INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS, VOL. 16, 891–914 (1993)

A COMPARISON OF COARSE AND FINE GRAIN PARALLELIZATION STRATEGIES FOR THE SIMPLE PRESSURE CORRECTION ALGORITHM

A. J. LEWIS AND A. D. BRENT

BHP Research, Newcastle Laboratories, P.O. Box 188, Wallsend NSW 2287, Australia

SUMMARY

The primary aim of this work was to determine the simplest and most effective parallelization strategy for control-volume-based codes solving industrial problems. It has been found that for certain classes of problems, the coarse-grain functional decomposition strategy, largely ignored due to its limited scaling capability, offers the potential for significant execution speed-ups while maintaining the inherent structure of traditional serial algorithms. Functional decomposition requires only minor modification of the existing serial code to implement and, hence, code portability across both concurrent and serial computers is maintained. Fine-grain parallelization strategies at the 'DO loop' level are also easy to implement and largely preserve code portability. Both coarse-grain functional decomposition and fine-grain loop-level parallelization strategies for the SIMPLE pressure correction algorithm are demonstrated on a Silicon Graphics 4D280S eight CPU shared memory computer system for a highly coupled, transient two-dimensional simulation involving melting of a metal in the presence of thermal-buoyancy-driven laminar convection. Problems requiring the solution of a larger number of transport equations were simulated by including further scalar variables in the calculation. While resulting in slight degradation of the convergence rate, the functional decomposition strategy exhibited higher parallel efficiencies and yielded greater speed-ups relative to the original serial code. Initially, this strategy showed a significant degradation in convergence rate due to an inconsistency in the parallel solution of the pressure correction equation. After correcting for this inconsistency, the maximum speed-up for 16 dependent variables was a factor of 5·28 with eight processors, representing a parallel efficiency of 67%. Peak efficiency of 76% was achieved using five processors to solve for 10 dependent variables.

KEY WORDS Computational fluid dynamics Parallel computing Parallel processing Functional decomposition SIMPLE algorithm Pressure correction schemes

1. INTRODUCTION

Parallel processing offers the potential for near real-time execution of Computational Fluid Dynamics (CFD) codes simulating real-world industrial problems. To date, efforts to achieve high performance in parallel implementations have often concentrated on arrays of inexpensive processors using domain decomposition techniques,[1] largely ignoring functional decomposition as a technique requiring code that in many cases is unable to take full advantage of machine parallelism. The speed-ups achievable using functional decomposition are constrained by problem-specific details. How many different tasks can be performed independently of one another? This makes the number of CPUs usable strictly limited. However, functional decomposition does

0271–2091/93/100891–24$17.00

Received 8 October 1991
Revised 4 June 1992

have the advantages of ease of application and largely maintaining the integrity of traditional serial algorithms. It can also be transferred without difficulty between different machines as its application generally does not require extensive modification of the solution algorithm, unlike other parallelization strategies such as domain decomposition methods.

For certain classes of problems, functional decomposition has been shown to be an effective parallelization strategy for execution on a project supercomputer class machine. These problems are, generally, governed by a comparatively large number of transport equations. Areas of specific interest lie in the modelling of the ironmaking blast furnace,[2] which involves the solution of over 15 dependent variables, including highly coupled systems of chemical reactions and multiphase flows. Other applications of interest include modelling of near net shape casting metal delivery systems,[3,4] and continuous casting operations[5] which require three-dimensional simulation of fluid flows with heat transfer and solidification mechanisms.

Functional decomposition offers a straightforward method of problem partitioning that can be easily implemented in segregated solution algorithms. Good parallel efficiencies are obtainable when load balance is reasonable. In this work, the use of two functional decomposition strategies for the solution of the Navier–Stokes equations in their primitive variable form using the SIMPLE pressure correction algorithm is described. The original serial algorithm is described briefly and the development of a concurrent strategy outlined. A standard, fully conservative control-volume-based numerical scheme[7] is used for all calculations. Degradation in the convergence rate was observed when the pressure correction equation was solved inconsistently in one of the functional decomposition strategies.

The coarse-grain functional decomposition approach is contrasted with fine-grain or loop-level parallelization. In this approach both coefficient assembly and solution of the transport equations are parallelized by spreading execution of DO loop iterations over several CPUs. Standard red–black ordering[6] is used for parallelization of the line-by-line Tri-Diagonal Matrix Algorithm (TDMA) solver.[7]

These strategies are demonstrated by solving a highly coupled, transient two-dimensional heat transfer and fluid flow problem. This simulation is numerically intensive and is representative of the class of problems presently under investigation. The effect on parallel efficiency of increasing the number of transport equations is simulated by multiple solution of a scalar transport equation. Since the additional equations introduced are uncoupled, this will give an upper bound on the speed-up obtainable for the functional decomposition strategies. Any coupling between additional transport equations would tend to reduce the rate of convergence to a greater degree for functionally decomposed code than for the original serial code. This is because the concurrent solution of several transport equations leads to their being 'decoupled in time', newly computed values of variables not being fed into the solution of subsequent equations as occurs in serial implementations.

2. SERIAL IMPLEMENTATION OF THE SIMPLE PRESSURE CORRECTION ALGORITHM

The SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm is based on a pressure correction scheme. The reader is referred to Patankar[7] for details of the algorithm derivation. The SIMPLE algorithm is implemented using a fully conservative control-volume-based finite-difference scheme[7] and the QUICK scheme of Leonard,[8] with flux limiting after Gaskell and Lau,[9] for representing convective transport terms. The solid/liquid phase change is

accommodated using the enthalpy porosity approach.[10] Note that the notation and conventions of Patankar[7] are used in this paper.

The analysis of a two-dimensional coupled heat transfer and fluid flow problem requires the simultaneous solution of two momentum equations, a continuity equation and an energy equation. Note that many industrial problems may require the solution of additional dependent variables such as concentration of chemical species. For example, a comprehensive two-dimensional CFD model of a blast furnace ironmaking operation[2] requires the simulation of a multitude of chemical reactions and the solution of over 15 dependent variables. These include chemical species in addition to the momentum and energy transport equations. It is in the segregated solution of problems such as this which involve a large number of transport variables that functional decomposition becomes an attractive parallelization strategy.

For each of the dependent variables considered, the governing convection/diffusion transport equation is of the general form (for dependent variable ϕ)[7]

$$\frac{\partial}{\partial t}(\rho\phi)+\operatorname{div}(\rho\mathbf{u}\phi)=\operatorname{div}(\Gamma\operatorname{grad}\phi)+S, \tag{1}$$

where Γ is the diffusion coefficient and S is the source term.

The governing differential equation by integration over a control volume yields the general discretization equation at the ith grid point of the form[7]

$$a_i\phi_i=\Sigma a_{\mathrm{nb}}\phi_{\mathrm{nb}}+b, \tag{2}$$

where the nb subscripts refer to the neighbouring points to the ith point.

In the segregated solution methodology inherent in the SIMPLE algorithm, the dependent variables are effectively decoupled and solved for independently. For each dependent variable, a separate system of linearized algebraic equations is derived with coefficients assembled from the velocity fields and diffusion coefficients.

The sequence of operations in the SIMPLE algorithm is, after Patankar:[7]

1. Guess the pressure field.
2. Solve the momentum equations (separately for each component of velocity).
3. Solve the pressure correction equation.
4. Calculate the pressure field from the pressure correction.
5. Correct the velocity fields.
6. Solve the discretization equations for any other variables (such as temperature and turbulence quantities) that influence the flow field, e.g. through fluid properties. (Otherwise calculate them after a converged solution for the flow field is obtained).
7. Treat the corrected pressure as a new guess, return to step 2 and repeat until convergence is reached.
8. If the problem is transient step forward in time and repeat.

For a two-dimensional, laminar, coupled heat transfer and fluid flow problem, the sequential form of the SIMPLE algorithm is shown in Figure 1. The specific governing equations for the two velocity components, mass conservation and energy are

$$\frac{\partial}{\partial t}(\rho u)+\operatorname{div}(\rho\mathbf{u}u)=\operatorname{div}(\mu\operatorname{grad}u)-\frac{\partial P}{\partial x}+S_u, \tag{3}$$

