A THESIS

entitled

THE SIMULATION OF FLUID FLOW PROCESSES USING VECTOR PROCESSORS

Submitted in partial fulfilment of the
requirements for the award of the

DEGREE OF DOCTOR OF PHILOSOPHY

of the

COUNCIL FOR NATIONAL ACADEMIC AWARDS

by

CONSTANTINOS SAVVAS IEROTHEOU
BSc, GIMA

Faculty of Technology
Centre for Numerical Modelling and Process Analysis
School of Mathematics, Statistics and Computing
Thames Polytechnic
LONDON

MAY 1990

To my family

The simulation of fluid flow processes using vector procesors

by

Constantinos Savvas Ierotheou

Abstract

In this thesis the potential gains in vectorisation of linear and non-linear systems of equations are investigated. Previous studies carried out on the suitability of algorithms for vectorisation have been based on the solution of Poisson's equation. In accordance with this, a range of algorithms are explored and compared using a VA-1 pipeline processor attached to a MASSCOMP MC5400. Analysis shows that almost full vectorisation is possible leading to speed-up factors of up to 90. Based on these results the vectorised conjugate gradient with a Jacobi preconditioner (JCGV) is the best of the algorithms considered.

This work is extended to the development of a two-dimensional fluid flow code which is used to solve the Navier-Stokes equations, SIMPLE is implemented to handle the non-linear nature of the equations. The first two problems are isothermal flows, viz, the 'moving lid cavity' and the 'sudden expansion in a duct' problem. A study of where the greatest computational effort is expended, and subsequent vectorisation leads to 98% of SIMPLE being modified. This results in speed-up factors of 6 for the cavity problem and 29 for the sudden expansion problem. In both problems the JCGV is marginally faster than the vectorised Jacobi with under-relaxation (JURV). However, the JCGV algorithm is not robust and it is necessary to relax carefully the approximation, otherwise high computation times or divergence is likely.

Two further problems are considered each with increasing complexity, these include scalar quantities of temperature and characteristics of k-ε turbulence. One problem is based on 'turbulent L-shaped flow in a duct' and the other on the 'natural convection in a square cavity'. A consequence of the higher scalar computation gives speed-up factors of 5 for the turbulent L-shaped flow and 11 for the natural convection problem. There is little to choose between the JCGV and JURV algorithms, however, the robustness problems with the JCGV algorithm remain.

A multigrid method (ACM) is used to improve the convergence rate of the algorithms, particularly as the size of problem is increased. Although it is more effective in scalar, it also provides worthwhile improvements for the vectorised algorithms with overall factors of 8.5. Convergence difficulties with the JCG algorithm also prevents the combination with the ACM method. Therefore, the vectorised JUR algorithm with the ACM method is not only more efficient and reliable, but also has scope for improvement as the grid is increased.

The potential gains in vectorisation of the SIMPLE family on pipeline architectures have been clearly demonstrated and indicate that such efforts on practical CFD codes should be well rewarded with regard to processor performance.

Acknowledgements

I would like to express my gratitude to Professor Mark Cross and to Dr Will Richards for their invaluable advice and encouragement over the last three years, particularly during the difficult phases of the work.

I would also like to take this opportunity to thank the staff at the School of Mathematics, Statistics and Computing, and to the postgraduates at the Centre for Numerical Modelling and Process Analysis of Thames Polytechnic. They contributed to many useful discussions and provided a stimulating working environment. In addition, I would like to acknowledge Mrs Berol Cooper and Mrs Irene Wilmot for providing me with adequate computer resources.

My sincerest thanks go to my parents, my sister Helen, and my brothers Andrew, George and Nicos who have supported and encouraged me throughout my education. Thanks too, to all my personal friends, particularly Hazel and Tony.

Finally, the financial support provided by the Science and Engineering Research Council is gratefully acknowledged.

Contents

CHAPTER ONE

1.0 INTRODUCTION

1.1 Overview of CFD

In recent years the field of Computational Fluid Dynamics (CFD) has evolved at a phenomenal rate. CFD has grown to such an extent that today it is used as a design tool which is capable of predicting complex flows in situations where experimentation is not feasible or too costly, or both. Currently, CFD simulations and experiments are both used as a means for investigating engineering applications. However, it may not be long before numerical simulation is considered more important than experimentation in many areas. The role of the experiment may be limited to the validation and necessary refining of CFD models and computation procedures. CFD simulations are certainly more informative and can cover a range of different complex fluid flow simulations many of which can not be performed experimentally, this makes CFD simulations essential.

