1974

Parallel Algorithms for Adaptive Quadrature III - Program Correctness

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
74-112

PARALLEL ALGORITHMS FOR ADAPTIVE QUADRATURE III -

PROGRAM CORRECTNESS

John R. Rice
Computer Science Department
Purdue University
West Lafayette, Indiana 47907

March 1974

CSD-TR-112

(REVISED DECEMBER, 1974)

1. INTRODUCTION.

This is the third of a sequence of papers on parallel algorithms for adaptive quadrature. The primary aim is to study the rate of convergence achieved by such algorithms. The speed-up achieved by parallelism has been a secondary topic but will be the primary topic of further studies.

Our goal is to prove that a specific algorithm (computer program) achieves a certain rate of convergence. The proof is developed in a top-down approach with three levels. The first [3] is a convergence theorem valid for all algorithms represented by a general metalgorithm. This theorem is very much like the traditional mathematical theorems of numerical analysis. The second level [4] involved a much more specific metalgorithm with 32 detailed attributes assumed. It is shown that any algorithm represented by this metalgorithm achieves the rate of convergence established by the first level theorem. A significant change in the nature of the second level theorem is from mathematical convergence to algorithmic convergence. Thus it is shown that any algorithm from this metalgorithm will terminate with a quadrature estimate accurate to within a prescribed input requirement. The amount of computation (measured in integrand evaluations) required is given by the convergence result. The present third level presents a specific computer program (for a hypothetical computer described later) and shows that it has all the 32 attributes assumed by the second level metalgorithm. We then conclude that the convergence result applies to this specific program.

It is important to note that the convergence result established is exceptionally strong and illustrates the surprising power of adaptive

---

This work was supported in part by Grant GP 32940X from the National Science Foundation.

quadrature. Results of this type were first established in [2] and say, roughly, that adaptive algorithms integrate functions with a finite number of singularities as efficiently as comparable traditional numerical methods integrate smooth functions. See Sections 5 and 6 for a precise technical statement.

Note that the convergence theorem established requires as a part of its proof a proof that the program is correct. The approach to proving program correctness used here is the one traditional to mathematics. We first identify the obvious and not-so-obvious arguments involved. We then state that the obvious arguments are, in fact, obvious and present detailed explanations for the not-so-obvious ones. Since we must establish 32 attributes of a longish program a complete proof would be too long and too boring to present. Thus we assume the reader becomes familiar enough with the program so that he can recognize those facts about it which are obvious. Further comments about the proof are made at the end of the paper.

The program is written in a pseudo-Fortran and is believed to be unambiguously defined. The non-standard Fortran constructions used are described in the program comments.

The hypothetical computer for executing this program has a number of general purpose processors capable of executing an arbitrary Fortran program. We make the following specific assumptions about this computer:

1. The arithmetic is exact.

2. The size of memory is unlimited.

3. All processors operate at the same speed; in one unit of time (called a statement) they can execute one Fortran statement of arbitrary type. Substatements of a statement are each counted separately.

Thus

```
IF(X.EQ.4.2) THEN Y=X,GO TO 5
   ELSE  X = COS(DX+Y**.42)/(7.1*X+3.2*ALOG(DX+.1)) + X,
         DX = AMAX1(DX,Y**.42)
```

requires three statements of time to execute: one for the test and two for whichever clause is executed.

A crucial element of any parallel program is the control of access to critical information which in this case is the interval collection and the area and bound estimates. The access mechanism used in this program depends essentially on the timing of certain segments of code. While the above assumption about the execution time is obviously unrealistic, it serves the purpose here. In any real parallel computer one would make adjustments in the mechanism based on the actual execution times for the relevant code segments.

The next section presents the program PAFAQ (Parallel Algorithm For Adaptive Quadrature) and the metalgorithm from [4]. The objective is to show that PAFAQ is represented by this metalgorithm. Section 3 contains a set of obvious or easy results. Section 4 presents the analysis of the parallel execution features of the program and Section 5 presents the numerical analysis of bounds and area estimation. The final section has the main results and some discussion of their implications.

2. THE METALGORITHM AND THE PROGRAM PAFAQ.

For the sake of completeness we reproduce the metalgorithm of [4]. That is then followed by the program PAFAQ.

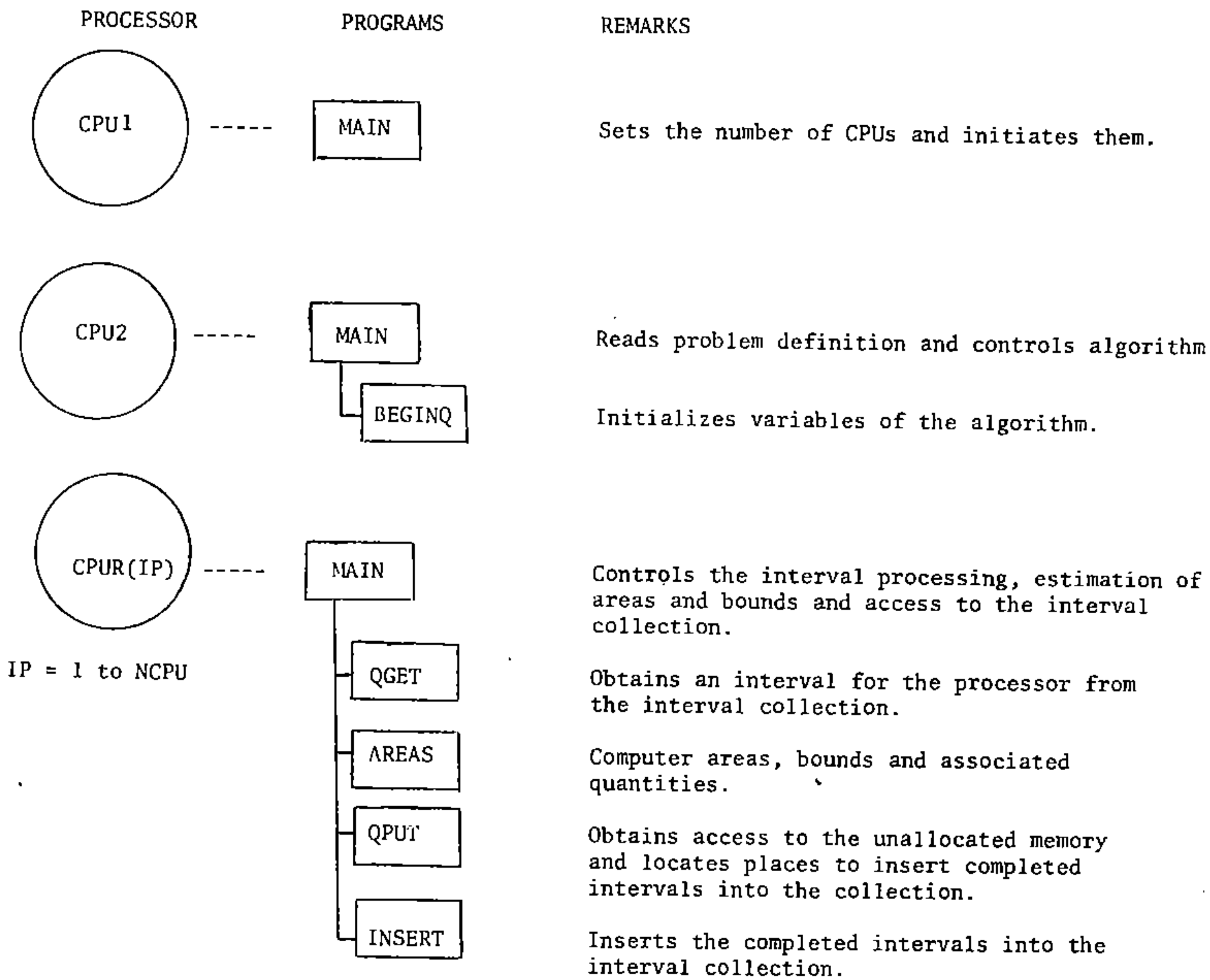


Figure 1. A schematic diagram of the parallel metalgorithm for adaptive quadrature. The components are described in more detail in [4].

We now list the 32 specific attributes assumed for the programs represented by this metalgorithm.

A. Attributes of MAIN - CPU1.

1. Assigns the value of NCPU.
2. Enables the other CPUs.
3. Initializes all control variables to be false and all numerical variables to be zero.

B. Attributes of MAIN - CPU2.

1. Obtains the variables that define the problem.
2. Initially invokes BEGINQ.
3. Monitors BOUNDA and terminates the algorithm (with output) when BOUNDA < EPS, when there is a memory overflow or when there are no more active intervals.

C. Attributes of BEGINQ.

1. Places the interval [A,B] into the interval collection, computes all associated values and initializes the collection properly.
2. Initializes variables for control of access to the interval collection.
3. Its final statement enables the other CPUs to proceed by designating the interval [A,B] as "free".

D. Attributes of MAIN - CPUR(IP). Once this CPU is activated it executes the following sequence of actions:

Invoke QGET

Invoke AREAS

Invoke QPUT

Invoke INSERT

Return to the top of this list

E. Attributes of AREAS.

1. Computes changes in AREA and BOUNDA. The resulting values of AREA and BOUNDA satisfy certain requirements (e.g. Assumptions 1 of [2]) provided F(x) satisfies certain requirements (e.g. Assumptions 2 of [2]).

2. Uses a proportional error distribution for BOUNDA and implements the restriction that the interval length be less than CHARF before BOUNDA is allowed to be less than EPS.

3. Determines how many, if any, intervals are to be discarded and identifies them.

4. Computes the variety of information about the two intervals that are obtained. This information, along with the other information generated, is temporarily placed in the memory PROCESSORS and associated with this CPU.

5. There are no unbounded computations in AREAS and its maximum execution time is bounded by a constant. It is the only program of CPUR(IP) that evaluates F(x) and it does this at most q times.

F. Attributes of INSERT.

1. Once places have been assigned in QUEUE by QPUT, it places all the relevant information about the new intervals into these places in QUEUE.

2. Prevents an interval from being assigned to another CPU before its insertion into the collection is complete.

3. There are no unbounded computations in INSERT and the maximum execution time is bounded by a constant.

G. Attributes of QGET.

1. This program gains sole access to an interval in the collection that is free to be assigned to a CPU. If the interval to be assigned is not free, then QGET waits in an idle loop.

2. Once access is gained to an interval, it is assigned to CPUR(IP) and so identified, and not assigned again. A new interval is designated as next to be assigned.

3. At most NCPU-1 CPUs gain access to the interval collection between the time a particular one tries for and the time it achieves access to the interval collection.

4. There is no conflict between QGET and QPUT.

5. Does not affect information about the interval itself, only about the interval's status in the algorithm.

6. No interlock occurs when more than one CPU is executing QGET and, in such a case, one of them gains access to the interval collection within a fixed time.

H. Attributes of QPUT.

1. This program gains sole access to the unallocated or available memory in QUEUE. It waits in an idle loop until this access is achieved.

2. Obtains places in the available memory of QUEUE for the new intervals to be returned and assigns these places to the interval collection. It updates the information about the available memory in QUEUE.