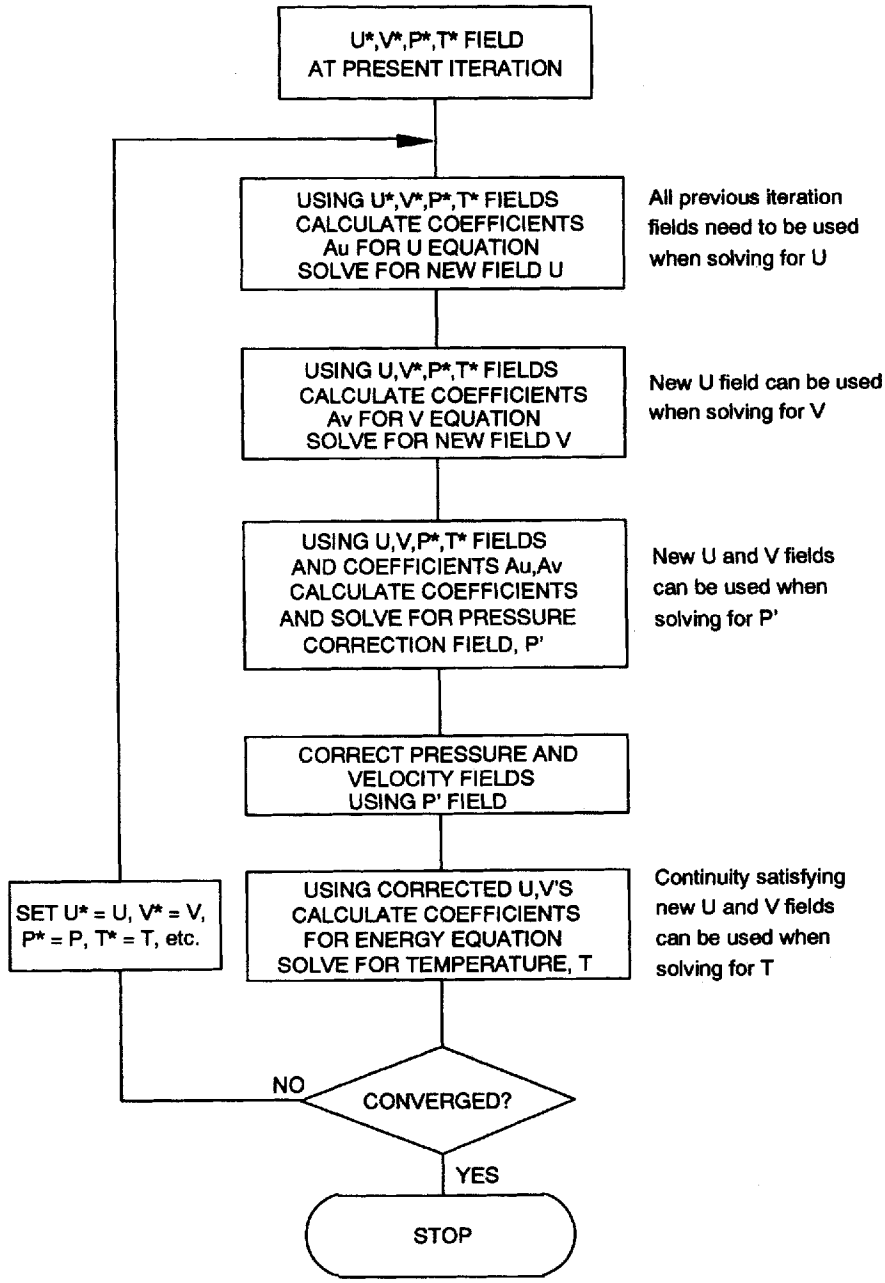

Figure 1. Fully sequential implementation of SIMPLE algorithm

where

$$S_u = Au,$$

$$\frac{\partial}{\partial t}(\rho v) + \operatorname{div}(\rho \mathbf{u} v) = \operatorname{div}(\mu \operatorname{grad} v) - \frac{\partial P}{\partial y} + S_v, \tag{4}$$

where

$$S_v = Av + \rho_{\text{ref}} g \beta (T - T_{\text{ref}}),$$

$$\frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} = 0, \tag{5}$$

$$\frac{\partial}{\partial t}(\rho c T) + \operatorname{div}(\rho c \mathbf{u} T) = \operatorname{div}(k \operatorname{grad} T) + S_T, \tag{6}$$

where

$$S_T = \rho \frac{\partial \Delta H}{\partial t}.$$

In this fully sequential version of the algorithm, the coefficients for each dependent variable are calculated using the latest values available for all other variables. For example, the coefficients for the v-momentum equation are calculated using previous iteration values of v, P and T, and newly calculated values of u. Similarly, the coefficients for the energy equation are calculated using the newly calculated present iteration values of both u and v. While assembling the coefficients for a particular variable, the fully sequential form of the algorithm is, thus, able to utilize the most recently calculated values of all the other variables. Note in Figure 1 that this effect 'cascades' down the solution sequence and that progressively less previous iteration fields (denoted by *) are used in coefficient calculation as the end of an iteration is neared. The final transport equation to be solved, in this case the energy equation, is, thus, able to utilize newly calculated fields for both u and v. This use of the most up-to-date data when solving each transport equation accelerates the rate of convergence of the algorithm.

The solution of each system of algebraic discretization equations may proceed in a number of ways. Direct methods such as Gaussian elimination are generally prohibitively costly in terms of computer resources and, hence, iterative methods are widely used. For this implementation, a combination of the Gauss–Seidel method[7] and the direct Tri-Diagonal Matrix Algorithm (TDMA) was used by employing the TDMA on a line-by-line basis as described by Patankar. The line-by-line sweeping was repeated in each of the co-ordinate directions within a single pass of the solving routine, similar to the ADI method of Peaceman and Rachford.[11] Each pass of the solving routine was augmented by a block correction procedure.[12] The whole procedure was performed iteratively, updating coefficients at each iteration.

Functional decomposition parallelizes the execution of the algorithm by placing the coefficient assembly and solution of a system of equations for a given dependent variable on a separate CPU. This becomes an economic method of utilizing machine parallelism when the number of transport equations approaches or exceeds the number of available CPUs. Maximum gains will be obtained when the number of transport equations matches the number of CPUs exactly.

In contrast, the loop-level parallelization strategy spreads execution of DO loop iterations in coefficient assembly and equation solution over all available CPUs. This easily utilizes as much of the computer as may be devoted to the problem and is readily scalable, the addition of more CPUs leading to immediate gains in execution speed. There is a limit to this characteristic in that no more CPUs can be used than there are iterations in the typical program DO loop. For real-world problems, however, this limit generally exceeds the CPUs available in machines of small to moderate parallelism.

The Thomas algorithm involves calculation of recurrence relations, making loop level parallelization impossible within each TDMA traverse. In addition, the line-by-line method of 'sweeping' the TDMA across the problem domain renders it in its simplest form inherently unparallelizable. However, red–black ordering[6] of the TDMA line-by-line technique allows concurrent calculation.

3. CONCURRENT IMPLEMENTATION—FUNCTIONAL DECOMPOSITION

3.1. Functional decomposition strategy I—independent solution of momentum and pressure correction equations

It should be recognized that functional decomposition strategy I (see Figure 2) is not a rigorous implementation of the true SIMPLE scheme.[7] Although this strategy results in a relatively severe penalty in convergence rate, it still offers the potential for significant computational speed-ups for certain problems. In this strategy, the u, v, P' and T equation are solved for independently on separate processors. The coefficients for the u, v and T equations are calculated using the previous iteration fields u^*, v^* and T^*. Calculation of the coefficients for the P' equation also requires using the previous iteration values of the coefficients for the u and v equations. The velocity and pressure corrections are made only after solving for u, v, P' and T. This results in an inconsistency in that the calculated pressure corrections are being applied to the new u and v fields, rather than to the previous iteration u^* and v^* fields which were actually used to calculate the pressure corrections. This inconsistency results in the loss of one of the convergence-promoting characteristics of the SIMPLE algorithm, namely, that convergence is approached through a series of continuity-sastisfying velocity fields.[7] Note, however, that as convergence is neared, this incon-

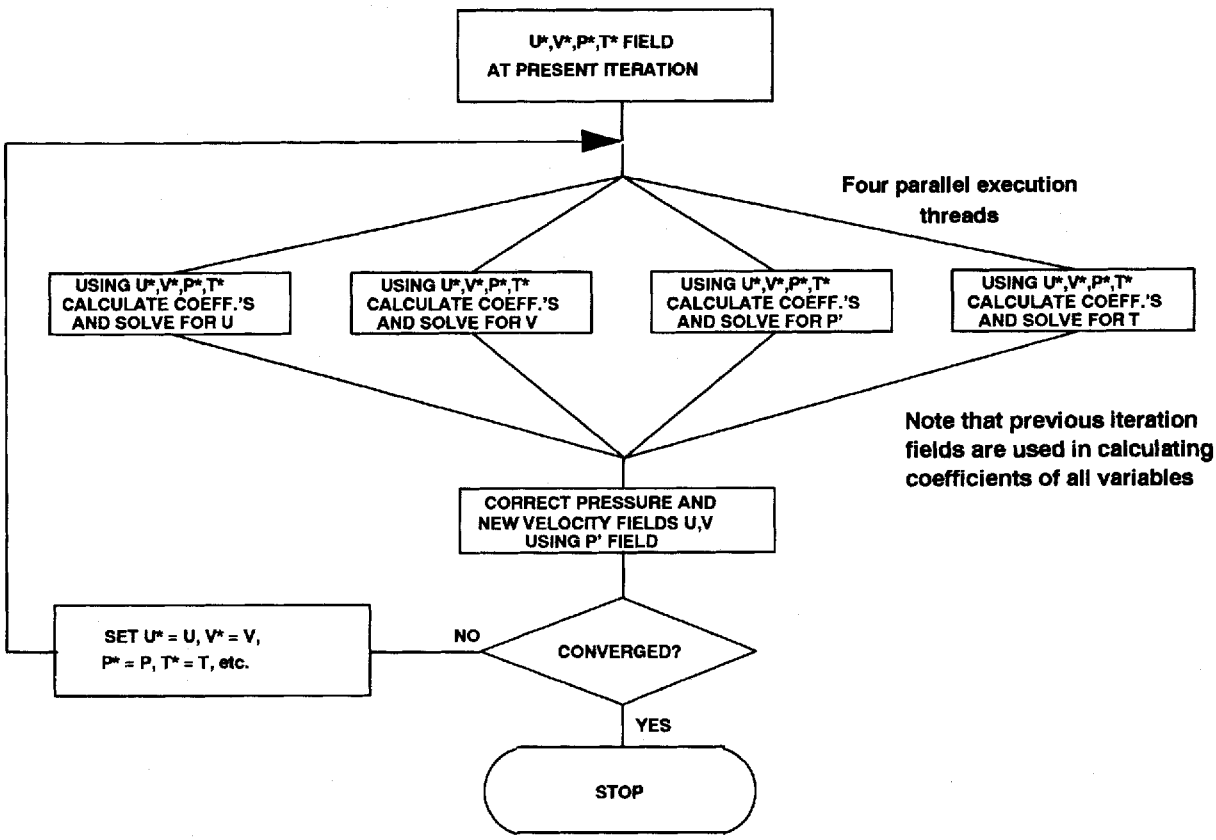

Figure 2. Functional decomposition strategy I—Independent solution of momentum and pressure correction equations

sistency disappears because the previous iteration values and newly calculated values approach each other. Although a certain degree of degradation in the convergence rate may be expected using this strategy, this can be offset by the increase in computational speed arising from the fact that the pressure correction equation is being solved at the same time as all the other dependent variables.