Consider the spread of smoke and fire in an underground station such as the King's Cross incident. Simulations of this type are extremely important. An attempt to carry out experiments for such a problem with different scenarios is extremely difficult. Even for a single numerical fire simulation this can be a very demanding computational task. The emergence of supercomputer architectures such as the CRAY family, CYBER 205 and IBM 3090 (Hockney and Jesshope [1988]) which can compute at very high speeds, coupled with the advances in numerical techniques and solution procedures, make such simulations possible. Indeed, CFD simulations relating to the King's Cross fire (Fennell [1988]) have been performed at Harwell using their own three-dimensional CFD code called HARWELL-FLOW3D (Jones et al [1985]).

Traditionally, CFD simulations have been computationally very expensive and although complex problems could be tackled, the accuracy of the solution or the resolution of the grid was not as high as the engineer would have liked. However, the introduction of pipeline vector processors as an alternative to the conventional scalar processors has begun to overcome these past difficulties. Today, many large and complex flow problems can be modelled using general purpose CFD codes such as HARWELL-FLOW3D (Jones et al [1985], Burns et al [1986]) and PHOENICS (Rosten and Spalding [1986]). In addition, the number of computation nodes which can be solved in a reasonable time is now approaching the order of hundreds of thousands. The introduction of these new architectures has also assisted in advancing several branches of CFD to such an extent that many have become research topics in their own right.

One branch which has become very fruitful is the refinement and modification to existing solution procedures which are used to solve the governing differential equations. The problem with solving the equations numerically lies in the fact that the equations are often coupled and that the pressure field (which drives the flow) is not known a priori. The use of a stream function - vorticity formulation will overcome the latter problem since the pressure is explicitly eliminated, however, this approach is currently restricted to flow problems where the pressure field is not dominant. A more common practice is to adopt a primitive variable approach. Here the velocity components and pressure (pressure-correction) equations are obtained from their governing equations. The SIMPLE solution procedure (Patankar and Spalding [1972]) and its derivatives are probably the most widely used within the CFD community and forms the basis of many commercial software packages.

Turbulence modelling is also an area of intense research. There are currently two main schools of thought for resolving the presence of turbulence in engineering. The first is based on large eddy simulation (Riley and Metcalfe [1980]) and involves the solution of the full Navier-Stokes equations. Even with the computer power currently available, the expected computation times needed to solve very simple problems are still too high. The second approach focuses on the solution of the time-averaged Navier-Stokes equations together with transport equations to model key characteristics of turbulence. Research on this approach has been more successful and continues to be popular particularly amongst engineers. Launder et al [1974, 1975] were amongst the first to adopt such an approach, and although the k-ε model is very popular, there is to date no general turbulence model.

The numerical representation of the convection term present in the governing equations has been of interest for many years, especially in flow problems dominated by the convection process. This has led to a number of different schemes, each attempting to correctly describe the convection process. The hybrid scheme (Spalding [1972]) switches between central and upwind differencing. The Quadratic Upstream Interpolation for Convective Kinematics (QUICK) scheme due to Leonard [1979] is more accurate at low grid Peclet numbers but at the expense of an increase in the computation time. This is evident in some turbulent flow simulations (Han, Humphrey and Launder [1981]). The Curvature Compensated Convective Transport (CCCT) scheme will guarantee the boundedness condition and can be used to derive all the schemes above (Gaskell and Lau [1988]). The Corner UPwInDing (CUPID) scheme (Patel, Markatos and Cross [1988]) copes particularly well with the problem of false-diffusion, again at the expense of some increase in computation time.

The way in which the computational domain is discretised will lead to either structured or unstructured grids. A finite-difference approach has been successfully used in the past, but more recently the control-volume approach has increased in popularity to such an extent that it now exists as a serious competitor to the finite-difference approach. Both of these methods have been applied extensively to structured grids and less so to body fitted grids. The finite-element method on the other hand is ideal for complex geometries but lacks the simplicity of the control-volume approach. Recently, work has been done on the use of a control-volume based finite-element method (Prakash and Patankar [1985], Lonesdale and Webster [1989]) and this could be a future trend. The control-volume approach has been adopted in this research because all the examples have straightforward rectangular geometries.

