because more than one task can be executed using the data residing in processor storage.

The essence of Table 2 is that with an increasing number of FORTRAN processors, the observed speedup approaches (asymptotically) the value of N_{task}, which can be understood as follows: If all tasks are computing on data that are relatively near to each other, the amount of data they span can be only a small fraction of the entire data set under consideration. Under these conditions the following scenario is possible. Some data, e.g., D_K, are paged in for processing under task T_1. Since T_2 follows T_1 closely in space (see Figure 1) and time, it is quite probable that D_K will be processed by T_2 before it is paged out. Under favorable conditions all N_{task} tasks will have swept over D_K before it is paged out. In this case D_K undergoes N_{task} times as much processing as in the serial case. Consequently, the total amount of paging is reduced by the corresponding amount. The most notable result of Table 2 is the speedup achieved in the $N_{proc} = 1$ case. This is an excellent example to show the importance of being able to define parallel tasks that can be dynamically allocated to available processors and to specify synchronization between the tasks. In the $N_{proc} = 1$ case, after computing a complete plane within T_l the

Table 2 Performance of the SLOR method on the IBM 3090 600S computer with 64 megabytes of available processor storage (problem size = 80M bytes; $N_x = 400$, $N_y = 400$, $N_z = 125$)

FORTRAN Processors	CPU Time (s)	Elapsed Time (s)	Speedup Ratio
Case 1. Parallel tasks initiated: $N_{task} = 4$			
Serial run	250	7416	1.00
$N_{proc} = 1$	247	2334	3.18
$N_{proc} = 2$	248	1997	3.71
$N_{proc} = 3$	248	1995	3.72
$N_{proc} = 4$	248	1928	3.85
$N_{proc} = 5$	248	1930	3.84
$N_{proc} = 6$	247	1915	3.87
$N_{proc} = 12$	247	1896	3.91
Case 2. Parallel tasks initiated: $N_{task} = 8$			
Serial run	250	7416	1.00
$N_{proc} = 1$	245	1455	5.10
$N_{proc} = 2$	246	1092	6.79
$N_{proc} = 3$	246	1029	7.21
$N_{proc} = 4$	246	1003	7.39
$N_{proc} = 5$	246	989	7.50
$N_{proc} = 6$	247	980	7.57
$N_{proc} = 12$	247	961	7.72
Case 3. Parallel tasks initiated: $N_{task} = 12$			
Serial run	250	7416	1.00
$N_{proc} = 1$	248	1158	6.40
$N_{proc} = 2$	248	785	9.45
$N_{proc} = 3$	249	726	10.2
$N_{proc} = 4$	249	702	10.6
$N_{proc} = 5$	249	711	10.4
$N_{proc} = 6$	249	691	10.7
$N_{proc} = 12$	249	664	11.2

FORTRAN processor switches to T_{l+1} in the queue. In a stand-alone environment this amounts to the processor visiting each task in a round-robin fashion. As the processor moves from one task to another, D_K is being subjected to several iterations between paging in and paging out. Parallel FORTRAN provides a means of easily representing the parallel nature of this algorithm and the synchronization required. While it is true the approach of reusing data in storage could be coded[11] in serial FORTRAN, the code would not be as easy to understand, implement, and debug.

Tables 1 and 2 refer to computer experiments carried out on three-dimensional array of 80 megabytes. To ascertain that the method used in formulating parallel runs scale well, two sets of experiments with $N_{task} = 12$ were repeated on data 2.5 times that size. The corresponding results are summarized in Tables

Table 3 **Performance of the SLOR method on the IBM 3090 600S computer with 256 megabytes of available processor storage (problem size = 200M bytes; $N_x = 500$, $N_y = 500$, $N_z = 200$)**

FORTRAN Processors	CPU Time (s)	Elapsed Time (s)	Speedup Ratio
Parallel tasks initiated: $N_{task} = 12$			
Serial run	590	600	1.00
$N_{proc} = 1$	596	607	0.988
$N_{proc} = 2$	596	306	1.96
$N_{proc} = 3$	597	206	2.91
$N_{proc} = 4$	597	157	3.82
$N_{proc} = 5$	597	127	4.72
$N_{proc} = 6$	598	108	5.56
$N_{proc} = 12$	598	115	5.22

