A Parallel Processing Approach to Transition Prediction for Laminar Flow Control System Design

R. W. FORD[1] AND D. I. A. POLL[2]

[1]*Centre for Novel Computing, Department of Computer Science, The University of Manchester, Oxford Rd., Manchester, M13 9PL, U.K.; e-mail: rupert@cs.man.ac.uk*

[2]*Department of Engineering, The University of Manchester, Oxford Rd., Manchester, M13 9PL, U.K.; e-mail: diapoll@fs1.eng.man.ac.uk*

ABSTRACT

The performance of transport aircraft can be considerably improved if the process by which the wing boundary layer becomes turbulent can be controlled and extensive areas of laminar flow maintained. In order to design laminar flow control systems, it is necessary to be able to predict the movement of the transition location in response to changes in control variables, e.g., surface suction. At present, the technique which is available to industry requires excessively long computational time—so long that it is not suitable for use in the "design process." Therefore, there is a clear need to produce a system which delivers results in near realtime, i.e., in seconds rather than hours. This article details how parallel computing techniques on a KSR-1 produce these performance improvements. © 1995 by John Wiley & Sons, Inc.

1 INTRODUCTION

In order to sustain an aircraft in straight and level flight, two fundamental conditions must be satisfied. The first is that the lifting force generated by the wings must be equal to the weight and the second is that the thrust from the engines must be equal to the drag.

The drag force opposes the motion of the aircraft and acts in the direction of flight. It is made up of two components, these being the pressure drag and the viscous drag. Pressure drag is produced by the variation of the air pressure acting on the aircraft surface and is closely related to the lift force—in fact, it is the penalty that must be paid to produce the lift. Viscous drag, on the other hand, is the result of tangential, or shearing forces, because, as a result of viscosity, air sticks to the surface.

On a typical transport aircraft in the cruise condition, the two components of the drag are approximately equal i.e., 50% of the drag is due to the action of viscosity. This means that a substantial fraction of the fuel which an aircraft carries is there to overcome viscous drag. It follows that, if the viscous drag can be reduced, a substantial saving in fuel and, consequently, operating cost is possible.

Received May 1994
Revised December 1994

In flight, the effect of air viscosity is confined to a very thin layer close to the aircraft—the so-called boundary layer. On the surface viscous forces require that the flow speed must be zero. However, a few millimeters above the surface the air must be traveling at almost the flight speed of the aircraft.

The motion of air within this thin boundary layer takes one of two possible forms. It may set up a simple, steady flow where each layer of air slips slowly over its neighbor—this is called the laminar condition. The alternative is an unsteady flow with almost random fluctuations causing the various layers of air to be mixed violently with one another—this is the turbulent condition.

Not surprisingly, a turbulent flow produces more viscous drag than the laminar one—the ratio is approximately 10 : 1. At very low speeds, flows are always laminar. However, as speed is increased, there comes a stage at which laminar flow becomes unstable to the small disturbances which are always present in reality. At higher speeds, these disturbances are amplified and, when the amplitudes are sufficiently large, there is a breakdown and turbulent flow is produced.

It follows that, for a given aircraft configuration, there is a flight speed above which the boundary layer flow will be turbulent and the viscous drag will be large. Unfortunately, for all but the smallest aircraft, the boundary layers are turbulent in the cruise condition.

Until recently this situation was accepted as unavoidable and aircraft designs have been optimized on the assumption that boundary layer flow would always be turbulent. However, it has been known for over 60 years that, if some of the air in the boundary layer could be sucked through the surface, then the speed at which laminar flow becomes unstable can be increased.

The effect of suction is sufficiently powerful for laminar flows to be achieved at typical aircraft cruise conditions when the suction velocity is only 0.05% of the flight speed (i.e., only about 10 cm/s!). If suction could be engineered, the drag of an aircraft could be reduced by as much as 30%. This would represent a quantum leap in aircraft performance since, in the current commercial climate, an aircraft which could deliver a 5% drag improvement relative to its competitors would capture the entire market.

There are two major obstacles to the development of an aircraft which uses the surface suction technique for laminar flow control (LFC). The first is the provision of a suitable porous surface through which the air can be drawn. This has always been a serious problem since, in the past, surfaces which were porous did not have good load-bearing properties. However, this difficulty is now effectively resolved because of the recent development of the laser perforating technique. This enables traditional aerospace materials—titanium, aluminium, steel, and even composite material—to be drilled with millions of holes as small as 50 μm in diameter, placed in any desired pattern, with any desired spacing. This leaves the second problem which is that, in order to produce a design for an LFC system, it is necessary to be able to estimate the conditions under which the boundary layer flow will undergo a "transition" from the laminar to turbulent state. Moreover, it is necessary to be able to produce these estimates sufficiently quickly so that a conventional design process is not slowed down.

The problem of predicting the conditions necessary for the onset of transition is a particularly challenging one. In fact, at present, there is no complete theory for transition. Nevertheless, various semi-empirical methods have been developed over the years and some are appropriate for use in design.

Of these, the one which is most accurate and allows for all the important parameters i.e., flow compressibility (mach number), surface temperature (heat transfer), and wall transpiration (suction) is the e^N method. This is based on an approximate formulation of the stability problem for a boundary layer which, when solved by a suitable numerical technique, produces dispersion relations for the unstable disturbances. These relations are used to track the disturbance amplitude development and an empirical criterion is used to determine the breakdown (transition onset) condition.

However, while the e^N technique allows for all the physical effects which can influence transition, it requires a great deal of computation—so much so that the elapse time between predictions is far too long for it to be described as a design tool. One possible solution to this problem is the application of parallel computing techniques.

This article chronicles the parallelization of a laminar to turbulent transition prediction code, developed by the LFC group, in the Department of Engineering, at the University of Manchester.

Section 2 introduces modeling techniques for the onset of turbulence and Section 3 describes the parallelism inherent in the solution method. Section 4 introduces the KSR-1 and discusses both the scalar optimizations and parallelization methods used. Finally, Section 5 summarizes the main conclusions of the article and Section 6 discusses future work.

2 ALGORITHM DESCRIPTION

2.1 Solution Methods

The general problem of predicting the onset of transition in a flow is extremely complex. Strictly speaking, the complete approach requires the full unsteady Navier–Stokes equations to be solved for a range of disturbances, which span the complete spectrum of freestream fluctuations, surface roughness, surface vibration, and sound. Such a calculation would have to be performed with very fine resolution of length and time scales and it would be necessary to specify every possible form of disturbance in order to ascertain which were amplified most rapidly. Moreover, the computations would have to be carried out sufficiently far downstream to capture the nonlinear processes which lead to laminar flow breakdown and the ultimate establishment of turbulent flow. Even with the most powerful machines currently available, such calculations are only possible for simple flows under highly restrictive and ultimately unrealistic conditions, e.g., fully developed pipe flow with temporally developing disturbances. For engineering purposes, when the basic flows are much more complex, an alternative approximate approach is called for.