3.2. Functional decomposition strategy II—independent solution of momentum equations followed by sequential solution of pressure correction equation

Functional decomposition strategy II maintains the integrity of the SIMPLE algorithm by treating the solution of the pressure correction equation as a sequential operation after solving the u and v equations (see Figure 3). The previous iteration fields u^*, v^* and T^* are used in calculating coefficients for the velocity and energy equations. These variables are solved for independently on separate processors, after which the updated u and v fields, along with the updated velocity equation coefficients, are used in the calculation of the pressure correction coefficients. After solving the pressure correction equation, the velocities and pressures are corrected using the pressure correction field. This strategy, therefore, ensures that the pressure corrections being applied to correct the velocities are fully consistent with these velocities and that convergence is again approached through a series of continuity-satisfying velocity fields. A slight degradation in convergence rate over the fully sequential scheme shown in Figure 1 may still be expected, however, as the previous iteration values of all variables must be used to calculate the coefficients of all transport equations except the pressure correction equation for the present iteration.

4. CONCURRENT IMPLEMENTATION—LOOP-LEVEL PARALLELIZATION

The loop-level parallelization strategy spreads execution of DO loop iterations in coefficient assembly and equation solution over all available CPUs. Parallelization of the coefficient assembly is straightforward, but modifications are required to the line-by-line iterative solver. Each grid line in a particular co-ordinate direction is solved sequentially. The values of the dependent variable being solved on the neighbouring lines are held to be the 'latest' values and constant during performance of the TDMA, and the values along the chosen line solved by the TDMA. The direction of traverse is changed so that the problem domain is swept in all four directions.

As mentioned earlier, calculation of the recurrence relations within the TDMA algorithm prohibits loop-level parallelization within each traverse. When attempting to introduce loop-level parallelism on a line-by-line basis, the assumption that values on neighbouring lines are constant introduces problems due to the non-deterministic updating of values in memory due to concurrent processing. Neighbouring lines are treated separately by different processes and, so, exactly *when* neighbouring values have been, or are to be, updated is unknown.

This problem is easily overcome by modifying the sweeping TDMA solver to use standard red–black ordering of the lines.[6] The stride through the loop performing the traverse of lines was changed to two, and the loop performed twice from two adjacent starting lines. In this way, every line is processed but, during any pass through the loop, every line solved has constant neighbours. This is shown diagramatically in Figure 4. The solution of lines within a loop can, thus, be safely performed concurrently.

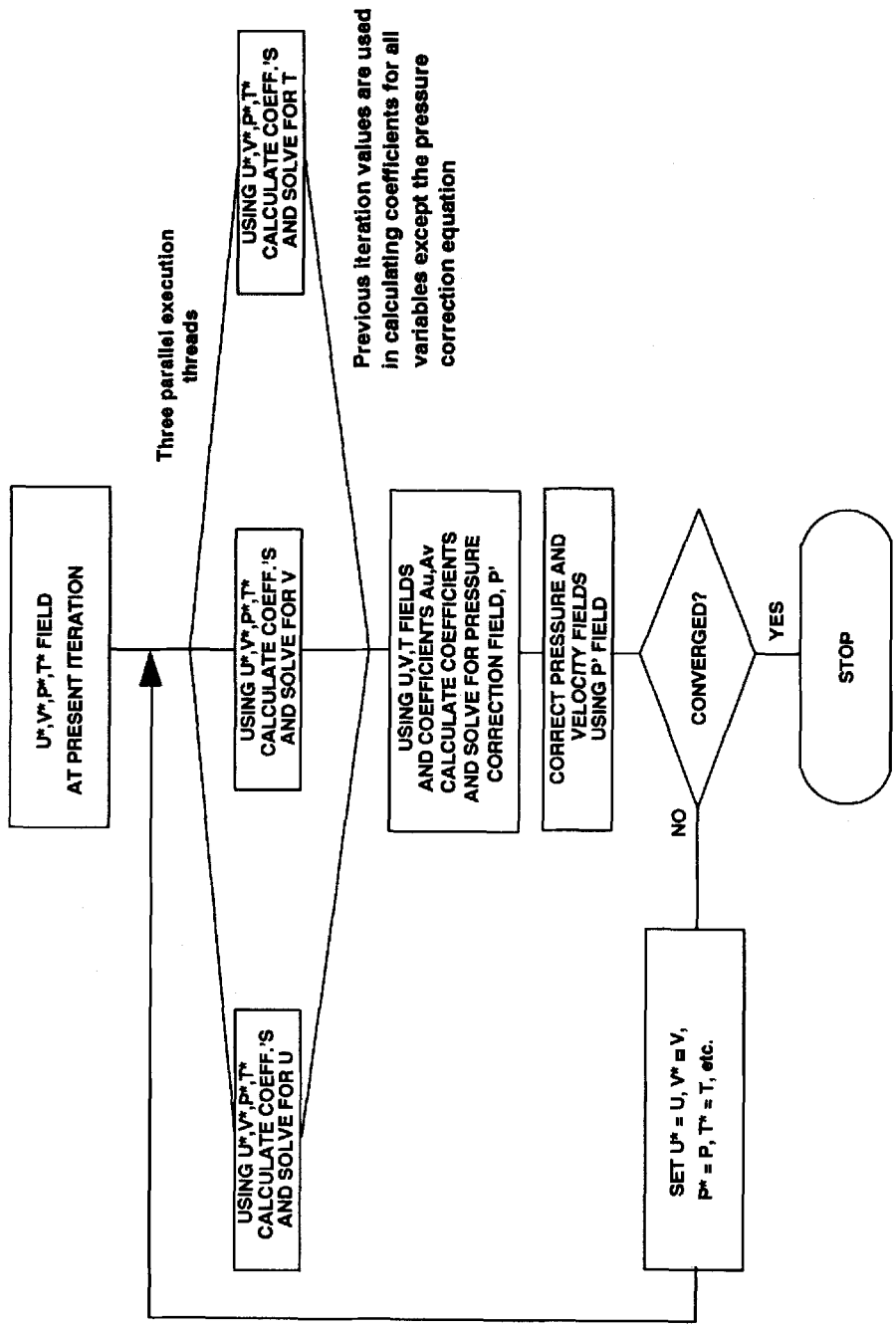

Figure 3. Functional decomposition strategy II—Independent solution of momentum equations, followed by sequential solution of pressure correction equations

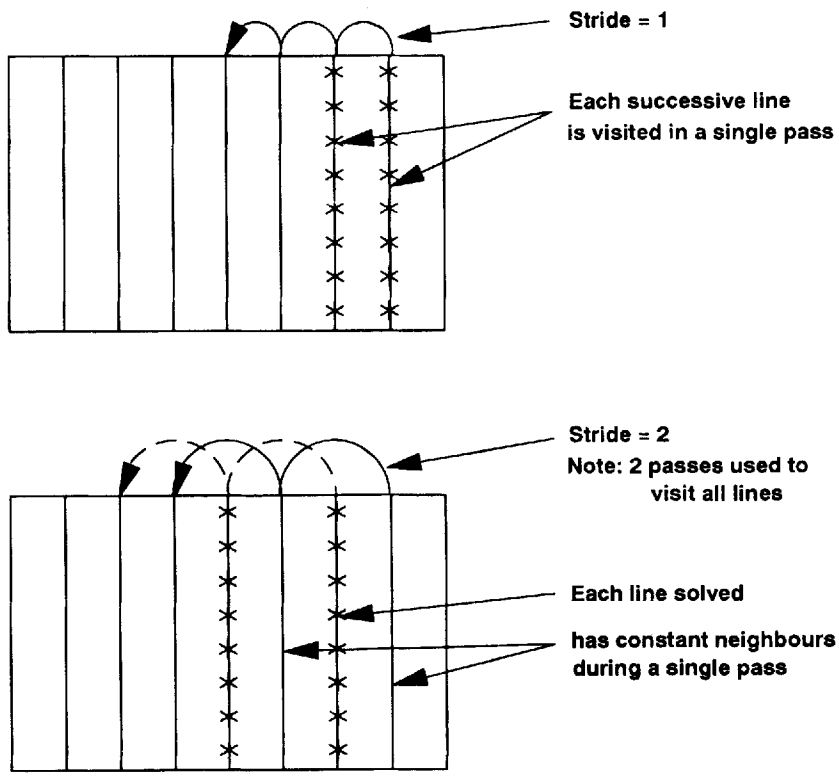

Figure 4. Red–black ordering of loop iteration

In addition, loop-level parallelization was implemented throughout the remainder of the code, consisting mainly of coefficient formulation and source term calculation. There were no inherent obstacles to parallelization in these tasks.

5. TEST PROBLEM

The two-dimensional melting of gallium in the presence of laminar thermal-buoyancy-driven convection was used as a benchmark problem to test the various parallelization strategies. This problem has previously been numerically investigated by Brent *et al.*[9] and was chosen as it may be considered typical of coupled, non-linear CFD problems which are computationally intensive. The governing equations specific to this problem are equations (3)–(6).

Note that the Boussinesq approximation is used in modelling the buoyancy term. The geometry, boundary and initial conditions for the problem are shown in Figure 5. Further details such as the physical properties used for gallium may be found in Reference 9.

