NASA TM-88200

NASA Technical Memorandum 88200

Experiences with Cray Multi-tasking

E.N. Miya

November 1985

National Aeronautics and
Space Administration

NASA Technical Memorandum 88200

Experiences with Cray Multi-tasking

E. N. Miya, Ames Research Center, Moffett Field, California

November 1985

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

Introduction and Outline

The search for improved performance has focused on using different forms of *parallelism* to achieve speed increases.[1] To this end, Cray Research, Inc. (CRI) introduced vector processing and, most recently, user-accessible multi-tasking (Larson, 1984, Research, 1985, Research, 1984). The Cray work on multi-tasking takes a "coarse *grain*" approach to parallelism in contrast to the "fine grain" parallelism of vector instruction sets or dataflow (Dennis, 1979). Multi-tasking was not introduced without tradeoffs such as this.

The issues raised with the introduction of multi-tasking and multiprocessing involve more than performance. Multi-task programs may require major changes in their algorithms storage management, and code. Toward this end, new or modified programming languages are needed.

Explicitly parallel languages must handle problems beyond the scope of conventional programming languages. These issues include data protection, non-determinism, process management (i.e., creation, scheduling, deletion), interprocess communication, synchronization (i.e., deadlock and starvation), and error and exception handling (Denning, 1985). These problems are well documented with the Carnegie-Mellon's multiprocessor research (Jones, 1980). There are few simple solutions,[2] and tradeoffs must be made. Grit and McGraw compare parallel applications programming to operating systems programming in sheer difficulty (Grit, 1983) thus creating more trouble.

System timing must receive careful consideration in multi-task codes to avoid inconsistent results and deadlock. A sequential code hacking style is dangerous in this environment. Care is required when dividing a problem into multiple tasks to avoid inconsistency. This division is called *partitioning* or *decomposition* as well as by other terms.

Several partitioning schemes can execute codes in parallel (Jones, 1980). The most common are *pipelining, spatial partitioning* (by problem space or machine storage), or *relaxation* that removes assumptions of data consistency. David Kuck is best known for his research on automatic partitioning (Kuck, 1980). This paper covers the subject of partitioning an existing application program by hand.

The program "TWING" is the vehicle that we use to explore the issues surrounding multi-tasking. This report covers:

Our programming style is conservative and defensive. We assume the multi-task program will not execute the first time. We chose a synchronous algorithm and sought results identical to results using uni-task TWING. This work stresses the importance of careful analysis, design, and testing.

Existing FORTRAN Drawbacks

As background, it is useful to understand the problems inherent with standard FORTRAN and multi-tasking. FORTRAN is not currently designed for or intended to run in a parallel environment. New problems arise in multi-tasking such as synchronization, communication, error handling, and deadlock. An excellent survey of language issues and various attempts at solving them appears in *Computing Surveys (Andrews, 1983)*.

First, the standard FORTRAN language lacks process-creation primitives and structures. The SUBROUTINE is the closest FORTRAN object resembling a process or a TASK. Second, the language lacks features for explicit synchronization and protection

[1] The terminology is varied, colorful, and highly confusing. Among other phrases, we have: parallel processing, multiprocessing, polyprocessing, distributed computing, decentralized computing, and so forth. Each phrase has a slightly different meaning: enough to make communications difficult. CRI makes the subtle distinctions that *multiprogramming* means multiple jobs working on a CPU [e.g., time-sharing], *multiprocessing* means work done on multiple physical CPUs working multiple jobs [i.e., without regard for jobs], and *multi-tasking* means multiple physical CPUs working *cooperatively* on a *single* problem.

[2] Jones and Gehringer specifically classify distributed system issues into problems of 1) consistency, 2) deadlock, 3) starvation, and 4) exception handling.

such as semaphores (Dijkstra, 1968) (i.e., ALGOL-68), monitors (Hoare, 1974) (i.e., concurrent Pascal), or rendezvous (i.e., Ada[3]) (DOD, 1980). It, also, lacks explicit communication features such as mailboxes.

Each of the aforementioned synchronization features has assumptions of *atomicity* (uninterruptability) which is critical for maintaining a degree of consistency that standard FORTRAN cannot currently provide. Synchronization is a technique normally reserved for operating system programming (using libraries) since it offers "hazardous" user facilities.[4]

Lastly, the software engineering problems associated with FORTRAN are accentuated in a multi-tasking environment. These problems are documented elsewhere (Dijkstra, 1968); they include GO TOs and the lack of modern data structures. An example of these tradeoffs is the inability for Cray multi-tasking FORTRAN to coherently perform multiple RETURNs.

It is not easy to add these features to the FORTRAN language. These features conflict with existing language semantics. The programmer must locate and manage *side effects* on globally referenced memory (such as COMMON variables), call-by-reference parameter passing, and manufacturer-dependent features. These side effects also occur at the lower vector-processing level; Cray users have modified their programming style to accommodate them. We can similarly expect users to adopt a multi-tasking programming style.

Cray Multi-tasking FORTRAN extensions

The existing Cray Research supercomputer line performs efficiently by using a vector instruction set. Performance improvement is achieved by using regular data-access patterns on arrays and their indices. Currently, multi-tasking seeks to achieve performance improvement using multiple processing units.

Cray Research has a set of primitive extensions to support multi-tasking in version 1.13 of their CFT FORTRAN compiler (Larson, 1984). These extensions currently allow several *virtual* CPUs to execute simultaneously on one to four *physical* CPUs. These primitives are invoked using subroutine CALLs. They are useful for creating more elaborate synchronization mechanisms such as monitors (Hoare, 1974).

The Cray primitives fall into three general categories:

TASK creation and control
EVENT creation and synchronization
LOCK creation and protection

The primitives are controlled using three basic data structures: a TASK control array (INTEGER type containing two or three elements), EVENTs, and LOCKs (both of type INTEGER) all explicitly assigned (i.e., created).

An extremely important semantic[5] difference is the handling of storage (primary memory) in this version of FORTRAN. Local storage in normal FORTRAN has a static allocation resulting in possible *side effects*.

The new multi-tasking CFT FORTRAN requires a dynamic or stack-based allocation of storage more characteristic of ALGOL-like languages such as Pascal or C. This is necessary for TASK creation and migration. Local storage (scalars or arrays) now has a finite lifetime and scope. A programmer cannot use a value left over from a previous subroutine CALL or assume values are initialized to zero (0). This is a radical departure from standard FORTRAN. The next four sections cover these primitives and their effects in greater detail.

TASK Control

We begin with TASK creation. A user controls a concurrent object called a *TASK* that is invoked like a SUBROUTINE. The TASK is defined like any other SUBROUTINE except that its name must explicitly appear in an EXTERNAL statement before a CALL, and its storage gets handled differently. The specific TASK syntax primitives are shown in figure 1 where SUBNAME is the SUBROUTINE name, and ITCA is an INTEGER *TASK control array*. Note,

[3]Ada is a trademark of the Ada Joint Project Office of the US DOD.

[4]There exists the potential for user-induced system deadlock.

[5]We mention this because there are no FORTRAN keywords (i.e., syntax) associated with this problem; it's semantic.

```
CALL TSKSTART(ITCA,SUBNAME,[arguments])
CALL TSKWAIT(ITCA)
```

Figure 1. Cray TASK primitives.

restricted, positional SUBROUTINE arguments are passable.

A *TASK control array* is a simple data structure that holds TASK control data for a scheduler that is loaded with the program on execution. This scheduler is distinct from the operating system's scheduler in that it governs user defined TASKs rather than JOBs.

The TASK is created using the TSKSTART call. TSKSTART is similar to a **fork** in languages like ALGOL-68 except a separate address space is created, much like a separate space for a FORTRAN subroutine. The effect is like a subroutine CALL with one major exception: subroutine CALLs are synchronous and consequently wait, unlike TSKSTART calls.

The following program fragment (figure 2), listed in parallel, illustrates the creation of a TASK. Note that the subprogram allocating the TASK control array must not lose the TASK control array storage! Severe problems will result!