A major simplification of the problem is produced by limiting the consideration to the development of small amplitude disturbances in a boundary layer flow, since this allows linearization of the governing equation. This approach was first proposed in the 1920s by Prandtl's group in Göttingen[1]. The complete analysis is available in many standard texts, e.g., Mack[2] but, in essence, the arguments run as follows.

1. The instantaneous fluid properties are expressed in terms of a mean component plus a fluctuating component, e.g., $U = \bar{u} + u'$, $P = \bar{p} + p'$, etc.
2. It is assumed that the complete unsteady flow satisfies the Navier–Stokes equations.
3. The amplitudes of the disturbed quantities are assumed to be sufficiently small for products of fluctuating components to be negligible.
4. The mean flow is assumed to satisfy the steady, boundary layer equations.
5. The normal-to-surface component of the velocity is assumed to be negligibly small compared with the streamwise component. Thus the flow is taken to be parallel.

Having taken the above steps the resulting equations for the disturbance components are found (by inspection) to have harmonic solutions. Since the problem has been linearized, a general disturbance can be constructed by superimposing normal modes of the form

$$u'(x, y, z, t) = F(y)e^{i(\alpha x + \beta z - \omega t)} \tag{1}$$

where, in general, α, β, and ω are complex quantities. By substituting expressions for the disturbance quantities of the form of Equation 1 into the governing equation, a system of equations is obtained which can be used to determine the characteristics of traveling waves propagating through the flow. These waves are known as Tollmien–Schlichting (T–S) waves. It is interesting to note that, originally, the above stability analysis was carried out in the absence of any experimental evidence that such waves could exist. Verification of the existence of T–S waves and their precursor role in the process of boundary layer transition was not provided until the 1940s by Schubauer and Skramstadt[3].

Finally, in order to produce further simplification, the flow may be assumed to be incompressible; it is then possible to show that, for a two-dimensional boundary layer, the most unstable wave propagate in the mean flow direction, i.e., $\beta = 0$[4]. Consequently, from the point of view of transition prediction, only two-dimensional disturbances need to be considered. This being the case the stability problem reduces to the solution of a single, fourth order ordinary differential equation:

$$\left(\frac{d^2}{dy^2} - \alpha^2\right)^2 \hat{v} = iR\left[(\alpha U - \omega)\left(\frac{d^2}{dy^2} - \alpha^2\right) - \alpha\frac{d^2U}{dy^2}\right]\hat{v} \tag{2}$$

subject to the boundary conditions:

$$\hat{v}(0) = 0,\ d\hat{v}(0)/dy = 0$$
$$\hat{v}(y) \to 0,\ d\hat{v}(y)/dy \to 0 \text{ as } y \to \infty \tag{3}$$

This is known as the Orr–Sommerfeld (O–S) equation. From the point of view of the present exercise it is important to note that this equation depends on local conditions only, i.e., there are no terms involving derivatives with respect to x. This means that the stability characteristics are not affected by the upstream history. For a wave of fixed frequency, ω_r, convecting through a given flow at a specified local Reynolds number, R, the O–S equation provides two relationships (real and imaginary part) among the three unknown quantities α_r, α_i, and ω_i. Therefore, in order to close the problem, an extra condition is required. In the early days of stability computation, the final step was to assume that the disturbance grew in time but not in space, i.e., α_i was zero. This produced a well-posed mathematical problem. However, in the physical world in which waves are observed to propagate in the mean flow direction a slightly more realistic approach is to assume that the disturbances grow in space but not in time, i.e., ω_i is zero. This spatial form for the T–S wave is the preferred option for use in transition prediction.

Solutions to the O–S equation permit the calculation of the dispersion relation for the disturbance waves of a specified frequency. The secondary problem then is how to use this information to predict the onset of transition. It has already been noted that breakdown of laminar flow occurs when the amplitude of amplified traveling disturbances becomes large. From Equation 1, it is immediately apparent that for a wave of fixed frequency, if, at the point of neutral stability, x_0, the disturbance amplitude is A_0 then at station x, where $x > x_0$:

$$A/A_0 = \exp\left(\int_{x_0}^{x} (-\alpha_i)dx\right) \tag{4}$$

In general, the boundary layer will change its thickness and Reynolds number between any two stations and, consequently, α_i will vary with x. However for the purposes of evaluating α_i, it is assumed that locally the flow does not vary with x. Hence, the amplitude ratio relation is only approximate. Nevertheless, it is a quantity which can be readily calculated and it does bear some relationship to the stability of the mean flow.

By examining a range of experiments in which transition was observed, it has been proposed that transition onset correlates with the condition where the wave which has undergone the greatest total amplification has just reached an amplitude ratio of e^9, i.e.,

$$\ln\left(\frac{A}{A_0}\right) = N = 9 \tag{5}$$

This is the basis of the so-called e^N transition method when the critical (transition onset) value for N in low disturbance environments is 9.

2.2 Numerical Method

The boundary layer (see Fig. 1) will, in general, vary in thickness along the aerofoil. Therefore, we solve the O–S equation at a number of equally spaced positions along the aerofoil, finding the wave amplification rates at each: these positions are denoted "stations." A particular wave's amplification rate corresponds to the imaginary part of its wavenumber α_i. At each station the O–S equation is solved (finding α_i) for a number of equally spaced frequencies: the number and bounds of these are specified.

An initial wavenumber approximation α must be supplied by the user for the lowest frequency at the first station. When the actual wavenumber corresponding to that frequency has been calculated, the frequency is perturbed and a new wavenumber found for the perturbed frequency. The original and perturbed values are then used to make an approximation to the wavenumber solution, for the next frequency, at the same station (Fig. 2a shows the dependencies, at the first station, denoted by vertical arrows).

When the instabilities for each frequency at a station have been calculated the results are stored. The wavenumber corresponding to the lowest frequency (f_1 in Fig. 2a) is then used as an approximation to the wavenumber for the lowest frequency at the next station (Fig. 2b shows the dependencies here, denoted by horizontal arrows): this method is possible as the boundary layer and velocity profile are slowly varying (therefore adjacent solutions have similar values). Thus,

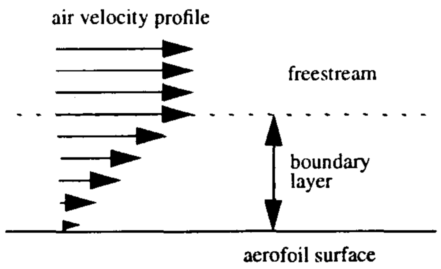


FIGURE 1 Typical boundary layer velocity profile.

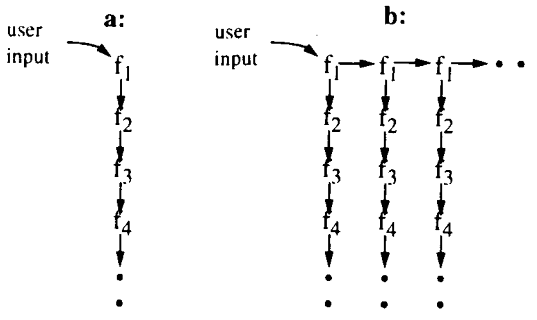


FIGURE 2 Dependencies of frequencies and stations.

once the initial wavenumber approximation has been supplied, subsequent approximations are generated automatically.