The early stages of melting were simulated. Computations were carried out up to a simulation time of 50 s using a constant time step of 0·5 s. The non-uniform computational grid of 24×40 control volumes used is shown in Figure 6. A fine grid is required in the fluid regions to resolve steep velocity and temperature gradients and essentially isothermal conditions exist ahead of the melt front. The melt front and streamlines at 50 s are plotted in Figure 7, indicating the complex flow structure arising.

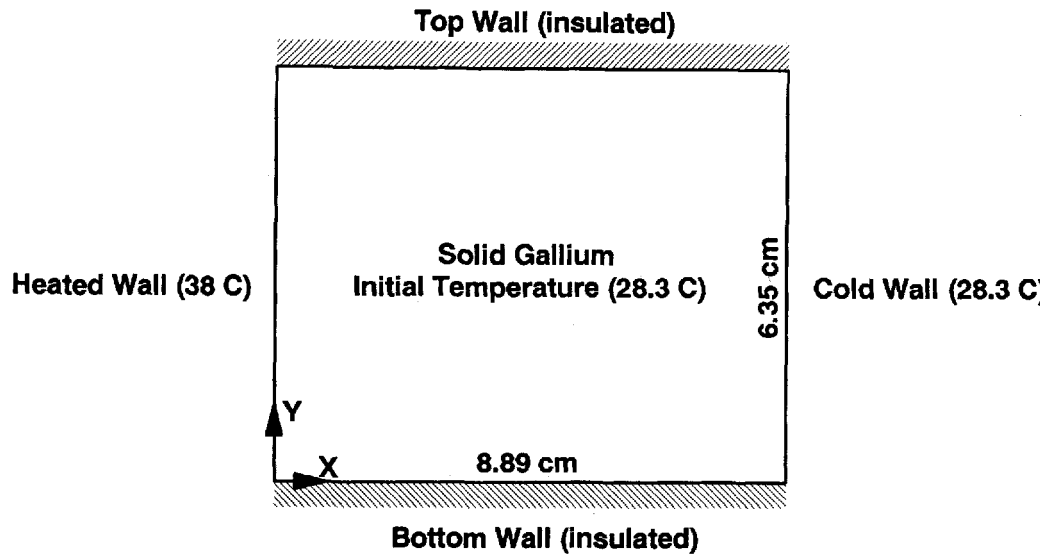

Figure 5. Benchmark problem description

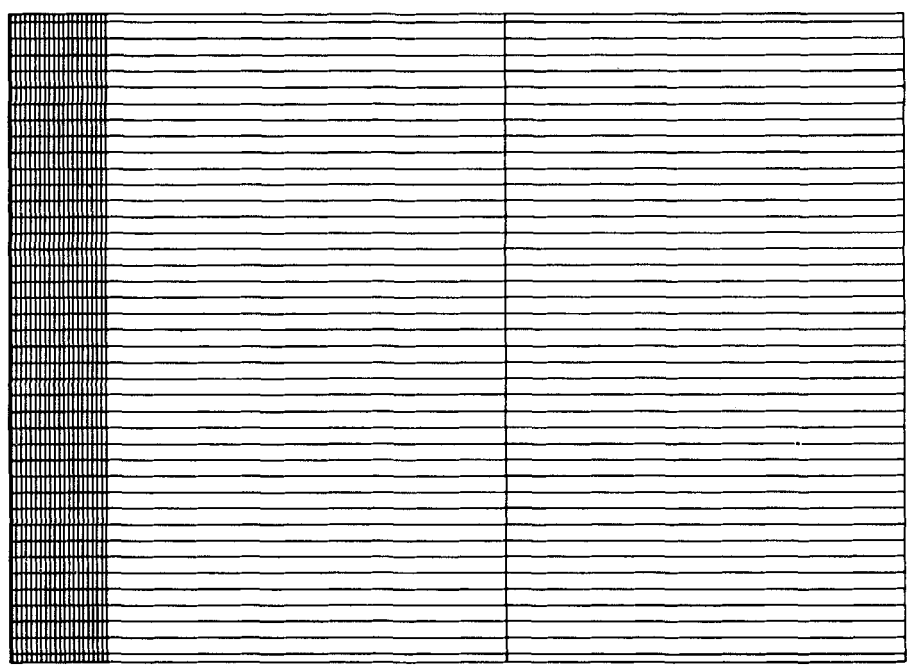

Figure 6. Computational mesh

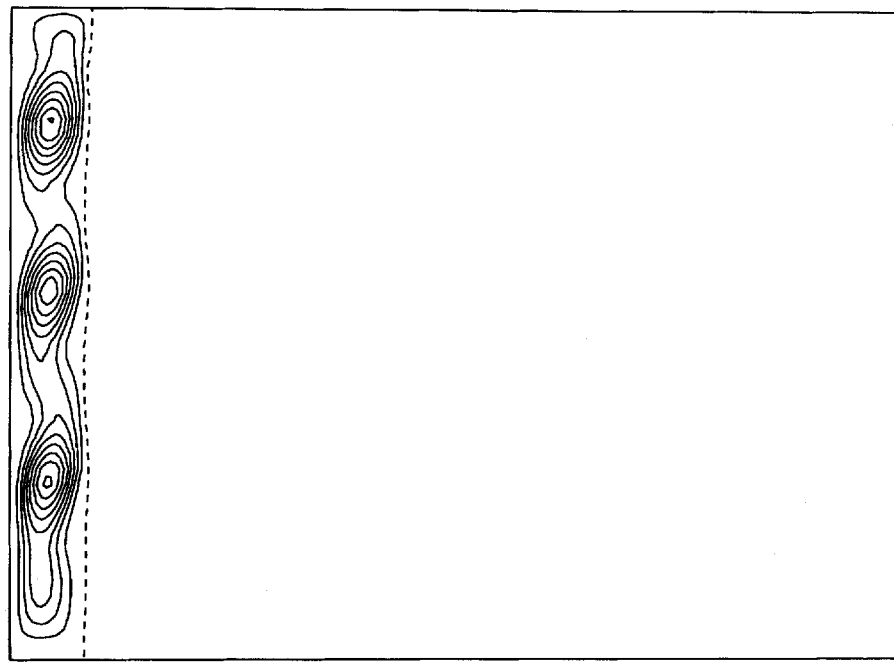

Figure 7. Melt front and streamlines at 50 s

6. RESULTS AND DISCUSSION

6.1. Load balance profiling

The Silicon Graphics 280S f77 FORTRAN compiler profiling option was used on the benchmark problem to generate profile reports to indicate relative proportions of CPU time spent on different operations. A number of profiles were generated to examine various aspects of CPU requirements. The first of these profiles gives an indication of the proportions of CPU time spent solving for each transport equation and is shown in Table I. It should be noted that all profiling runs were conducted without any other users on the SGI 280S system and, hence, CPU clashes with other jobs (apart from the system accounting which is insignificant) were avoided.

Table I. Profile giving breakdown of CPU requirements by dependent variable

Dependent variable	Proportion of CPU time spent of each variable
u-velocity	27%
v-velocity	29%
Pressure correction P'	17%
Temperature	27%
Totals	100%

It can be seen from Table I that solution of the pressure correction equation requires significantly less computational effort than does the solution of the other transport equations. This imbalance arises because calculation of the pressure correction equation coefficients does not require computation of the flux-limited QUICK convective differencing terms. The small discrepancies in computational requirements for coefficient calculation of the other dependent variables result from differences in source term calculations; for example, the calculation of the Boussinesq buoyancy source term for the v-equation results in the v-equation requiring more CPU time in coefficient calculation than the u-equation (29% compared to 27%). Table II shows the proportion of total CPU time spent on the tasks of coefficient assembly and matrix solution for all the variables.

Note that approximately 39% of the computational effort is used in the solution of the linear discretization equations (i.e. in the TDMA line-by-line solver). The remaining CPU time, approximately 7%, involves convergence checking of heat and mass balance, enthalpy updates for phase changes and like tasks.

6.2. *Speed-up analysis*

Amdahl's Law[13] can be used to determine the maximum theoretical speed-up for different parallelization strategies. Although the maximum theoretical speed-up can generally not be attained in practice due to system overheads, it is a useful tool in evaluating parallelization strategies. Amdahl's Law states

$$\frac{t_s}{t_p} = \frac{n}{(F_s \times n) + F_p}, \tag{11}$$

where t_s is the sequential run-time, t_p the parallel run-time, F_s the sequential portion of the code, F_p the parallel portion of the code and n the number of processors used.

Note that, although machines with large numbers of processors may be valuable, some parallelization strategies restrict the number of processors which can be used. For example, functional decomposition strategy II allows the use of only three CPUs (one each for u, v and T-equations). The maximum theoretical speed-up attainable for different parallelization strategies are given in Table III. Note that this analysis assumes that no change occurs in convergence rates for different implementations.

Profiles of execution indicated that some 90% of the code was parallelized in the loop-level implementation and this is reflected in the speed-up shown in Table III. It should be recognized that the values in Table III apply only to the benchmark problem in which u, v, P' and T are the only transport equations being solved. When a problem incorporates a large number of depend-

Table II. Proportion of total CPU time required for coefficient assembly and line-by-line solver for benchmark problem

Operation	Proportion of total CPU time
Coefficient and source term calculation for all variables	54%
Iterative solution of discretization equations for all variables (four-sweep line-by-line TDMA solver with block correction)	39%

Table III. Maximum theoretical speed-ups for different parallelization strategies

Parallelization strategy	Number of CPUs used	Maximum theoretical speed-up
Strategy I	4	3·41
Strategy II	3	2·48
Loop level	8	4·74

Table IV. Summary of results for parallelization strategies compared to sequential code for benchmark problem

Code	Elapsed time (s)	Average number of iterations per timestep	Total CPU time (s)	System overhead (s)
Fully sequential	483	24·4	483	1
Loop level	220	25·0	1567	35
Strategy I	312	29·0	1027	274
Strategy II	256	25·9	784	29

ent variables (e.g. the blast furnace model[2]), the theoretical speed-up for strategies I and II may be expected to be significantly higher.