A "TSKWAIT" statement could force a crude explicit synchronization on execution of a RETURN statement within task A. The section on **Debugging** will touch on the usefulness of TSKWAIT. More refined synchronization is available using EVENTs and LOCKs. There are also TSK calls covered in the Cray documentation that report TASK information or statistics (Research, 1985).

Cray support of multi-tasking includes a simple deadlock-detection mechanism. Deadlock occurs when all user TASKs are waiting for a condition that never occurs. This goes for synchronization using TSKWAIT, EVENTs, or LOCKs. Care is required, particularly, in using EVENTs because these functions are not necessarily atomic (indivisible). [Deadlock is discussed further in the section on **Debugging**.]

EVENTs and LOCKs

Synchronization and consistency protection use combinations of EVENTs and LOCKs. Both are useful for simple synchronization. The key difference between an EVENT and a LOCK is that a LOCK forces tasks to run in a First-In, First-Out (FIFO) order. An EVENT is comparable to a "broadcast," and many TASKs can run at once. It is also important to clear or reset a LOCK or EVENT at appropriate times.

```
PROGRAM
INTEGER TA(2)
EXTERNAL A
...
CALL TSKSTART(TA,A,arguments)     SUBROUTINE A(parameters)
...                               ...
END                               END
```

Figure 2. An illustration of simple TASK creation.

EVENTs and LOCKs are created by using subroutine CALLs which assign special protection in the same manner in which TASKs are created. Basic arithmetic and logical operations are disabled for these objects until they are released. The specific primitive SUBROUTINE CALLs are:

EVENT Control	LOCK Control
EVASGN(IEVAR)	LOCKASGN(LCK)
EVPOST(IEVAR)	LOCKON(LCK)
EVWAIT(IEVAR)	LOCKOFF(LCK)
EVCLEAR(IEVAR)	LOCKREL(LCK)
EVREL(IEVAR)	

in which IEVAR and LCK are INTEGERs assigned as EVENTs or LOCKs. The following is a simple two-TASK synchronization using EVENTs in two separate executing TASKs. The scope is shown by the bounding boxes of figure 3. If an EVENT or a LOCK is CLEARed or RELeased while some TASK is waiting, the consequences are nondeterministic and can be disastrous.

If combinations of EVENTs, LOCKs, and COMMON memory are used, it is possible to make more elaborate synchronization mechanisms such as semaphores and monitors. Sequential *critical sections* of code and data need protection using these synchronization primitives. Problems of inconsistent synchronization are covered in the next section.

Communications

Communication takes place though one of three mechanisms:

CALL-by-Reference parameter passing
Global COMMON memory
TASK COMMON memory

Data is passed using shared (e.g., COMMON) variables. This is the principal means of communication and requires care in use.

A TASK-local COMMON (e.g., TASK COMMON) is available in version 1.14 of the CFT compiler. It is similar to the more global COMMON except that its data is accessible only to objects (SUBROUTINEs) within a particular TASK. Maintaining a consistent system state is a chore left to the user.

Consistency is threatened by three basic *hazards*. Suppose A and B are two TASKs running in parallel and sharing a variable V. The hazards are based on the order in which processes access V: a timing problem. The first hazard is the *read-write* hazard – having one TASK prematurely reading a stale value before the appropriate write. The next is the *write-read* hazard: having one TASK prematurely "clobbering" a value before it could be read. The last hazard is the *write-write* hazard in which one TASK writes over values that never get a chance to be read [particularly difficult to detect][6]. The Cray is not responsible for these potential user errors of timing.

Storage and Subroutine Linkage

The actual handling of storage differs vastly from conventional static FORTRAN. This has its greatest effect on SUBROUTINE

[6]The memory on the Denelcor Heterogeneous Processor [HEP] is an attempt to solve this problem. If variables receive a special declaration, they are forced to alternate reads and writes using a unique semaphore memory system.

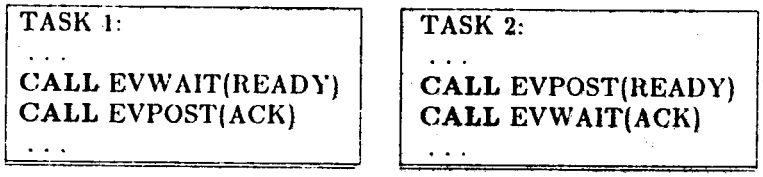

Figure 3. Synchronization of two TASKs using EVENT flags. Boxes represent different address spaces.

and FUNCTION linkages. The semantics of these new linkages prompt some users to name this an entirely different language (e.g., "not-FORTRAN") Old memory-saving tricks such as statically defined and allocated variables left for a second subroutine CALL are now undefined and may contain unreliable data. Users cannot assume values are initialized to zero (0). Expressions in parameter lists involve similar problems.

Those readers familiar with dynamic storage management in scoped languages such as ALGOL, C, Pascal, or LISP should grasp these concepts easily. FORTRAN simply does not offer the protection mechanisms to ensure consistency of data in a multiprocess environment. The user must actively manage the data consistency and program defensively.

The Mathematical Basis for TWING

TWING is a program that solves the conservative full-potential equation, using a fully implicit, approximate-factorization algorithm. The program solves for stable state airflow over a wing flying at transonic velocity. TWING is the development of Dr. Terry Holst and Scott Thomas (Thomas, 1983) at the Applied Computational Aerodynamics Branch, NASA Ames Research Center.

Figure 4 is a schematic of the finite difference mesh over which the flow solver operates. From this representation in "physical space", the problem is transformed into a "computational space" [figure 5] which preserves the orthogonality of the mesh lines throughout the computational domain.

A mathematical representation of this flow solver is given in the derivation of equation 1.c. The three-dimensional, full potential equation (in x,y.z coordinates) is presented in equation 1.a. The transformation into computational coordinates (ξ,η,ς coordinates) yields equation 1.b. In this equation, U, V, and W are terms composed of Φ_x, Φ_y, and Φ_z combined with assorted metric quantities. J represents the Jacobian of the transformation. The finite-difference approximation of this transformed equation (1.c) employs backward difference operators in the ξ,η, and ς directions. This yields the finite-difference approximation in equation 1.c. The special density coefficients $\tilde{\rho}$, $\bar{\rho}$, and $\hat{\rho}$ introduce an artificial viscosity term into the calculation. The residual term $L(\Phi)$ obtained from this equation is used in the first step of the factorization scheme outlined below.

An outline of the three-step approximate-factorization scheme is shown in the derivation of equation 2.c. In step one (equation 2.a), an intermediate term G(i,j) is computed for each point on a given "k-shell" of the mesh by solving a tridiagonal linear system along each η line (i.e., ξ = a constant) extending from the symmetry plane out to the freestream sidewall. In step two (equation 2.b), G(i,j) computes another intermediate term F(i,j,k) for each point in the "k-shell." This step requires the solution of a tridiagonal linear system along each ξ line (i.e., constant η) extending from the upper vortex sheet around the leading edge to the lower vortex sheet (figure 6). Finally, when F(i,j.k) has been computed for every point in the three-dimensional mesh, the correction factor

---

$$(\rho\Phi_x)_x + (\rho\Phi_y)_y + (\rho\Phi_z)_z = 0 \quad (1.a)$$

The three-dimensional full potential equation (x,y,z coordinates).

$$(\rho U / J)_\xi + (\rho V / J)_\eta + (\rho W / J)_\varsigma = 0 \quad (1.b)$$

The full potential equation in computational space (ξ, η, ς).

$$\bar{\delta}_\xi\left(\tilde{\rho} U / J\right)_{i+\frac{1}{2},j,k} + \bar{\delta}_\eta\left(\bar{\rho} V / J\right)_{i,j+\frac{1}{2},k} + \bar{\delta}_\varsigma\left(\hat{\rho} W / J\right)_{i,j,k+\frac{1}{2}} = 0 \quad (1.c)$$

The resultant finite-difference approximation.