To solve the OS equation, finding a wavenumber solution from an initial approximation at each frequency and station, we use a shooting method[5]. The shooting method solves boundary value problems (to find the solution for an ordinary differential equation between two points with known boundary conditions). In this case the freestream gives one boundary condition and the aerofoil the other (see Equation 3). An approximation is supplied at one boundary and the system of ordinary differential equations integrated to the other boundary (from the freestream to the aerofoil). This is repeated with another approximation. In our algorithm the two approximations, Z^1 and Z^3, are integrated at the same time. The integrator used is a fourth order Runge–Kutta integrator.

The shooting method we use suffers from the problem of parasitic error growth. For this case (a two-dimensional wave in a two-dimensional boundary layer) the two solutions Z^1 and Z^3 each consists of four components. Z^3 grows more rapidly with decreasing y than Z^1. The parasitic error follows Z^3 and when the difference in magnitude of Z^3 and Z^1 becomes sufficiently large $Z^{(1)}$ is no longer independent of Z^3. Before this occurs we apply Gram-Schmidt orthonormalization: in fact we do this for each iteration. The large solution $Z^{(3)}$ is normalized component by component to give the new solution:

$$Z^{(3)}_{new} = Z^{(3)}/\{\overline{Z}^{(3)}Z^{(3)}\}^{1/2} \tag{6}$$

where the overbar refers to a complex conjugate and $\{\}$ a scalar product. $Z^{(1)}$ is then replaced by:

$$Z^{(1)}_{new} = (Z^{(1)} - \{\overline{Z}^{(3)}_{new}Z^{(1)}\}\, Z^{(3)}_{new})/(\underline{Z}^{(1)}\underline{Z}^{(1)})^{1/2} \tag{7}$$

where the underbar refers to the quantity in the numerator. The numerical integration proceeds with the new values of Z^1 and $Z^{(3)}$.

A linear combination of $Z^{(1)}$ and Z^3 can be found which satisfies the boundary condition $\hat{u}(0) = 0$ at the aerofoil but will not satisfy the condition $\hat{v}(0) = 0$ unless the wavenumber approximation is an eigenvalue of the equation (the correct value). The residual $\hat{v}(0)$ can therefore be found.

The real part of the wavenumber approximation, α_r is perturbed by a small amount $\Delta\alpha_r$ and the integration repeated. The imaginary part of the wavenumber approximation α_i is then perturbed by a small amount $\Delta\alpha_i$ and the integration repeated. Corrections $\delta\alpha_r$ and $\delta\alpha_i$ to the initial approximations α_i and α_r are obtained from the residual and numerical approximations to derivatives using Equations 8 and 9:

$$\left[\frac{\partial}{\partial\alpha_r}\hat{v}_r(0)\right]\delta\alpha_r - \left[\frac{\partial}{\partial\alpha_i}\hat{v}_r(0)\right]\delta\alpha_i = -\hat{v}_r(0) \tag{8}$$

$$\left[\frac{\partial}{\partial\alpha_r}\hat{v}_i(0)\right]\delta\alpha_r - \left[\frac{\partial}{\partial\alpha_i}\hat{v}_i(0)\right]\delta\alpha_i = -\hat{v}_i(0) \tag{9}$$

The corrected α_i and α_r are used to start a new iteration and the process continues until $\delta\alpha_r$ and $\delta\alpha_i$ are reduced below a preset criterion. This method is a quasi Newton–Raphson search.

2.3 Algorithm in Context

So far we have not discussed how initial data, such as wavenumber and velocity profiles, for the aerofoil are calculated. This section overviews this process.

We begin with an aerofoil and its corresponding pressure distribution. A simplified diagram of an aerofoil is given in Figure 3. This shape has a surface static pressure distribution which is of the form given by Figure 4.

Once a particular aerofoil shape is chosen the pressure distribution, corresponding to that shape, is used as input to a mean flow code. This

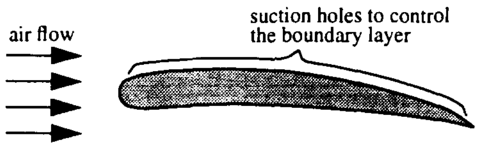


FIGURE 3 Aerofoil.

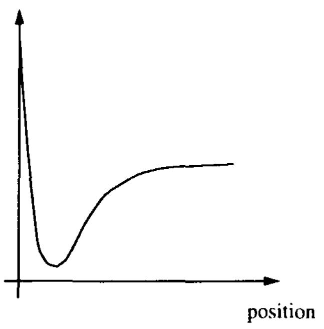


FIGURE 4 Pressure distribution along aerofoil.

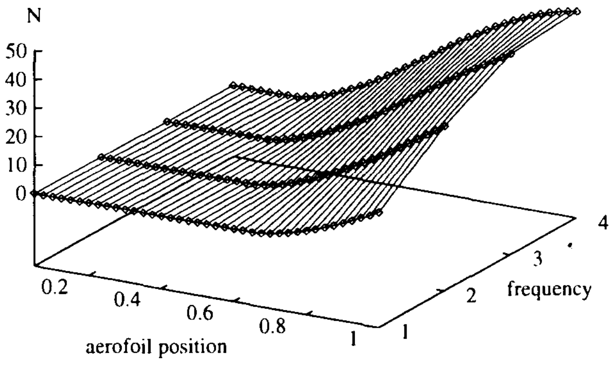


FIGURE 5 Amplitude ratios.

problem calculates the various boundary layer parameters needed at each station, such as Reynolds number, boundary layer thickness, and velocity profile; its output is used as input to the stability code. These two programs are separated so that certain control parameters can be set and others checked before running the time consuming stability program.

The control parameters we set are the initial wavenumber approximation for the first frequency at the first station, the frequency range to be examined, and the number of frequencies within this range.

It is known from previous experience that in practice two-dimensional instability waves tend to occur within the region 500–5,000 Hz. Therefore, the first time the instability program is run, a spread of frequencies across this range is examined for a number of stations along the aerofoil, typically 50–100 stations and 5–10 frequencies. The first run is effectively exploratory, to find which frequencies and stations to concentrate on. We then rerun the program with a greater number of frequencies and stations over the range of interest, typically 100–400 stations and 10–40 frequencies. This process may be repeated two or three times before a sufficiently accurate picture is obtained.

The output consists of two files, a large diagnostic file which records virtually all relevant variables, and a file which gives the amplification rate (α_i) for each frequency at each station.

α_i is the amplification rate at a station (for a particular frequency). To convert this into an amplitude ratio, α_i needs to be integrated along the aerofoil. The natural log of the amplitude ratio is then plotted against position along the aerofoil for each frequency; an example is shown in Figure 5.