6.3. *Comparison of results for functional decomposition*

Table IV contains a summary of the results obtained for both functional decomposition parallelization strategies I and II and the results obtained running a fully sequential code on a single CPU of a SG280S.

Simulation of the benchmark problem on a fully sequential code (see Figure 1 for sequential SIMPLE) took 483 CPU seconds and an average of 24·4 iterations to reach convergence at each time step. Note that the CPU time is equivalent to the elapsed time in this case as only one CPU is used. The total CPU time for the loop-level parallelized code includes loop-scheduling overheads. The classical definition of efficiency of a parallelization strategy, as shown by Akl,[14] is

$$\text{Efficiency} = \frac{\text{Sequential run-time}}{\text{Parallel run-time} \times n}, \tag{12}$$

where n is the number of processors and the denominator gives the 'cost' of the parallel algorithm.

Strategy I gave a speed-up relative to the sequential code of 1·55. This implementation, thus, yielded only 45% of the theoretical maximum speed-up given by Amdahl's law (see Table III) and had an efficiency of only 39%. The poor performance of this strategy is due to the significant degradation in convergence rate (19% more iterations than the fully sequential code). This decrease in convergence rate is primarily the result of the inconsistency in the velocity corrections discussed above.

Strategy II gave a speed-up of 1·89, which is 76% of the theoretical maximum. The efficiency of this strategy is 63%. Note that only a slight degradation in convergence rate was noted using this strategy, with only 6% more iterations being required for convergence within each time step. The

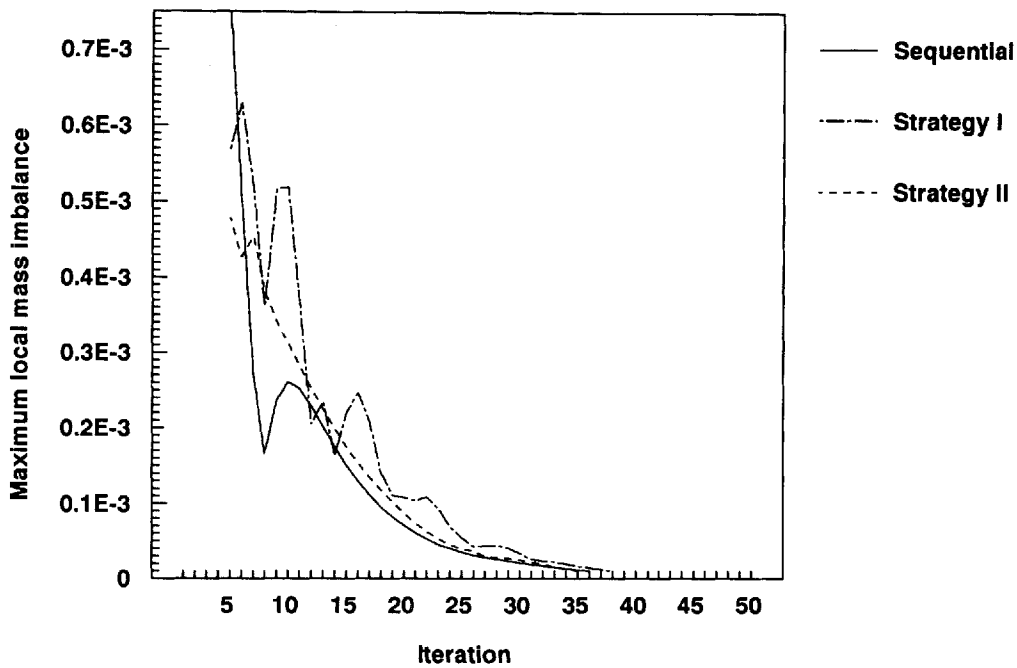

Figure 8. Maximum mass residuals for functional decomposition strategies at timestep 82

improved load balancing in this implementation also results in considerably lower system overheads than in strategy I.

The rates of convergence of the two functional decomposition strategies and the sequential code are compared for a typical timestep involving significant velocity fields in Figure 8. Note that convergence testing was performed only from the fifth iteration to ensure a minimum of five iterations per timestep. The convergence criteria were a maximum local mass imbalance of less than $3{\cdot}0 \times 10^{-3}\%$ of the total mass present in the cavity and an absolute error in the unsteady heat balance of less than 0·05%. The maximum local mass imbalance is derived from the mass source term, given by[7]

$$b = (\rho u_w - \rho u_e)\Delta y + (\rho v_s - \rho v_n)\Delta x. \tag{14}$$

It may be noted that strategy I shows significant oscillatory behaviour caused by the inconsistent application of pressure corrections. As a result, the velocity fields no longer satisfy continuity at the end of each iteration. Strategy II, in which this problem was resolved, shows near monotonic decrease in residual imbalance.

The iterations required to reach convergence at each timestep for the entire simulation are compared in Figure 9.

In Figure 9 it can be seen that the solution of the test problem proceeds through three distinct phases. Up to about timestep 20, a significant number of iterations are required at each timestep to resolve the high temperature gradients and rapid melting near the left wall. During this phase, the problem is essentially a classical Neumann problem. Between timesteps 20 and 50, the problem is still heat-conduction-dominated since fluid flow is constrained by the small size of the liquid volume. However, temperature gradients are smaller than those encountered within the first 20 timesteps and, hence, fewer iterations are required to reach convergence. As melting

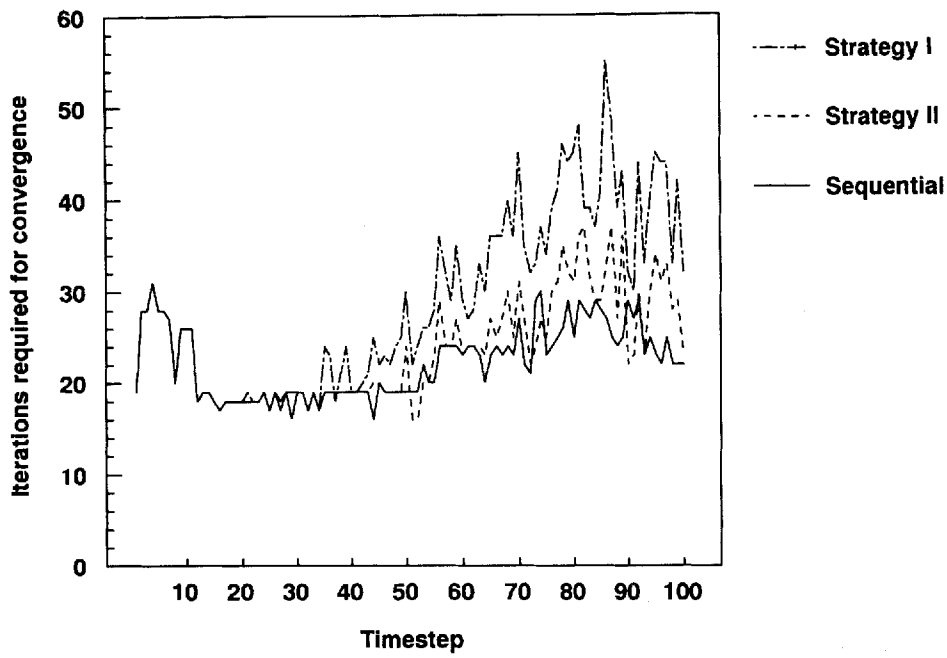

Figure 9. Iterations required for convergence for functional decomposition strategies

proceeds, buoyancy forces accelerate fluid motion and the number of iterations required for convergence increases to resolve the complex, transient flow structures arising.

The pressure correction inconsistency in strategy I is also evident from Figure 9. It can be seen that while strategy I is similar in performance to the other strategies when the problem is conduction-controlled, as the fluid flow becomes more significant it requires considerably more iterations. The effect of increasing iterations also becomes apparent earlier than for the other strategies.

From this evidence and for the reasons stated earlier, strategy I could be expected to experience convergence difficulties in highly coupled, flow-dominant problems such as those dealing with turbulence, phase change and chemical reaction. This strategy, therefore, received no further consideration.