3. At most NCPU-1 CPUs gain access to the available memory between the time a particular one first tries and the time it achieves access to the available memory.
4. While it has access to the available memory it updates the values of AREA and BOUNDA. Thus access to the available memory is required and made even if both new intervals are discarded.
5. If the interval collection is empty when this CPU is obtaining places for the return of intervals to the collection, then QPUT designates one of the returned intervals as the next one to be assigned.
6. There is no conflict between QGET and QPUT.
7. Does not affect information about the interval itself, only about the intervals' status in the algorithm.
8. No interlock occurs when more than one CPU is executing QPUT and, in such a case, one of them gains access to the available memory within a fixed time.

```
C          PARALLEL ALGORITHM FOR ADAPTIVE QUADRATURE
C                            BY
C              JOHN R. RICE, PURDUE UNIVERSITY
C
C     THIS ALGORITHM IS WRITTEN IN A PSEUDO-FORTRAN IN ORDER TO MAKE IT
C  MORE UNDERSTANDABLE AND CONCISE.  THE CHANGES FROM FORTRAN ARE SUCH
C  THAT A TRANSLATION INTO FORTRAN CAN BE MADE IN A MECHANICAL WAY. THE
C  CHANGES USED ARE LISTED BELOW.
C
C  1.  GLOBAL VARIABLES --- ARE DECLARED AS A SEPARATE . . . . .
C            OF STATEMENTS CALLED -- DECLARE   IS DEFINED AND THE . . .
C            SIMPLY HAVE THE STATEMENT
C                  ##CALL DECLARE
C            TO NOTE THAT THESE DECLARATIONS ARE IN . . .
C            VARIABLES NOT DECLARED GLOBAL ARE LOCAL TO EACH INVOCATION
C            OF A SUBPROGRAM.
C
C  2.  IF-THEN-ELSE   -- IS USED (NOT COMPOUNDED) IN THE NATURAL WAY AND
C            ALLOWING MULTIPLE STATEMENTS IN EACH CLAUSE.  COMMAS ARE
C            USED TO SEPARATE STATEMENTS UNLESS AN AMBIGUITY RESULTS, IN
C            WHICH CASE STATEMENTS ARE SEPARATED BY LINE ENDINGS. E.G.
C
C               IF( K .EQ. 3 ) THEN  J=0, NK=NK+1, GO TO 40
C              *                   ELSE  K=K+1
C
C            THE WORDS THEN AND ELSE ARE RESERVED KEYWORDS
C
C  3.  ARBITRARY EXPRESSIONS ---- ARE ALLOWED AS ARRAY INDEXES, DO-LOOP
C            RANGES, ETC.
C
C  4.  FORMAT FREE I/O --- IS ASSUMED.  THERE IS VERY LITTLE I/O.
C
C  5.  MULTIPLE ASSIGNMENT STATEMENTS --- ARE ALLOWED.
C
C  6.  VARIABLE DIMENSIONS ARE USED FOR ARRAYS
C            LIMQ   = FOR THE INTERVAL COLLECTION INFORMATION
C            LIMCPU = FOR THE CPU DEPENDENT INFORMATION
C
C
C
C     ************  ALGORITHM STRUCTURE  ************
C
C     PROCESSOR     ASSOCIATED
C       NAME         PROGRAMS                  REMARKS
C
C     CPU1 ---- -- MAIN            THIS IS THE OPERATING SYSTEM. IT ASSIGNS
C                                  THE PROCESSORS AND PASSES CONTROL TO THE
C                                  ALGORITHM ITSELF.
C
C     CPU2 ---- -- MAIN            THIS PROGRAM CONTROLS THE COMPUTATION
C
C                  BEGINQ          THIS PROGRAM INITIALIZES THE ALGORITHM
C                                  ONCE CONTROL IS PASSED TO CPU2
C
C     CPUR(LIMCPU) MAIN            THIS IS AN ARRAY OF PROCESSORS WITH AN
C                                  ARRAY OF IDENTICAL PROGRAMS.  THIS
C                                  PROGRAM CONTROLS THE PROCESSING OF
C                                  INDIVIDUAL INTERVALS
C
C                  AREAS           THIS PROGRAM COMPUTES AREAS AND BOUNDS.
C                                  IT HAS TWO SUBPROGRAMS
C                                     TRIANGL - FORMULAS FOR TRIANGLE AREA
C                                     SPECIAL - FORMULAS FOR INFLECTION PT.
C
C                  INSERT          THIS PROGRAM  PLACES THE NEW INTERVALS
```

```
C                                     INTO THE QUEUE
C
C
C                                     IS TRUE
C
C                        QGET         THIS PROGRAM GETS AN INTERVAL FROM THE
C                                     QUEUE.
C
C                        QPUT         THIS PROGRAM OBTAINS A PLACE TO PUT THE
C                                     COMPLETED INTERVALS BACK INTO THE QUEUE
C
C     ****** THE FUNCTION F(X) IS NOT EXPLICITLY PRESENT IN THIS
C            PROGRAM. IT IS ASSUMED AVAILABLE TO ANY CPU AT ANY TIME
C-----------------------------------------------------------------------
C
      PROGRAM DECLARE
C     *********** THIS PROGRAM CONTAINS THE VARIABLE DECLARATIONS
      COMMON / OPSYS   / NCPU, CPU1ON, CPU2ON, CPURON(LIMCPU)
C               NCPU    - NUMBER ACTIVATED CPU'S LESS TWO
C               CPU1ON  - SWITCH TO ACTIVATE CPU1
C               CPU2ON  - SWITCH TO ACTIVATE CPU2
C               CPURON  - SWITCH TO ACTIVATE CPUR(IP), IP = 1 TO NCPU
C
      COMMON / PROBLEM / A, B, EPS, CHARF
C               A, B    - END POINTS OF INTERVAL OF INTEGRATION
C               EPS     - ACCURACY REQUIRED IN QUADRATURE ESTIMATE
C               CHARF   - CHARACTERISTIC LENGTH FOR F(X). F(X) HAS NO
C                         INFLECTION POINTS CLOSER THAN 2.*CHARF.
C
      COMMON / CONTROL / AREA, BOUNDA, DISCARD, LIMQ, LIMCPU, FINISH
C               AREA    - CURRENT AREA EXTIMATE
C               BOUNDA  - CURRENT BOUND ON ERROR IN AREA
C               DISCARD - AN INTERVAL IS DISCARDED WHEN THE BOUND ON ITS
C                         AREA ESTIMATE (AEST) IS LESS THAN THIS CONSTANT
C               LIMQ    - LIMIT ON THE NO. OF INTERVALS IN THE COLLECTION
C                         INEXT = LIMQ MARKS THE END OF THE QUEUE
C               LIMCPU  - LIMIT ON THE NO. OF CPU'S (BESIDE CPU1 + CPU2)
C               FINISH  - SWITCH TO TERMINATE EXECUTION IF TRUE
C
      COMMON / QUEUE   / LEADER, NQ, LASTQ, INEXT(LIMQ),
     1                   AEST(LIMQ), BOUND(LIMQ), INFLECT(LIMQ),
     2                   COTAN(LIMQ), COTR(LIMQ), COTL(LIMQ),
     3                   XRIGHT(LIMQ), FRIGHT(LIMQ), IRIGHT(LIMQ),
     4                   XLEFT (LIMQ), FLEFT (LIMQ), ILEFT (LIMQ),
     5                   QFREE, IDQ, WAITING, NEXTQ, THREE, IDT, TAILING, NEXTF
     6                   , INQUEUE(LIMQ)
C
C               LEADER  - NEXT INTERVAL IN COLLECTION TO BE ASSIGNED TO
C                         A CPU.  IT IS THE HEAD OF THE QUEUE
C               NQ      - LAST INDEX USED IN THE ARRAYS FOR THE QUEUE.
C               LASTQ   - TAIL OF THE QUEUE
C               INEXT   - INDEX OF THE NEXT INTERVAL IN THE QUEUE
C               INQUEUE - SWITCH FOR INTERVAL STATUS
C                         TRUE  = INTERVAL IS IN COLLECTION
C                         FALSE = INTERVAL BEING PROCESSED BY SOME CPU
C               AEST    - AREA ESTIMATE FOR THIS INTERVAL
C               BOUND   - BOUND ON ERROR IN AEST
C               INFLECT - SWITCH INDICATING INFLECTION POINT IS POSSIBLE
C                         = 0      - NO INFLECTION POINT
C                         = LEFT   - LEFT INTERVAL OF A SET OF THREE
C                         = CENTER - CENTER INTERVAL OF A SET OF THREE
C                         = RIGHT  - RIGHT INTERVAL OF A SET OF THREE
C               COTAN   - COTANGENT OF SEGMENT IN THIS INTERVAL
C               COTR    - COTANGENT OF BOUNDING LINE ON THE RIGHT
C               COTL    - COTANGENT OF BOUNDING LINE ON THE LEFT
C                         THESE ARE NORMALLY COTAN VALUES OF NEIGHBORS,
C                         BUT MAY BE MERELY COTANGENTS OF BOUNDING LINES.
C               XRIGHT, XLEFT - ABSCISSA OF END POINTS OF INTERVAL
```

```
C                FRIGHT,FLEFT - F(XRIGHT),F(XLEFT)
C                IRIGHT,ILEFT - INDEX OF RIGHT, LEFT NEIGHBORS
C               QFREE    - SWITCH FOR FREE ACCESS TO LEADER =HEAD OF QUEUE
C               IDQ      - INDEX OF CPU CURRENTLY ACCESSING LEADER
C               WAITING  - SWITCH INDICATES THAT CPU WITH PRIORITY IS
C                          WAITING FOR ACCESS TO THE LEADER.
C               NEXTQ    - INDEX OF CPU WITH PRIORITY TO ACCESS LEADER
C               TFREE    - SWITCH FOR FREE ACCESS TO LASTQ = TAIL OF QUEUE
C               IDT      - INDEX OF CPU CURRENTLY ACCESSING LASTQ
C               TAILING  - SWICH INDICATES THAT CPU WITH PRIORITY IS
C                          WAITING FOR ACCESS TO THE TAIL OF THE QUEUE.
C               NEXTT    - INDEX OF CPU WITH PRIORITY TO ACCESS TAIL
C
      COMMON / PROCSRS / ACHANGE(LIMCPU), BCHANGE(LIMCPU), AREAR(LIMCPU),
     1                   AREAL(LIMCPU), BOUNDR(LIMCPU), BOUNDL(LIMCPU),
     2                   IASSIGN(LIMCPU), IRETURN(LIMCPU), KORETRN(LIMCPU)
C               ACHANGE - CHANGE IN AREA COMPUTED BY INTERVAL PROCESSOR
C               BCHANGE - CHANGE IN BOUNDA COMPUTED
C               AREAR,AREAL   - AEST VALUES FOR RIGHT, LEFT HALVES
C               BOUNDR,BOUNDL - BOUND VALUES FOR RIGHT, LEFT HALVES
C               IASSIGN - INDEX OF INTERVAL ASSIGNED TO CPU
C               IRETURN - NO. OF INTERVALS RETURNED BY CPU
C               KORETRN - INDEX OF NEW PLACE IN QUEUE FOR INSERTION OF
C                         AN INTERVAL RETURNED BY AN INTERVAL PROCESSOR
C
C                          LOCAL VARIABLES IN THE PROCESSORS
C                          NEEDED GLOBALLY AMONG SUBPROGRAMS OF CPUR(IP)
     3                   , DX(LIMCPU), XMID(LIMCPU), FMID(LIMCPU),
     4                   COTANR(LIMCPU), COTANL(LIMCPU), INFL(LIMCPU)
C
      LOGICAL CPU1ON, CPU2ON, CPURON, FINISH, INQUEUE
     1        QFREE, WAITING, TFREE, TAILING
C
      DATA LEFT, CENTER, RIGHT / 4HLEFT , 6HCENTER , 5HRIGHT /
C-----------------------------------------------------------------------
C
      PROGRAM CPU1