Figure 5 shows a number of amplitude ratio plots for various frequencies. It is the profile of these frequencies, i.e., the largest amplitude ratio at any given position along the aerofoil, that is important when predicting where turbulence starts. This is because turbulence only begins above a certain (experimentally determined) value of N, regardless of which frequency first reaches it.

2.4 Available Parallelism

This section discusses the parallelism apparent from the numerical method described in Section 2.2.

Pipelined Station Parallelism

When examining Figure 2b it is clear that computation for frequencies at a station can begin once the wavenumber for the first frequency (f_1) of the previous station has been calculated: this gives a parallel pipeline effect demonstrated in Figure 6.

In Figure 6 the maximum overlap (number of stages in the pipeline) is the number of frequen-

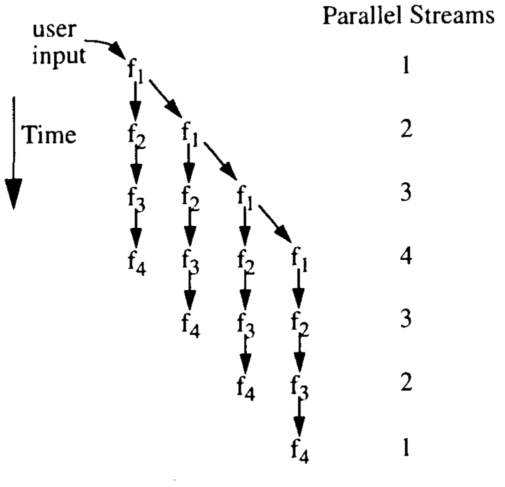


FIGURE 6 Overlapped solutions.

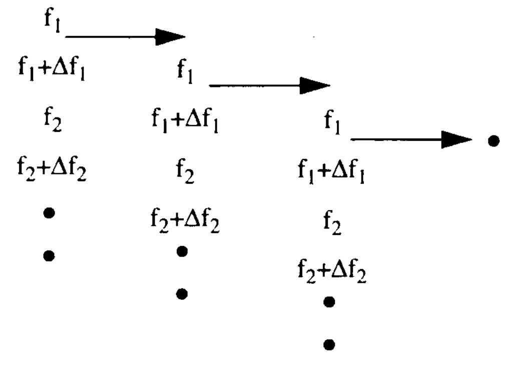


FIGURE 7 Maximum station overlap.

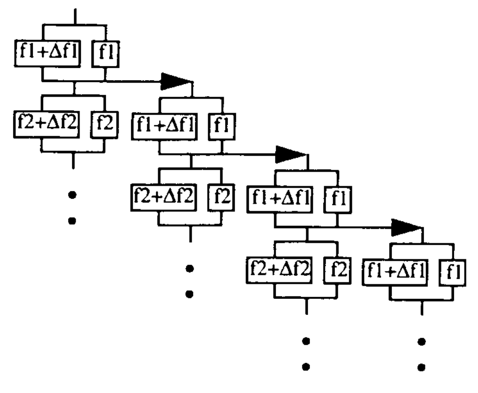


FIGURE 8 Station and frequency parallelism.

cies per station. In fact the overlap is greater than this as a station does not rely on the perturbed frequencies wavenumber solution for the previous station; it can proceed as soon as the actual wavenumber for the frequency at the previous station has been calculated. This effect is shown in Figure 7.

Thus the potential number of stages in the pipeline (assuming all wavenumber frequency pairs take the same time) is two times the number of frequencies per station. As will be seen in the next section the pipeline is effectively reduced to that shown in Figure 6 when the frequency and perturbed frequency are parallelized.

Frequency and Perturbed Frequency

As described in Section 2.2, the wavenumber is solved for each frequency and for that frequency perturbed. The perturbation allows a wavenumber approximation to be extrapolated for the next frequency at that station.

At first glance one would expect these two solution to be independent; however to decrease the number of iterations needed to converge for the perturbed frequency, our sequential algorithm uses the wavenumber solution of the frequency itself; this imposes a dependency.

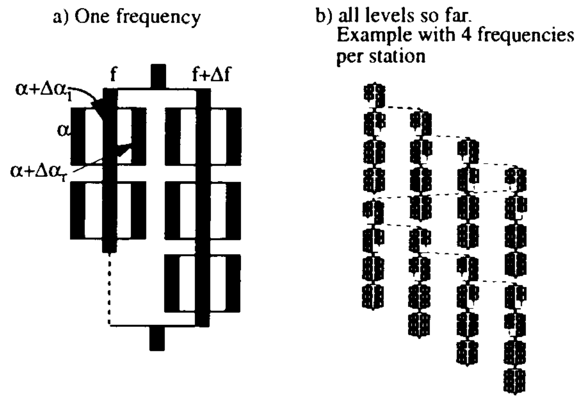
To enable these two solutions to proceed in parallel the same wavenumber approximation used for the frequency can be used for the perturbed frequency. This change in the algorithm may increase the amount of computation as the solution to the perturbed frequency could take longer to converge.

Another possible problem is that for a particularly bad wavenumber approximation the perturbed frequency may not converge—therefore the parallel algorithm is potentially less stable than the sequential algorithm. This problem can be eliminated as, on failure, the parallel algorithm can revert to its sequential form. For all test cases examined so far this has not been required. The parallelism so far described is shown in Figure 8.

Wavenumber Approximations

For each frequency, the wavenumber approximation and perturbations of its real and imaginary parts (α_r and α_i) are integrated through the boundary layer. As suggested in Section 2.2, these are independent and can therefore be calculated in parallel.

The parallelism so far described is shown in Figure 9. It shows a potential loss of efficiency when parallelizing at the frequency and perturbed frequency level described in the previous section. In the example given one solution converges in two iterations while the other takes three.



a) One frequency
b) all levels so far. Example with 4 frequencies per station
f
f+Δf
α+Δα_i
α
α+Δα_r

FIGURE 9 Nested parallelism.

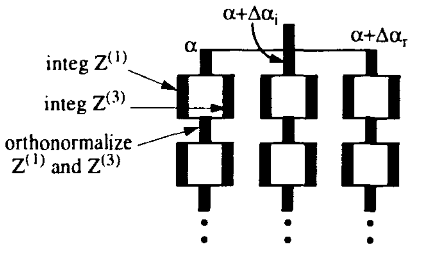


FIGURE 10 Integration of solutions.

Integration of Solutions

Each wavenumber approximation and perturbed values of its real and imaginary parts has two dependent solutions $Z^{(1)}$ and $Z^{(3)}$. These are integrated from the edge of the boundary layer to the aerofoil (see Section 2.2). Each integration is independent, however orthonormalization is applied at each step (see Fig. 10).

3 PROGRAMMING TECHNIQUES

The codes described in this article were written and are maintained by the LFC group in the aeronautics department of The University of Manchester, U.K. The stability code is called Melissa.

Melissa is written in standard Fortran 77 and as a result has run without modification on all platforms tried. The code itself is approximately 1,000 lines long.

The O–S solver utilized in Melissa was taken from an earlier, more general purpose code, written in FortranIV. Due to both the language used and the original authors' coding practice in this earlier code, the O–S solver section of Melissa has a typical "dusty deck" form.