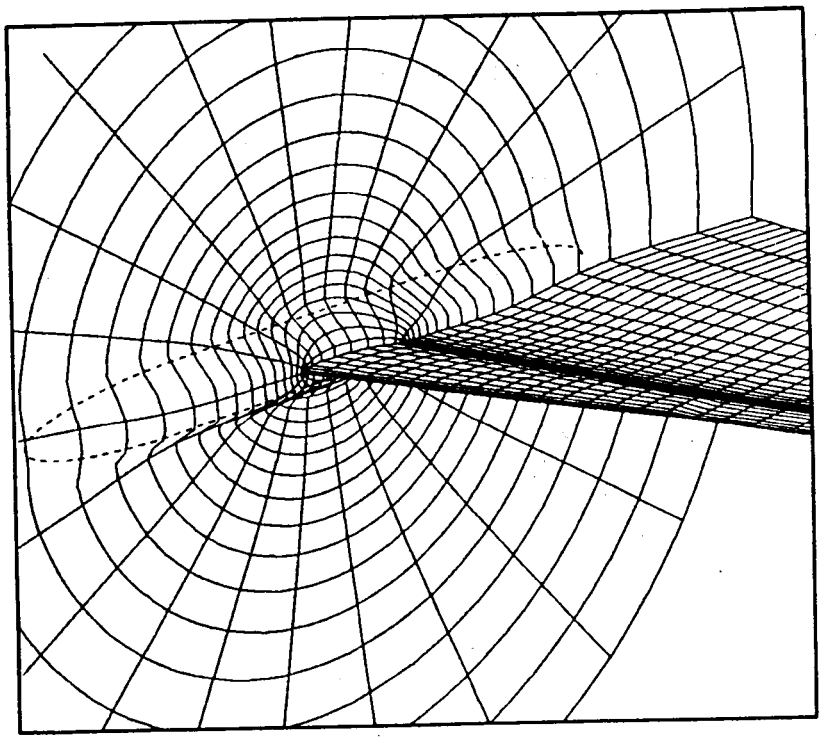

Figure 4. Sample finite difference mesh.

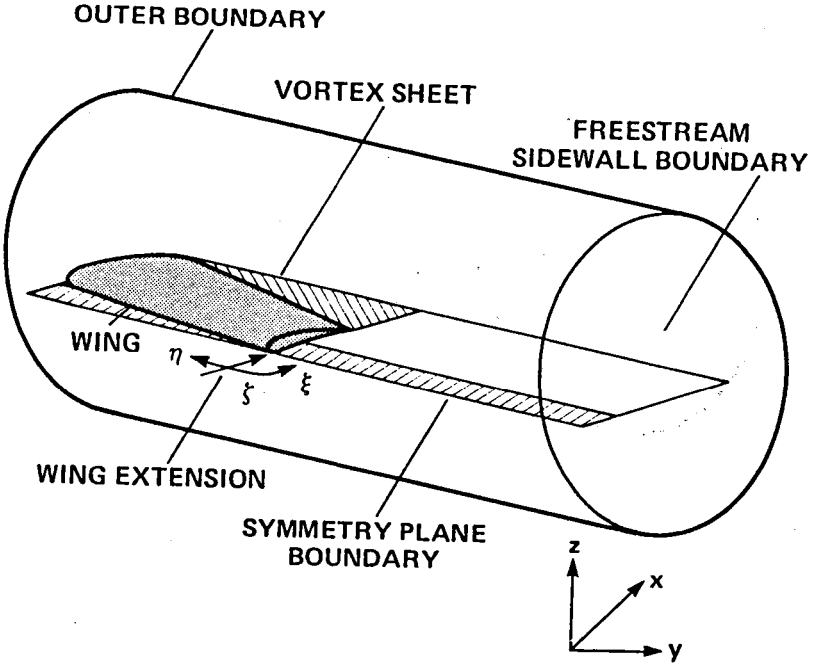

Figure 5. Transformation to computational space.

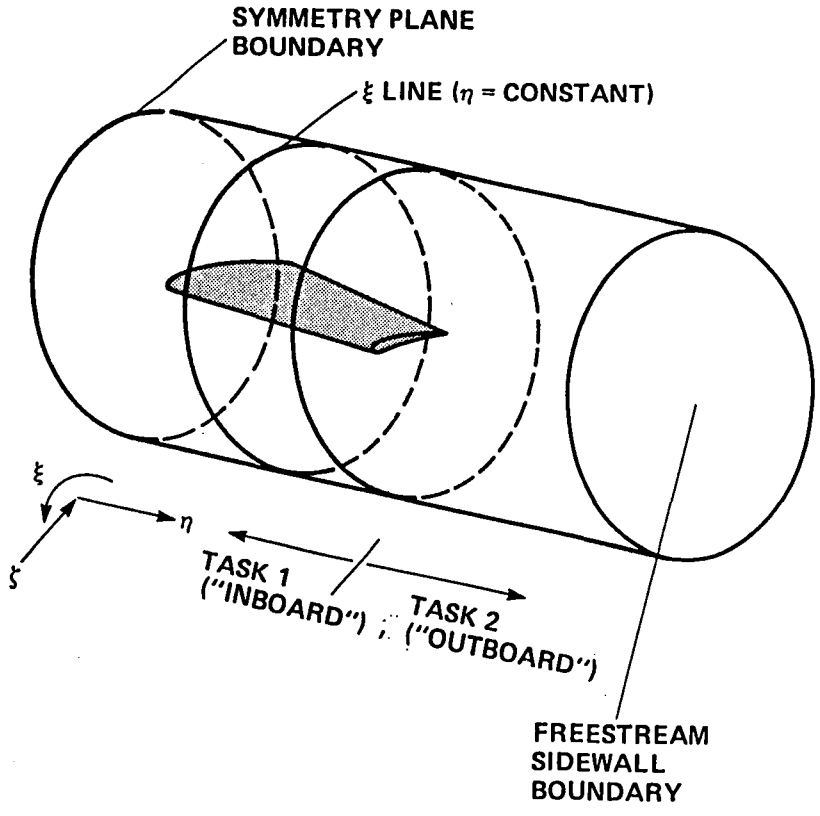

Figure 6. Computation divided into two tasks.

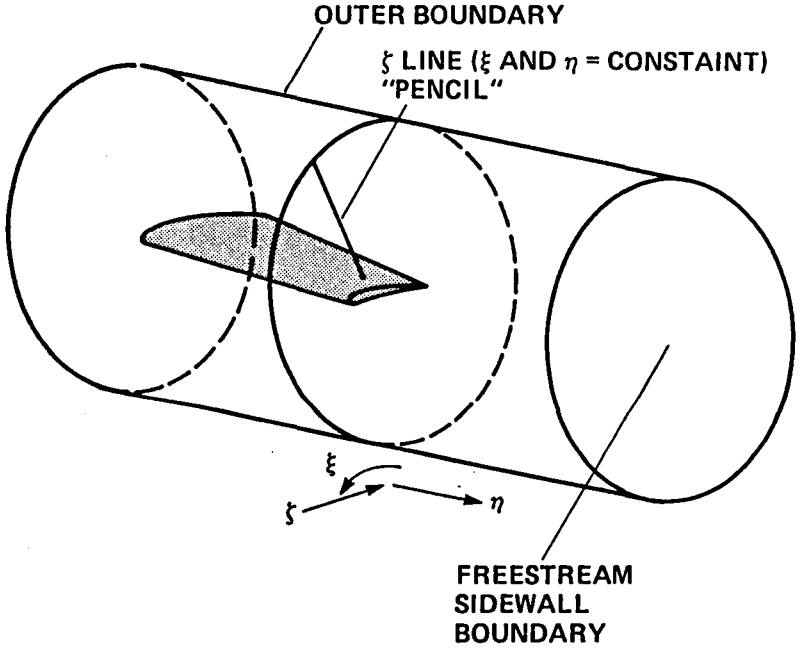

Figure 7. Computation done as a region of pencils.

