1974

Parallel Algorithms for Adaptive Quadrature III - Program Correctness

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

Report Number:
74-112

https://docs.lib.purdue.edu/cstech/64

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
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

\[
\text{IF}(X \text{EQ} 4.2) \text{ THEN } Y = X, \text{GO TO 5}
\]

\[
\text{ELSE } X = \cos(DX + Y^{**.42})/(7.1*X + 3.2*ALOG(DX + .1)) + X,
\]
\[
DX = \text{AMAX1}(DX, Y^{**.42})
\]

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 PAFAP (Parallel Algorithm For Adaptive Quadrature) and the metaalgorithm from [4]. The objective is to show that PAFAP is represented by this metaalgorithm. 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.

<table>
<thead>
<tr>
<th>PROCESSOR</th>
<th>PROGRAMS</th>
<th>REMARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU1</td>
<td>MAIN</td>
<td>Sets the number of CPUs and initiates them.</td>
</tr>
<tr>
<td>CPU2</td>
<td>MAIN</td>
<td>Reads problem definition and controls algorithm.</td>
</tr>
<tr>
<td>CPU(IP)</td>
<td>MAIN</td>
<td>Initializes variables of the algorithm.</td>
</tr>
<tr>
<td></td>
<td>QGET</td>
<td>Controls the interval processing, estimation of areas and bounds and access to the interval collection.</td>
</tr>
<tr>
<td></td>
<td>AREAS</td>
<td>Obtains an interval for the processor from the interval collection.</td>
</tr>
<tr>
<td></td>
<td>QPUT</td>
<td>Computer areas, bounds and associated quantities.</td>
</tr>
<tr>
<td></td>
<td>INSERT</td>
<td>Obtains access to the unallocated memory and locates places to insert completed intervals into the collection.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inserts the completed intervals into the interval collection.</td>
</tr>
</tbody>
</table>

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.
PARALLEL ALGORITHM FOR ADAPTIVE QUADRATURE

BY

J. P. P. FORBES UNIVERSITY

This algorithm is written in a pseudo-FORTRAN II form to make it
more understandable and concise. The changes from FORTRAN II such
that a translation into Fortran can be made in a minimum number of
changes used are listed below.

1. Global variables --- are declared as a separate section in each
   group of statements called "Declaration" is deleted and replaced
   simply have the statement

   DECLARE

   To note that these declarations are in Fortran II.

   Variables not declared global are local to their declaration
   or a subroutine.

   IF ( K = EQ. 3 ) THEN J = 0, NK = NK+1, GO TO 20
   ELSE K = K+1.

   THE HANDS THEN AND ELSE ARE RESERVED KEYWORDS.

2. Arbitrary expressions --- are allowed as array indices, do-loop
   ranges, etc.

3. FORMAT FREE I/O --- is assumed. There is very little I/O.

4. Multiple assignment statements --- are allowed.

5. Variable dimensions are used for arrays.

   LINK --- FOR THE INTERNAL COLLECTION INITIATION
   LINKF --- FOR THE CPU DEPENDENT INITIATION

***ALGORITHM STRUCTURE ***

PROCESSOR NAME PROGRAMS REMARKS

CPU1 --- MAIN

   THIS IS THE OPERATING SYSTEM. IT ARRANGES
   THE PROCESSES AND Passes CONTROL TO THE
   ALGORITHM ITSELF.

CPU2 --- MAIN

   BEGIN

   THIS PROGRAM CONTROLS THE INITIATION
   ONCE CONTROL IS PASSED TO LINK.

CPU2LINKCPU MAIN

   BEGIN

   THIS IS AN ARRAY OF PROCESSES WITH AN
   ARRAY OF IDENTICAL PROGRAMS. THIS
   PROGRAM CONTROLS THE PROCESSING OF
   INDIVIDUAL INTERVALS.

   BEGIN

   THIS PROGRAM COMPUTES AREAS AND BOUNDS
   IT HAS THIS SUBPROGRAMS:
   INITIAL -- FOR INITIAL AREA
   FINAL -- FOR THE INTERVALS
   INTERNAL, -- FOR THE INITIATION.

   INITIAL

   THIS PROGRAM PAKTS THE INTERVALS.
PROGRAM DECLARE

********** THIS PROGRAM CONTAINS THE VARIABLE DECLARATIONS:

COMMON / CPSYS / NPUL, CPUH, CPUQ, CPUH, CPUQ, CPUH, CPUQ,
NPUL - NUMBER ACTIVATED CPU'S LESS 100
CPUH - SWITCH TO ACTIVATE CPU
CPUQ - SWITCH TO ACTIVATE CPU
CPUQH - SWITCH TO ACTIVATE CPU(H), IP = 1 TO NPUL

COMMON / PROBLEM / A, B, EPS, CHARE
A, B - END POINTS OF INTERVAL OF INTEGRATION
EPS - ACCURACY REQUIRED IN NUMERICAL ESTIMATE
CHARE - CHARACTERISTIC LENGTH FOR F(x), F(x) HAS NO
INFLECTION POINTS CLOSER THAN 2, CHARE.

COMMON / CONTROL / AREA, BOUND, DISCARD, LINCOL, LPUL, FINISH
AREA - CURRENT AREA ESTIMATE
BOUND - CURRENT BOUND ON ERROR IN AREA
DISCARD - IN INTERVAL IS DISCARD WHEN THE BOUND ON ITS
AREA ESTIMATE (BEST) IS LESS THAN CURRENT
LIMIT - LIMIT ON THE NO. OF INTERVALS IN THE COLLECTION
INCREASE LIMIT MARKS THE END OF THE LOOP
LPUL - LIMIT ON THE NO. OF CPU'S (SIDE CPU + CPU)
FINISH - SWITCH TO TERMINATE EXECUTION IF TRUE.