6.4. Effect of increasing number of transport equations

From profiling of the execution of both sequential and functional decomposition implementations of the algorithm, it could be expected that solving for further dependent variables would lead to little increase in the parallel-code run-time while the sequential-code run-time would be expected to increase by about 27% of the current value for each additional variable (see Table I). There may be expected some increase in the parallel code run-time due to increased contention for the global lock used in the computer's memory management scheme. This would lead to some decreases in the efficiency of the code and the degree to which it approaches the theoretical maximum speed-up given by Amdahl's law. However, these conditions could not be expected to hold for systems where the number of dependent variables being solved approaches or exceeds the parallelism of the machine architecture. In the case of the number of variables exceeding the

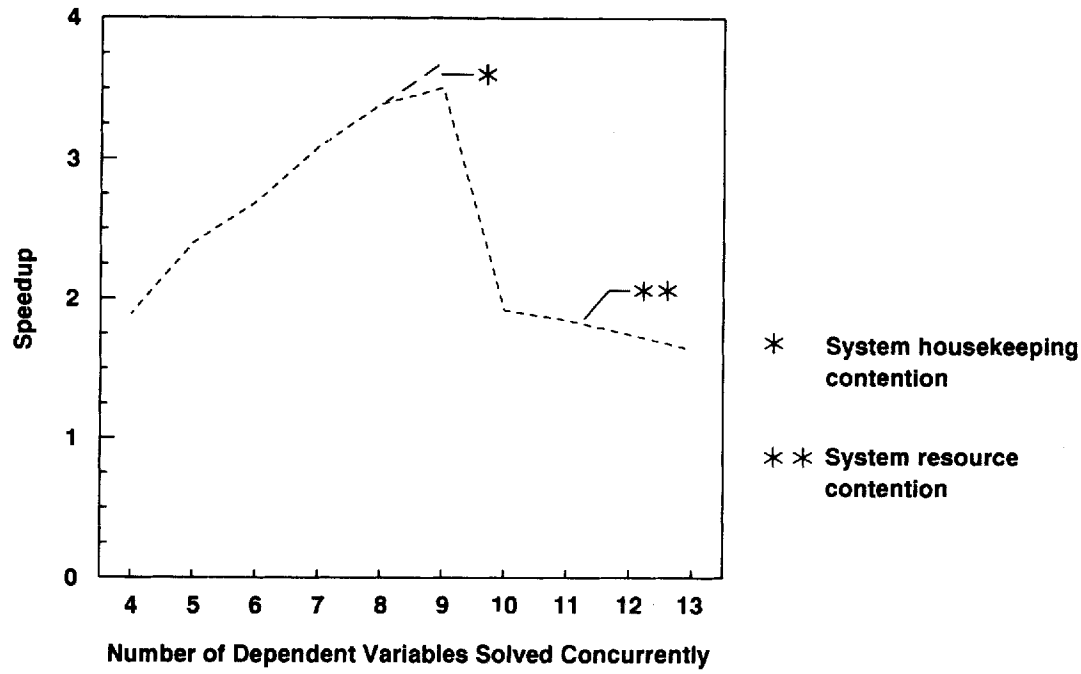

Figure 10. Simulated speed-up for strategy II as a function of the number of dependent variables solved concurrently

number of CPUs, even if only by one, it could be expected that the run-time would essentially double.

To test these hypotheses, variants of the test problem were run in which the energy equation was solved a number of times within each iteration. In effect, this attempts to simulate problems which require the solution of a number of dependent scalar variables. Note that this simulation ignores possible degradation in convergence rates arising from any possible coupling between the scalar transport equations. This simulation of increasing the number of scalar transport equations does, however, provide an estimate of the maximum speed-up which would be obtained using a functional decomposition strategy on problems governed by a large number of transport equations. The results of running these simulations are shown in Figure 10.

From these results it can be seen that as the number of dependent variables solved increases, there is an approximately linear increase in the speed-up relative to the projected sequential run-time. This is to be expected as each additional transport equation is assigned to an idle CPU. When the number of variables reaches the number of CPUs available (eight in the case of the SGI280S), there is a loss of speed-up due to contention with the system clock and accounting functions allocated to one of the CPUs.

When the number of variables exceeds the number of CPUs, as expected, there is an approximate halving of the speed-up obtained. However, as the number of variables is increased further, the speed-up continues to decrease. This was thought to be due to the simplistic code structure used initially, which allowed all the execution threads spawned, one for each variable, to contend for the available CPUs simultaneously. An explicit scheme for handling this situation, using two parallel sections one after the other on a lesser number of CPUs was implemented, to assist in achieving greater speed-ups and increased efficiency. The rearranged code structure is shown in Figure 11.

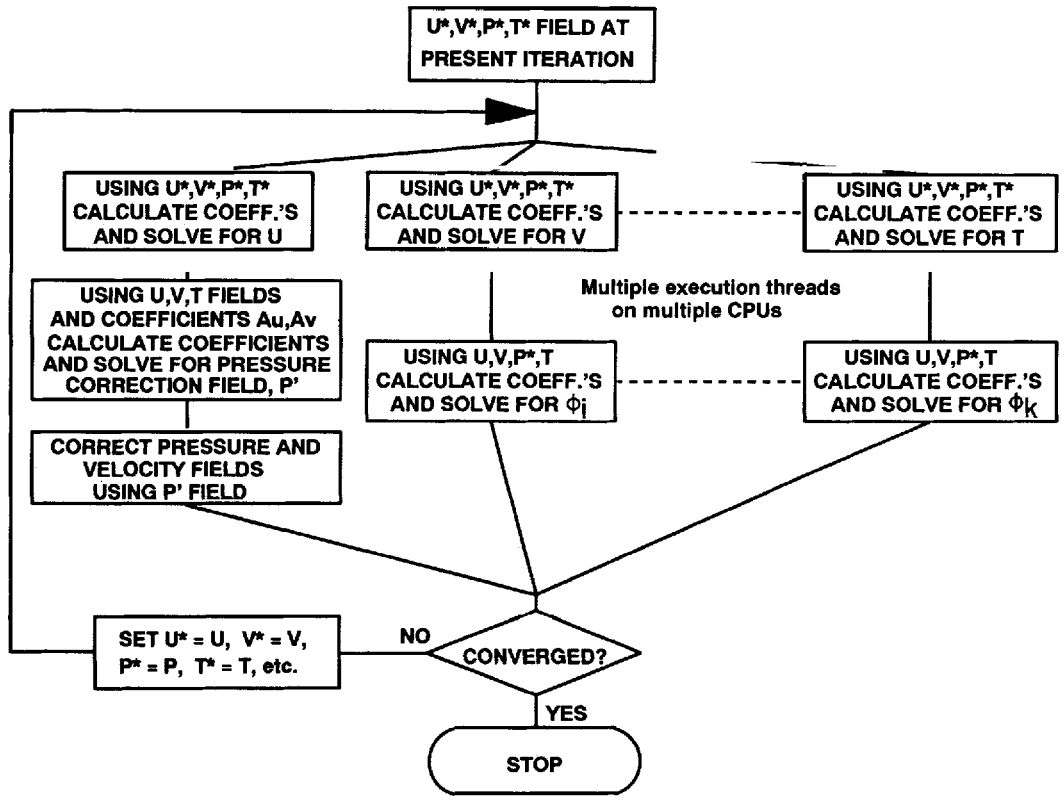

Figure 11. Code structure—two parallel sections, multiple variables

A side-effect of this approach is that pressure correction, which must be performed after the u and v velocity calculations, can now be solved in the second parallel section. In effect, the additional transport equations are solved in parallel with the already sequentially solved pressure correction equation and, so, there is no significant impact on run-time from their inclusion. Reduced overheads from resource contention compensated for any impact there may have been. When the number of dependent variables exceeded 16, a third parallel section had to be added and a drop in speed-up was observed.

6.5. Comparison of functional decomposition and loop-level parallelization results

Figure 12 gives the run-times for each method for various numbers of transport equations. While the increase in run-time for the sequential and fine-grain parallel methods are basically linear in their increase, it may be noted that the coarse-grain method is starting the expected stepped increase.

Figure 13 shows the corresponding speed-ups for the coarse and fine-grain parallelization methods. The staircase increase in run-time of the functional decomposition method seen at 17 transport equations in Figure 12 gives rise to a sawtooth pattern of speed-up, which can be seen beginning in Figure 13. Note that the fine-grain loop-level parallelization method is better for problems involving only a few transport equations. As the number of transport equations becomes greater, the coarse-grain functional decomposition method shows less increase in

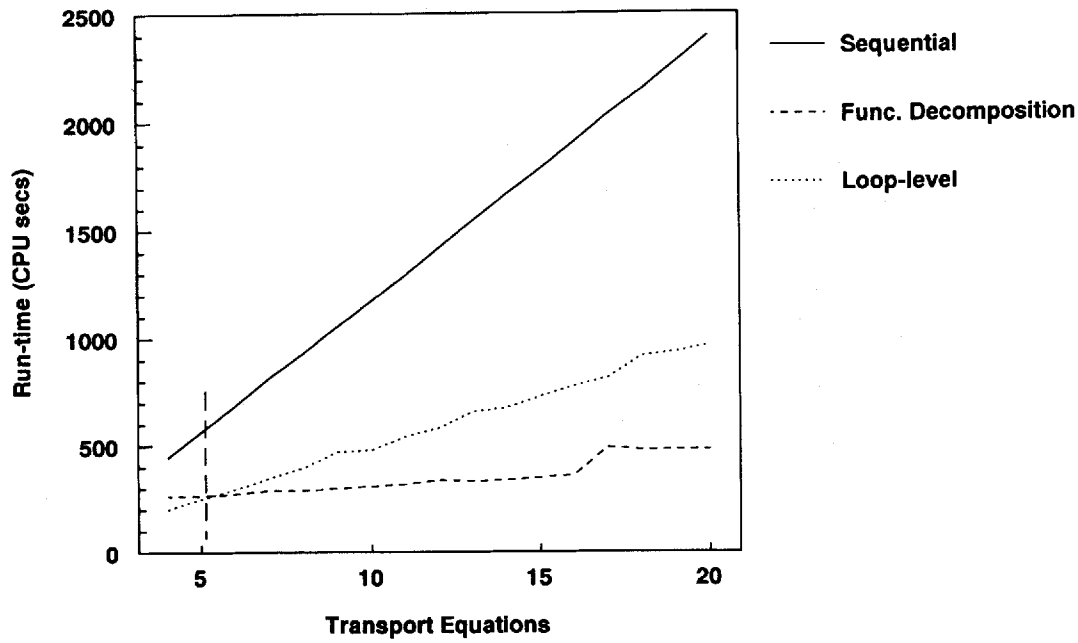

Figure 12. Run-times of functional decomposition and loop-level parallelization methods