Step 1:

$$\left(\alpha + \alpha\beta_\eta \left|\frac{V}{J}\right|_{i,j,k} \vec{\delta}_\eta - \frac{1}{A_k} - \vec{\delta}\eta A_j \bar{\delta}_\eta\right) g^n{}_{i,j} = \alpha\omega L \Phi^n{}_{i,j,k} + \alpha A_{k+1} f^n{}_{i,j,k+1} \quad (2.a)$$

Step 2:

$$\left(A_k + \beta_\xi \vec{\delta}_\xi - \frac{1}{\alpha}\dot{\delta}_\xi A_i \bar{\delta}_\xi\right) f^n{}_{i,j,k} = g^n{}_{i,j} \quad (2.b)$$

Step 3: Correction factor C.

$$\left(\alpha + \vec{\delta}_\zeta\right) C^n{}_{i,j,k} = f^n{}_{i,j,k} \quad (2.c)$$

Steps in the finite differencing scheme.

Program VTWING
  Input **subroutine** (INPUT)
    READ mesh
    READ run-time parameters
  Initialization **subroutine** (INIT)
    initialize the solution
    compute and store metrics
  Flow Solver: (SOLVE)
    for each iteration **do**
      for each k-shell in mesh **do**
        get metrics
        compute density and density coefficients
        compute residuals
        solve for $g^n{}_{i,j}$ and $f^n{}_{i,j,k}$
      end k-loop
      calculate and apply $C^n{}_{i,j,k}$
      output maximum residual and correction for iteration
      check convergence
    end iteration loop
  output solution

Figure 8. Sequential structure of the TWING Program:

C(i,j,k) is computed in step three (equation 2.c). This calculation proceeds from the outer boundary down to the wing surface, requiring the solution of a bidiagonal system for each ζ line (i.e., ξ and η = constants, figure 7) of the mesh. This correction factor is then added to the solution from the previous iteration, generating a new solution. This three-step process is repeated iteratively until convergence is achieved or a preset maximum iteration is reached.

An outline showing the code structure itself is presented in figure 8. The program first reads the physical coordinates of the finite difference mesh and its run-time parameters. The program then computes the metric quantities defining the transformation of the problem into "computational space" and writes these to disk.

At this point, the main iteration loop of the program begins. The program completes steps one and two (equations 2.a and 2.b) of

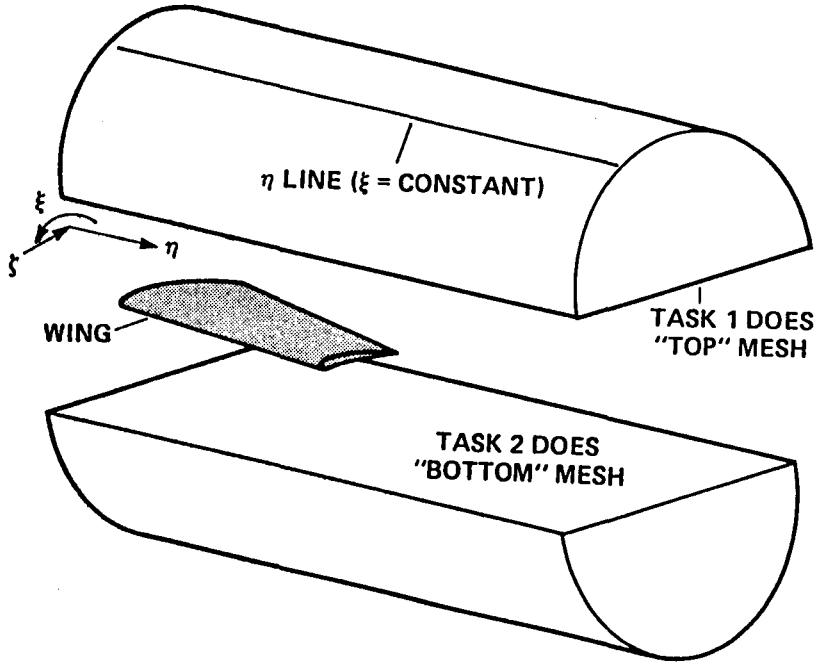

Figure 9. Computation divided in two different regions.

the three-step approximate-factorization scheme outlined above operating on successive "k-shells" in the mesh, beginning at the surface of the wing and progressing to the outer boundary. For each k-shell, the code:

(1) fetches the appropriate subset of metrics from the disk

(2) computes the density at each point

(3) generates the special density coefficients

(4) computes the residual terms resulting from equation 1.c

(5) solves for G(i,j) and F(i,j,k)

After completing this "k-loop," the code completes step three of the scheme by calculating the correction C(i,j,k) and applies it to each mesh point to generate a new solution. A convergence check follows: when satisfactory convergence is achieved, the final solution is written to disk.

The Modification of TWING

TWING is written in portable FORTRAN 66 and executes on Cray, CDC 7600, and VAX CPUs. The program was rewritten to be well-structured. Its control flow is serial (i.e., few GO TOs jumping control around). Although it was possible to partition the computation along functional lines in a sort of high-level *pipeline*, this approach was not pursued because it needs either substantial additional memory or elaborate internal buffering to store intermediate results. Pipelining may also hinder efficient execution-time load-balancing with some stages of a pipeline executing longer than other stages of the pipe.

This problem was exacerbated in TWING by the extensive use of EQUIVALENCE statements in the original code, employed in an effort to squeeze the largest possible problems into the limited core memory of a CDC 7600 or a Cray 1S. Since a functional partitioning of the problem seemed unsuited to the limited shared memory available, a static spatial-partitioning scheme was employed.

Our restructuring took advantage of existing code and attempted as little algorithm change as possible. In this scheme, each step in the algorithm was examined in an effort to determine if several portions of the mesh could be operated on simultaneously at that step. Execution profiling using the Cray FLOWTRACE facilities showed dominant run times in three SUBROUTINEs. Vectorized TWING executed three times faster than scalar TWING with input-output overhead included. Since distinct steps in the algorithm tend to correspond to separate modules in the finished code, this process resulted in a body of code that formed the skeleton of the concurrent processing portion of the modified TWING.

The calculations of the density (subroutine RO), the special density coefficients (subroutine ROCO), and the residuals (subroutine RESID) were all split along the η axis for each "K-shell" in the computational mesh (figures 6 and 7). This resulted in splitting loops (figure 10). One processor generated these results for points on or between the symmetry plane boundary and the wingtip. The other processor handled points on the wing extension, out to the freestream sidewall boundary. This "inboard-outboard" partitioning scheme was chosen because the algorithm employed in each of these calculations is usually constant for a given ξ line (η = a constant) but varied with position along the η axis. An inboard-outboard scheme was therefore constructed using processor-dependent *branches* such as:

```
C
      IF (TASKID .EQ. 2) GOTO 12
      DO 10 I = 1,NIM
      . . .
   10 CONTINUE
C     This continue added for multi-tasking
   12 CONTINUE
```

Mathematically, however, each point in the mesh was operated on independently during these preliminary calculations. We can replace the mesh with different divisions if there were reasons for favoring it.

A more fundamental relationship between the underlying mathematics of the algorithm and the spatial decomposition of the problem for Multiple-Instruction stream, Multiple-Data stream (MIMD) execution is illustrated by the three-step approximate-factorization scheme outlined in a previous section. Recall that in equation 2.a, the backward differencing is performed only about η, which generates tridiagonal linear systems along η lines (ξ= a constant). This makes the inboard-outboard partitioning scheme used above unworkable for this step.

Table 1. Execution Time Profiling

Subroutine	Vectorized TWING†, % Total Run Time	Scalar TWING % Total Run Time
RO	15.93	14.92
ROCO	13.45	15.91
RESID	23.92	17.99
Total %	53.3	48.82

†To clarify: this is not % of vector execution.

```
      . . .
C  Variables declared as integer, TASKID obtained from TSKVALUE.
      IF (TASKID .EQ. 1) THEN
         ROJSTART = 2
         ROJSTOP = NJTM
      ELSEIF (TASKID .EQ. 2) THEN
         ROJSTART = NJT
         ROJSTOP = NJM
      ENDIF
C  The values of NJTM, NJT, and NJM are preset parameters
C  in uni-tasked TWING.
      . . .
C  Now, each process works on the j-lines defined by
C  the initial assignment block.
C
      DO 20 J=ROJSTART1,ROJSTOP1
C     DO 20 J=2,NJM  -- old statement
        DO 15 I=1,NIM
        . . .
   15 CONTINUE
   20 CONTINUE
      . . .
```

Figure 10. Code illustrating the splitting of a loop.