C  *************** THIS IS THE OPERATING SYSTEM SIMULATION
**CALL,DECLARE
C
C     IT IS ASSSUMED THAT ALL NUMERIC VARIABLES ARE INITIALLY ZERO AND
C     ALL LOGICAL VARIABLES ARE INITIALLY FALSE.
C
      NCPU = 5
C
C          TURN ON THE TWO SPECIAL CPU'S
      CPU1ON = CPU2ON = .TRUE.
C
C          TURN ON SOME PROCESSOR CPU'S
      DO 100 K = 1,NCPU
         CPURON(K) = .TRUE.
  100 CONTINUE
C
C          IDLE LOOP WAITING FOR ALGORITHM TO TERMINATE
  200 IF( .NOT. FINISH )      GO TO 200
C
C          TURN OFF ALL CPU'S
      DO 300 K = 1,NCPU
         CPURON(K) = .FALSE.
  300 CONTINUE
      CPU1ON = CPU2ON = .FALSE.
      STOP
      END
C-----------------------------------------------------------------------
C
      PROGRAM CPU2
C
```

```
C  ************ THIS IS THE ALGORITHM CONTROLLER
C
**CALL,DECLARE
C
C          PROBLEM DEFINITION
      READ A,B,EPS,CHARF
C
      CALL BEGINQ
C
   10 IF( BOUNDA .LE. EPS )  FINISH = .TRUE.
C
C          CHECK FOR ABNORMAL TERMINATION
      IF( NQ .GE. LIMQ) THEN FINISH = .TRUE., PRINT 'ABNORMAL STOP'
C
      IF( .NOT. FINISH )                  GO TO 10
      PRINT ' VALUE OF THE INTEGRAL OF F(X) FROM' A ' TO ' B ' IS ' AREA
      PRINT ' ACCURATE TO WITHIN ' BOUNDA
      STOP
      END
C-----------------------------------------------------------------------
C
      SUBROUTINE BEGINQ
C
C  ********** ** THIS PROGRAM INITIALIZES THE ALGORITHM
C               THE ORIGINAL INTERVAL IS BROKEN UP INTO LENGTHS OF CHARF
C               AND THUS THIS TEST IS NEVER NEEDED LATER.  THE INTERVALS
C               WITH INFLECTION POINTS ARE DETERMINED.
**CALL,DECLARE
C
C               MISC. INITIALIZATIONS
      AREA = 0.
      DISCARD = EPS/(B-A)
C             FIND THE INITIAL INTERVAL LENGTH
      DX1 = AMIN1(CHARF/5.,B-A)
      NQ = (B-A)/DX
      IF( DX1*NQ .LT. B-A )   NQ = NQ + 1
      DX1 = (B-A)/NQ
      LASTQ = NQ
      LEADER = 1
C
C          FIRST SET OF QUANTITIES FOR INITIAL INTERVALS
      DO 100 K = 1,NQ
         XRIGHT(K) = A + K*DX1
         XLEFT (K) = XRIGHT(K) - DX1
         FRIGHT(K) = F(XRIGHT(K))
         FLEFT (K) = F(XLEFT(K))
         AEST  (K) = .5*DX1*(FLEFT(K) + FRIGHT(K))
         AREA      = AREA + AEST(K)
         COTAN (K) = DX1/(FRIGHT(K)-FLEFT(K))
         IRIGHT(K) = K+1
         ILEFT (K) = K-1
         INEXT (K) = K+1
  100 CONTINUE
C
C          FIX ITEMS FOR END INTERVALS NOT SET CORRECTLY ABOVE
      ILEFT(1)  = LIMQ
      IRIGHT(NQ)= LIMQ
      INEXT(NQ) = LIMQ
C
C          SECOND SET OF QUANTITIES FOR INITIAL INTERVALS
      COTL(1) = COTR(NQ) = 0.
      INFLECT(1) = INFLECT(NQ) = 0
C        DETERMINE INTERVALS WHERE COTANGENT DIFFERENCES CHANGE SIGN
C        THESE ARE CENTER INTERVALS OF TRIPLET WHICH MAY HAVE INFLECTION
C             SKIP IF WE ONLY HAVE 1 OR 2 INTERVALS
      IF( NQ .EQ. 1 )                  GO TO 201
      COTR(1) = COTAN(2)
```

```
      C0TL(NQ)= C0TAN(NQ-1)
      IF( NQ .LE. 2 )                   G0 T0 201
      D0 200 K = 2,NQ-1
          C0TL(K) = C0TAN(K-1)
          C0TR(K) = C0TAN(K+1)
C                   THIS EQUALITY TEST ASSUMES EXACT ARITHMETIC
          IF( ABS(C0TL(K)-C0TR(K)) .EQ.
     1              ABS(C0TL(K)-C0TAN(K)) + ABS(C0TAN(K)-C0TR(K)))
     2          THEN INFLECT(K) = 0
C                   AND THERE IS N0 INFLECTI0N P0INT DETECTED.
C
     3          ELSE INFLECT(K) = CENTER
C                   AND THERE IS AN INFLECTI0N P0INT DETECTED.
     4              INFLECT(K-1) = LEFT,  INFLECT(K+1) = RIGHT
  200 C0NTINUE
  201 C0NTINUE
C
C           N0W C0MPUTE THE INITIAL ERR0R B0UND
      B0UNDA = 0.
      D0 300 K = 1,NQ
          IF( INFLECT(K) .EQ. CENTER ) THEN
     1          C0TREC    = 1./C0TR(K) + 1./C0TL(K)
     2          B0UND(K) = DX1*ABS( FLEFT(K) - FRIGHT(K) + C0TREC )
C
     3        ELSE
     4          DF = FRIGHT(K) - FLEFT(K)
     5          IF( INFLECT(K) .EQ. LEFT )
     6             B0UND(K) = TRIANGL(C0TL(K),C0TAN(K),0,DX1,DF)
     7          IF( INFLECT(K) .EQ. RIGHT )
     8             B0UND(K) = TRIANGL(0,C0TAN(K),C0TR(K),DX1,DF)
     9          IF( INFLECT(K) .EQ. 0 )
     A             B0UND(K) = TRIANGL(C0TL(K),C0TAN(K),C0TR(K),DX1,DF)
C
          B0UNDA    = B0UNDA + B0UND(K)
  300 C0NTINUE
C
C
C           FINALLY FREE ALL INTERVALS AND MARK THEM AS IN THE QUEUE
      D0 400 K = 1,NQ
          INQUEUE(K) = .TRUE.
  400 C0NTINUE
      QFREE = TFREE = .TRUE.
      RETURN
      END
C-----------------------------------------------------------------------
C
      PR0GRAM CPUR(IP)
C
C ********** THIS IS THE INTERVAL PR0CESS0R PR0GRAM T0 ESTIMATE AREAS
C               IT IS AN ARRAY 0F PR0GRAMS (WITH INDEX IP) F0R THE ARRAY
C               0F CPU'S
-*CALL,DECLARE
C
C           ATTEMPT T0 GET AN INTERVAL FR0M THE QUEUE
   10 CALL QGET(IP)
C
C           HAVE 0NE, C0MPUTE AREAS AND B0UNDS
      CALL AREAS(IP)
C
C           ATTEMPT T0 GET PLACES IN THE QUEUE T0 PUT INTERVALS
      CALL QPUT(IP)
C
C           INSERT THE INTERVALS INT0 THE QUEUE
      CALL INSERT(IP)
C
C           RESTART THE PR0CESS
      G0 T0 10
```

```
      END
C-----------------------------------------------------------------------
C
      SUBROUTINE AREAS(IP)
C
C ************ THIS PROGRAM COMPUTES AREA ESTIMATES
**CALL,DECLARE
      IAI = IASSIGN(IP)
C
C              PRELIMINARY QUANTITIES
      DX(IP)     = .5*(XRIGHT(IAI) - XLEFT(IAI))
      XMID(IP)   = XRIGHT(IAI) - DX(IP)
      FMID(IP)   = F(XMID(IP))
      COTANR(IP) = DX(IP)/(FRIGHT(IAI)-FMID(IP))
      COTANL(IP) = DX(IP)/(FMID(IP) -FLEFT(IAI))
C
C         CHECK INTERVAL SITUATION AND SELECT AREA FORMULAS
C         INFLECT = 0 IS THE NORMAL CASE
      IF( INFLECT(IAI) .EQ. 0 ) THEN
     1           BOUNDL(IP) = TRIANGL(COTL(IAI),COTANL(IP),COTANR(IP),
     2                              DX(IP),FLEFT(IAI)-FMID(IP))
     3           BOUNDR(IP) = TRIANGL(COTANL(IP),COTANR(IP),COTR(IAI),
     4                              DX(IP),FMID(IP)-FRIGHT(IAI))
     5           INFL(IP) = 0
C
     6    ELSE
C              USE SPECIAL FORMULAS
     7           CALL SPECIAL(COTL(IAI),COTANL(IL),COTANR(IP),COTR(IAI),
     8                      INFLECT(IAI),IP)
C
C         CHECK DISCARDING OF INTERVALS
      IRETURN(IP) = 2
      IF( BOUNDR(IP) .LT. DISCARD*DX(IP) ) IRETURN(IP) = 1
      IF( BOUNDL(IP) .LT. DISCARD*DX(IP) ) IRETURN(IP) = IRETURN(IP) - 1
C
C         COMPUTE CHANGES IN AREA AND BOUNDA
      AREAR(IP) = .5*DX(IP)*(FMID(IP) + FRIGHT(IAI))
      AREAL(IP) = .5*DX(IP)*(FMID(IP) + FLEFT (IAI))
      BCHANGE(IP) = BOUND(IAI) - BOUNDR(IP) - BOUNDL(IP)
      ACHANGE(IP) = AEST (IAI) - AREAR(IP)  - AREAL(IP)
      RETURN
      END
C-----------------------------------------------------------------------
C
      FUNCTION TRIANGL(CLFT,CENT,CRGT,XBASE,YBASE)
C
C ************ THIS FUNCTION FINDS TRIANGLE AREAS FROM COTANGENTS
C              OF THE SIDES AND THE HORIZONTAL AND VERTICAL PROJECTIONS
C              OF THE BASE
**CALL,DECLARE
      COTRGT = (CRGT*CENT + 1.)/(CRGT - CENT)
      COTLFT = (CENT*CLFT + 1.)/(CENT - CLFT)
      BASE2  = XBASE**2 + YBASE**2
      TRIANGL= ABS( .5*BASE2/(COTRGT + COTLFT))
      RETURN
      END
C-----------------------------------------------------------------------
C
      SUBROUTINE SPECIAL(CLL,CL,CR,CRR,NFLECT,IP)
C
C ************ THIS PROGRAM COMPUTES AREA ESTIMATES FOR AN INTERVAL
C              WHERE AN INFLECTION POINT MAY BE PRESENT
C              THERE ARE THREE CASES ACCORDING TO THE VALUE OF INFLECT
C              THE ARGUMENTS CORRESPOND TO GLOBAL VARIABLES AS FOLLOWS
C                CLL     = COTL(IAI)
C                CL      = COTANL(IP)
C                CR      = COTANR(IP)
```

```
C                  CRR    = CØTR(IAI)
C                  NFLECT = INFLECT(IAI)
C              THE CORRECT VALUES FØR INFLECT ØF THE TWØ HALVES ARE SET
C                  INFL(IP)      - FØR THE LEFT HALF
C                  INFLECT(IAI) - FØR THE RIGHT HALF
C              THE MØDIFICATIØN ØF INFLECT FØR NEIGHBØRING INTERVALS IS
C              AN APPARENT VIØLATIØN ØF ATTRIBUTE 4 ØF APEAS.  HØWEVER,
C              IT IS SHØWN THAT THIS ACTIØN DOES NØT INVALIDATE THE
C              ALGØRITHM.  IT IS CUMBERSØME AND PØINTLESS TØ SAVE THE
C              INFØRMATIØN AND MØDIFY INFLECT LATER.
**CALL,DECLARE
C
C         ARITHMETIC STATEMENT FUNCTIØNS
C         DETERMINE MØNØTØNICITY ØF CØTANGENT SEQUENCE ØN THREE PØINTS
      CHANGE3(K) =     ABS(CLL-CRR)
     1               -( ABS(CLL-CL) + ABS(CL-CR) + ABS(CR-CRR) )
C         AREA ØF QUADRILATERAL FØR CENTER INTERVAL
      QUADRIL(CLFT,CRGT,DX,DF) = DX*ABS( DF + 1./CLFT + 1./CRGT )
C
      IAI = IASSIGN(IP)
      DFL = FMID(IP) -FLEFT(IAI)
      DFR = FRIGHT(IAI) - FMID(IP)
C
C         SELECT ØNE ØF THREE CASES FØR INFLECT
      IF(  NFLECT .EQ.  CENTER )             GØ TØ 300
      IF(  NFLECT .EQ.  RIGHT  )             GØ TØ 200
C
C                INFLECT = LEFT
      IF( CHANGE3(IP) .EQ.  0. ) THEN
C                THE INFLECTIØN PØINT MAY ØNLY BE IN THE LEFT HALF
     1           BØUNDL(IP) = TRIANGL(CLL,CL,CR,DX(IP),DFL)
     2           BØUNDR(IP) = TRIANGL(CL,CR,0.,DX(IP),DFR)
     3           INFL(IP) = 0
C
     4    ELSE
C                THE INFLECTIØN MAY BE IN EITHER HALF
     5           BØUNDL(IP) = TRIANGL(CLL,CL,0.,DX(IP),DFL)
     6           BØUNDR(IP) = QUADRIL(CL,CRR,DX(IP),DFR)
     7           INFL(IP) = LEFT ,  INFLECT(IAI) = CENTER
C                      UPDATE INFLECT VALUES TØ THE RIGHT
     8           INFLECT(IRIGHT(IAI)) = RIGHT
     9           INFLECT(IRIGHT(IRIGHT(IAI))) = 0