Table 4 **Performance of the SLOR method on the IBM 3090 600S computer with 64 megabytes of available processor storage (problem size = 200M bytes; $N_x = 500$, $N_y = 500$, $N_z = 200$)**

FORTRAN Processors	CPU Time (s)	Elapsed Time (s)	Speedup Ratio
Parallel tasks initiated: $N_{task} = 12$			
Serial run	612	19,517	1.00
$N_{proc} = 1$	598	2,960	6.59
$N_{proc} = 2$	598	2,033	9.60
$N_{proc} = 3$	599	1,912	10.2
$N_{proc} = 4$	599	1,820	10.7
$N_{proc} = 5$	599	1,771	11.0
$N_{proc} = 6$	599	1,746	11.2
$N_{proc} = 12$	610	1,699	11.5

3 and 4. For this larger problem size, all parallel runs resulted in greater speedup values than those measured for smaller data sets. This is an expected result, since the overhead associated with the parallel problem formulation is distributed over larger tasks.

The seismic migration method under consideration lends itself superbly to parallelization by domain decomposition. This is seen immediately by considering that the migration algorithm is formulated in the temporal frequency domain ω and the operations governed by Equations 11 and 12 are linear. Therefore, in principle, data associated with different frequencies can be processed independently of each other. Thus, the method based on the sequential staging of tasks does not open up new vistas for parallel execution but provides only an alternate approach to parallel computing. Nevertheless, in this textbook example of domain decomposition, one can find subtle differences in favor of the SST method.

- The sequence of operations for computing the migrated data in the SST method remains the same as that in the serial case. Therefore, the results should agree for any N_{task}, N_{proc}, and real CPUs available.
- The debugging of the parallel code is facilitated to a great extent.
- In conventional domain decomposition applied to seismic migration, all processors compute the same part (depth level) of the output (migrated) data from different parts (frequencies) of the input data.
- In the present method based on the sequential staging of tasks applied to seismic migration, different processors compute different parts (depth level) of the output (migrated) data from different parts (frequencies) of the input data.

The speedup figures for the compute-intensive example represented by Table 5 are excellent considering that they correspond to over 98 percent parallel execution of the code. In the I/O-intensive example shown in Table 6 the speedup figures are even better, but the additional speedups are due to reduced paging resulting from improved data management just as in the SLOR method discussed in detail earlier.

Reflections on parallel computing trends

The class of problems considered in this paper is characterized by repetitive operations applied to multidimensional data arrays representing the computational domain D. A sweep over the data corresponds to numerical integration whereby physical fields of interest are extrapolated or advanced by a $\Delta\tau$ amount. The integration variable τ depends on the problem under consideration. In time-dependent partial differential equations, τ is time. In iterative solution of elliptic problems, τ represents the iteration sequence number. In inverse problems, such as the migration of seismic reflection data discussed earlier, τ becomes the depth variable.

A very carefully researched approach to parallel formulation of such computational problems is based on domain decomposition which attempts, according to Gonzalez and Wheeler,[4] "to break up the domain of integration into many pieces, then somehow construct the global solution from these local solutions." To arrive from an indefinite approach to

a well-defined approach is not an easy matter. One source of the difficulty is that students of domain decomposition techniques usually confine themselves to working at a single τ level. An inquiry into the possibilities of formulating parallel tasks in τ, or more importantly in the $D \times \tau$ domain, can open up new horizons and untapped opportunities. This paper details an approach to exploit this τ domain parallelism by pipelining or sequential staging of tasks. The numerical experiments reported herein support the ideas set forth in the earlier part of the paper and substantiate the feasibility of the method. The computer experiments also yielded some new results: a verification of the authors' hypothesis that Parallel FORTRAN can be used to reduce elapsed time of certain I/O-intensive applications even on a single-processor system.