COMMON / QUEUE / LEADER, NO, LAST, INDEX, LIMIT
BEST, LIMIT, BOUND, LIMIT, INFLCT, LIMIT
RIGHT, LIMIT, LEFT, LIMIT, ILEFT, LIMIT
FREE, INDEX, NEXT, THRESH, IDT, TELLING, NEXT

LEADER - NEXT INTERVAL IN COLLECTION TO BE ASIGNED TO
A CPU. IT IS THE HEAD OF THE QUEUE.
NO - LAST INDEX USED IN THE ARRAYS FOR THE QUEUE.
LAST - TAIL OF THE QUEUE
INDEX - INDEX OF THE NEXT INTERVAL IN THE QUEUE
INFLCT - SWITCH FOR INTERVAL STATUS
THRESH - INTERVAL IS IN COLLECTION
FALSE - INTERVAL BEING PROCESSED BY SOME CPU
LIMIT - AREA ESTIMATE FOR THIS INTERVAL
BOUND - BOUND ON ERROR IN BEST
INFLECT - SWITCH INDICATING INFLECTION POINT IS POSSIBLE
= 0 - NO INFLECTION POINT
= LEFT - LEFT INTERVAL OF A SET OF THREE
= CENTER - CENTER INTERVAL OR A SET OF THREE
= RIGHT - RIGHT INTERVAL OF A SET OF THREE

COTHN - COTANGENT OF SEGMENT IN NPUL INTERVAL
COTH - COTANGENT OF BOUNDING LINE ON THE PATH
COT - COTANGENT OF BOUNDING LINE ON THE HIST

THRESH - ARE NORMALLY COTANGENT VALUES OF NEIGHBORS,
BUT MAY BE MERELY COTANGENT OF BOUNDING LINES.

RIGHT, LEFT - ABSCISSA OF END POINTS OF INTERVAL.
PROGRAM CPUL

**THIS IS THE OPERATING SYSTEM SIMULATION**

IT IS ASSUMED THAT ALL NUMERIC VARIABLES ARE INITIALLY ZERO AND ALL LOGICAL VARIABLES ARE INITIALLY FALSE.

NCPU = 5

TURN ON THE TWO SPECIAL CPU'S
CPU0ON = CPU20N = .TRUE.

TURN ON SAME PROCESSOR CPU'S
DO 100 K = 1, NCPU
CPU0RKN(K) = .TRUE.
100 CONTINUE

IDLE LOOP WAITING FOR ALGORITHM TO TERMINATE
200 IF (.NOT. FINISH) GO TO 200

TURN OFF ALL CPU'S
DO 300 K = 1, NCPU
CPU0R(K) = .FALSE.
300 CONTINUE
CPU0N = CPU20N = .FALSE.
STOP
END

PROGRAM CPU2
C ********** THIS IS THE ALGORITHM CONTROLLER
C**CALL DECLARE
C
C PROBLEM DEFINITION
READ A, B, EPS, CHAP
C CALL BEGIN
C
10 IF( BOUND . L.E. EPS ) FINISH = .TRUE.
C
C CHECK FOR ABNORMAL TERMINATION
IF( NO . GE. LIM ) 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 ' BOUND
STOP
END

C------------------ SUBROUTINE BEGIN ----------------------
C
C ********** THIS PROGRAM INITIALIZES THE ALGORITHM
C THE ORIGINAL INTERVAL IS BROKEN UP INTO LENGTHS OF CHAP
C AND THIS TEST IS NEVER NEEDED LATER. THE INTERVALS
C WITH INFECTION POINTS ARE DETERMINED.
C
(*CALL DECLARE
C
C MISC. INITIALIZATIONS
C AREA = 0.
C DISCARD = EPS/(B-A)
C FIND THE INITIAL INTERVAL LENGTH
DOL = AMIN1(CHAP/.5.,B-A)
NO = (B-A)/DOX
IF( DOL+NO .LT. B-A ) NO = NO + 1
DOX = (B-A)/NO
LAST = NO
LEADER = 1
C
C FIRST SET OF QUANTITIES FOR INITIAL INTERVALS
DO 100 K = 1,NO
XRIGHT(K) = A + (K-1)*DOX
XLEFT(K) = XRIGHT(K) - DOX
FRIGHT(K) = F(XRIGHT(K))
FLEFT(K) = F(XLEFT(K))
REST(K) = .5*(XRIGHT(K)-XLEFT(K) + FRIGHT(K))
AREA = AREA + REST(K)
C0HIN(K) = DOL/FRIGHT(K)-FLEFT(K))
IGNIT(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(NO) = LINO
IRIGHT(NO) = LINO
INEXT(NO) = LINO
C
C SECOND SET OF QUANTITIES FOR INITIAL INTERVALS
COTR(K) = COTR(K-1)
COTL(K) = COTR(K-1) = 0.
INFECT(K) = INFECT(K-1) = 0.
Determine intervals where cotangent differences change sign
These are center intervals of triplet which may have inflection
 Skip if we only have 1 or 2 intervals
IF( NO .LE. 1 ) GO TO 201
COTR(1) = COTR(2)
201 CONTINUE
$C\text{O}\text{T}(N) = \text{COT}(N-1)$

IF ( $N$ .LE. 2 )
GO TO 201

$C\text{O}\text{T}(K) = \text{COT}(K-1)$
$C\text{O}\text{T}(K) = \text{COT}(K+1)$

1 IF (ABS($C\text{O}\text{T}(K) - \text{COT}(K)$) .EQ. 1)
2 THEN INFLECT(K) = 0
3 ELSE INFLECT(K) = CENTER
4 INFLECT(K-1) = LEFT, INFLECT(K+1) = RIGHT
5 CONTINUE
6
7 Call AREAS(IP)
8 Attempt to get an interval from the queue
9 Call GGET(IP)
10 Have one; compute areas and bounds
11 Call AREAS(IP)
12 Attempt to get places in the queue to put intervals
13 Call CPU1(IP)
14 Insert the intervals into the queue
15 Call INSERT(IP)
16 Restart the process
GO TO 10
SUBROUTINE AREAS(IP)

**FUNCTION**

THIS PROGRAM COMPUTES AREA ESTIMATES.