However, adjacent η lines are computationally independent at this step, implying that the mesh could partition into "top" and "bottom" sections, each handled by a separate processor (figure 9). Similarly, in step two (equation 2.b), the backward differences are taken about ξ, generating tridiagonal linear systems along ξ lines (η = a constant) through the mesh. Here, each ξ line is computationally independent, and the resulting tridiagonal systems are solved concurrently by dividing the mesh into the inboard and outboard sections described in the last paragraph (see figure 6).

Finally, in step three (equation 2.c), bidiagonal systems are generated along lines in the ζ direction (ξ and η both = constants) (see figure 9). Again, concurrent processing of multiple ζ "pencils" is a simple and powerful way to use an MIMD machine at this step.

Note that true MIMD capacity was required to use such a spatial partitioning scheme. A vector architecture alone would not suffice because there was no guarantee that the instruction stream to be executed would be the same at different points in the mesh. Split difference schemes have sometimes proved useful. The wing root could

have used a more complex differencing scheme than employed near the outer boundary of a mesh. It is also possible that the values of some program parameters might also be position dependent.

Another code sequence commonly encountered in TWING was the selection of the maximum or minimum value in an array following an operation on the elements of the array. While this search has been conducted in a serial mode by the main program after the subprocesses return, this considerably degraded the resulting speedup. A better approach was to have each subprocess locate the maximum or minimum element in its portion of the data base, and pass the indices of this value back to the main program. The main program needed only to compare the two passed elements to obtain a maximum or minimum over the entire data base. An example of such a coding sequence is shown with in the next code section (figure 11) where numbered variables are TASK determined and nonnumbered variables are global shared variables.

VAX Modification

Our first MIMD testbed used two VAX 11/780 minicomputers linked to one MA780 multi-ported, shared memory unit. Because the operation of the processors was asynchronous, each with its own copy of the operating system running on a local clock, the configuration was best described as a "loosely coupled" multiprocessor. Although each processor retained its large virtual address space as local memory, the shared memory in the MA780 was not virtually addressable. Each MA780 unit could accommodate up to two megabytes of physical memory. The unit employed for this study was equipped with 256 kilobytes of physical memory.

The operating system in use at the time of the study was VAX/VMS (Version 3.1). VAX/VMS provides three facilities for interprocess communication across the shared memory link: event flags, mailboxes, and global data sections.

Event flags are allocated in thirty-two bit clusters and are manipulated using a variety of system-supplied routines. A process could set or clear individual flags and could wait for the logical AND or OR of a multiple flag mask. One drawback of VMS-event flag services for MIMD programming was that the flag operations were not indivisible (atomic). This can cause difficulties when an MIMD program uses shared memory. It required protection from simultaneous access by more than one process, especially if the number of competing processes is great. In the present study this problem did not arise, both because, at

```
IF (ABS(RMAX1) .GE. ABS(RMAX2)) THEN
  IF (ABS(RMAX1) .GT. ABS(RMAX)) THEN
     RMAX  = RMAX1
     IRMAX = IRMAX1
     JRMAX = JRMAX1
     KRMAX = KRMAX1
   END IF
ELSE IF (ABS(RMAX1) .LT. ABS(RMAX2)) THEN
  IF (ABS(RMAX2) .GT. ABS(RMAX)) THEN
     RMAX  = RMAX2
     IRMAX = IRMAX2
     JRMAX = JRMAX2
     KRMAX = KRMAX2
  END IF
END IF
```

Figure 11. Selecting a maximum value from two locally determined maxima.

most, two processes were active simultaneously and also because they generally operated on different parts of the statically partitioned data base.

The VAX/VMS system was not intended to be a multiprocessor operating system. Programming the shared memory was clumsy. Since our shared memory was small, we reduced the resolution of the program to fit the space of the memory. This was a development measure that did not happen on our Cray. This paper does not cover the VAX specific version in any greater detail.

The other MIMD testbed consists of a Cray X-MP/22 running version 1.13 of the Cray Operating System (COS). The Cray, by way of contrast, is a "tightly coupled," shared memory multiprocessor. This creates problems not faced on our VAX testbed such as more memory contention but simplifies programming.

Cray Modifications

The VAX version of TWING was a "stripped-down" version of the production Cray code designed to fit into the small shared memory system. We, therefore, did not count on the VAX version to reach convergence. The mesh was too coarse, and we did not get a chance to truly debug the VAX version. The mathematical basis for partitioning the vector version of TWING (VTWING) was identical to the VAX-specific version. This time, we sought realistic convergence. Debugging was a major problem not only for TWING, but also for the new STACK allocation and multi-tasking of the CFT compiler we were testing.

One important side step, was a quick set of checks regarding the new SUBROUTINE linkages. We should mention this was not a problem for TWING. To do this, a user compiled the complete, existing program using the ALLOC=STACK option on the new CFT compiler. The program was then run using the associated new loader given adequate stack and heap sizes (see the manual) (Research, 1985). The results were compared to the original STATICally compiled run. A useful variation of this was to create simple TASKs that START then immediately WAIT as a CALL to a SUBROUTINE would:

change from:	to:
CALL RO	**CALL** TSKSTART(TA,RO)
	CALL TSKWAIT(TA)

The timing differences between STATIC and STACK runs are included in the section on **Performance.** The compiler changes do effect program execution without source code changes.

The next stage entailed converting the existing code into a multi-tasking body of code. This was not as easily as it appeared as subtle errors required detection and correction. It is possible to do this at different levels or stages such as converting the entire program, converting subroutines, or converting blocks of code. Converting a code in large sections is like writing a large program and expecting it to run correctly the first time.

It was important to have good comparison data, since fast execution did not imply correct execution. A machine-readable output was created from an unmodified, running version of TWING. Once the code was running, we tested the output of the multi-task run with our uni-task output using a differential file comparator (the UNIX[7] *diff* program). This insured that our conversion was precise.

Our third and last attempt at conversion was to break a subroutine into two smaller subroutines: a parallel portion and a serial portion. Since most of the data was stored in COMMON blocks, parameter passing was minimized to simplify these problems. The parallel subroutines were run and synchronized before the serial portion as shown in Figure 12.

Portions of serial subroutine code (typically loops) then migrated to the parallel subroutines. This technique successfully identified subscripting oversights, branching problems, and so on. It was painfully slow, but it was effective. Initially, task synchronization was performed using TSKSTART and TSKWAIT, not the more complex EVENT flags. We used the "Make it right before you make it faster" philosophy from the *The Elements of Programming Style (Kernighan, 1978).*

We stress the following point: make certain that the existing code is bug-free. There

[7]UNIX is a trademark of AT&T Bell Laboratories.

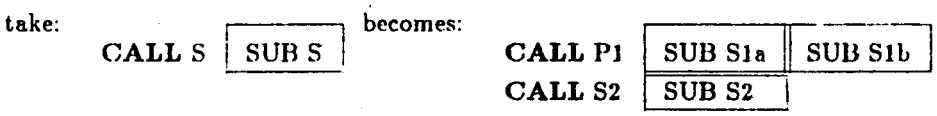

Figure 12. Code migration from serial into parallel, where S is the subprogram, the numbered portions refer to the halves (1 and 2) of S. P1 represents the set of CALLs that are invoked for parallel TASKs S1a and S1b.

is little sense trying to multi-task bug ridden code. Multi-tasking the code made programs harder to debug. The programmer has to distinguish the original bugs from the newly introduced linkage and multi-task bugs.

Each SUBROUTINE was individually converted to two parallel TASKs giving three versions of the program. The next step was to get combinations of two different TASKs running within a program. This was used to locate side effects between any two different TASKs. We still used the crude START and WAIT CALLs at this point. Finally, we had all three CALLs converted.

Once all TASKs were operating using crude synchronization, it was a simple matter to get barrier synchronization using EVENTs. We moved one TASK at a time to EVENT structures. After EVENTs replaced the TSKSTART and TSKWAIT CALLs, we wrote a simple user-level TASK scheduler (figure 13) that worked on simple message-passing.