3.1 Code Restructuring

To help understand the algorithm used to solve the O–S equation and make the code more readable certain code restructuring was performed.

Loops implemented with a counter and conditional branch using a GOTO were converted into DO loops. Redundant loops and code segments associated with the equation solver performing obsolete functions were removed. Tangled control flow and conditional GOTOs were converted into their IF THEN ELSE form. A number of redundant input variables and input variables read more than once were removed. Large code fragments were converted to subroutines to aid readability. Finally some COMMON blocks were removed and variables passed as arguments.

3.2 The Kendall Square Research KSR-1

The scalability of shared memory multiprocessors has traditionally been limited to tens of processors due to memory access contention. As a result it has been widely accepted that distributed memory is the key to scalable parallel machines, however these machines have been notoriously difficult to program.

The KSR-1 is a distributed memory machine that provides a single address space, supported by proprietary hardware [6], the advantage being a shared memory programming model for the user. This technique has been termed virtual shared memory (VSM). This term can cause confusion as the KSR-1 also supports virtual memory (VM) with an address space of 1 million Mbytes (2^{40}).

Each KSR-1 processor is a 20 MHz RISC-style superscalar 64-bit unit operating at 20 Mips and 40 Mflop/s (peak). A KSR-1 system contains from 8 to 1,088 processors with a peak performance range from 320 to 43,520 Mflop/s.

Each processor has 0.5 Mbyte of subcache, split equally between instructions and data, and 32 Mbyte of cache. It is therefore a nonuniform memory access (NUMA) style memory system. In this system instructions and data are not bound to specific physical locations, rather they migrate to where they are being referenced; this is termed a cache-only memory architecture (COMA).

The interconnect topology is a two-level hierarchy of slotted unidirectional rings, known as ring0 and ring1. Each ring0 can have a maximum of 32 processor memory pairs and has a bandwidth of 1 Gbyte/s. The ring1 connects up to 34 ring0s and has a bandwidth of 1–4 Gbyte/s depending on configuration. The KSR-1 at Manchester is a 64-processor machine.

A thread (termed pthread by KSR) is a sequential flow of control within a process and is the underlying mechanism used to execute the parallel constructs available to Fortran programmers. These constructs—parallel regions, parallel sections, and tile families—form a high-level interface to pthreads. The user inserts these parallel constructs, seen as comments to other compilers, around appropriate blocks of codes. A pthread library for thread creation, barriers, locks, condition variables, etc., can be accessed directly by the programmer if a finer level of control is required.

3.3 Scalar Optimization on the KSR-1

The core element of Melissa, the O–S equation solver originally included the case of oblique waves (three-dimensional waves); the values of these were set to zero in the input files. The redundant code associated with this was removed.

Two core functions, in which nearly all computation takes place, were each called twice with the same input parameters. These two calls were replaced by a single call and the result shared.

The innermost functions are called millions of times and have little work within them. These were manually inlined reducing the calling overhead.

In combination these scalar optimizations produced over fourfold improvement in solution time (see Section 4.3).

3.4 Parallelization on the KSR-1

Parallel Stations

In "Pipelined Station Parallelism" we describe the potential parallelism available by overlapping station solutions (see Fig. 6). As the overlap is equal to the number of frequencies per station the maximum parallelism that can be usefully employed is equal to the number of frequencies.

If, for example there are 4 frequencies, thread 1 will be used to calculate stations 1, 5, 9, etc., thread 2 will calculate stations 2, 6, 10, etc., and so on (see Fig. 11).

This ensures that all threads are kept as busy as possible and minimizes the number of threads used. As mentioned in "Pipelined Station Parallelism" this is, in fact, an oversimplification. To create the appropriate number of threads on the KSR-1 we use a parallel region directive. A function is called within the parallel region which returns a unique value to each thread and is used in combination with an explicit modulo function to ensure each thread only calculates the appropriate stations.

We now need to delay the thread at the next station until the thread at the current station has calculated the α solution for its first frequency. To implement the above we use a "mutex"; this allows only one thread through a section of code at a time (the first frequency calculation is effectively an ordered critical section) and a condition variable to ensure that the pthreads obtain the mutex in the correct order. These were implemented using calls to appropriate KSR pthread libraries. The code implementing this is shown below.

```
C*KSR* parallel region(numthreads=
C*KSR*&NFREQ,private=(I,mynum,istat))
  mynum=ipr_mid()
  DO I=1,NSTAT
      IF(mod(I,NFREQ).EQ.mynum)THEN
          call pthread_mutex_lock
  &       (mul, istat)
          IF(flag(I).EQ..false.)THEN
               call pthread_ cond_
  &            wait(icond(I), mul, istat)
          ENDIF
          CALL STATIONS(..)
      ENDIF
  END DO
C*KSR* end parallel region
```

The condition flag (flag(I)) is initialized to false for all instances except for I=1. If the "wrong" thread grabs the mutex it yields on a pthread_cond_wait() until it is woken by the thread which has calculated the previous station wavenumber. When a thread has finished calculating the wavenumber required for the next station to start, it unlocks the mutex, sets flag(I+1) to true, and wakes the next thread if it is sleeping on the condition variable; this is shown below:

```
IF (I.NE.NSTAT) flag(I+1)=.true.
call pthread_mutex_unlock(mul, istat)
IF (I.NE.NSTAT) THEN
     call pthread_cond_signal (icond
&    (I+1), istat)
ENDIF
```

A problem found in the use of parallel regions and condition flags is in the KSR-1's Fortran77

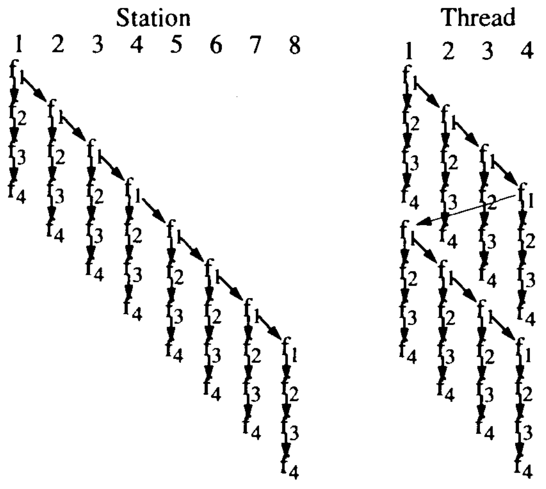


FIGURE 11 Maximum station parallelism.

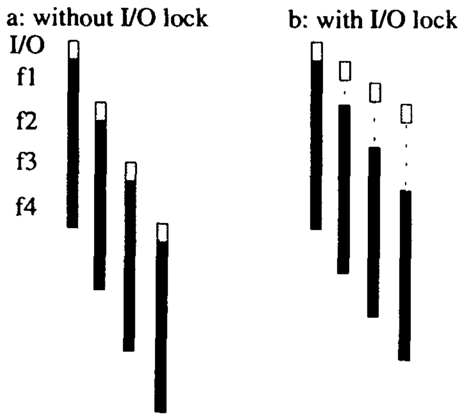


FIGURE 12 Overlapping I/O.

optimization; it assumes the code runs sequentially making appropriate optimizations. It is therefore possible for the compiler to "optimize out" a condition variable as. sequentially. the condition is always true. On the KSR-1 such explicit locks need to be declared `volatile`. This is a Fortran77 extension which stops the compiler from optimizing that variable.