                                           RETURN
C
C                INFLECT = RIGHT
  200 CONTINUE
      IF( CHANGE3(IP) .EQ.  0.  )  THEN
C                THE INFLECTIØN PØINT MAY ØNLY BE IN THE RIGHT HALF
     1           BØUNDL(IP) = TRIANGL(0.,CL,CR,DX(IP),DFL)
     2           BØUNDR(IP) = TRIANGL(CL,CR,CRR,DX(IP),DFR)
     3           INFL(IP) = RIGHT ,  INFLECT(IAI) = 0
C
     4    ELSE
C                THE INFLECTIØN MAY BE IN EITHER HALF
     5           BØUNDL(IP) = QUADRIL(CRR,CL,DX(IP),DFL)
     6           BØUNDR(IP) = TRIANGL(0.,CR,CRR,DX(IP),DFR)
     7           INFL(IP) = CENTER ,  INFLECT(IAI) = RIGHT
C                      UPDATE INFLECT VALUES TØ THE LEFT
     8           INFLECT(ILEFT(IAI)) = LEFT
     9           INFLECT(ILEFT(ILEFT(IAI))) = 0
                                           RETURN
C
C                INFLECT = CENTER
  300 CONTINUE
      IF( ABS(CL-CRR) .LT.  ABS(CL-CR) + ABS(CR-CRR) ) THEN
C                CØTANGENTS ARE NØT MØNØTØNE ØN THE RIGHT HALF
C                AND IT IS THE NEW CENTER
```

```
     1         BOUNDL(IP) = TRIANGL(CLL,CL,0.,DX(IP),DFL)
     2         BOUNDR(IP) = QUADRIL(CL,CRR,DX(IP),DFR)
     3         INFL(IP) = LEFT,  INFLECT(IAI) = CENTER
C                   UPDATE INFLECT TO THE LEFT
     4         INFLECT(ILEFT(IAI)) = 0
C
     5   ELSE
C              THE LEFT HALF IS THE NEW CENTER
     6         BOUNDL(IP) = QUADRIL(CLL,CR,DX(IP),DFL)
     7         BOUNDR(IP) = TRIANGL(0.,CR,CRR,DX(IP),DFR)
     8         INFL(IP) = CENTER,  INFLECT(IAI) = RIGHT
C                   UPDATE INFLECT TO THE RIGHT
     9         INFLECT(IRIGHT(IAI)) = 0
C
      RETURN
      END
C-----------------------------------------------------------------------
C
      SUBROUTINE INSERT(IP)
C
C     ********** THIS PROGRAM INSERTS THE INTERVAL INTO ASSIGNED PLACES
C                OF THE INTERVAL COLLECTION.  IF 2 INTERVALS ARE KEPT THEN
C                       LEFT  GOES INTO IASSIGN(IP) = IAI = IL
C                       RIGHT GOES INTO KQRETRN(IP) = IR
*CALL,DECLARE
C
      IL = IAI = IASSIGN(IP)
      IR       = KQRETRN(IP)
C
C          CHECK ABOUT DISCARDING RIGHT INTERVAL
      IF( BOUNDR(IP) .LT. DISCARD*DX(IP) ) THEN
C              DISCARD THE RIGHT INTERVAL, SKIP ITS INSERTION
     1                ILEFT(IRIGHT(IAI)) = LIMQ ,  IR = LIMQ
     2                                 GO TO 200
C
C          INSERT RIGHT INTERVAL INTO IAI = IL IF LEFT ONE IS DISCARDED.
      IF( BOUNDL(IP) .LT. DISCARD*DX(IP) ) THEN IR = IL,  IL = LIMQ
C
C          INSERT RIGHT INTERVAL INFORMATION INTO THE COLLECTION
      XRIGHT(IR) = XRIGHT(IAI)
      FRIGHT(IR) = FRIGHT(IAI)
      XLEFT (IR) = XMID(IP)
      FLEFT (IR) = FMID(IP)
      BOUND (IR) = BOUNDR(IP)
      AEST  (IR) = AREAR(IP)
      COTAN (IR) = COTANR(IP)
      ILEFT (IR) = IL
      IRIGHT(IR) = IRIGHT(IAI)
      COTR  (IR) = COTR(IAI)
      COTL  (IR) = COTANL(IP)
      INFLECT(IR)= INFLECT(IAI)
      INQUEUE(IR)= .TRUE
C
C          CHECK ABOUT DISCARDING LEFT INTERVAL
  200 IF( BOUNDL(IP) .LT. DISCARD*DX(IP) ) THEN
C              DISCARD THE LEFT INTERVAL, SKIP ITS INSERTION
     1                IRIGHT(ILEFT(IAI)) = LIMQ
     2                                 GO TO 300
C
C          INSERT LEFT INTERVAL INFORMATION INTO THE COLLECTION
      XRIGHT(IL) = XMID(IP)
      FRIGHT(IL) = FMID(IP)
      XLEFT (IL) = XLEFT(IAI)
      FLEFT (IL) = FLEFT(IAI)
      BOUND (IL) = BOUNDL(IP)
      AEST  (IL) = AREAL(IP)
      COTAN (IL) = COTANL(IP)
```

```
      ILEFT (IL) = ILEFT(IAI)
      IRIGHT(IL) = IR
      COTR  (IL) = COTANR(IP)
      COTL  (IL) = COTL(IAI)
      INFLECT(IL)= INFL(IP)
      INQUEUE(IL)= .TRUE.
C
C          INSERTION COMPLETED
  300 RETURN
      END
C-----------------------------------------------------------------------
C
      SUBROUTINE QGET(IP)
C
C  ************ THIS PROGRAM GAINS ACCESS TO THE HEAD OF THE QUEUE IN
C                  ORDER TO OBTAIN AN INTERVAL.
-*CALL,DECLARE
C
C          CHECK TO SEE IF THE QUEUE IS NOT EMPTY AND THE LEADER IS FREE
   10 IF( LEADER .EQ. LIMQ )             GO TO 10
C
C          ENTER COMPETITION FOR ACCESS TO THE QUEUE LEADER
      IF( .NOT. QFREE )  THEN
C                CHECK TO SEE IF THIS IS THE NEXT CPU IN ORDER
C                IF NOT, RETURN TO COMPETITION FOR QUEUE ACCESS
    1       20 IF( IP .NE. NEXTQ )       GO TO 10
C
C          ------- PRIORITY WAITING LOOP BEGINS --------
C                   THIS IS THE NEXT CPU IN ORDER
    2          WAITING = .TRUE.
C
C                    WAIT, SEE IF WAITING WAS TESTED WHILE BEING CHANGED
    3          CONTINUE
C                    IF SO, THEN EXIT AND RETURN TO COMPETITION
C                  NEXT STMT. IS A 1-LINE IF-THEN-ELSE
    4          IF( QFREE )  THEN WAITING = .FALSE. , GO TO 10
C                     IDLE LOOP AWAITING TURN
    5       30 IF( WAITING )             GO TO 30
C
C                CHECK THAT PREVIOUS ACCESS DID NOT EXHAUST COLLECTION
C                IF COLLECTION IS EMPTY, WAIT IN IDLE LOOP
            35 IF( LEADER .EQ. LIMQ )    GO TO 35
C
C                IT IS NOW THIS CPU'S TURN TO GAIN ACCESS TO THE QUEUE
    7                                    GO TO 50
C          ------- PRIORITY WAITING LOOP ENDS    --------
C
C          HAVE ENTERED GATE TO THE QUEUE, NOW CLOSE GATE BEHIND US.
   40 QFREE = .FALSE.
      IDQ   = IP
C               DELAY LONG ENOUGH SO ALL CPUS THAT FALL THRU THE ABOVE
C               'IF' ARE BETWEEN THE PREVIOUS AND THE FOLLOWING STMTS.
      CONTINUE
C          CHECK TO SEE IF THIS WAS THE LAST CPU TO SET IDQ
C          IF NOT, RETURN TO COMPETITION FOR QUEUE ACCESS
      IF( IDQ .NE. IP )                  GO TO 20
C
C ----------- START CRITICAL PART -------------------
C          CHECK THAT LEADER IS ACTUALLY AVAILABLE, WAIT IF NOT
   50 IF( .NOT. INQUEUE(LEADER) )        GO TO 50
C          HAVE SOLE ACCESS TO THE QUEUE LEADER
      IASSIGN(IP) = LEADER
C
C          MARK LEADER AS NOT IN THE QUEUE
      INQUEUE(LEADER) = .FALSE.
      LEADER      = INEXT(LEADER)
C          DELAY 1 STATEMENT TO ALLOW TIME FOR CPU WITH IP = NEXTQ TO BE
```

```
C           PUT INTO WAITING STATUS AT 30 ABOVE
      CONTINUE
C
C              OPEN GATE IF NO CPU IS WAITING INSIDE IT
      IF( WAITING )  THEN WAITING = .FALSE.
     1                    ELSE OFREE    = .TRUE.
C
C           INCREMENT THE INDEX FOR THE NEXT CPU IN ORDER
      NEXTO = MOD(NEXTO,NCPU) +  1
C   ------------ END   CRITICAL PART -------------------
C
C           HAVE FINISHED WITH QUEUE ACCESS
      RETURN
      END
C----------------------------------------------------------------------
C
      SUBROUTINE QPUT(IP)
C
C   ************ THIS PROGRAM GAINS ACCESS TO THE TAIL OF THE QUEUE IN
C                ORDER TO OBTAIN PLACES TO PUT THE NEW INTERVALS.
*CALL,DECLARE
C
C           ENTER COMPETITION FOR ACCESS TO THE TAIL
   10 IF( .NOT.  TFREE )  THEN
C                CHECK TO SEE IF THIS IS THE NEXT CPU IN ORDER
C                IF NOT, RETURN TO COMPETITION FOR TAIL ACCESS
     1      20 IF( IP .NE.  NEXTT )        GO TO 10
C
C           ------- PRIORITY TAILING LOOP BEGINS -------
C                 THIS IS THE NEXT CPU IN ORDER
     2          TAILING = .TRUE.
C
C                  WAIT, SEE IF TAILING WAS TESTED WHILE BEING CHANGED
     3          CONTINUE
C                  IF SO, THEN EXIT AND RETURN TO COMPETITION
C                NEXT STMT. IS A 1-LINE IF-THEN-ELSE
     4          IF( TFREE )  THEN TAILING = .FALSE.  , GO TO 10
C                  IDLE LOOP AWAITING TURN
     5      30 IF( TAILING )               GO TO 30
C
C               IT IS NOW THIS CPU'S TURN TO GAIN ACCESS TO THE TAIL
     6                                     GO TO 50
C           ------- PRIORITY TAILING LOOP ENDS    -------
C
C           HAVE ENTERED GATE TO THE TAIL, NOW CLOSE IT BEHIND US.
   40 TFREE = .FALSE.
      IDT   = IP
C                     DELAY 1 STATEMENT
      CONTINUE
C           CHECK TO SEE IF THIS WAS THE LAST CPU TO SET IDT
C           IF NOT, RETURN TO COMPETITION FOR TAIL ACCESS
      IF( IDT .NE.  IP )                  GO TO 20
C
C           HAVE SOLE ACCESS TO THE TAIL OF THE QUEUE
C   --------------- -- START CRITICAL PART ---------
C
   50 IF( IRETURN(IP) .EQ.  0 )  THEN
C           NO INTERVALS RETURNED
C
      IF( IRETURN(IP) .EQ.  1 ) THEN
C           PICK UP INFO TO PUT NEW INTERVAL IN OLD PLACE
     1      INEXT(NQ) = IASSIGN(IP)
     2      NQ        = IASSIGN(IP)
     3      INEXT(NQ) = LIMQ
C
      IF( IRETURN(IP) .EQ.  2 ) THEN
C           PICK UP INFO TO PUT 1 NEW INTERVAL IN OLD PLACE AND EXTEND
```

```
C           QUEUE APEA BY 1 FØR THE ØTHER NEW INTERVAL
     1      LASTQ = LASTQ + 1, INQUEUE(LASTQ) = .FALSE.
     2      KORETRN(IP) = INEXT(IASSIGN(IP))  = LASTQ
     3      INEXT(NQ) = IASSIGN(IP)
     4      NQ = LASTQ, INEXT(NQ) = LIMQ
C
C       REASSIGN THE QUEUE LEADER IF THE QUEUE WAS EMPTY
      IF( IRETURN(IP) .GT. 0 .AND. LEADER .EQ. LIMQ )
     1                                   LEADER = IASSIGN(IP)
C
C         UPDATE THE AREA AND BØUND ESTIMATES
      AREA   = AREA - ACHANGE(IP)
      BØUNDA = BØUNDA - BCHANGE(IP)
C
C         READY TO RELINQUISH ACCESS TØ THE TAIL
C
C            ØPEN GATE TØ TAIL IF NØ CPU IS WAITING INSIDE
      IF( TAILING )  THEN TAILING = .FALSE.
     1               ELSE TFREE   = .TRUE.
C
C       INCREMENT THE INDEX FØR THE NEXT CPU IN ØRDER
      NEXTT = MØD(NEXTT,NCPU) +  1
C  ---------------- END  CRITICAL PART ---------
      RETURN
      END
```