**CALL DECALRE**

IRI = INSEIGN(IP)

**PRELIMINARY QUANTITIES**

DX(IP) = .5*(RIGHT(IP) - LEFT(IP))
XMID(IP) = RIGHT(IP) - DX(IP)
FXMID(IP) = F(XMID(IP))
COTANK(IP) = D(X(IP))/(RIGHT(IP) - XMID(IP))
COTANL(IP) = D(X(IP))/(XMID(IP) - LEFT(IP))

**CHECK INTERVAL SITUATION AND SELECT AREA FORMULAS**

INFLECT = 0 IS THE NORMAL CASE

IF (INFLECT(IP).EQ. 0) THEN
1 BOUNDL(IP) = TRIANGL(COL(IP), COTL(IP), COTANL(IP), COTR(IP)),
2 D(X(IP), F(L(EFT(IP) - XMID(IP))
3 BOUNDR(IP) = TRIANGL(COTANK(IP), COTR(IP), CR(IP)),
4 D(X(IP), F(RIGHT(IP) - F(LIGHT(IP))
5 INFLECT(IP) = 0

ELSE

USE SPECIAL FORMULAS

CALL SPECIAL(COL(IP), COTL(IP), COTANL(IP), COTR(IP), INSENG(IP), IP)

**CHECK DISCARDING OF INTERVALS**

IF (BOUNDL(IP) .LT. DISCARD*DX(IP) ) IRETURN(IP) = 1
IF (BOUNDR(IP) .LT. DISCARD*DX(IP) ) IRETURN(IP) = IRETURN(IP) + 1

**COMPUTE CHANGES IN AREA AND BOUNDARY**

AREAR(IP) = .5*DX(IP)*COTTAN(IP) + F(LIGHT(IP))
AREAL(IP) = .5*DX(IP)*COTTAN(IP) + LEFT(IP)
CHANGE(IP) = BOUND(IP) - BOUNDR(IP) - BOUNDL(IP)
ACHANGE(IP) = AEST(IP) - AREAR(IP) - AREAL(IP)
RETURN
END

**FUNCTION**

TRIANGLES(CLFT, CENT, CRGT, XBASE, YBASE)

**FUNCTION**

THIS FUNCTION FINDS TRIANGLE AREAS FROM COTANGENTS OF THE SIDES AND THE HORIZONTAL AND VERTICAL PROJECTIONS OF THE BASE.

**CALL DECLARE**

COT = ((CRGT*CENT + 1.)/(CRGT - CENT))
COTLFT = ((CENT+CLFT + 1.)/(CENT - CLFT))
BASE2 = YBASE*YBASE + YBASE*YBASE + 2
TRIANGLE = .5 BASE2*COT(COT + COTLFT))
RETURN
END

SUBROUTINE SPECIAL(COL, CL, CR, COT, INFLECT, IP)

**FUNCTION**

THIS PROGRAM COMPUTES AREA ESTIMATES FOR AN INTERVAL WHERE AN INFLATION POINT MAY BE PRESENT.

THERE ARE THREE CASES ACCORDING TO THE VALUE OF INFLECT.

THE ARGUMENTS CORRESPOND TO GLOBAL VARIABLES AS FOLLOWS:

COL = COTL(IP)
CL = COTANL(IP)
CR = COTANK(IP)