Our last act scaled the grid from VAX shared memory-size to Cray memory, production size. During this final work, we corrected one VAX-scale dependency problem. This problem involved a partial correctness proof mentioned further in the section on **Debugging.**

Time and Effort

This work took several months. We reported our many compiler problems to CRI. Meanwhile, Cray Research migrated from CFT Release 1.13 to 1.14, solving many of our problems.

To reiterate the degree of change, TWING actually consisted of two separate programs[5]: a grid generator and the vectorized version of the TWING flow solver. The multi-tasking took place only on the flow solver.

We document the GRIDGEN program here only for completeness. The GRIDGEN program consisted of

2123 total lines of commented FORTRAN
1195 executable lines of code in
1031 executable statements

An instrumented uni-task version of the TWING solver consisted of

3926 lines of commented code
3840 lines without instrumentation
2529 total executable lines
1906 executable statements

An instrumented multi-task version of TWING came to

4450 lines of commented code
4399 lines without instrumentation
2870 lines total executable
2188 executable statements

Note that additions and modifications do not sum to the totals because there is overlap. Additions and modifications took the form of replication and addition of statements to handle problems such as parameter passing.

Our experience with converting this and other NASA codes [LES and ARC3D] currently has us modifying about 10% of the code (if the grid generator is counted, slightly more if not). Most of these codes have fewer

[5]The two programs are combined as one for machines with large memory.

```
...
MESG = 1
CALL SCHED
...
SUBROUTINE SCHED        SUBROUTINE PROCES
...                     ...
CALL EVPOST(GO)         CALL EVWAIT(GO)
...                      IF(MESG.EQ.1) THEN
CALL EVWAIT(DONE)       CALL RO
...                     ...
                        CALL EVPOST(DONE)
END                     ...
                        END
```

Figure 13. Structure of our simple scheduler.

loops split across processors compared to TWING. We split a total of 19 loops in three SUBROUTINEs. This includes new code for loop splitting, new per-process branches, TASK-EVENT creation and control code, and a small TASK scheduler. About 210 lines of control flow code were added (excluding comments). 70 more lines were replaced or modified into 160 lines to handle problems of parameter passing, or changes to array indices.

During the development of each TASK, good version control proved useful. A good tool requires parallel branching versions; linear version control such as UPDATE was not adequate. Maintaining the successful, intermediate stages of multi-task TWING made debugging and scale-up easier through the isolation of changes. It was always possible to easily fall back to some parallel, executable code.

Debugging

Sequential debugging is generally regarded as a *black art*. Bugs occur during compile-time and run-time: with the latter, the non-fatal ones are the hardest to find. The basic techniques for debugging are categorized into: 1) traces, 2) snapshots or 3) dumps. These techniques have problems in multiprocess environments lacking consistency or having deadlock. Multi-task debugging is plagued by a lack of reproducibility, synchronization, and good tools. The literature on run-time debugging in multiprocess environments is scarce (Model, 1979) and more work is needed in this area.

Numerous users tell us to "force multi-task execution into a single stream of execution[9]" as if simple user-controlled reduction would solve hazard problems.

This does not help!

Normal debugging depends on a machine being in a reasonably consistent state. A multi-task program crash may not occur at the same location as with a uni-task program. This is true for uniprocessors executing multi-task code as well.

Consider a simple example to illustrate the conceptual difficulties of debugging using the CFT traceback facility. A program creates a *child* TASK. When the child TASK dies, should the traceback trace through the point where the child process began, or should it trace through the synchronization routines (if any)? The tangled nondeterministic web makes this decision difficult. There are situations where one trace is preferable over the other. One condition is when the child dies because of the actions of its *parent* or *sibling* processes [side effects]. So, traces are not simple. What about snapshots?

[9]This is accomplished using the TSKTUNE call and setting the MAXCPU parameter to '1.'

Inserting WRITE statements into programs might not help. First, the execution order of these statements may vary (e.g., nondeterminism). Second, I/O is another shared resource, and the user must have LOCKs that protect that resource like any other shared resource

One surprising effect of inserting WRITE statements at key points was the migration of bugs from one location to another! We solved this debugging problem by modifying our technique of migrating code between serial and parallel development subroutines. Our new technique was to remove data structures and code immediately following the breakage point to isolate program and compiler bugs. This sometimes worked to locate bugs. The problem at this point becomes: is the program crashing because of the original bug or the bugs introduced by cutting code?

In the I/O locking process, it would help users debug codes if the system could hide I/O locking details from users. Better yet, a small library of simple routines would help. It should have traceable ERROR and ASSERTION routines. If a user resorts to adding WRITE statements to follow the execution of a program, the user should have a similar trace of a serial code for advanced comparison. A simple filter could take a source program and insert a WRITE with the subprogram's name. More elaborate and more powerful debugging tools would also help.

Dumps, the method of last resort, are frequently less consistent than traces or breakpoints. We avoided dumps at all cost.

One technique tried in the latter stages of multi-task conversion was program proving. Toward completion of program scale-up, we had a tricky change to a SUBROUTINE call. Precondition and postcondition assertions were compiled surrounding critical code changes. Proof techniques had limitations in a parallel environment, but it was a useful technique for checking changes. Program proving was not regarded as a cure-all and was regarded as controversial.

The last set of problems involve synchronization and timing. A new diagnostic message for first-time multi-tasking programmers is compressed from a real CRAY job in figure 14. *Race conditions* occur whenever two or more TASKs or processors are sharing data (or code). This is the time when *deadlock* can occur. There are no general solutions, but there is a mountain of research literature. Multi-tasking CFT provides limited deadlock detection and traceback. Keeping TASK scheduling and timing constraints simple is currently the best way to avoid deadlock. The most difficult deadlock problems should occur when there are indirect deadlocks.

Testing Multiprocessor Outputs

A running multi-task program was not enough; we sought numerical results identical to our uni-task TWING. There were many occasions where our program ran to completion, but our numbers did not agree at lesser digits of precision. A standard file comparator was used to test output between TWING runs. The importance of tools such as a good file comparator was not underestimated. A single, incorrect, boundary subscript could "poison" an entire array. Testing asynchronous methods [e.g., chaotic relaxation] is more difficult.

Fortunately, our program is completely synchronous. However, newer asynchronous, chaotic algorithms remove the consistency assumption and approximate a solution. If such asynchronous methods are used, file comparator programs are completely inadequate. Better comparison tools are needed. Output testing tools must approximate floating-point comparisons within a specified tolerance.

The Cray multi-task version of TWING had proved our concept by reaching convergence with results identical to a uni-task version of vectorized TWING.

Other Generally Useful Tools

While mentioning debugging tools, we should also mention other generally useful tools. Among these we could include tools to search for STATIC allocation and data dependence. Data dependence tools can also provide help when recursion is added to FORTRAN. A good cross-referencing tool could aid this search process. Other tools could possibly identify linkage problems. Such tools are useful in the analysis and compilation phases of development. All these programs should execute independently (i.e., from a compiler) in the style of other good software tools.

```
USER UT024 - DEADLOCK - ALL USER TASKS WAITING FOR LOCKS OR EVENTS
USER TB001 - BEGINNING OF TRACEBACK
USER       - $TRBK   WAS CALLED BY UTERP%  AT 1715731a
USER       - UTERP%  WAS CALLED BY $SUSTSK% AT 1705627a
USER       - $SUSTSK% WAS CALLED BY EVWAIT  AT 1701511b
. . .
USER TB002 - END OF TRACEBACK
```

Figure 14. A frequent error message for new users of multitasking.

Performance and Execution Behavior

The measurement of parallel programs is conceptually complicated by several factors. The Cray measurement facilities, if used, record the length of all parallel execution traces as if they were measured sequentially. For instance, two cycles run in parallel take one cycle to execute, but they are still counted as two cycles. The Cray documentation (Research, 1985) notes that flow tracing facilities do not work properly with multi-tasking environments. We resort to the direct use of the system real-time clock and flow tracing of the uni-task version of TWING to give us run-time characteristics.