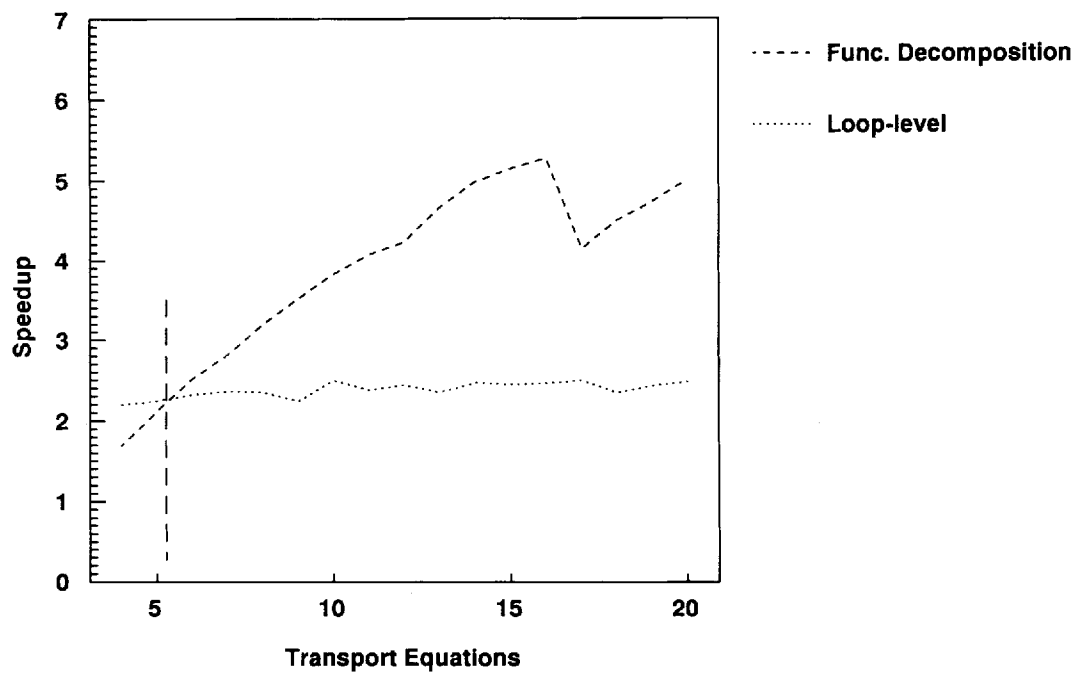

Figure 13. Relative speed-up of functional decomposition and loop-level parallelization methods

run-time than the fine-grain method. The crossover in speed-up between the two methods for the current implementations lies at about the five-transport-equation case. The position of this crossover might be expected to be related to problem size as the functional decomposition implementation suffers an effective message passing overhead in maintaining data integrity in memory.

For the test problem, the maximum achievable speed-up given by Amdahl's law for the fine-grain parallel code was 4·74. So, the measured speed-up of 2·2 is only 46% of the theoretical maximum. This result would appear to indicate problems with the application of fine-grain parallelization to the particular test problem considered; so, care must be taken in drawing comparisons of the two parallelization methods. The loop parallelization is implemented as a single-master and multiple-slave execution threads. All execution threads perform part of the loop iterations but the master thread performs in addition serial portions of the code and certain parallelization scheduling. While the master thread is occupied with these latter tasks, the slave threads enter busy-wait states. The maximum variation in execution time between slave threads is 1·7%, which would indicate that load balancing is not a significant factor in the poor parallel performance obtained. A decrease in the rate of convergence due to the less efficient solution algorithm adopted also affected speed-up. However, only about 5% more iterations were required to reach convergence, relative to the original algorithm. It would asppear that the dominant factor reducing speed-up in the problem analysed is the ratio of multiprocessing system overhead to user code in parallelized loops.

As can be seen, the effect of increasing the number of transport equations in the fine-grain method was quite different from the response observed for the functional decomposition method. The speed-up measured increased slightly, as could be expected due to the incrementally greater portion of code in parallel sections. The remaining serial code sections, comprising such tasks as unsteady heat balance calculation and convergence testing, are essentially independent of the number of transport equations. Thus, these serial sections will decrease as a proportion of execution time as more transport equations are added to the parallel-code sections. The percentage of the theoretical maximum speed-up achieved remained virtually constant. Figure 14 shows the relative parallel efficiencies of the coarse- and fine-grain methods.

The dips in efficiency noticeable for the odd-number cases are due to one-child process spin-waiting for the entire period of the second parallel section, there being no work for it to do. For example, in the case involving the solution of 11 dependent variables concurrently, six CPUs are used. First six variables are solved and then five, allowing one CPU to idle. Similarly, for 13 variables, seven CPUs are used, first solving seven variables and then solving the remaining six with one idle CPU consuming time in spin-wait (see Figure 15).

The general trend down in the efficiency of the functional decomposition code for a given number of parallel sections is due to the increasing overheads in managing concurrent processes, particularly, memory management. In order to maintain independent data partitions, data arrays must be shared under global locks and significant overheads are generated by loading of local copies of data to individual child processes.

The method of multiple parallel sections could be extended to cases of fewer dependent variables in order to use less processors. As this, in effect, makes the code more serial, there is a trade-off between parallel efficiency and speed-up obtainable.

6.6. Experimentally determined serial fraction

To investigate further the relative performance of the different approaches, the serial-code fraction was experimentally derived, after the method of Karp and Flatt.[15] This metric, a measure

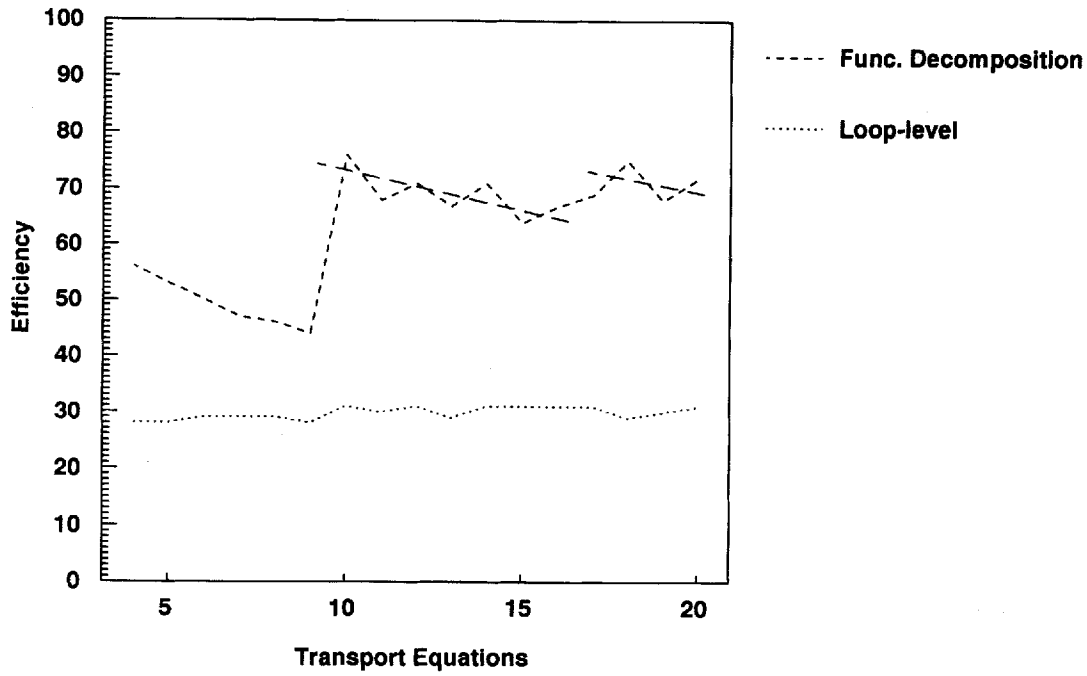

Figure 14. Comparison of parallel efficiencies for functional decomposition and loop-level parallelization methods

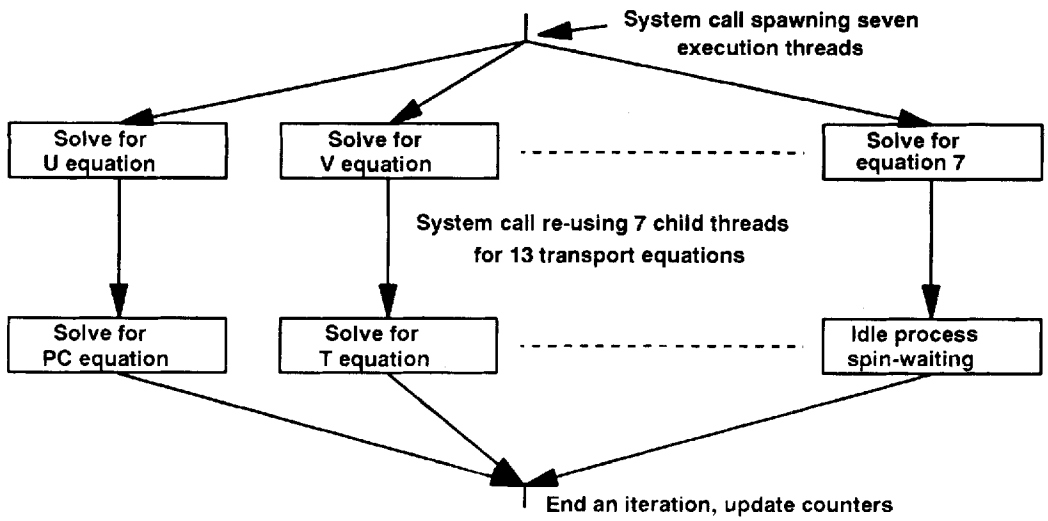

Figure 15. Process spin-waiting for odd numbers of dependent variables with two parallel sections