**END**
C CRR = C0TR(IAI)
C INFLECT = INFLECT(IAI)
C
THE CORRECT VALUES FOR INFLECT OF THE TWO HALVES ARE SET
C INFLECT(IP) = FOR THE LEFT HALF
C INFLECT(IAI) = FOR THE RIGHT HALF
C
THE MODIFICATION OF INFLECT FOR NEIGHBORING INTERVALS IS
AN APPARENT VIOLATION OF ATTRIBUTE 4 OF AREAS. HOWEVER,
IT IS SHOWN THAT THIS ACTION DOES NOT INVALIDATE THE
ALGORITHM. IT IS CONVENIENT AND POINTLESS TO SAVE THE
INFORMATION AND MODIFY INFLECT LATER.
C
DECLARE
C ARITHMETIC STATEMENT FUNCTIONS
C
DETERMINE MONOTONICITY OF COTANGENT SEQUENCE ON THREE POINTS
CHANGE(K) = ABS(CR+CL-CR)
1 - (ABS(CR-CL)+ABS(CR-CL)+ABS(CL-CL))
C AREA OF QUADRILATERAL FOR CENTER INTERVAL
QUADRIL(CLFT,CRFT,DX,DF) = DX*ABS(DF+1/CLFT+1/CRFT)
C
IAI = IASSIGN(IP)
DFL = FMDX(IP) -FLECT<IAI)
DFR = RIGHT<IAI) - FMDX(IP)
C
SELECT ONE OF THREE CASES FOR INFLECT
IF( INFLECT EO CENTER ) GO TO 300
IF( INFLECT EO RIGHT ) GO TO 200
C
INFLECT = LEFT
IF( CHANGE(IP) EO 0 ) THEN
1 THE INFLECTION POINT MAY ONLY BE IN THE LEFT HALF
BOUND(IP) = TRIANGL(CR,CL,DX(IP),DFL)
2 BOUNDR(IP) = TRIANGL(CR,CL,DX(IP),DFR)
3 INFLECT(IP) = 0
C 4 ELSE
THE INFLECTION MAY BE IN EITHER HALF
5 BOUND(IP) = TRIANGL(CR,CL,DX(IP),DFL)
6 BOUNDR(IP) = TRIANGL(CR,CL,DX(IP),DFR)
7 INFLECT(IP) = LEFT, INFLECT(IAI) = CENTER
C UPDATE INFLECT VALUES TO THE RIGHT
8 INFLECT(RIGHT<IAI) = RIGHT
9 INFLECT(RIGHT<RIGHT<IAI)) = 0
RETURN
C
200 CONTINUE
C INFLECT = RIGHT
IF( CHANGE(IP) EO 0 ) THEN
1 THE INFLECTION POINT MAY ONLY BE IN THE RIGHT HALF
BOUND(IP) = TRIANGL(CR,CL,DX(IP),DFL)
2 BOUNDR(IP) = TRIANGL(CR,CL,DX(IP),DFR)
3 INFLECT(IP) = RIGHT, INFLECT(IAI) = 0
C 4 ELSE
THE INFLECTION MAY BE IN EITHER HALF
5 BOUND(IP) = TRIANGL(CR,CL,DX(IP),DFL)
6 BOUNDR(IP) = TRIANGL(CR,CL,DX(IP),DFR)
7 INFLECT(IP) = CENTER, INFLECT(IAI) = RIGHT
C UPDATE INFLECT VALUES TO THE LEFT
8 INFLECT(LEFT<IAI)) = LEFT
9 INFLECT(LEFT<LEFT<IAI)) = 0
RETURN
C
300 CONTINUE
C INFLECT = CENTER
IF( ABS(CL-CR) LT. ABS(CL-CR) + ABS(CR-CR) ) THEN
C COTANGENTS ARE NOT MONOTONE ON THE POINT HALF
C AND IT IS THE NEW CENTER.
16

1 \text{BOUND\(\langle IP\rangle\)} = \text{TRIANGL}(\text{CLL}, \text{CLR}, \text{DX}(\text{IP}), \text{DFL})
2 \text{BOUND\(\langle IP\rangle\)} = \text{QUADRIL}(\text{CLL}, \text{CCR}, \text{DX}(\text{IP}), \text{DFR})
3 \text{INF\(\langle IP\rangle\)} = \text{LEFT}, \text{INFLECT\(\langle IRI\rangle\)} = \text{CENTER}
4 \text{UPDATE INFLECT TO THE LEFT}
5 \text{INFLECT\(\langle \text{LEFT}(\langle IAI\rangle)\rangle\)} = 0
6 \text{ELSE}
7 \text{THE LEFT HALF IS THE NEW CENTER}
8 \text{BOUND\(\langle IP\rangle\)} = \text{QUADRIL}(\text{CLL}, \text{CLR}, \text{DX}(\text{IP}), \text{DFL})
9 \text{BOUND\(\langle IP\rangle\)} = \text{TRIANGL}(\text{CLL}, \text{CCR}, \text{DX}(\text{IP}), \text{DFR})
10 \text{INF\(\langle IP\rangle\)} = \text{CENTER}, \text{INFLECT\(\langle IAI\rangle\)} = \text{RIGHT}
11 \text{UPDATE INFLECT TO THE RIGHT}
12 \text{INFLECT\(\langle \text{RIGHT}(\langle IAI\rangle)\rangle\)} = 0
13 \text{RETURN}
14 END

SUBROUTINE INSERT\(\langle IP\rangle\)

******* THIS PROGRAM INSERTS THE INTERVAL INTO ASSIGNED PLACES 
OF THE INTERVAL COLLECTION. IF 2 INTERVALS ARE KEPT THEN 
LEFT GOES INTO \text{ASSIGN\(\langle IP\rangle\)} = \text{IAI} = \text{IL}
RIGHT GOES INTO \text{KORETRAN\(\langle IP\rangle\)} = \text{IR}

\text{CALL DECLARE}

\text{IL} = \text{IAI} = \text{ASSIGN\(\langle IP\rangle\)}
\text{IR} = \text{KORETRAN\(\langle IP\rangle\)}

\text{CHECK ABOUT DISCARDING RIGHT INTERVAL}
\text{IF} (\text{BOUND\(\langle IP\rangle\)}, \text{LT. DISCARD\(\langle DX\(\langle IP\rangle\rangle\)}) \text{THEN}
\text{DISCARD THE RIGHT INTERVAL, SKIP ITS INSERTION}
1 \text{LEFT\(\langle \text{RIGHT}(\langle IAI\rangle)\rangle\)} = \text{LIMQ}, \text{IR} = \text{LIMQ}
2 \text{GO TO 200}

\text{INSERT RIGHT INTERVAL INTO IAI = II IF LEFT ONE IS DISCARDED, IF (BOUND\(\langle IP\rangle\), \text{LT. DISCARD\(\langle DX\(\langle IP\rangle\rangle\)}) \text{THEN IR} = \text{IL}, \text{IL} = \text{LIMQ}