Taking advantage of τ domain parallelism via the SST method does not preclude or negate the benefits of domain decomposition methods. The two ideas can coexist. For example, in both applications treated earlier, the tasks as defined herein could be split into many smaller subtasks following the ideas of domain decomposition. One must, however, ponder how much parallelism is really needed in most realistic large-scale computing environments. The seismic migration problem considered can be partitioned into N_ω equal tasks, where $100 < N_\omega < 1000$, and in case of a three-dimensional migration, each ω plane can be (domain) decomposed into some 100 complex rows and/or columns. This 10 000-fold parallelism creates excitement only among researchers of parallel hardware and software. The geophysicists remain unimpressed, and there is no evidence of production codes running in parallel. One reason is clear and simple. In realistic computing environments many users compete for few processors, and one user seldom can justify using four or six processors in a dedicated mode. Other reasons have to do with implementation complexity.

The situation is expected to change in the near future when, for example, 10 or 20 users will access a shared-memory MIMD system with 128 processors. This does not really need to be a true shared-memory system as long as the user perceives it as one. Consider again the above-mentioned 10 000-fold parallel seismic migration problem. By the time one creates that many tasks with corresponding I/O, one can still expect 95 percent parallelism, but not more. A problem like this on a dedicated 128 processor system would result in no more than 14 percent total CPU utilization. If, however, 10 such identical problems were executed simultaneously on the above system, disregarding all overhead related to support Parallel FORTRAN and multitasking to simplify matters, system utilization might be around 99.9 percent. As a result, each job could achieve a speedup of 12. Systematic extensions made to current architectures and other means of taking advantage of normal technological advances of the future will result in the achievement of improved performance without requiring exotic architectures, revolutionary parallel numerical methods, and special software to detect parallelism automatically.

Table 5 Performance of a seismic migration on the IBM 3090 600S computer with 256 megabytes of available processor storage (problem size = 192M bytes; $N_x = 4096$, $N_t = 4096$, $N_z = 48$, $N_\omega = 2048$)

FORTRAN Processors	CPU Time (s)	Elapsed Time (s)	Speedup Ratio
Parallel tasks initiated: $N_{task} = 12$			
Serial run	3,223	3,273	1.00
$N_{proc} = 1$	3,220	3,271	1.00
$N_{proc} = 2$	3,224	1,659	1.97
$N_{proc} = 3$	3,221	1,122	2.92
$N_{proc} = 4$	3,222	856	3.82
$N_{proc} = 5$	3,224	698	4.69
$N_{proc} = 6$	3,226	593	5.52

Table 6 Performance of a seismic migration on the IBM 3090 600S computer with 128 megabytes of available processor storage (problem size = 192M bytes; $N_x = 4096$, $N_t = 4096$, $N_z = 48$, $N_\omega = 2048$)

FORTRAN Processors	CPU Time (s)	Elapsed Time (s)	Speedup Ratio
Parallel tasks initiated: $N_{task} = 12$			
Serial run	3,214	17,410	1.00
$N_{proc} = 1$	3,220	6,128	2.84
$N_{proc} = 2$	3,222	3,616	4.81
$N_{proc} = 3$	3,223	2,722	6.40
$N_{proc} = 4$	3,225	2,432	7.16
$N_{proc} = 5$	3,225	2,399	7.26
$N_{proc} = 6$	3,225	2,361	7.37

Conclusion

An approach to parallel formulation of scientific problems on shared-memory multiprocessors has been described. In the class of problems considered, repetitive operations are applied over the computational domain D, whereby some fields of interest are

integrated with respect to an independent variable τ, which may be time, distance, or iteration sequence number, depending on the physical nature of the problem. The technique known as sequential staging of tasks (SST) is based on defining concurrently executable tasks with respect to τ rather than with respect to D alone at a single τ level. Partitioning the computational workload with respect to τ, or more precisely with respect to the $D \times \tau$ domain, has significant advantages. This study substantiates that parallel formulation of tasks with respect to the τ variable and their sequential staging is characterized by simplicity that has noteworthy practical consequences, such as minimized paging I/O. Since each task is defined by a serial program which requires no further debugging, and the tasks are executed in a well-defined sequence, the necessity of extensive analysis for data dependencies is avoided. Concurrency is achieved by means of a sequential arrangement of tasks that has a profound effect on minimizing implementation complexity, which is probably the single most important factor determining user acceptance of parallel computing.