of the effective serial portion of the code, is shown in Figure 16. Since the effective serial fraction will limit the speed-up obtained, a lower value is better. From the formulation for the serial fraction, f,

$$f=\frac{1/s-1/n}{1-1/n}, \tag{15}$$

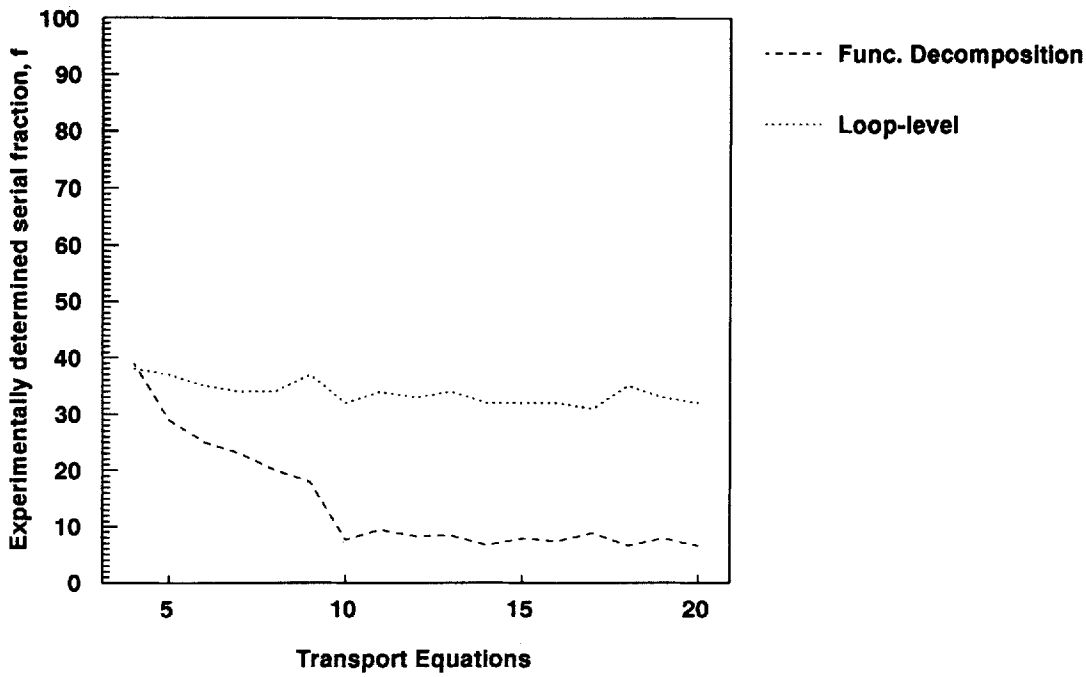

Figure 16. Comparison of experimentally derived serial fraction for functional decomposition and loop-level parallelization methods

where s is the experimentally obtained speed-up and n the number of processors, it can be seen that this quantity is closely related to the percentage of the theoretical maximum speed-up already mentioned. It is, however, much easier to formulate, being derived entirely from experimentally obtained quantities, requiring no code analysis. The same features already discussed can be seen, in particular, the load-balancing peaks; in addition, the extent of the advantage gained by moving to double parallel sections is clearly visible. Care must be taken in interpretation of these results, since the sensitivity of this metric also means statistical variations in run-time, due to changing system overheads, load or cache usage, can be of the same order as those variations attributable to load-balancing effects.

6.7. *Effect of problem grid size and storage array size*

All of the cases described were performed with a single computational mesh. The effect of increasing the number of control volumes was investigated very briefly. It was found that solving for more grid points had a greater effect on the coarse-grain method than the fine-grain method, with the result that the crossover point between the two methods seen in Figure 12 moved progressively to higher numbers of transport equations.

The interpretation of this effect was complicated by the fact that due to the explicit memory management techniques required by the functional decomposition method, it suffered from a considerable overhead in handling larger arrays. Given a particular problem size, storage in larger arrays had little or no effect on the serial or fine-grain codes but increased the run-time of

the coarse-grain method significantly. This problem could be ameliorated by an alternative approach to maintaining data integrity.

6.8. *Memory usage*

All the parallelization strategies used more memory than the serial code, trading memory usage against processing speed. For the base test case, the serial code used approximately 400 kB, the functional decomposition code running on three processors used 2·6 MB, and the loop-level parallelized code running on eight processors used 6·4 MB. For the parallelized code, memory usage increased roughly as the number of code segments duplicated across processors. For problems of this size, it would appear that functional decomposition is the most memory-efficient method of increasing processor speed.

7. CONCLUDING REMARKS

It should be noted that all parallelization strategies will reflect problem dependencies. However, it has been shown that functional decomposition offers an easily implementable and effective alternative to code parallelization for certain classes of problems of industrial interest. The method is particularly effective for problems governed by a relatively large number of transport equations and its efficiency depends on matching machine and problem parallelism. Ease of implementation in the existing codes and maintenance of the inherent structure of the traditional serial algorithms are seen as the major advantages of this approach. In comparison with a fine-grain parallelization method, the coarse-grain functional decomposition method readily gave better speed-ups and efficiencies for cases with more than five independent transport equations, for similar or less programming effort.

Further work is required to assess the effects of varying machine configuration and convergence rate degradation in complex problems in which coupling actually exists between many or all of the dependent variables. In this regard, the application of both loop level and functional decomposition strategies are presently being evaluated for a blast furnace model. A number of computer platforms, including both shared and distributed memory machines, are being studied. The primary aim of this work is to achieve rapid execution of the model on low-cost computer platforms to allow widespread on-site implementation of the model in ironmaking operations, Parallelization studies on other models, including three-dimensional casting models, are also being undertaken.

APPENDIX: NOMENCLATURE

a	coefficients in the discretized energy equation
A	porosity function for the momentum equations (see Reference 10)
b	source term in the discretized energy equation
c	specific heat
f	experimentally determined serial fraction of the code
F_s	serial code fraction
F_p	parallel code fraction
g	acceleration due to gravity
k	thermal conductivity
n	number of processors used

P	effective pressure
P'	pressure correction
s	experimentally obtained code execution speed-up
S	source term for governing transport equation
S_T	source term for energy equation in terms of temperature
S_u	source term for u momentum equation
S_v	source term for v momentum equation
t	time
t_p	parallel run-time
t_s	sequential run-time
T	temperature
$\mathbf{u}$	velocity vector
u, v	velocity components in x and y directions
x, y	co-ordinates
X	length of rectangular cavity
Y	height of rectangular cavity
β	volumetric coefficient of thermal expansion
ΔH	nodal latent heat
Γ	diffusion coefficient for governing transport equation
μ	dynamic viscosity
ρ	density
ϕ	generalized dependent variable

Subscripts

nb	neighbouring node points
n, s, e, w	co-ordinate directed neighbouring node points
i	ith node point or control volume
ref	reference conditions

Superscript

*	values at previous iteration

REFERENCES

1. M. E. Braaten, 'Solution of viscous fluid flows on a distributed memory concurrent computer', *Int. j. numer. methods fluids*, **10**, 889–905 (1990).
2. W. B. U. Tanzil, D. G. Mellor and J. M. Burgess, 'Application of a two-dimensional flow, heat transfer and chemical reaction model for process guidance and gas distribution control on port Kembla no. 5 blast furnace', *Proc. 6th Internat. Iron and Steel Congress*, ISIJ, Nagoya, 1990.
3. J. S. Truelove, 'Fluid dynamics in strip-casting liquid-delivery systems', *RNT/CP/R/90/005*, BHP Central Research Laboratories, Shortland, 1990.
4. J. S. Truelove, 'Fluid dynamics, heat transfer, and solidification in strip-casting liquid-delivery systems', *RNT/CP/R/90/006*, BHP Central Research Laboratories, Shortland, 1990.
5. P. J. Flint, 'A three-dimensional finite difference model of heat transfer, fluid flow and solidification in the continuous slab caster', *ISS 73rd Steelmaking Conf. Proc.*, Detroit, Michigan, 1990.
6. R. W. Hockney and C. R. Jesshope, *Parallel Computers—Architecture, Programming and Algorithms*, Adam Hilger, Bristol, 1988.
7. S. V. Patankar, *Numerical Heat Transfer and Fluid Flow*, Hemisphere, New York, 1980.
8. B. P. Leonard, 'A stable and accurate convective modelling procedure based on quadratic upstream interpolation', *Comput. methods appl. mech. eng.*, **19**, 59–98 (1979).
9. P. H. Gaskell and A. K. C. Lau, 'Curvature-compensated convective transport: SMART, A new boundedness-preserving transport algorithm', *Int. j. numer. methods fluids*, **8**, 617–641 (1988).

10. A. D. Brent, V. R. Voller and K. J. Reid, 'The enthalpy–porosity approach for modelling convection–diffusion phase change', *Numer. Heat Transfer*, **13**, 297–318 (1988).
11. D. W. Peaceman and H. H. Rachford, 'The numerical solution of parabolic and elliptic differential equations', *J. Soc. Ind. Appl. Math.*, **3**, 28 (1955).
12. A. Settari and K. Aziz, 'A generalization of the additive correction methods for the iterative solution of matrix equations', *SIAM J. Numer. Anal.*, **10**, 506–520 (1973).
13. G. M. Amdahl, 'Validity of the single processor approach to achieving large scale computer capabilities', *Proc. AFIPS Spring Joint Computer Conf.*, Vol. 30, Atlantic City, NJ, 1967.
14. S. G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, NJ, 1989.
15. A. H. Karp and H. P. Flatt, 'Measuring parallel processor performance', *Comm. ACM*, **33**, 539–543 (1990).