\text{INSERT RIGHT INTERVAL INFORMATION INTO THE COLLECTION}
\text{RIGHT\(\langle IR\rangle\)} = \text{RIGHT\(\langle IAI\rangle\)}
\text{RIGHT\(\langle IRI\rangle\)} = \text{RIGHT\(\langle IAI\rangle\)}
\text{LEFT\(\langle IR\rangle\)} = \text{LEFT\(\langle IAI\rangle\)}
\text{LEFT\(\langle IRI\rangle\)} = \text{LEFT\(\langle IAI\rangle\)}
\text{BOUND\(\langle IR\rangle\)} = \text{BOUND\(\langle IAI\rangle\)}
\text{BOUND\(\langle IRI\rangle\)} = \text{BOUND\(\langle IAI\rangle\)}
\text{COSTAN\(\langle IR\rangle\)} = \text{COSTAN\(\langle IAI\rangle\)}
\text{COSTAN\(\langle IRI\rangle\)} = \text{COSTAN\(\langle IAI\rangle\)}
\text{COST\(\langle IR\rangle\)} = \text{COST\(\langle IAI\rangle\)}
\text{COST\(\langle IRI\rangle\)} = \text{COST\(\langle IAI\rangle\)}
\text{INFLECT\(\langle IR\rangle\)} = \text{INFLECT\(\langle IAI\rangle\)}
\text{INQUEUE\(\langle IR\rangle\)} = \text{TRUE}

\text{CHECK ABOUT DISCARDING LEFT INTERVAL}
\text{IF (BOUND\(\langle IP\rangle\), \text{LT. DISCARD\(\langle DX\(\langle IP\rangle\rangle\)}) \text{THEN}
\text{DISCARD THE LEFT INTERVAL, SKIP ITS INSERTION}
1 \text{RIGHT\(\langle \text{LEFT}(\langle IAI\rangle)\rangle\)} = \text{LIMQ}
2 \text{GO TO 200}

\text{INSERT LEFT INTERVAL INFORMATION INTO THE COLLECTION}
\text{RIGHT\(\langle IL\rangle\)} = \text{XMIN\(\langle IP\rangle\)}
\text{RIGHT\(\langle IRI\rangle\)} = \text{XMIN\(\langle IP\rangle\)}
\text{LEFT\(\langle IL\rangle\)} = \text{LEFT\(\langle IAI\rangle\)}
\text{LEFT\(\langle IRI\rangle\)} = \text{LEFT\(\langle IAI\rangle\)}
\text{BOUND\(\langle IL\rangle\)} = \text{BOUND\(\langle IAI\rangle\)}
\text{BOUND\(\langle IRI\rangle\)} = \text{BOUND\(\langle IAI\rangle\)}
\text{COST\(\langle IL\rangle\)} = \text{COST\(\langle IAI\rangle\)}
\text{COST\(\langle IRI\rangle\)} = \text{COST\(\langle IAI\rangle\)}
\text{INFLECT\(\langle IL\rangle\)} = \text{INFLECT\(\langle IAI\rangle\)}
\text{INQUEUE\(\langle IL\rangle\)} = \text{TRUE}
ILEFT (IL) = ILEFT (IL-1)
IRIGHT (IL) = IR
CSTR (IL) = CSTR (IP)
CCTL (IL) = CCTL (IP)
INFLCT (IL) = INFL (IP)
INQUEUE (IL) = TRUE.

INSERTION COMPLETED
300 RETURN
END

SUBROUTINE OGET (IP)

************ THIS PROGRAM GAINS ACCESS TO THE HEAD OF THE QUEUE IN
ORDER TO OBTAIN AN INTERVAL.

CALL DECLARE

CHECK TO SEE IF THE QUEUE IS NOT EMPTY AND THE LEADER IS FREE
10 IF (LEADER .EQ. LINO) GO TO 10

ENTER COMPETITION FOR ACCESS TO THE QUEUE LEADER
IF (.NOT. QFREE) THEN
CHECK TO SEE IF THIS IS THE NEXT CPU IN ORDER
IF NOT, RETURN TO COMPETITION FOR QUEUE ACCESS
20 IF (IP .NE. NEXTQ) GO TO 10

------- PRIORITY WAITING LOOP BEGINS -------
THIS IS THE NEXT CPU IN ORDER
2
WAITING = .TRUE.

WAIT, SEE IF WAITING WAS TESTED WHILE BEING CHANGED
3 CONTINUE
IF 30, THEN EXIT AND RETURN TO COMPETITION
NEXT STAT. IS A 1-LINE IF-THEN-ELSEF
4 IF (QFREE) THEN WAITING = .FALSE., GO TO 10

IDLE LOOP WAITING TURN
5 30 IF (WAITING) GO TO 30

CHECK THAT PREVIOUS ACCESS DID NOT EXHAUST COLLECTION
IF COLLECTION IS EMPTY, WAIT IN IDLE LOOP
35 IF (LEADER .EQ. LINO) GO TO 35

IT IS NOW THIS CPU'S TURN TO OBTAIN ACCESS TO THE QUEUE
40 GO TO 50

------- PRIORITY WAITING LOOP ENDS -------

HAVE ENTERED GATE TO THE QUEUE, NOW CLOSE GATE BEHIND US.
49 QFREE = .FALSE.
COND = IP
DELAY LONG ENOUGH SO ALL CPUS THAT FALL THRU THE ABOVE
IF STATEMENTS ARE BETWEEN THE PREVIOUS AND THE FOLLOWING STATS.
CONTINUE

CHECK TO SEE IF THIS WAS THE LAST CPU TO SET IDQ
IF NOT, RETURN TO COMPETITION FOR QUEUE ACCESS
54 IF (IP .NE. IP) GO TO 40

--------- START CRITICAL PART ----------

CHECK THAT LEADER IS ACTUALLY AVAILABLE, WAIT IF NOT
56 IF (.NOT. INQUEUE (LEADER) ) GO TO 50

HAVE GAINED ACCESS TO THE QUEUE LEADER
LEADER = IP
ASSIGN (IP) = LEADER

MARK LEADER AS NOT IN THE QUEUE
INQUEUE (LEADER) = .FALSE.
LEADER = INERT (LEADER)

DELAY A STATEMENT TO ALLOW TIME FOR CPU WITH IP = NEXTQ TO BE
PUT INTO WAITING STATUS AT 30 ABOVE
CONTINUE

OPEN GATE IF NO CPU IS WAITING INSIDE IT
1 IF( WAITING ) THEN WAITING = .FALSE.
ELSE FREE = .TRUE.