3. SOME ATTRIBUTES THAT ARE OBVIOUS OR EASILY ESTABLISHED

Many of the attributes claimed for the program PAFAQ may be verified by inspection. We list these in the first theorem and indicate the appropriate parts of the program to inspect.

THEOREM 3.1. The program PAFAQ has the following attributes:

Program	Attributes
MAIN-CPU1	1,2,3
MAIN-CPU2	1,2,3
MAIN-CPUR	
AREAS	4,5
NEIGH	5,6
INSERT	2,3
QGET	5
QPUT	4,7

Proof. The main programs for CPU1 and CPU2 are so short that we merely inspect them to see that they have the attributes claimed. The attribute for the main program of CPUR is in fact a specification of this program and we see that CPUR has the four subprogram invocations as required.

An inspection of AREAS shows that it only assigns values to variables indexed by IP and that it (and its two subprograms TRIANGL and SPECIAL) are straight line programs. F(x) is evaluated exactly once by AREAS and this is the only subprogram that evaluates F(x) (i.e. $q = 1$ in Attribute 5 of AREAS) except for the algorithm initialization in BEGINQ.

Attribute 5 of QGET and Attribute 7 of QPUT are the same and one inspects the list of variables assigned values to see that QGET and QPUT do not affect any information about an interval other than its status in the algorithm.

Finally, QPUT is seen to have Attribute 4 by virtue of two statements near the end of the critical section of QPUT. This concludes the proof.

In the remainder of this section we establish that the program has a variety of attributes which are considered easy but not obvious.

LEMMA 3.1. The program PAFAQ has Attributes 1, 2 and 3 of BEGINQ.

Proof. BEGINQ initializes the interval collection by dividing [A,B] into equal segments of length CHARF/5. Three passes are made through this initial collection. The first (DO 100 loop) computes basic quantities for each interval (e.g. end points, cotangents). The second pass (DO 200 loop) then detects intervals which are the center of triplets which contain an inflection point. Since the intervals are short, there is no overlap in these triplets. The third pass (DO 300 loop) then computes the initial error bound for each interval in the collection and the total for [A,B]. A direct verification shows that the miscellaneous quantities associated with the interval collection are initialized properly. This establishes that BEGINQ has Attribute 1.

That PAFAQ has Attribute 2 and 3 may be verified by inspection.

LEMMA 3.2. The program PAFAQ has Attribute 1 of INSERT.

Proof. The action of INSERT required for this attribute is made primarily by the two long sequences of simple assignment statements. The only delicate operation is to switch the right interval to the left interval's location in case the left interval is discarded. This is accomplished by the switch in index IR = IL made just before the assignment statements for the right interval.

LEMMA 3.3. The program PAFAQ has Attribute 4 of QGET and Attribute 6 of QPUT.

Proof. These two attributes are the same and an inspection of QGET and QPUT indicates that their domains of action only intersect in the variables LEADER, INEXT and INQUEUE. The situation where QPUT assigns a value to LEADER is analyzed in more detail in Section 4, but even so it is readily apparent that no conflict can occur. That is, QPUT can modify LEADER only if its current value is LIMQ (which indicates the queue is empty) and QGET cannot reach the critical section when the value of LEADER is LIMQ.

The only modification of INEXT by QPUT that could affect QGET is that of LEADER. However, QPUT modifies INEXT only for intervals assigned to CPUs or ones newly created by subdivision. None of these can be the queue leader so no conflict occurs here. A similar argument shows that INQUEUE cannot lead to a conflict and this concludes the proof.

LEMMA 3.4. The program PAFAQ has Attribute 2 of QGET.

Proof. We see that the variable INQUEUE is used by QGET to mark an interval assigned to a CPU as unavailable for further assignment. INQUEUE is initialized to be true by BEGINQ. A perusal of the program shows that INQUEUE is only reassigned by INSERT as the last operation on an interval after it is placed in the interval collection. It is clear that a new value of LEADER is assigned and this concludes the proof.

LEMMA 3.5. The program PAFAQ has Attribute 2 of QPUT.

Proof. The critical section of QPUT contains three IF statements, one for each possibility of returning intervals. One possibility is that no intervals are returned and no action is required in this case. Note that this program does not do any garbage collection in memory, so the program loses the use of memory space of an interval when both halves are discarded.

If one interval is returned, then it is placed in the memory used by its predecessor and this interval is made the end of the queue.

If two intervals are returned, then QPUT extends the memory allocated to the collection (LASTQ marks the extent of this memory), updates the links INEXT for the queue and moves the end of the queue to the newly created queue position (i.e. NQ = LASTQ). This concludes the proof.

4. CONFLICTS AND DELAYS DUE TO PARALLEL EXECUTION.

This section deals with the fundamental question of integrity of the interval collection during the multiple, unsynchronized access by various interval processors. The main responsibility for maintaining this integrity is taken by the subprograms QPUT and QGET and, in particular, the algorithm at the beginning of each of them. We begin with some technical lemmas about the mechanism to control this access.

LEMMA 4.1. Consider the K-th interval which has priority for access to the head or tail of the queue, i.e. K = NEXTQ or K = NEXTT and further which has entered the priority waiting loop of QPUT or QGET. The shortest time lapse for this interval's processor to change NEXTQ or NEXTT from the previous change is 10 statements. The longest time lapse for this interval's processor to enter the critical section is 2 statements after NEXTQ (or NEXTT) is changed.

Proof. We list in tabular form the statements executed by CPU(INSIDE), the CPU currently in the critical section, and by CPU(IP), the CPU processing the K-th interval. An examination of the program shows that the shortest time lapse occurs in the following case (we use the statements from QGET here).

Table 4.1. Statements executed for the shortest time lapse to enter the critical sections of QGET and to change NEXTQ.

Time	CPU-INSIDE	CPU-IP
0	IF(WAITING)	IF(WAITING)
1	WAITING = .FALSE.	GO TO 30
2	NEXTQ =	IF(WAITING)
3		IF(LEADER.EQ...
4		GO TO 50
5		IF(.NOT. INQUEUE...
6		INQUEUE(LEADER) = .FALSE.
7		IASSIGN(IP) = LEADER
8		LEADER = INEXT(LEADER)
9		CONTINUE
10		IF(WAITING)
11		WAITING = or QFREE =
12		NEXTQ =

An examination of QPUT shows that the critical section has at least 6 statements to execute (compared to 5 for QGET) but does not have one of the statements in the waiting section. This establishes the first conclusion.

A similar table for the time required for the interval with priority to reach the critical section is given below. This table shows the longest possible delay in QGET (QPUT has one less statement for CPU(IP) to execute).

Table 4.2. The longest delay in exiting the priority waiting loop.

Time	CPU-INSIDE	CPU-IP
0	IF(WAITING)	IF(WAITING)
1	WAITING = .FALSE.	GO TO 30
2	NEXTQ =	IF(WAITING)
3		IF(LEADER.EQ....
4		GO TO 50
5		IF(.NOT.INQUEUE

This concludes the proof.

LEMMA 4.2. Consider an interval which does not have priority for access to the head or tail of the queue. The shortest time lapse for this interval's processor to change NEXTQ (or NEXTT) from the previous change is 11 statements. The longest time lapse for this interval's processor to enter the critical section is 6 statements after NEXTQ or NEXTT is changed.

Proof. We consider two cases for the CPU processing this interval. In case 1 the CPU (denoted by IP) is continually finding QFREE to be false. In case 2 the CPU has found QFREE to be true along with the processor INSIDE, but it did not gain access to the critical section. An inspection shows that in the second case the CPU cannot change NEXTQ or NEXTT faster than in the first case. Likewise, the second case cannot generate a longer time lapse because by the time QFREE is set true, this processor has already exited to the group of CPUs testing QFREE. Thus we need only consider the first case here and the table below shows the situation where the fastest change occurs for QGET.

Table 4.3. Statements executed to achieve the fastest change in NEXTQ.

Time	CPU-INSIDE	CPU-IP
0	IF(WAITING)	
1	QFREE = .TRUE.	IF(LEADER...
2	NEXTQ =	IF(.NOT.QFREE)...
3		QFREE = .FALSE.
4		IDQ = IP
5		CONTINUE
6		IF(IDQ...
7		IF(.NOT.INQUEUE
8		INQUEUE(LEADER) =
9		IASSIGN(IP) =
10		LEADER =
11		IF(WAITING)
12		QFREE = .TRUE.
13		NEXTQ =

Again the critical section for QPUT executes at least one more statement but the waiting portion has one less statement. This establishes the first conclusion.

The situation for the longest time lapse possible for CPU-IP to enter the critical section is shown in the next table for QGET.

Table 4.4. Statements executed for the longest time lapse to enter the critical section of QGET.

Time	CPU-INSIDE	CPU-IP
0	IF(WAITING)	IF(IP.NE.NEXTQ)
1	QFREE = .TRUE.	GO TO 10
2	NEXTQ =	IF(LEADER

Table 4.4 (Continued)

Time	CPU-INSIDE	CPU-IP
3		IF(.NOT.QFREE)
4		QFREE = .FALSE.
5		IDQ = IP
6		CONTINUE
7		IF(IDQ.NE.IP)
8		IF(.NOT.INQUEUE

There is one less statement to execute in QPUT and this concludes the proof.

These timing lemmas enable us to establish a key property of the algorithm to control access to the queue.

LEMMA 4.3. There is at most one CPU waiting in QGET (or in QPUT) for access and which is executing the priority waiting loop. There is at most one CPU executing the critical section of QGET (or of QPUT).

Proof. We first consider the possibility that two CPUs are idle and designated as having priority, i.e. they will enter the critical section as soon as WAITING or TAILING is set false. During a period while NEXTQ or NEXTT is fixed, it is clear that only one CPU can achieve this status. Thus the only possibility to have two CPUs in this status is for one to achieve it, then have NEXTQ or NEXTT change and another achieve it before the first has entered the critical section. The first possible uncertainty revolves about WAITING and TAILING which are critical values but which have not been protected by an elaborate mechanism. Such a mechanism is not required because at most two CPUs can simultaneously (or nearly simultaneously) process WAITING and TAILING. This is seen from the table below where we display the statements executed by the CPU–INSIDE and the CPU with IP = NEXTQ (we consider QGET here for concreteness).

Table 4.5. Statements executed while entering the priority waiting loop.

Time	CPU–INSIDE	Time	CPU–IP
0	CONTINUE	t	IF(IP.NE.NEXTQ)
1	IF(WAITING)	t+1	WAITING = .TRUE.
2	WAITING = .FALSE. or QFREE = .TRUE.	t+2	CONTINUE
3	NEXTQ =	t+3	IF(QFREE)
4		t+4	IF(WAITING) or WAITING = .FALSE.
5		t+5	IF(LEADER...
6		t+6	GO TO 50

When t = 0 in this match-up between statements we see that WAITING is tested by INSIDE at the same time its value is changed by IP. This fact is detected by the test of QFREE and CPU–IP exits the priority waiting loop. A similar exit occurs when t = 1, 2 or 3. If $t \geq 4$ then IP is not the priority CPU as the test at time t occurs after NEXTQ is changed.