Other necessary modifications to the code involved making wavenumber results for the first frequency global so that the next station could read them and sequentializing the output to a file: this occurs after all the wavenumbers have been calculated at a station. This latter change was implemented with a naive spin lock condition variable.

It ensues that we cannot neglect the time taken for each thread to complete I/O before calculating the wavenumber solutions for its frequencies. This overhead can be reduced by adding another lock and condition variable, allowing the thread for a station to proceed with its I/O as soon as the I/O has been dealt with by the previous station (see Fig. 12).

Parallel Frequencies

As discussed in "Frequency and Perturbed Frequency" the algorithm requires modification to calculate the wavenumber for the frequency and perturbed frequency in parallel. This involves employing the same wavenumber approximation by both the frequency and perturbed frequency. To implement this we need only change one `IF` statement. To run this in parallel we add the KSR tile directive given below:

```
C*KSR* TILE (FREQ, teamid=iteam1)
   DO FREQ=1,2
        CALL PERT(...)
   END DO
C*KSR* END TILE
```

The teamid argument to the tile directive is discussed in "Teams and Nested Parallelism."

Parallel Wavenumber Approximations

As discussed in "Wavenumber Approximations" the wavenumber approximation and perturbations of its real and imaginary parts are integrated through the boundary layer. As in the previous section to implement this we simply need to add the KSR tile directive given below:

```
C*KSR* TILE(PERT, teamid=iteam2)
   DO PERT=1,3
        CALL WAVE(...)
   END DO
C*KSR* END TILE
```

The teamid argument to the tile directive is discussed in "Teams and Nested Parallelism."

Integration of Solutions

As discussed in "Integration of Solutions" each integration step of the two dependent solutions $Z^{(1)}$ and $Z^{(3)}$ can be executed in parallel. however. this is not true for the whole integration due to the orthonormalization of the solutions (see Fig. 10). This was implemented using the KSR parallel sections directive. see below:

```
   DO J=I,NSTEPS
C*KSR* parallel sections
& (teamid=iteam3)
C*KSR* section
         CALL INTEG(AZ,..)
C*KSR* section
         CALL INTEG(CZ,..)
C*KSR* end parallel sections
         CALL MODOR(CZ,..)
         CALL ORTHO(AZ, CZ,..)
   END DO
```

The teamid argument to the parallel sections directive is discussed in "Teams and Nested Parallelism."

Inner Functions

When the code is examined we observe some inner functions which could be called in parallel. For example, functions U and U1 independently search for the appropriate velocity and acceleration data, respectively. However, although a large proportion of the computational time is spent in these routines, the time per call is too small to obtain any benefit on the KSR-1.

Teams and Nested Parallelism

When a parallel construct is encountered, the appropriate number of threads are allocated to the work within it. To achieve this, threads which have already been created and have finished their previously allocated work are utilized: such threads are kept in an "idle pool." If there are not enough threads in the idle pool then threads are created to make up the shortfall.

As threads start referencing instructions and variables, the appropriate data are fetched from remote processors (if there is no instance of those data in the correct state locally). When threads finish their work they return to the idle pool.

If another parallel construct is encountered later, which accesses the same variables, it is sensible to use the same threads and processors that executed previously, as the processors' local memory may still have valid copies of data. If the same construct is encountered, then the instruction cache may also still have valid copies of instruction code, and the thread will have the correct data structures associated with it. The idea of data reuse here is, of course, the same as reuse of cache on a single processor machine.

To ensure the same threads and processors are used for subsequent parallel constructs we use "teams" of threads. A team is created with a defined number of threads. This team can then be referenced in association with a number of parallel constructs and the same threads associated with the team will be utilized each time it is called.

Teams of threads were implemented for each level of parallelism in the code. Another important effect of using teams is observed when the granularity of work is close to that of the start up cost of the construct and the construct is called many times; this is because a team reduces subsequent construct start up overheads by a significant margin.

A subtle problem in the use of teams arises when parallelism is nested. This is due to run-time thread creation. A team groups together a number of threads including the currently active thread, thus sets of calls to create appropriately sized teams at the beginning of the code are not desired. It must be the active thread, arriving at the parallel construct, that creates the team. Therefore the team is created (once) at the same nested level as the parallel construct.

4 PERFORMANCE REALIZATION

4.1 Test Data

As outlined in Section 2.3, we first apply a low resolution search to find the frequencies and aerofoil positions of interest; we then do a more detailed analysis around these areas. The low resolution search typically uses 5–10 frequencies and 50–100 stations. The high resolution analysis typically uses 10–40 frequencies and 100–400 stations.

In this article we use two test cases to represent these different resolutions. Test case A has 4 frequencies and 40 stations. Tese case B has 40 frequencies and 100 stations.

4.2 Results

The results reported in this article were achieved on the KSR-1/64 at Manchester University. Timing runs were taken in a multiuser environment. To ensure the exclusive use of the appropriate number of processors during the runs and to minimize interference from other users, cells were allocated exclusively to the program during its execution. This was achieved with the command `'allocate_cells -A n'`, where n is the number of processors and -A avoids 'loaded' cells such as those with ethernet cards. All results shown in the next sections are the average of three consecutive runs and all timings were made using the unix timer "time."

Results for varying numbers of processors are presented as temporal performance graphs[7]. The dotted line in each graph is the "naive ideal" line $t_p = t_s/p$ (often termed "linear speedup"), where t_s is the elapsed time for the serial program for a fixed problem size and t_p is the elapse time for the same problem size on p processors.

Other results are presented in tabular form in which we give absolute time and solutions per second. A solution is defined as the calculation of the wavenumber for one frequency (and its perturbed complement) at a station.

The results described in this article extend work reported in Ford[8].

4.3 Single Cell

A number of scalar optimizations have been applied to the code during the course of our work. Table 1 shows the performance improvements obtained. The solution rates differ for the two cases as case B has a better convergence rate for the iterative method used to solve the O–S equation.

4.4 Parallel Stations

The temporal performances for cases A and B for increasing numbers of processors are shown in Figures 13 and 14, respectively.

The two results in each graph (S and Sio) represent the advantage of performing I/O as soon as it is possible to do so: this effectively increases the pipeline length (see Fig. 12).

The performance results "drop away" from the naive ideal line due to two major factors. The first is that we run out of overlapped work to do; note that this is not a sharp cut off, as the pipeline overlap varies during the run due to the iterative method used. The second effect is due to the overhead of "filling up" the pipeline. This effect is more prominent in case B (Fig. 14), as the ratio of the pipeline length (a function of the number of frequencies) to the number of stations is larger than in case A.