Acknowledgments

The performance data are from runs performed at the IBM Washington System Center. The authors extend thanks to: the dedicated assistance of Alan Karp, whose patient cooperation in guiding the programs to and through a stand-alone Model 600 system for dedicated runs was invaluable; Randolph Scarborough for his expert advice on Parallel FORTRAN implementations; Baxter Armstrong for his patience in discussing and defining terminology for new concepts; Horace Flatt for his constant encouragement and interest in this work; and the reviewers who made many valuable and constructive suggestions.

Appendix A: SST code for concurrent SLOR iterations

```
C ... SLOR iterations in parallel for a 3D
C ... Poisson equation computed by sequential
C ... staging of tasks managed by parallel
C ... events
C
C ... set up problem size and parameters
C ...       grid size: (NX,NY,NZ)
C ...       NITER = total number of iterations
C ...               to be computed
C ...       NTASK = number of tasks
C
C ... Note that in the following code, the
C ... Parallel FORTRAN constructs are boxed in.
C
C ...
      DIMENSION PHI(NX,NY,NZ)
      DIMENSION IEVENT(NTASK,NZ-1)
C ...
C ... originate tasks, events, and lock
C
      DO 10 I = 1, NTASK
        ORIGINATE ANY TASK IT
10    CONTINUE
      DO 20 J = 1, NZ-1
      DO 20 I = 1, NTASK
        CALL PEORIG (IEVENT(I,J))
20    CONTINUE
      CALL PLORIG (LOCK)
C
C ... set initial and boundary values
C ...
C ... set the global iteration count: ITER to
C ... 0 and schedule all the tasks
C
      ITER = 0
      DO 30 I = 1, NTASK
        SCHEDULE ANY TASK IT,
     $      CALLING SLOR (PHI,NX,NY,NZ,NITER,
     $                    ITER,IEVENT,LOCK)
30    CONTINUE

      WAIT FOR ALL TASKS
C ...
      STOP
      END
C

C
      SUBROUTINE SLOR (PHI,NX,NY,NZ,NITER,
     $                 ITER,IEVENT,LOCK)
      DIMENSION PHI(NX,NY,NZ)
      DIMENSION IEVENT(NTASK,NZ-1)
C
C ... critical section, obtain a unique
C ... iteration 'MINE' for each task
C
100   CALL PLLOCK (LOCK)
      ITER = ITER + 1
      MINE = ITER
      CALL PLFREE (LOCK)
C
```

```
C ... end critical section
C
      IF (MINE .GT. NITER) GO TO 300
C
C ... IWAIT identifies the task to issue the
C ... wait.  IPOST identifies which task to
C ... post.  The following shows first few
C ... values for the variables if NTASK=4
C ...
C ...          ITER :  1  2  3  4  5  6  ...
C ...          IWAIT:  1  2  3  4  1  2  ...
C ...          IPOST:  2  3  4  1  2  3  ...
C ...
C ... thus iteration 5 will be picked up by
C ... the first task and iteration 6 by the
C ... second task, etc.
C
      IWAIT = MOD(MINE-1, NTASK) + 1
      IPOST = MOD(MINE  , NTASK) + 1
C
C ... start the iteration (NOTE: first
C ... iteration can start immediately)
C
      IF (MINE .NE. 1 ) CALL PEWAIT
     $  (IEVENT(IWAIT,1))
C
C.... solve planes 2 thru NZ-1
C
      DO 400 K = 2, NZ-1
      IF (MINE .NE. 1 ) CALL PEWAIT
     $  (IEVENT(IWAIT,K))

C
C.... compute new values for plane K
C
      DO 500 J = 2, NY-1
C
C ... compute new values for line J by solving
C ... a tridiagonal system and perform SOR
C ... extrapolation according to Eqns (8)
C ... and (9)
C ...
500   CONTINUE
      IF (MINE .NE. NITER) THEN
      CALL PEPOST (IEVENT(IPOST,K-1))
C
C ... Note that the following call to the
C ... library routine 'PXDISP' is optional.
C ... The effect of the routine is to
C ... temporarily halt the execution of the
C ... current task by putting the current task
C ... at the bottom of the 'queue'.  The next
C ... waiting task at the top of the queue will
C ... then be picked up for execution.  In this
C ... way, the locality of the working set is
C ... maintained. It enables this code to
C ... execute much faster (less paging I/O)
C ... than the serial code when the problem
C ... size exceeds the primary memory. This
C ... effect is most noticeable when the number
C ... of FORTRAN processors is 1. See Tables
C ... 2, 4, and 6 in the text.
C
      CALL PXDISP
      ENDIF
400   CONTINUE
      IF (MINE .NE. NITER) THEN
      CALL PEPOST (IEVENT(IPOST,NZ-1))
C
C ... same comment as above
C
      CALL PXDISP
      ENDIF
      GOTO 100
300   RETURN
      END
```