INCREASE THE INDEX FOR THE NEXT CPU IN ORDER
NEXT = MOD(NEXT+CPU) + 1

------ END CRITICAL PART ------

HAVE FINISHED WITH QUEUE ACCESS
RETURN
END

SUBROUTINE CPU(IP)

-- THIS PROGRAM OBTAINS ACCESS TO THE TAIL OF THE QUEUE
-- IN ORDER TO OBTAIN PLACES TO PUT THE NEW INTERVALS.

CALL DECLARE

ENTER COMPETITION FOR ACCESS TO THE TAIL
10 IF( .NOT. TFREE ) THEN

CHECK TO SEE IF THIS IS THE NEXT CPU IN ORDER
IF NOT, RETURN TO COMPETITION FOR TAIL ACCESS

20 IF( .IP .NE. NEXT ) GO TO 10

-------- PRIORITY TAILING LOOP BEGINS -----

THIS IS THE NEXT CPU IN ORDER

2 TAILING = .TRUE.

WAIT, SEE IF TAILING WAS TESTED WHILE BEING CHANGED
CONTINUE

IF SO, THEN EXIT AND RETURN TO COMPETITION
NEXT STM. IS A 1-LINE IF-THEN-ELSE

4 IF( TFREE ) THEN TAILING = .FALSE.
GO TO 10

IDLE LOOP WAITING TURN

5 30 IF( TAILING ) GO TO 20

IT IS NOW THIS CPU'S TURN TO GAIN ACCESS TO THE TAIL
GO TO 50

-------- PRIORITY TAILING LOOP ENDS --------

HAVE ENTERED GATE TO THE TAIL, NOW CLOSE IT BEHIND US.

40 TFREE = .FALSE.

IDT = IP

DELAY 1 STATEMENT

CHECK TO SEE IF THIS IS THE LAST CPU TO SET IDT
IF NOT, RETURN TO COMPETITION FOR TAIL ACCESS

50 IF( IDT .NE. IP ) GO TO 20

HAVE SOLE ACCESS TO THE TAIL OF THE QUEUE

-------- START CRITICAL PART -------

55 IF( RETURN(IP) .EQ. 0 ) THEN

NO INTERVALS RETURNED

IF RETURN(IP) .EQ. 1 ) THEN

PICK UP INFER TO PUT NEW INTERVAL IN OLD PLACE

NEXT(IP) = INFER(IP)