4.5 Parallel Frequencies

To implement parallel frequencies we modified the algorithm (see Section 2.4, "Frequency and Perturbed Frequency"). Table 2 shows performance comparisons for the serial code and the parallel version run both serially, and in parallel, for test cases A and B. The parallel algorithm runs more slowly than its sequential counterpart. The temporal performance of cases A and B with the parallelism of parallel frequencies and parallel stations combined is shown in Figures 15 and 16, respectively. Again the two graphs show the advantage of performing I/O as soon as it is possible to do so.

Parallelizing at the frequency level has the advantage of reducing the pipeline length but has the disadvantage of modifying the algorithm to a less efficient form. Figures 15 and 16 show that parallel frequencies are only beneficial when the pipeline length is large, i.e., close to the number of

Table 1. Sequential Optimizations

Version	Elapsed Time	sol/s
A: Original	6m 0s	0.44
A: Optimized	1m 28s	1.82
B: Original	1h 51m 31s	0.60
B: Optimized	25m 17s	2.64

Table 2. Parallel Frequencies

Version	Elapsed Time	sol/s
A: sequential	1m 27.7s	1.82
A: par 1 cell	1m 34.1s	1.70
A: par 2 cells	55.6s	2.88
B: sequential	25m 16.8s	2.64
B: par 1 cell	29m 19.0s	2.27
B: par 2 cells	15m 14.5s	4.37

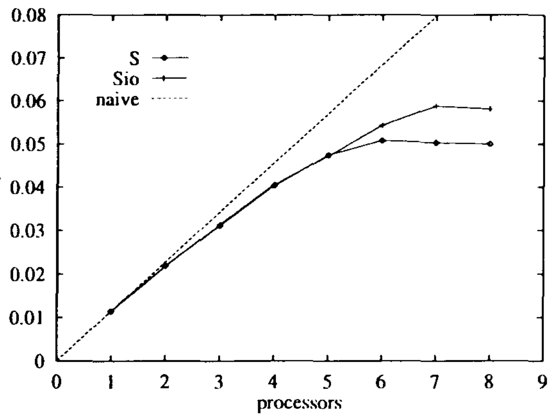


FIGURE 13 Case A parallel stations.

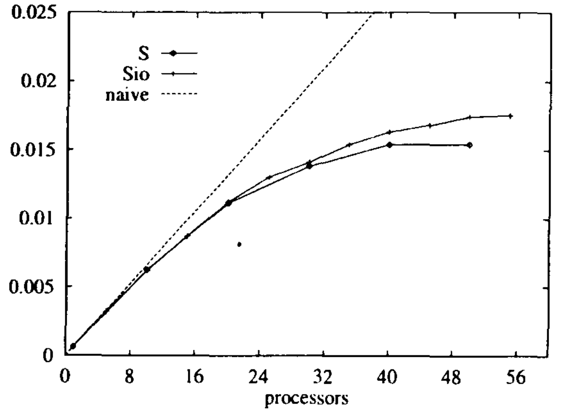


FIGURE 14 Case B parallel stations.

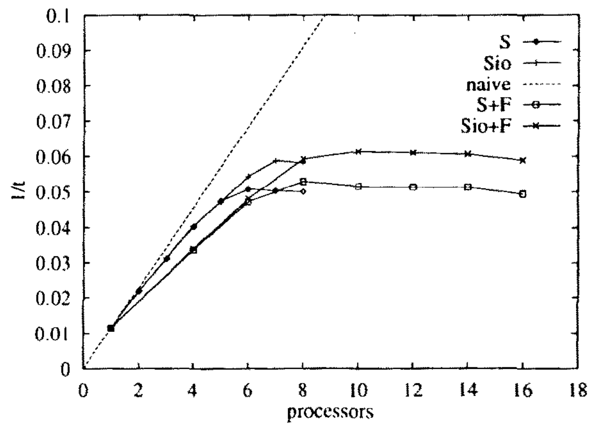


FIGURE 15 Case A parallel stations and frequencies.

stations and that for small numbers of processors it is better to parallelize at the station level only.

The interesting features in the parallel station and parallel frequency line (S + F) in Figure 16 are the subject of ongoing work.

4.6 Parallel Wavenumber Solutions

Table 3 shows the performance results when solving wavenumber solutions in parallel. The temporal performances of cases A and B, when the parallelism of parallel wavenumbers and parallel stations is combined, as shown in Figures 17 and 18, respectively.

The results of parallelizing at the station level only are also shown for comparison. There is a clear advantage in parallelizing at the wavenumber level as we obtain improved performance in both cases A and B. In case A the improvement is due to the increase in the amount of parallelism

Table 3. Parallel Wavenumbers

Version	Elapsed Time	sol/s
A: sequential	1m 27.7s	1.82
A: 3 cells	33.9s	4.72
B: sequential	25m 16.8s	2.64
B: 3 cells	8m 57.9s	7.44

available. In case B this improvement results in better peformance for the number of processors used; a major part of this is due to the reduction of the pipeline overhead.

4.7 Parallel Integration

When the integration was parallelized as described in Section 2.4, "Integration of Solutions," the program ran more slowly. The degradation in performance is due to the startup overhead of the

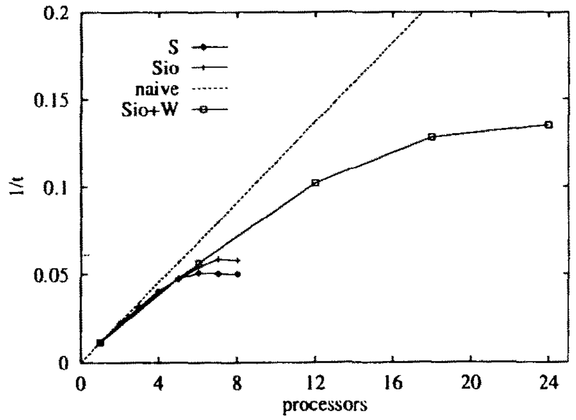


FIGURE 17 Case A parallel stations and wavenumbers.

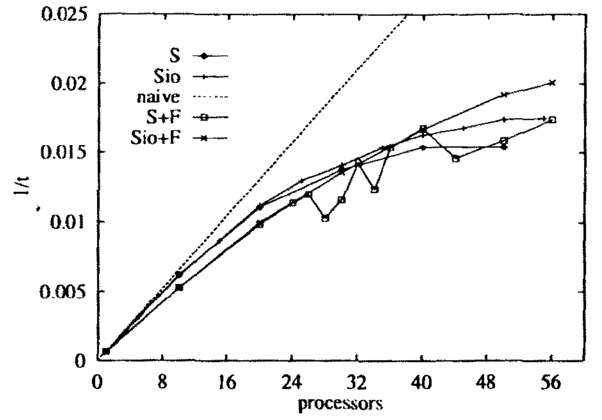


FIGURE 16 Case B parallel stations and frequencies.