If $t < 0$, we see that WAITING is set false after having been set true and CPU-IP gains access to the priority waiting loop. Then WAITING is set false and CPU-IP exits the priority waiting and enters the critical section within four statements. The CPU whose index is NEXTQ as set in statement 3 can start to enter the priority waiting loop so both are not in the loop simultaneously.

The other possible uncertainty may occur if NEXTQ is changed, a CPU enters the priority waiting loop, then NEXTQ is changed again and another admitted before the first can leave the priority waiting loop and enter the critical section. It is seen from Lemma 4.1 that a change of NEXTQ requires that at least 10 statements be executed while the exit from the priority waiting loop requires at most two statements. This establishes the first conclusion of the lemma.

An examination of QGET and QPUT shows that the critical section can only be entered from the priority waiting loop or from the "gate" governed by QFREE or TFREE. The two programs are essentially identical in operation and, for concreteness, we only consider QGET here. Entry into the critical section is allowed by the CPU exiting it when it sets QFREE true or WAITING false. If WAITING is set false only one CPU can start execution of the critical section because only one CPU is executing the priority waiting loop.

If QFREE is set true then there is no CPU in the priority waiting loop and if one enters just before QFREE is set true then, as shown above, it exits the priority waiting loop. This CPU may attempt to enter the critical section in this case only via the normal route. An arbitrary number of CPUs may start to enter and each of them sets QFREE false so that a group of CPUs is executing the code almost simultaneously. Each sets IDQ equal to the CPU's index and then delays one statement. Since all the CPUs of the group are within one statement of one another in executing the program, there is an instance when all are executing the CONTINUE statement and the value of IDQ is that of the last CPU to set it. This last CPU is the only one where the test IDQ.NE.IP is false. This CPU enters the critical section and all others exit to statement 20 where the test for identifying the priority CPU is made. All those that fail this test rejoin the CPU's competing for access to the queue. One CPU might enter the priority waiting loop at statement 20, but it is easily seen that it would stay there until the CPU with access to the critical section exits from the critical section. This concludes the proof.

COROLLARY. The program PAFAQ has Attribute 1 of QGET and QPUT.

Proof. The corollary follows directly from Lemma 4.3 for QPUT. In the case of QGET there is the additional condition that the LEADER of the queue exist and be available for assignment. If this condition is not satisfied it is seen that a CPU executing the priority waiting loop continues to wait in an idle loop until the LEADER is available. All CPUs attempting to gain initial access to the critical section execute an idle loop as long as the LEADER is unavailable and, once it becomes available, they behave as described in Lemma 4.3.

THEOREM 4.1. The program PAFAQ has Attribute 3 of QGET and QPUT.

Proof. Let the CPU which attempts to gain access have index IPX. We consider only the case of QGET as the one for QPUT is essentially identical. It is readily seen that each CPU that exits the critical section increments NEXTQ by 1 modulo NCPU+1. Thus it is clear that whenever NPCU CPUs have executed the critical section, the variable NEXTQ will have taken on all values from 1 to NCPU. It remains to show that whenever NEXTQ=IPX then the CPU IPX does enter the priority waiting loop and thence enters the critical section.

It follows from Lemmas 4.1 and 4.2 that the shortest time lapse between changes of NEXTQ is 10 statements. When the variable NEXTQ is set to IPX, then CPU IPX will be attempting to gain access without being in the priority waiting loop. It might achieve access when QFREE is set true and this would occur in 6 statements. In this case CPU IPX would achieve access within the specified time without entering the priority waiting loop.

We now need to know the longest time lapse possible for CPU IPX to enter the priority waiting loop. If this time lapse is less than the smallest possible time lapse between changes in NEXTQ, then we have established the theorem. The situation giving the longest time lapse is shown in Table 4.6.

Table 4.6. Statements executed which give the longest time lapse for entry into the priority waiting loop.

Time	CPU-INSIDE	CPU-IPX
0	IF(WAITING)	
1	QFREE = .TRUE.	IF(LEADER
2	NEXTQ =	IF(.NOT.QFREE)
3		QFREE = .FALSE.
4		IDQ = IP
5		CONTINUE
6		IF(IDQ.NE.IP)
7		GO TO 25
8		IF(IP.NE.NEXTQ)
9		WAITING = .TRUE.
10		CONTINUE
11		IF(QFREE)
12		IF(WAITING)

The longest time lapse for IPX to enter the priority waiting loop is thus 10 statements, but it is seen from Lemma 4.2 that NEXTQ cannot be changed before time 13 (a time lapse of 11 statements). We also see from Table 4.3 that WAITING cannot be tested before time 11 and thus WAITING is set false by the CPU which does gain access to the critical section. This concludes the proof.

THEOREM 4.2. The program PAFAQ has Attribute 6 of QGET and Attribute 8 of QPUT.

Proof. These two attributes have almost been established during the preceding proofs. Thus from the proof of Theorem 4.1 we know that the delay between the exit of one CPU from QGET (on QPUT) and the entry of another to the critical section is quite short. Further we have seen that no CPU is blocked from access to the critical sections of QGET and QPUT. The only delay of uncertain magnitude is in QGET which may be caused when the queue is empty (LEADER = LIMQ) or the LEADER has not yet been inserted into the interval collection (INQUEUE(LEADER) is false).

We claim that the total time to execute the subprogram MAIN for CPU-IP is bounded by the sum of the following times:

1. MAIN - 6 statements
2. QGET - NCPU times QGET execution time without delays
3. AREAS - 17 statements plus 1 execution of SPECIAL
4. QPUT - NCPU times QPUT execution time without delays
5. INSERT - 41 statements

Suppose now that the interval collection is empty. Then all intervals must be assigned to processors (otherwise the algorithm is terminated) and thus for some CPU we have execution occurring in or after the critical section of QGET. This CPU then proceeds to execute AREAS and starts to execute QPUT. Either it or another CPU then gains access to the available memory. However, the CPU that gains access might not return any intervals to the collection and thus not designate a new LEADER. Even so, the other CPUs which are processing intervals gain access to the available memory and may return an interval. If none of them do (all intervals are discarded) then the algorithm termination criterion is met. Otherwise one of them does obtain space for an interval and proceeds to execute INSERT. There are only NCPU processors so unless the computation terminates successfully, we have that within a fixed time the test of LEADER = LIMQ is made and a new LEADER is assigned. As soon as INSERT terminates the queue leader is unblocked, INQUEUE(LEADER) is true and execution proceeds. This concludes the proof.

We may summarize the results of this section by saying that there are no indefinite delays in the execution of PAFAQ. Every delay made in order to avoid conflicts from parallel execution is bounded in length by some constant times NCPU.

5. THE AREA AND BOUND ESTIMATES.

This section deals with the basic numerical analysis procedures of the algorithm, namely Attributes 1, 2 and 3 of AREAS. These attributes essentially state that if the integrand f(x) is in the domain of applicability as defined by Assumption 1 below then the area estimates and bounds on the area estimates satisfy the conditions of Assumption 2 of [4] which is one of the hypotheses of the convergence proof.

ASSUMPTION 1. (Integrand) f(x) has singularities.

$$S = \{s_i \mid i = 1, 2, \ldots, R;\ R < \infty\}$$

Let

$$w(x) = \prod_{i=1}^{R} (x - s_i)$$

(i) $x_0 \notin S$ implies that $f''(x)$ is continuous in a neighborhood of x_0.

(ii) there are constants K and $\alpha > 0$ so that

$$|f''(x)| \leq K|w(x)|^{\alpha-2}$$

(iii) f(x) has a finite number of inflection points.

(iv) f(x) has no cusps.

(v) the minimum separation between singularities and/or inflection points is CHARF.

The limitation implied by the fourth part of this assumption is for the sake of simplicity. One could, as indicated in [2], expand the subprogram AREAS to accommodate cusps.

The first step is to locate the inflection points.

LEMMA 5.1. Let f(x) satisfy Assumption 1. Every subinterval which might contain an inflection point has INFLECT not zero and every interval with INFLECT zero has no inflection point.

Proof. First consider BEGINQ where the interval [A,B] is partitioned in subintervals of length CHARF/5 and the broken line interpolant to f(x) is found. Specifically, the cotangent COTAN(K) of the K-th line segment is computed and then the monotonicity of the sequence COTAN(K) is checked. It is easily seen geometrically that any set of three intervals where monontonicity is absent contains an inflection point. The assumption that the partition is in intervals of CHARF/5 insures that only one inflection point is contained in any such set of intervals and that such sets do not overlap. After the iteration 200 is terminated all the center intervals of such sets are marked with INFLECT = CENTER and INFLECT = LEFT or RIGHT on the appropriate sides of these center intervals. Thus we have established the lemma to be correct for the initial situation.

An examination of PAFAQ shows that INFLECT is thereafter changed only in the subprogram SPECIAL of AREAS. There is a technical violation of Attribute 4 of AREAS in this subprogram because the value of INFLECT might be changed for neighboring intervals during the execution of SPECIAL. This violation does not invalidate the effectiveness and correctness proof for two reasons. First, if an interval has started being processed with one value of INFLECT and then a change of CENTER to LEFT or RIGHT or of LEFT/RIGHT to 0 is made at some point, no error results. Specifically, such a change could only affect SPECIAL itself and one sees that there is only one test of INFLECT (per possible case) and a change in its value has no effect after the test. That is, the result from SPECIAL is the same as if no change had been made. Second, the possible changes in INFLECT can only decrease the value of the error bound and there decreased values are correct if the change is made. Thus, if SPECIAL changes a neighboring interval's value of INFLECT, the worst that can happen is that PAFAQ computes a larger than necessary bound on the quadrature error. Incidently, as noted in the comments of PAFAQ, it is possible, but surprisingly cumbersome, to avoid this technical violation of Attribute 4 by saving the changes to be made in INFLECT and then modifying INFLECT later.

The proof is then completed by examining the partition of subintervals which occurs in AREAS, or more exactly in its subprogram SPECIAL. There are three cases corresponding to INFLECT = LEFT, CENTER or RIGHT. There is complete symmetry between the LEFT and RIGHT cases and we only consider the LEFT case here. These three cases are processed separately by SPECIAL and in each case an examination shows that there are two possible outcomes of the subdivision which are indicated in the following table:

Case	New Value of INFLECT for Left Subinterval	Right Subinterval	Action Required for Neighbors
INFLECT=LEFT, #1	0	LEFT	None
#2	LEFT	CENTER	Change INFLECT to RIGHT for right neighbor Change INFLECT to 0 for second right neighbor
INFLECT=CENTER, #1	CENTER	RIGHT	Change INFLECT to 0 for right neighbor
#2	LEFT	CENTER	Change INFLECT to 0 for left neighbor

The subprogram SPECIAL sets the value of INFLECT for the two halves of the interval being processed and also makes the modifications of the appropriate neighboring values of INFLECT. The values saved in SPECIAL for INFLECT are then assigned in INSERT as the subintervals are returned to the interval collection. This concludes the proof.

LEMMA 5.2. Let f(x) satisfy Assumption 1. The program PAFAQ has Attributes 2 and 3 of AREAS.

Proof. An inspection of AREAS shows that the proportional error distribution is used, that is BOUNDR and BOUNDL are always compared to DISCARD * DX = EPS * DX/(B-A). This is equivalent to having ERROR of Assumption 2 equal to BOUND(I) divided by DX. Those intervals with BOUND less than DISCARD * DX are identified and counted in AREAS.

The condition of Attribute 2 that intervals be shorter than CHARF is implemented in BEGINQ by the initial partitioning of the interval [A,B]. This concludes the proof.

The key point of this section is that PAFAQ computes true bounds on the errors in the trapezoidal rule. Figures 5.1 and 5.2 illustrate the different situations and the geometric constructions used to bound the quadrature errors. These figures also indicate the correspondence with names in the program.

THEOREM 5.1. Let f(x) satisfy Assumption 1. The values of BOUNDR and BOUNDL computed by AREAS are true bounds on the error in the trapezoidal rule.

Proof. There are two distinct situations. First is where the interval is known not to contain an inflection point. Then the quadrature error is bounded by the area of the triangle as shown in Figure 5.1. The program computes this area using the function TRIANGL and assigns this value to the bounds when INFLECT is zero.