NEXT(IP) = LIMIT

IF RETURN(IP) .EQ. 2 ) THEN

PICK UP IDT TO PUT 1 NEW INTERVAL IN OLD PLACE AND EXTEND

RETURN


```
1. QUEUE AREA BY 1 FOR THE OTHER NEW INTERVAL
   LASTQ = LASTQ + 1, INQUEUE(LASTQ) = .FALSE.
2. INDEX = INQUEUE(IP) = INQUEUE(INQUEUE(IP)) = LASTQ
3. NO = LASTQ, INEXT(NO) = LING

REASSIGN THE QUEUE LEADER IF THE QUEUE WAS EMPTY
   IF ( IRETURN(IP) .EQ. 0 .AND. LEADER .EQ. LING )
      LEADER = IASSIGN(IP)

UPDATE THE AREA AND BOUND ESTIMATES
   AREA = AREA - CHANGE(IP)
   BOUND = BOUND - CHANGE(IP)

READY TO RELINQUISH ACCESS TO THE TAIL
   IF ( TAILING ) THEN TAILING = .FALSE.
   ELSE TAILING = .TRUE.

INCREMENT THE INDEX FOR THE NEXT CPU IN ORDER
   NEXTT = MOD(NEXTT, NCPU) + 1

----------------- 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:

<table>
<thead>
<tr>
<th>Program</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN-CPU1</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>MAIN-CPU2</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>MAIN-CPUR</td>
<td></td>
</tr>
<tr>
<td>AREAS</td>
<td>4, 5</td>
</tr>
<tr>
<td>NEIGH</td>
<td>5, 6</td>
</tr>
<tr>
<td>INSERT</td>
<td>2, 3</td>
</tr>
<tr>
<td>QGET</td>
<td>5</td>
</tr>
<tr>
<td>QPUT</td>
<td>4, 7</td>
</tr>
</tbody>
</table>

**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. \( NQ = \text{LASTQ} \)). 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. \( K = \text{NEXTQ} \) or \( K = \text{NEXTT} \) and further which has entered the priority waiting loop of QPUT or QGET. The shortest time lapse for this interval's processor to change \( \text{NEXTQ} \) or \( \text{NEXTT} \) 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 \( \text{NEXTQ} \) (or \( \text{NEXTT} \)) 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.

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU-INSIDE</th>
<th>CPU-IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IF(WAITING)</td>
<td>IF(WAITING)</td>
</tr>
<tr>
<td>1</td>
<td>WAITING = .FALSE.</td>
<td>GO TO 30</td>
</tr>
<tr>
<td>2</td>
<td>NEXTQ =</td>
<td>IF(WAITING)</td>
</tr>
<tr>
<td>3,</td>
<td></td>
<td>IF(LEADER.EQ...)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>GO TO 50</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>IF(.NOT. INQUEUE...)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>INQUEUE(LEADER) = .FALSE.</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>IASSIGN(IP) = LEADER</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>LEADER = INEXT(LEADER)</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>CONTINUE</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>IF(WAITING)</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>WAITING = or QFREE =</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>NEXTQ =</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU-INSIDE</th>
<th>CPU-IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IF(WAITING)</td>
<td>IF(WAITING)</td>
</tr>
<tr>
<td>1</td>
<td>WAITING = .FALSE.</td>
<td>GO TO 30</td>
</tr>
<tr>
<td>2</td>
<td>NEXTQ =</td>
<td>IF(WAITING)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>IF(LEADER.EQ...)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>GO TO 50</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>IF(.NOT.INQUEUE)</td>
</tr>
</tbody>
</table>

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 IF) 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.

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU-INSIDE</th>
<th>CPU-IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IF(WAITING)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>QFREE = .TRUE.</td>
<td>IF(LEADER...)</td>
</tr>
<tr>
<td>2</td>
<td>NEXTQ =</td>
<td>IF(.NOT.QFREE)...</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>QFREE = .FALSE.</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>IDQ = IF</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>CONTINUE</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>IF(IDQ...)</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>IF(.NOT.INQUEUE</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>INQUEUE(LEADER) =</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>IASSIGN(IP) =</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>LEADER =</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>IF(WAITING)</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>QFREE = .TRUE.</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>NEXTQ =</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU-INSIDE</th>
<th>CPU-IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IF(WAITING)</td>
<td>IF(IP.NE.NEXTQ)</td>
</tr>
<tr>
<td>1</td>
<td>QFREE = .TRUE.</td>
<td>GO TO 10</td>
</tr>
<tr>
<td>2</td>
<td>NEXTQ =</td>
<td>IF(LEADER</td>
</tr>
</tbody>
</table>
Table 4.4 (Continued)

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU-INSIDE</th>
<th>CPU-IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>IF(.NOT.QFREE)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>QFREE = .FALSE.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>IDQ = IP</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>CONTINUE</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>IF(IDQ.NE.IP)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>IF(.NOT.INQUEUE)</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU-INSIDE</th>
<th>Time</th>
<th>CPU-IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CONTINUE</td>
<td>t</td>
<td>IF(IP.NE.NEXTQ)</td>
</tr>
<tr>
<td>1</td>
<td>IF(WAITING)</td>
<td>t+1</td>
<td>WAITING = .TRUE.</td>
</tr>
<tr>
<td>2</td>
<td>WAITING = .FALSE. or QFREE = .TRUE.</td>
<td>t+2</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>3</td>
<td>NEXTQ =</td>
<td>t+3</td>
<td>IF(QFREE)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>t+4</td>
<td>IF(WAITING) or WAITING = .FALSE.</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>t+5</td>
<td>IF(LEADER...)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>t+6</td>
<td>GO TO 50</td>
</tr>
</tbody>
</table>

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 > 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.

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

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 NCPUs processors so unless the computation terminates successfully, we have that within a fixed time the test of \( \text{LEADER} = \text{LIMQ} \) is made and a new LEADER is assigned. As soon as INSERT terminates the queue leader is unblocked, \( \text{INQUEUE(LEADER)} \) 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 \( \text{PAFAQ} \). Every delay made in order to avoid conflicts from parallel execution is bounded in length by some constant times NCPUs.

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) \( x_0 \in S \) 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 \( \text{INFLECT} \) not zero and every interval with \( \text{INFLECT} \) zero has no inflection point.