Cited references

1. J. J. Hack, "On the promise of general-purpose parallel computing," *Parallel Computing* **10,** No. 3, 261–275 (1989).
2. R. W. Hockney and C. R. Jesshope, *Parallel Computers 2*, Adam Hilger Ltd., Bristol, Great Britain (1988).
3. R. Glowinsky and M. F. Wheeler, "Domain decomposition and finite element methods for elliptic problems," *Proceedings, First International Symposium on Domain Decomposition Methods for Partial Differential Equations* (Paris, France, 1987), SIAM, Philadelphia (1988), pp. 144–172.
4. R. Gonzalez and M. F. Wheeler, "Domain decomposition for elliptic partial differential equations with Neumann boundary conditions," *Parallel Computing* **5,** No. 1 & 2, 257–263 (1987).
5. R. W. Hockney, private communication, Rome, Italy (1985).
6. L. Adams and J. Ortega, "A multi-color SOR method for parallel computation," *Proceedings, 1982 International Conference on Parallel Processing* (Silver Spring, MD), IEEE Computer Society (1982), pp. 53–56.
7. L. J. Toomey, E. C. Plachy, R. G. Scarborough, R. J. Sahulka, J. F. Shaw, and A. W. Shannon, "IBM Parallel FORTRAN," *IBM Systems Journal* **27,** No. 4, 416–435 (1988).
8. *IBM Parallel FORTRAN Language and Library Reference*, SC23-0431, IBM Corporation; available through IBM branch offices.
9. J. Gazdag, "Wave equation migration with the phase-shift method," *Geophysics* **43,** 1342–1351 (1978).

10. J. Gazdag and P. Sguazzero, "Migration of seismic data," *Proceedings of the IEEE* **72,** 1302–1315 (1984).
11. E. L. Wachspress, *Iterative Solution of Elliptic Systems,* Prentice-Hall Inc., Englewood Cliffs, NJ (1966), p. 279.

Jeno Gazdag *IBM Palo Alto Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304.* Dr. Gazdag is currently a manager at the Palo Alto Scientific Center with research objectives that include the development of practical, robust, and parallel algorithms for shared-memory MIMD computers. He began his undergraduate studies at the Polytechnic University of Budapest, received his B.E. from McGill University, Montreal, his M.A.Sc. from the University of Toronto, and his Ph.D. in electrical engineering from the University of Illinois in 1966. From 1961 to 1962 he worked at the National Research Council of Canada in Ottawa. He joined IBM's Research Division in 1966 and subsequently transferred to the IBM Palo Alto Scientific Center, where he conducted research in diverse areas of computational physics. His research interest is in numerical solution methods for partial differential equations with applications in seismic data processing. Dr. Gazdag is a member of the European Association of Exploration Geophysicists, the Society of Exploration Geophysicists, and the Institute of Electrical and Electronics Engineers.

Hsuan-Heng Wang *IBM Palo Alto Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304.* Dr. Wang is currently a staff member at the Palo Alto Scientific Center. He joined IBM in Poughkeepsie after receiving a Ph.D. in mathematics from the University of Texas, Austin, in 1964. After joining IBM, Dr. Wang worked in the area of scientific computing and machine evaluation, and recently he has been active in parallel and vector processing. He has received an IBM Outstanding Technical Achievement Award for his work on developing numerical algorithms for vector machines. He is a member of the Institute of Electrical and Electronics Engineers Computer Society and the Society for Industrial and Applied Mathematics.

Reprint Order No. G321-5380.