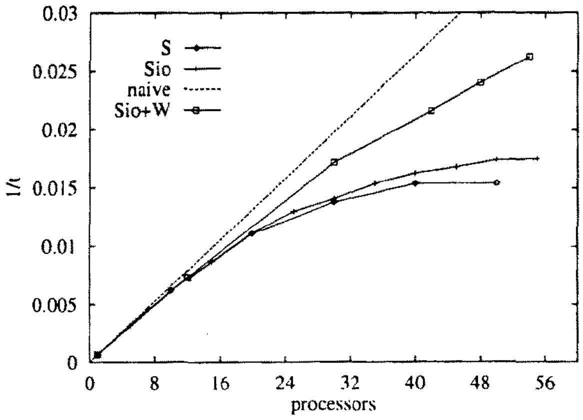


FIGURE 18 Case B parallel stations and wavenumbers.

Table 4. Parallel Integrations

Version	Elapsed Time	sol/s
A: sequential	1m 27.7s	1.82
A: sections 2 cells	4m 9.4s	0.64
A: reduced 2 cells	1m 13.5s	2.18

KSR parallel section construct being close or greater than the work inside each section. To reduce this overhead we expanded the KSR parallel sections (see Section 2.4, "Integration of Solutions") outside the integration loop, thus reducing the number of section startups. This was achieved by duplicating the loop and synchronizing manually via shared variables*, see below.

```
C*KSR* parallel sections(teamid=
C*KSR*& iteam3, private=J)
C*KSR* section
  DO J=1,NSTEPS
       CALL INTEG(AZ,..)
       DONE_INTEG(J)=.TRUE.
 20    IF (.NOT.DONE_ORTHO(J)) GOTO 20
   END DO
C*KSR* section
  DO J=1,NSTEPS
       CALL INTEG (CZ,..)
 10    IF (.NOT.DONE_INTEG(J)) GOTO 10
       CALL MODOR (CZ,..)
       CALL ORTHO (AZ,CZ,..)
       DONE_ORTHO(J)=.TRUE.
  END DO
C*KSR* end parallel sections
```

Table 4 shows the performance of these different versions. Clearly we only obtain a modest improvement in the solution time when parallelizing sections.

5 CONCLUSIONS

We have shown that for a typical low resolution search, we can reduce the solution time from minutes to near interactive times. In test case A, we have reduced the solution time from 6 minutes to under 8 seconds on 24 processors (this is the maximum number of processors that can be usefully employed on this problem size).

We have also demonstrated that for a typical high resolution search, we can reduce the solution time from hours to seconds. In test case B, we have reduced the solution time from over 1 hour 51 minutes to under 40 seconds on 54 processors.

These results were obtained by a combination of scalar optimization and parallelization. Scalar optimization accounts for over fourfold improvements in the results.

These results open up the possibility of much larger runs involving more stations and frequencies. However, more significantly, we demonstrate near iteractive performance for smaller runs, enabling a transformation of search methods and opening up new possibilities for this methods use in industry.

The algorithm has a reasonable amount of functional parallelism (for the standards of today's machines) but would not be classified as being massively parallel; useful parallelism on the KSR-1 is limited to six times the number of frequencies in a run. The majority of this parallelism is coarse grained with little communication and should therefore be efficient on both shared memory and distributed memory machines.

There is further parallelism available, within which a large amount of the total execution time is spent; however, the time per call is too small to exploit on the KSR-1.

Conceptually, a message-passing style would be a natural way to exploit the functional parallelism available and synchronization needed in this algorithm, particularly at the level of parallel stations. However, the combination of a shared memory model and dynamic thread control on the KSR-1, coupled with automatic thread and data migration, offered significant advantages when parallelizing the existing sequential program. These features enabled each level of parallelism to be separately parallelized with little code modification; they also enabled each level and combinations of levels to be incrementally developed and tested; finally, we could effective ignore thread placement and the migration of shared data.

6 FUTURE WORK

The algorithm described here models incompressible fluid flow. A compressible version is now operational; this has similar characteristics but is more computationally intensive. The compressible code will be the subject of future parallelization efforts.

* The premise here is that manual synchronization incurs less overhead than parallel sections.

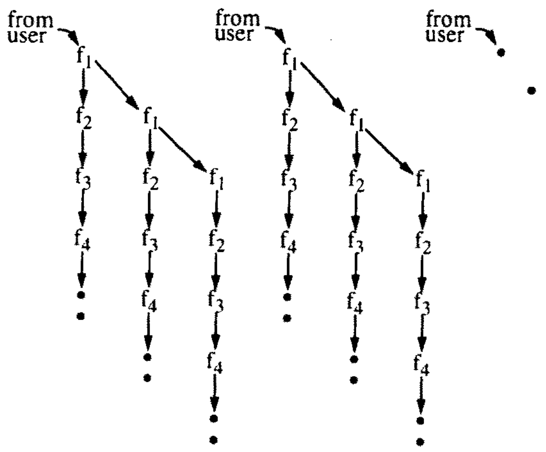


FIGURE 19 New parallelism strategy.

The dependency of a station on the previous station may be broken by "inputting" an accurate initial wavenumber approximation either at every station or, more feasibly, groups of stations. This would increase the amount of parallelism available and reduce the functional pipeline overhead (see Fig. 19). The accuracy of initial wavenumber approximations is one of the current focuses of the LFC group.

The algorithm described here is also constrained to a two-dimensional flow. A three-dimensional version of their model is now operational. The three-dimensional code will again increase the computational requirements. In this case a different parallelization strategy will also have to be adopted, as the three-dimensional version imposes new orderings on the computation (such as dependencies across the wing).

Work aimed at improving the efficiency of a search procedure in one of the core routines, eliminating any further redundant work and reducing the parallel overheads of thread start-up, may also yield further performance benefits.

ACKNOWLEDGMENTS

This work was supported by the Esprit Projects 2716-AMUS and 6253-SHIPS. The authors would also like to thank members of the Centre for Novel Computing and Laminar Flow Control Group for their invaluable advice.

REFERENCES

[1] H. Schlichting, *Boundary-Layer Theory*. New York: McGraw-Hill.

[2] L. M. Mack, "Boundary layer linear stability theory, special course on stability and transition of laminar flow." Advisory Group for Aerospace Research and Development, AGARD Report No. 709, March 1984.

[3] G. B. Schubauer, and H. H. Skramstad, "Laminar boundary oscillations and stability of laminar flow," NACA Report 909.

[4] H. B. Squire, "On the stability of three-dimensional disturbances of viscous fluid between parallel walls," *Proc. R. Soc.* [*A*], vol. 142, pp. 621–628, 1933.

[5] R. L. Burden, *Numerical Analysis* (4th ed.). PWS-KENT, 1989.

[6] KSR-1 Programming Manuals, 170 Tracer Lane, Waltham, MA, 1993.

[7] R. A. Hockney, "Framework for benchmark performance analysis," *Supercomputer*, vol. 48, pp. 9–22, 1992.

[8] R. W. Ford, "Proceedings of Super Computing Europe '93," Utrecht, Holland, February 22–24, 1993.