**Proof.** First consider BEGINQ where the interval \([A,B]\) is partitioned in subintervals of length \( \text{CHAP} /5 \) and the broken line interpolant to \( f(x) \) is found. Specifically, the cotangent \( \text{COTAN}(K) \) of the \( K \)-th line segment is computed and then the monotonicity of the sequence \( \text{COTAN}(K) \) is checked. It is easily seen geometrically that any set of three intervals where monotonicity is absent contains an inflection point. The assumption that the partition is in intervals of \( \text{CHAP} /5 \) 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 \( \text{INFLECT} = \text{CENTER} \) and \( \text{INFLECT} = \text{LEFT} \) or \( \text{RIGHT} \) 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 \( \text{INFLECT} \) 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 \( \text{INFLECT} \) 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 \( \text{INFLECT} \) and then a change of \( \text{CENTER} \) to \( \text{LEFT} \) or \( \text{RIGHT} \)
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. Incidentally, 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:
<table>
<thead>
<tr>
<th>Case</th>
<th>New Value of INFLECT for Left Subinterval</th>
<th>New Value of INFLECT for Right Subinterval</th>
<th>Action Required for Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFLECT=LEFT, #1</td>
<td>0</td>
<td>LEFT</td>
<td>None</td>
</tr>
<tr>
<td>#2</td>
<td>LEFT</td>
<td>CENTER</td>
<td>Change INFLECT to RIGHT for right neighbor</td>
</tr>
<tr>
<td>INFLECT=CENTER, #1</td>
<td>CENTER</td>
<td>RIGHT</td>
<td>Change INFLECT to 0 for right neighbor</td>
</tr>
<tr>
<td>#2</td>
<td>LEFT</td>
<td>CENTER</td>
<td>Change INFLECT to 0 for left neighbor</td>
</tr>
</tbody>
</table>

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 \( x = XLEFT(I) \)

\[
d = XRIGHT(I) - XLEFT(I)
\]
COTR(IAI) = cot \gamma_R
COTL(IAI) = cot \gamma_L
COTANR(IP) = cot \beta_R
COTANL(IP) = cot \beta_L

COTRGT(IP) = cot \alpha_R
COTMID(IP) = cot \alpha_m
COTLFT(IP) = cot \alpha_L
COTAN(IAI) = cot \theta

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.
INFLECTION POINT: \( \gamma_{LL} < \gamma_L \) and \( \gamma_R > \gamma_{RR} \)

or \( \gamma_{LL} > \gamma_L \) and \( \gamma_R < \gamma_{RR} \)

Figure 5.2 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

\[
\text{BOUND}(I) \leq 2|f''(x)|d^3
\]

**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 \( \frac{bh}{2} \) where  

\[
b^2 = d^2 + (\text{FRIGHT}(I) - \text{FLEFT}(I))^2 \\
h = b/(\cot \alpha + \cot \beta)
\]

\( \alpha, \beta \) = 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 \( \xi_L \) (and \( \xi_R \)) 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 \( \eta_L \) (and \( \eta_R \)) are mean values between \( \xi_L \) (and \( \xi_R \)) and the point \( \xi_I \)  
where \( f'(x) = 0 \). Set \( d^* = \min(d_R, d_L) \) and note that \( d^* \leq 2d \) 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 \( 0 < \theta < 1 \)

\[
h \leq 2d^2/(\theta/f''(\eta_L) + 1/f''(\eta_R)) \leq 2d^2|f''(\eta_L)| \\
\leq 4d^2|f''(x)|
\]

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

\[
\text{BOUND}(I) \leq K(x)d^2
\]

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 \( \theta \) 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 \( 2\theta \). As in the proof of Lemma 5.3 we may assume that

\[
\text{FRIGHT}(I) - \text{PLEFT}(I) = 0
\]

and with the formulas used there we find that

\[
\begin{align*}
    b &= d \\
    h &= \frac{d}{\cot \alpha + \cot \beta} \\
    \cot \alpha, \cot \beta &> \cot 2\theta
\end{align*}
\]

so that the area of the triangle is bounded by

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

which establishes the conclusion.

LEMMA 5.5. Let \( f(x) \) satisfy Assumption 1. Assume that \( \text{INFLECT}(I) \neq 0 \) 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

\[
\text{BOUND}(I) \leq K(x)d^{\alpha + 1}
\]

where \( K \) is independent of \( d \).
Note that this lemma gives an unduly pessimistic value for \( \text{BOUND}(I) \) 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 \( d = \text{XRIGHT}(I) - \text{XLEFT}(I) \) 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 \( Kd^\alpha \) and consequently we have \( \text{BOUND}(I) \leq Kd^\alpha \) 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 \( d \tan \beta \) where \( \tan \beta \) is the slope of the secant line for the next to the end interval. Thus \( \tan \beta = f'(\xi) \) for some mean value point \( \xi \) and for \( d \) sufficiently small \( \tan \beta \) is bounded independently of \( d \). Consequently in this case we have \( \text{BOUND}(I) \leq Kd^{\alpha + 1} \) for some constant \( K \).

Now consider one of the three intervals near an inflection point with \( \text{INFLECT}(I) \neq 0 \). 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 \( \text{BOUND}(I) \) 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 $Kd^\alpha$ and consequently we have $\text{BOUND}(I) \leq Kd^{\alpha+1}$ and
then 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 compari-
sions of $\text{BOUND}(I)$ with DISCARD * DX rather than merely DISCARD makes
these two relations equivalent to $\text{ERROR}(x,k) \leq k|f''(x)|d^2$, $\text{ERROR}(x,k) \leq kd^\alpha$
as given in [4].

**ASSUMPTION 2.** Consider the $I$-th interval of length $d$. There are
constants $K$ and $\alpha$ (the same as in Assumption 1) so that when $d < \text{CHARF5}$
we have

(i) If the $I$-th interval contains no singularities then

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

(ii) If the $I$-th interval contains a singularity then

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

The objective is, of course, to show that if $f(x)$ satisfies Assumption 1 then
the computed values of $\text{BOUND}$ 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 inflec-
tion points. Third, a larger constant may be required than given in Assump-
tion 1. Thus we introduce the following

**TERMINOLOGY:** We say that the $I$-th interval contains a singularity if
it (i) is an end interval, (ii) has $\text{INFLECT}(I) \neq 0$ or (iii) contains an
actual singularity of $f(x)$. We take $\text{CHARF5}$ to be the smallest value re-
quired in the above proofs, namely one-fifth of the minimum of
(i) separation between singular points and/or inflection points

\( (= \text{CHARF}) \)

(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), \( \tan \frac{2\theta}{4} \) 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 \( \text{CHARF} \) 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 $\text{AREA}$ requiring $N$ evaluations of $f(x)$ so that

$$|\text{If - AREA}| \leq \text{EPS}$$

with

$$N = \Theta(\text{EPS}^{-\frac{1}{2}})$$
or, equivalently,
\[ |\text{If - AREA}| \leq \Theta\left(\frac{1}{N^2}\right) \]

If \( N > \text{NCPU}^2 \) then the total computation time \( T_N^f \) satisfies, for constants \( K_1, C_0 \) and \( C_1 \) as defined in [4],
\[ T_N^f \leq K_1 \frac{N^4(4C_0 + 2C_1\text{NCPU})}{\text{NCPU}} \]

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 not speeded up by the parallelism of the algorithm. The constant \( C_0 \) is seen to be substantially larger, about 100. For large values of \( \text{NCPU} \) 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 \((\log \text{NCPU})/\text{NCPU}\). 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 TRIANGLE may be called TRIANGL at some point and provisions might not be made for an input of EPS = -0.001 (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


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

1. \( \int_0^1 (x-.5)^3 + 5 \, dx \quad \varepsilon = 10^{-3} \)
2. \( \int_0^1 \ln x \, dx \quad \varepsilon = 10^{-3} \)
3. \( \int_0^1 \sqrt{x} \, dx \quad \varepsilon = 10^{-3} \)
4. \( \int_0^1 \sqrt{x} \, dx \quad \varepsilon = 10^{-4} \)

NUMBER OF PROCESSORS