When the interval might contain an inflection point then the quadrature error is still bounded by the area of a triangle when INFLECT is LEFT or RIGHT (see Figure 5.2). If INFLECT is CENTER then the quadrature error is bounded by the area of a quadrilateral (actually a trapezoid). These calculations are carried out in SPECIAL using the functions TRIANGL and QUADRIL.

COROLLARY. Let f(x) satisfy Assumption 1. The values for AEST(K) BOUND(K), BOUNDA and AREA are correctly computed by PAFAQ.

Proof. The previous arguments establish this result for BOUNDA and BOUND(K) and the computations of AEST(K) and AREA may be verified as correct by inspecting BEGINQ (where initialization takes place), AREAS (where AREAR and AREAL are computed), INSERT (where AREAR and AREAL values are assigned to AEST(K)) and QPUT (where the value of AREA is updated).

LEMMA 5.3. Let f(x) satisfy Assumption 1. Assume the I-th interval and its two neighbors have neither an inflection point nor a singularity of f(x) and it is not one of the two end intervals. Then with x=XLEFT(I)

$$d = XRIGHT(I) - XLEFT(I)$$

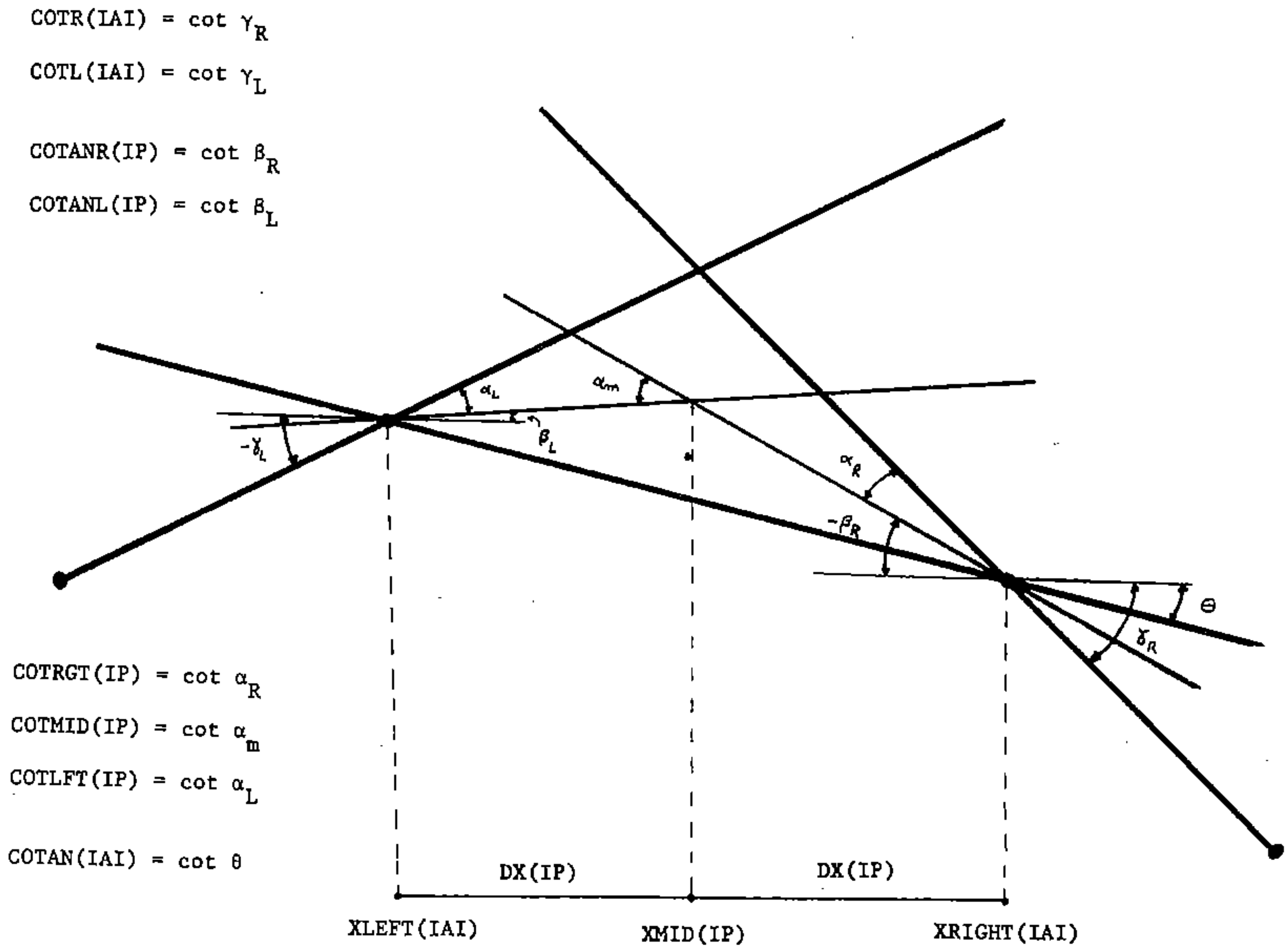
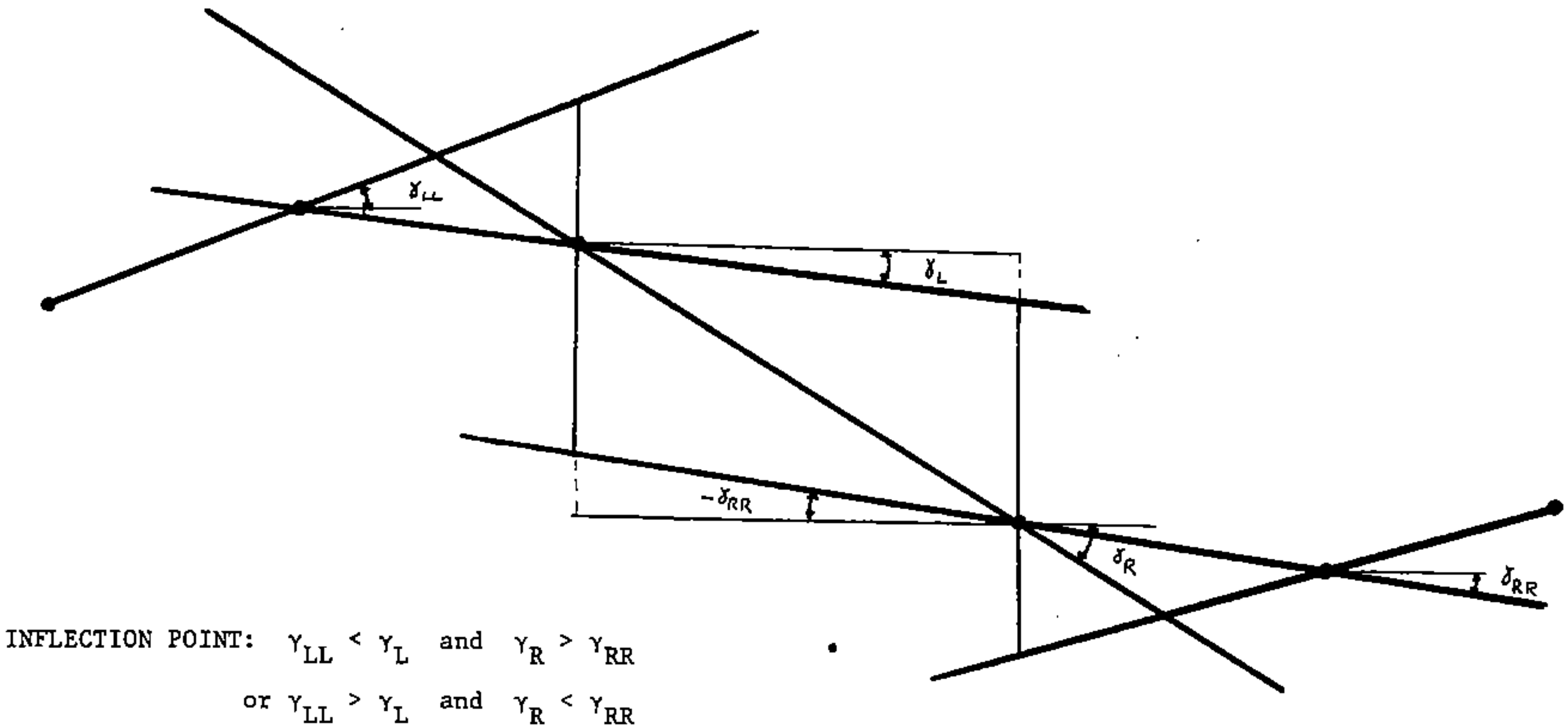


Figure 5.1 The geometric construction used to calculate the bounds on the quadrature errors in subdivision of a normal interval. The notation used in PAFAQ is also defined and the function TRIANGL computes the areas of the two interior triangles.



<u>Figure 5.2</u> The geometric construction used to calculate the bounds on the quadrature errors in the subdivision of intervals near an inflection point. The function QUADRIL computes the area of the quadrilateral that occurs.

we have, for d sufficiently small, that

$$\text{BOUND}(I) \le 2|f''(x)|d^3$$

Proof. Recall that BOUND(I) is just the area of the bounding triangle (see Figure 5.1) so we need to estimate the area of this triangle. Its area is given by bh/2 where

$$b^2 = d^2 + (\text{FRIGHT}(I) - \text{FLEFT}(I))^2$$

$$h = b/(\cot\alpha + \cot\beta)$$

α, β = angles of the triangle at left and right vertices

The geometry is invariant under rotation, so we may assume that FRIGHT(I) − FLEFT(I) = 0. Let ξ_L (and ξ_R) be mean values in interval I and its left (and its right) neighbor interval so that

$$\tan\alpha = |f'(\xi_L)| = (\xi_I - \xi_L)|f''(\eta_L)| = d_L|f''(\eta_L)|$$
$$\tan\beta = |f'(\xi_R)| = (\xi_R - \xi_I)|f''(\eta_R)| = d_R|f''(\eta_R)|$$

where η_L (and η_R) are mean values between ξ_L (and ξ_R) and the point ξ_I where $f'(x) = 0$. Set $d^* = \min(d_R, d_L)$ and note that $d^* \le 2d$ since at least one of the neighbors of I has length d or smaller. Further let d be small enough so that $f''(x)$ does not change by a factor more than 2 in the interval $[x-d, x+2d]$. Then we have

$$h = b/(1/d_L|f''(\eta_L)|) + 1/(d_R|f''(\eta_R)|))$$
$$= dd^*/d^*/(d_L|f''(\eta_L)|) + d^*/(d_R|f''(\eta_R)|))$$

For concreteness assume that $d^* = d_R$ and then we have, with $0 < \theta < 1$

$$h \le 2d^2/(\theta/f''(\eta_L) + 1/f''(\eta_R)) \le 2d^2|f''(\eta_L)|$$
$$\le 4d^2|f''(x)|$$

The area in this case is then bounded by $2d^3|f''(x)|$ which establishes the lemma.

LEMMA 5.4. Let $f(x)$ satisfy Assumption 1. Assume the I-th interval and its two neighbors have a singularity of $f(x)$ but not an inflection point and the I-th interval is not one of the end intervals. Then with the notation of Lemma 5.3 we have, for d sufficiently small, that

$$\mathrm{BOUND(I)} \leq K(x)d^2$$

where $K(x)$ is independent of d but dependent upon x.

Proof. Since no inflection point is involved, the function $f(x)$ is convex or concave in the I-th interval and hence BOUND(I) is again the area of the bounding triangle. It is clear that $f(x)$ cannot be infinite except at an inflection point or at the end points. Thus the worst discontinuity that can occur in such an interval is a jump discontinuity of $f'(x)$. For d sufficiently small we see that at most one such singularity exists in the I-th interval or its two neighbors.

Let θ be the jump in $f'(x)$ in these intervals and for d sufficiently small we have that the total variations in $f'(x)$ in these intervals is bounded by 2θ. As in the proof of Lemma 5.3 we may assume that FRIGHT(I) - FLEFT(I) = 0 and with the formluas used there we find that

$$b = d$$

$$h = d/(\cot\alpha + \cot\beta)$$

$$\cot\alpha,\ \cot\beta \geq \cot 2\theta$$

so that the area of the triangle is bounded by

$$\tfrac{1}{2}bh \leq \tfrac{1}{2}d^2/(2\cot 2\theta) = \frac{d^2}{4\cot 2\theta}$$

which establishes the conclusion.