There are no standard metrics for determining multiprocessor performance improvement. The most common in use is [simple] *speed up* defined by:

$$Simple\ speed-up = \frac{Serial\ execution\ time}{Parallel\ execution\ time}$$

The simple speed up of TWING is illustrated in the next table.

Another conceptual measurement problem is where and how measurements are taken. We simply throw two CPUs at a problem, so the maximum simple speed-up is one-half the total serial execution time. I/O wait time is a significant portion of the program that cannot multi-task. We recognize that we don't use two CPUs for the entire time: we have serial code, and we have wait-time for TASKs to finish and synchronize. Also, we need more cycles to cover overhead.

Since we were able to multi-task only 50% of the total serial execution, the best improvement we could gain would be 25% of total execution. We might term this performance figure as *proportional*, simple, speed-up. As we multi-task more code, this figure should slowly increase.

Still another problem is that with two or more CPUs sharing common resources - memory and I/O - collisions become inevitable. Processors are forced to wait, and this expends more overhead cycles. This contention is visible when running a uni-task version of the code in one processor, and running a second code in another processor. By varying the work load in the second program between a CPU intensive versus memory intensive JOB, we can see the simple, but significant effects of memory contention (See the Table below). These are interference effects not found on uniprocessors. A problem arises in shared memory multiprocessors such as on our VAX and Cray that local memory multiprocessors do not have. Memory contention significantly slows down memory performance. Designers of future multiprocessors must balance processor- versus memory-performance rates.

Another performance issue is the addition of overhead cycles required to control TASKs. Figure 15 shows the cost in cycles versus the iterations toward solution for our VAX version. This cost occurs similarly on the Cray.

Load balancing is a significant problem since TASKs vary in work load, and we have seen that measurement of load has problems. The output from the Cray day files shows a considerable imbalance of work. Cray tools discovered that TASK 1 did more work (executed significantly longer) than TASK 2. The timing output of a single day-file illustrates the difference on our two CPU system:

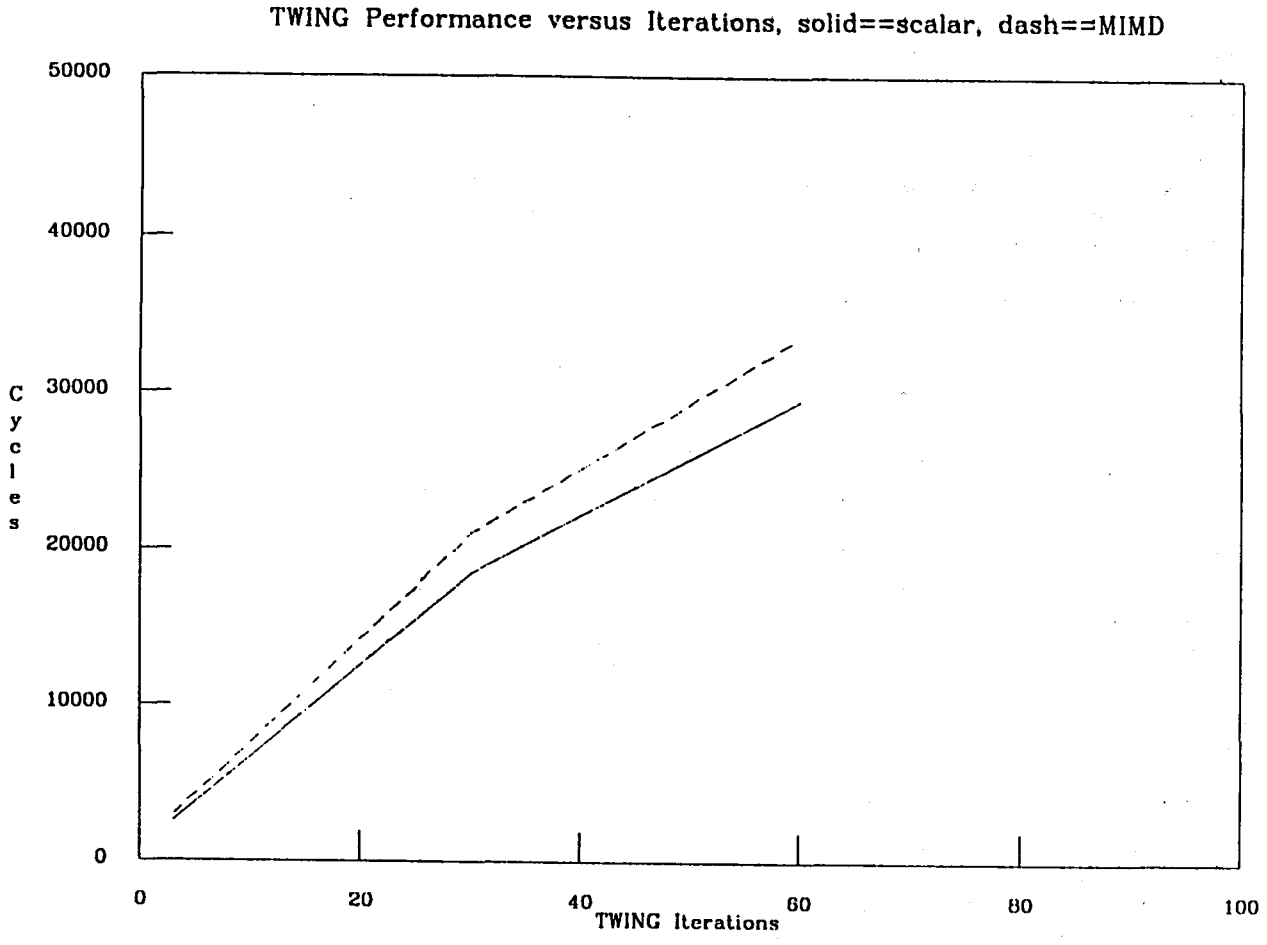

Figure 15. Graph showing the additional overhead (near linear) between sequential versus parallel code versions.

Table 2. TWING Execution time

CPUs used	STATIC Compile one	STACK Compile one	Multi-tasked two	Speedup **
Real [wall] time	7.76	7.17	7.10	10%
Total system time	7.36	6.76	9.67	n/a‡
Input	0.0244	0.0255	0.0263	n/a‡
Init	0.250	0.210	0.211	n/a‡
Solve	7.08	6.52	9.43	n/a‡
RO	1.25	1.06	1.23	2%
ROCO	0.985	0.891	0.886	10%
RESID	1.75	1.73	1.42	20%

‡not applicable: sequential FLODUMP timings added only for completeness: the difference in wall clock time versus what the operating system reports.

Table 3. Uni-task TWING Execution Times (in Seconds)

	Low Memory Contention		High Memory Contention	
	STATIC compile	STACK compile	STATIC compile	STACK compile
Real [Wall] Time	7.94	7.36	8.49	8.04
Total-System Time	7.35	6.76	8.03	7.35
Subroutines:				
Input‡	0.0244	0.0255	0.0264	0.0276
Init‡	0.250	0.210	0.270	0.225
Solve	7.08	6.52	7.73	7.09
RO	1.26	1.06	1.35	1.13
ROCO	0.995	0.893	1.07	0.970
RESID	1.77	1.74	1.91	1.89

‡These SUBROUTINEs were not converted to use multi-tasking. They are included here for control reasons to show the effect of changing to a STACK compilation.

TASK	CP TIME
1	11.85
2	4.83

This is because the work areas were not partitioned evenly between the two TASKs based on hand analysis of array proportions. Work was partitioned based on existing, somewhat lopsided DO-loop parameters in three-dimensional arrays.

To change these parameters would require more computation and potentially further array-subscript change. Additional algorithm modifications are required for boundary regions. Dynamic load balancing is harder still.

Discussion

Further Research

This research has not covered other forms of multiprocess partitioning. Pipelines are a common proposal: easily constructed and debugged, but difficult to tune or load balance. (See **Scale-Up.**) The program's author (Thomas) is considering this approach, but it requires extensive rewriting.

Micro-tasking is another Cray-proposed multiprocessing construct (Booth, 1985). Micro-tasking involves a simpler, more restrictive set of control primitives. Another important issue is the area of scale-up (See next section).