LEMMA 5.5. Let $f(x)$ satisfy Assumption 1. Assume that INFLECT(I) $\neq 0$ or that the I-th interval is one of the two end intervals. Then, with the notation of Lemma 5.3 we have, for d sufficiently small, that

$$\mathrm{BOUND(I)} \leq K(x)d^{\alpha+1}$$

where K is independent of d.

Note that this lemma gives an unduly pessimistic value for BOUND(I) if the intervals in question do not contain singularities of f(x). One can establish bounds comparable to those of Lemma 5.3 for the end or inflection point intervals if f(x) is not singular. However, the trigonometry is tedious and the final conclusions are unchanged so this situation is not considered here.

Proof: First consider the two end intervals. Assume that $d = XRIGHT(I) - XLEFT(I)$ is small enough so that $f''(x)$ does not change sign in this end interval. There are two cases: first when $f'(x)$ and $f''(x)$ have the same sign near the end point (the end point may be a singularity in this case). It is easily seen that in this case the triangle area is bounded by d times the difference in the f(x) values at the two end points of the interval. Assumption 1 implies that this difference is at most Kd^{α} and consequently we have $BOUND(I) \leq Kd^{\alpha}$ in this case. In the second case where $f'(x)$ and $f''(x)$ have opposite signs there is no possibility of a singularity. The triangle area is seen to be bounded by d times d tan β where tan β is the slope of the secant line for the next to the end interval. Thus $\tan \beta = f'(\xi)$ for some mean value point ξ and for d sufficiently small tan β is bounded independently of d. Consequently in this case we have $BOUND(I) \leq Kd^{\alpha+1}$ for some constant K.

Now consider one of the three intervals near an inflection point with $INFLECT(I) \neq 0$. We may assume that d is small enough that $f'(x)$ is of constant sign in these three intervals (including the possibility that $|f'(x)| = \infty$ at the inflection point). In each of these three intervals it is seen that the triangle area or quadrilateral area used in computing BOUND(I) has its area bounded by d times the difference in the f(x) values at the two end points of the intervals. Assumption 1 implies that this difference is at most Kd^{α} and consequently we have $\mathrm{BOUND(I)} \leq Kd^{\alpha+1}$ and this concludes the proof.

We now recall Assumption 2 from [4] concerning error estimates and state it in the particular situation of this paper. The use of comparisons of BOUND(I) with DISCARD * DX rather than merely DISCARD makes these two relations equivalent to $\mathrm{ERROR}(x,k) \leq k|f''(x)|d^2$, $\mathrm{ERROR}(x,k) \leq kd^{\alpha}$ as given in [4].

<u>ASSUMPTION 2</u>. <u>Consider the I-th interval of length d. There are constants K and α (the same as in Assumption 1) so that when d < CHARF5 we have</u>

(i) <u>If the I-th interval contains no singularities then</u>

$$\mathrm{BOUND(I)} \leq K|f''(x)|d^3$$

(ii) <u>If the I-th interval contains a singularity then</u>

$$\mathrm{BOUND(I)} \leq Kd^{\alpha+1}.$$

The objective is, of course, to show that if f(x) satisfies Assumption 1 then the computed values of BOUND satisfy Assumption 2. The preceding lemmas achieve this is essence but there are three technicalities. First, the analysis and program treat some intervals as containing singularities even when they do not contain singularities. Second, the analysis restricts the length d in ways other than the separation of singularities and inflection points. Third, a larger constant may be required than given in Assumption 1. Thus we introduce the following

<u>TERMINOLOGY</u>: <u>We say that the I-th interval contains a singularity if it (i) is an end interval, (ii) has INFLECT(I) ≠ 0 or (iii) contains an actual singularity of f(x). We take CHARF5 to be the smallest value required in the above proofs, namely one-fifth of the minimum of</u>

(i) separation between singular points and/or inflection points (= CHARF)

(ii) distance of inflection or singular points to the end of the interval (unless the end point is itself a singularity).

The value of K in Assumption 1 is increased, if necessary, to be larger than 2 (for Lemma 5.3), $\tan 2\theta/4$ for each jump discontinuity of $\tan \theta$ in $f'(x)$ (for Lemma 5.4) and $\tan \theta = 2f'(x)$ for $x = A$ and $x = B$ (for Lemma 5.5). Note that this terminology still leaves us with a finite number of intervals containing a singularity and K is still finite because the number of jump discontinuities in $f'(x)$ is finite and CHARF5 is still positive.

We now state a crucial result concerning the effectiveness of this program.

THEOREM 5.3. With the terminology introduced above we have that if $f(x)$ satisfies Assumption 1 then the computed values of BOUND(I) satisfy Assumption 2.

Proof. Theorem 5.1 and its corollary establish that PAFAQ computes the areas of the triangles and quadrilaterals correctly and correctly obtains values for local and global error estimates. Lemmas 5.3, 5.4 and 5.5 establish that these error estimates satisfy Assumption 2 provided that $f(x)$ satisfies Assumption 1.

We summarize the results of this section by

COROLLARY. The program PAFAQ has Attributes 1, 2 and 3 of AREAS.

6. THE CORRECTNESS AND CONVERGENCE RESULT FOR PAFAQ.

We first summarize one of the consequences of the previous section's analysis by saying that the program is correct in the sense that it has the attributes to be represented by the parallel metalgorithm of [4]. This fact is stated explicitly in the following:

THEOREM 6.1. The program PAFAQ is represented by the parallel metalgorithm of [4].

Proof. In order to establish this we must show that the program has the structure specified by the metalgorithm and that the elements of this structure have the required attributes. A comparison of the description in [4] with the program shows that the same structure is present and, in fact, the same names are used. Some subprograms of [4] have been implemented by using additional subprograms (TRIANGL and SPECIAL), but this does not alter the situation.

To see that the attributes are present as specified one has to check that all 32 of them have been established in the preceding three sections. This is in fact the case. Since PAFAQ is specific, certain variables in the metalgorithm description have constant values here. In particular we have $q = 1$ (in Attribute 5 of AREAS) and $p = 2$ in Assumption 1 about the integrand $f(x)$. Assumption 1 is made more specific in two other ways, namely that $f(x)$ has no cusps, and has a finite number of inflection points. Thus the attributes in AREAS are valid with respect to this more restrictive Assumption 1. This concludes the proof.

With this result we may now apply the main result (Theorem 5) of [4] to establish

THEOREM 6.2. Assume that $f(x)$ satisfies Assumption 1 and the computer operation is as described in Section 1. Then the program PAFAQ terminates with an estimate AREA requiring N evaluations of $f(x)$ so that

$$|If - AREA| \le EPS$$

with

$$N = \mathcal{O}(EPS^{-\frac{1}{2}})$$

<u>or, equivalently,</u>

$$|If - AREA| \le \mathcal{O}\left(\frac{1}{N^2}\right)$$

<u>If $N > NCPU^2$ then the total computation time $T_N f$ satisfies, for constants K_1, C_0 and C_1 as defined in [4],</u>

$$T_N f \le K_1 \frac{N*(4C_0 + 2C_1*NCPU)}{NCPU}$$

This theorem is very satisfactory in several ways. First, it specifies the result of the actual operation of the program, namely the program will terminate and print out a result for which these estimates are valid. This is a substantial improvement over the more usual result of mathematical convergence which merely states that a program will eventually compute a number for which these estimates are valid. Second it shows that the adaptive nature of the program enlarges the domain of efficiency of this program to include virtually all functions of interest in applications. Third, it shows explicitly the speed up achieved by parallelism in the computation. The constant C_1 equals $t_1 + t_2$ where t_1 is the maximum time in the critical parts of QPUT and QGET (17 statements). The time t_2 is the time for one attempt at access to the queue. Under certain circumstances this latter time can be as much as 41 statements (the maximum execution time of INSERT). The time for an attempt otherwise is seen from Lemma 4.2 to be 6 statements. Thus the maximum value of C_1 the order of 60 statements but the average value is likely to be 20-25 statements. These statements represent the portion of the computation which is <u>not</u> speeded up by the parallelism of the algorithm. The constant C_o is seen to be substantially larger, about 100. For large values of NCPU this implies a speed up of a factor of about 9.

The result if disappointing in that it shows that there is a definite limitation on the speed-up obtained from parallelism and that one must provide CHARF as input data to the program. The speed-up obtained here is not the best and deserves further analysis. On the other hand, it is not likely that the dependence on NCPU can be made better than (log NCPU)/NCPU. The input CHARF is essential to obtaining valid results from this (or any other) quadrature program. Without a knowledge of CHARF (or some equivalent information) there is no way to bound the error in the number returned by a quadrature program.

Finally there are two other troublesome questions: Is the program actually correct? and: How much computational efficiency has been sacrificed to obtain a completely reliable program? It is now realized that the answer to the first question is (for any program): "We do not know". Even so, there is a variety of program errors which the approach of this paper is not likely to detect. There are "clerical" errors and trivial omissions or oversights. Thus the program TRIANGL may be called TRIANGLE at some point and provisions might not be made for an input of EPS = -.001 (they are not in this case) or CHARF = 1000.*(B-A) (they are in this case). This variety of errors is much more likely to be detected by testing than by proving and testing presents a problem for a program written in a non-standard language for a hypothetical computer.

Some testing can be made in this case by using three approaches. First, the program can be translated into Fortran without much effort and executed in the usual sequential fashion. This does not test any of the parallelism of the algorithm, but it does check the initialization, the management of the data structure and the numerical analysis subprograms. Second, the parallelism can be simulated for the Fortran version. The simulation is straight-forward but tedious. One labels each Fortran statement and sets up an instruction counter for each CPU. One can then cycle through the CPUs executing one Fortran statement in each. The beginning of each subprogram is a large computed GOTO and a RETURN follows each statement executed in the algorithm. Special steps are required for subprogram calls and logical statements, but these are obvious. This approach was carried out on an earlier, more complex algorithm which allowed the number of CPUs to vary dynamically. Finally, one can translate the algorithm into a language which includes parallel simulation. Such language systems are primarily designed for modeling operating systems, but they may be quite suitable to test this program. One such language system is ASPOL available on CDC 6000 computers. Neither of these simulations gives truly asynchronous parallel operation as assumed in this paper. This approach has been carried out and a substantial number of tests made. The speed-up actually observed for several cases is shown in Figure 6.1. The speed-up is quite acceptable for this small number of CPUs.

We conclude that the combination of detailed proof and substantial testing via simulation leads to a very high level of confidence in the correctness of the program.

The answer to the question about efficiency is not so satisfactory. Everyone, of course, realizes that high reliability must cost something in efficiency for routine integrands. Experiments show that the algorithm normally detects oscillations and obtains correct answers even if CHARF is omitted or is much too large. For example with [A,B] = [0,1] the function f(x) might have a peak with two inflection points at x = .49 and x = .51. This forces the program to use subintervals of length .004

everywhere even though they are unlikely to be required to be that short for most of the interval. The adaptive nature of the algorithm normally detects the peak and arrives at a much more logical choice of subintervals. However, there is then no way to avoid the exceptional case where fine oscillations are missed and incorrect results produced. CADRE [1] is an example of an adaptive quadrature program which is almost certain to detect fine oscillations but integrands can be constructed where it fails.

It is clear that one can gain efficiency by allowing the information provided about f(x) to be more detailed. This complicates the program development and use but, if well done, probably would result in a more satisfactory algorithm.

REFERENCES

[1] Carl de Boor, CADRE: An algorithm for numerical quadrature, in Mathematical Software (J. R. Rice, Ed.), Academic Press, 1971, Chapter 7, pp. 417-449.

[2] John R. Rice, A metalgorithm for adaptive quadrature. To appear in J. Assoc. Comp. Mach.

[3] ____________, Parallel algorithms for adaptive quadrature - Convergence, Proceedings IFIP Congress '74, North-Holland Publishing (1974) pp. 600-604.

[4] ____________, Parallel algorithms for adaptive quadrature II - metalgorithm correctness, CSD-TR 107, Computer Science Department, Purdue University, November 1973. (Revised, November 1974), 23 pages.

Figure 6.1. Measured speed-up obtained for PAFAQ by simulation.