Scale-up

Certain aspects of scaling up programs are trivial. Increasing problem size is not typically a problem: our VAX case was not a necessary prerequisite to move the program to the Cray. Adding more processors, however, is not trivial. The work on the TWING code began before there was any consideration of generalizing the program to use more than two processors.

The current multi-task work on TWING will not generalize to an n-processor case. The code used to determine maxima is one problem that will not easily scale. If more than two processors are used, different partitioning schemes become preferable.

Probably the key issue of multi-tasking is whether the performance gained was worth the effort expended. There is a conflicit (or tradeoff) between the need to have large multi-task sections for performance and small multi-task sections for ease of development and debugging.

The multi-tasking programmer must also confront the need to have large protected critical sections and many asynchronous processes running. Our scale-up of the code uncovered many machine-dependent assumption problems. For the scale-up of code, the parallel-serial divide-and-conquer approach again worked.

Open Issues

The problems of automatic partitioning are not addressed in this study. Our future intent is to extend FORTRAN by using a simple preprocessor to add support for simpler constructs (e.g., COBEGIN, COEND) like Cray micro-tasking. The preprocessor should ideally hide low-level details and machine dependent processing. It is tempting for programmers to be parochial about particular constructs, so we wish to avoid this by using preprocessors. Similar research is under study on different architectures at other sites (e.g., LANL, ANL, Bell Labs, CMU, U. of Ill.).

There are dozens of issues left open: different synchronous and asynchronous algorithms, translation into an intermediate language for dataflow-style execution, measurement and load balancing. Parallel processing has many difficult problems remaining which will take years to research.

Conclusions

The introduction of parallelism is as significant a tool as either Cray multi-tasking or micro-tasking. The problems of parallelism are not new. They are typically thought to inhabit that realm called *systems programming*. Users intending to add parallelism to their collection of tools are advised to learn from experience of others.

Good software tools would help programmers. These tools must provide multiprocessing support. Many programmers would probably desire a standardization of multiprocessing syntax, but this is premature.

Programmers should recognize that with adding parallelism and achieving better performance, there will come some loss of the coherent sequence that makes sequential programming such a powerful tool.

Programs designed to use parallelism from their inception are more likely to use parallelism efficiently. This was clearly the case with the introduction of vectorization, i.e., vector-designed programs tend to use vectors more efficiently. We should soon see more multi-task programs, but it is an open question whether these programs are scale-able into the hundred- and thousand- (proposed) processors range.

Acknowledgements

Michael Johnson deserves special recognition for doing the preliminary VAX work (Johnson, 1983). Scott Thomas and Alan Fernquist provided valuable assistance with

uni-task TWING. Ken Stevens, Eric Barszcz, and Cathy Schulbach assisted with debugging. Dave Robertson of Zero-One relayed our many CFT problems back to Cray Research.

References

[Larson, 1984]
John L. Larson. "Multitasking on the CRAY X-MP-2 Multiprocessor." *Computer* **17**(7), pp. 62-69 IEEE, (July 1984).

[Research, 1985]
Cray Research, Inc., "Multitasking User's Guide," Technical Note SN-0222, Rev. A (January 1985).

[Research, 1984]
Cray Research, Inc., "Multitasking and the X-MP," Technical Note (January 1984).

[Dennis, 1979]
Jack B. Dennis, "The Varieties of Data-Flow Machines," *1st International Conference on Distributed Computing Systems*, , pp. 430-439 IEEE, (October 1979).

[Denning, 1985]
Peter J. Denning, "The Science of Computing: Parallel Computing," *American Scientist* **73**(4), pp. 322-323 (July-August 1985).

[Jones, 1980]
Anita K. Jones and Edward F. Gehringer, eds., "The Cm* Multiprocessor Project: A Research Review," CMU-CS-80-131, Carnegie-Mellon University, Pittsburgh, PA (July 1980).

[Grit, 1983]
Dale H. Grit and James R. McGraw, "Programming Divide and Conquer on a Multiprocessor," UCRL-88710, Lawrence Livermore National Laboratory, Livermore, CA (May 1983).

[Jones, 1980]
Anita K. Jones and Peter Schwarz, "Experience using multiprocessor systems — a status report," *Computing Surveys* **12**(2), pp. 121-165 (June 1980).

[Kuck, 1980]
David J. Kuck, Robert H. Kuhn, Bruce Leasure, and Michael W ..e, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Computer Software and Applications Conference (COMPSAC80)*, , pp. 709-715 IEEE, (October 1980).

[Andrews, 1983]
Gregory R. Andrews and Fred B. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys* **15**(1), pp. 3-43 (March 1983).

[Dijkstra, 1968]
E. Dijkstra, Multiprogramming System"" "The Structure of the "THE" Multiprogramming System," *Communications of the ACM* **11**(5), pp. 341-346 (May 1968).

[Hoare, 1974]
C. A. R. Hoare. "Monitors: An Operating System Structuring Concept," *Communications of the ACM* **17**(10), pp. 549-557 (October 1974).

[DOD, 1980]
DOD, *Reference Manual for the Ada Programming Language*, U.S. Department of Defense (July 1980).

[Dijkstra, 1968]
E. Dijkstra, "Go To Considered Harmful," *Communications of the ACM* **11**(3), pp. 147-148 (March 1968).

[Fong, 1984]
Kirby Fong, *Personal Communication*, LLNL, Magnetic Fusion Energy Computer Center (1984).

[Thomas, 1983]
Scott D. Thomas and Terry L. Holst, "Numerical Computation of Transonic Flow About Wing-Fuselage Configurations on a Vector Computer," *AIAA 21st Aerospace Sciences Meeting*, (January 1983).

[Kernighan, 1978]
Brian W. Kernighan and P. J. Plauger, *The Elements of Programming Style, 2nd edition*, McGraw-Hill, New York, NY (1978).

[Model, 1979]
Mitchell L. Model, "Monitoring System Behavior in A Complex Computational Environment," CSL-79-1, Xerox Palo Alto Research Center, Palo Alto, CA 94306 (January 1979).

[Booth, 1985]
Mike Booth, "Microtasking Presentation," Internal Report, Cray Research

Inc.,
Dallas. TX (1985).

[Johnson, 1983]
Michael S. Johnson, "Modification of the TWING Full Potential Code for Execution on an MIMD Computer," Tech. Memo., NASA Ames Research Center, Moffett Field, CA (1983).

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-88200		

4. Title and Subtitle	5. Report Date
EXPERIENCES WITH CRAY MULTI-TASKING	November 1985
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
E. N. Miya	A-85424
	10. Work Unit No.

9. Performing Organization Name and Address	11. Contract or Grant No.
Ames Research Center Moffett Field, CA 94035	
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Technical Memorandum
National Aeronautics and Space Administration Washington, DC 20546	14. Sponsoring Agency Code 505-37-01

15. Supplementary Notes

Point of Contact: E. N. Miya, Ames Research Center, MS 233-14, Moffett Field, CA 94035 (415)694-6453 or FTS 464-6453

16. Abstract

This paper covers the issues involved in modifying an existing code for multi-tasking. These include Cray extensions to FORTRAN, an examination of the application code under study, designing workable modifications, specific code modifications to the VAX and Cray versions, performance, and efficiency results. The finished product is a faster, fully synchronous, parallel version of the original program.

A "production" program is partitioned by hand to run on two CPUs. Loop splitting (performed manually) multi-tasks three key subroutines.

Simply dividing subroutine data and control structure down the middle of a subroutine is not safe. Simple division produces results that are inconsistent with uniprocessor runs. The safest way to partition the code is to transfer one block of loops at a time and check the results of each on a test case. Other issues include debugging and performance. Task startup and maintenance (e.g., synchronization) are potentially expensive.

Future research considerations involve the development and integration of a FORTRAN preprocessor for higher-level, explicit control of multi-tasking. Despite these problems, the partitioning of certain pre-existing programs looks promising.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Parallel processing Program decomposition Software engineering Algorithms	Unlimited Subject category - 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	24	A02

*For sale by the National Technical Information Service, Springfield, Virginia 22161

End of Document