1.2 Literature survey

The ability to perform large scale simulations particularly in CFD would have been near unthinkable fifteen years ago. A select few had access to supercomputers, the most successful machines being the CRAY-1 (Russell [1987]) and a derivative of the original CDC STAR-100 machine called the CYBER 205 (Kascic [1979]). These machines were significantly faster than any other machines available at that time. The spectacular improvements in computer speed were achieved as a direct result of combining the technological advances in hardware with the introduction of a higher level of concurrency or parallelism in the architecture. By the early eighties the CRAY and CDC machines had become world leaders and had allowed CFD practitioners to become more adventurous. This in turn stimulated other computer manufacturers to market their own vector and parallel based machines, these include the IBM 3090, AMT DAP, FACOM VP-100 and VP-200, the NEC SX-1 and SX-2 and the Sequent balance 8000 and 21000 machines. Not surprisingly, a vast amount of literature has appeared in the last decade on the solution of practical engineering problems using supercomputers. This has also led to new journals dedicated entirely to the computer science of vector and parallel processing the most notable being '*Parallel Computing*'.

In the past, a large number of the publications have been based on work carried out on CRAY-based machines and a smaller proportion on CYBER 205 machines. Although some of these machines can be used to perform both vector and coarse-grain parallelism operations, attention is primarily focused on the use of a single pipeline vector processor.

A number of different questions need to be answered about the use of vector processing in the solution of CFD problems. For example, how fast can a CFD code run on a given vector processing architecture? How much faster (or slower) is the vectorised execution compared to the execution of the equivalent scalar code? and how much improvement in speed can one ever hope to achieve using a particular vector processor, given the characteristics of a typical CFD code? The answers to these questions will help to reveal and characterise different aspects of vector processing and vector processors.

1.2.1 Vectorised tridiagonal algorithms

In the past much attention has been given to the solution of a system of equations since it has become apparent that this constitutes a high proportion of the total computation time. Lambiotte and Voigt [1975] consider the solution of a tridiagonal system of *nxn* equations using a number of direct and iterative algorithms. One of the direct algorithms considered is the Gaussian elimination algorithm with LU factorisation. For the purposes of vectorisation the implicit steps are replaced by explicit steps. When coded on a CDC STAR-100 vector machine the modified Gaussian elimination algorithm (using the vector hardware instructions) is more efficient than the conventional scalar algorithm for matrix systems $n>13$. The vectorised algorithm of Stone [1973] was implemented and found to be slower than the vectorised Gaussian elimination algorithm. Lambiotte and Voigt [1975] also consider the vectorised cyclic reduction algorithm (Hockney [1965]), their study reveals that the cyclic reduction algorithm is up to seven times faster than the Gaussian elimination algorithm for large matrix systems $n>125$. As well as direct algorithms, iterative algorithms such as the Jacobi, red-black SOR and a Traub factorisation [1973] are also studied by Lambiotte and Voigt [1975]. Results are presented for the solution of the tridiagonal system of equations $A\mathbf{x}=\mathbf{r}$ where the ith row of A is given by $(0,...,0,b,1,b,0,...,0)$, $\mathbf{r}=(1,...,1)^T$ and b is varied to change the diagonal dominance of the matrix system. The settings are those used by Traub [1973] where b=0.24, 0.4 and 0.49 for the cases where n=100 and 1000. The red-black SOR algorithm is the most efficient iterative algorithm, but overall, the cyclic reduction algorithm is found to be the best of all the algorithms for the problem considered on the CDC STAR-100 machine.

Masden and Rodrigue [1976] carried out a similar investigation to that of Lambiotte and Voigt [1975] based on the solution of a tridiagonal matrix system. They restricted their study to direct solvers only and compared the performances of the vectorised Gaussian elimination algorithm, Jordan's algorithm [1974] and the cyclic reduction algorithm (Hockney [1965]). The calculations were also performed on a CDC STAR-100 machine and therefore similar conclusions were obtained to those of Lambiotte and Voigt [1975]. Masden and Rodrigue then proceeded to define a hybrid 'Super-STAR-Algorithm' which takes advantage of the fact that at an advanced stage of the cyclic reduction process the computation becomes inefficient on the CDC STAR-100. Instead, the process switches to a more efficient low-order tridiagonal solver such as the vectorised Gaussian elimination algorithm. The super-STAR-Algorithm was faster than the Gaussian elimination algorithm (implemented on a CDC 7600 scalar machine) for n>750.

Swarztrauber [1979] considers the vectorised implementation of Cramer's rule for the solution of a tridiagonal system of equations. The performance of the algorithm was compared to the Gaussian elimination algorithm with partial pivoting. On a CDC 7600 scalar machine the Gaussian elimination algorithm is faster, but despite having a higher operation count than the cyclic reduction algorithm, the vectorised Cramer's rule is faster than the Gaussian elimination on the CRAY-1. This is purely because the Gaussian elimination algorithm is vectorised to a lesser degree.

1.2.2 Vectorised algorithms for large sparse systems of equations

The early eighties saw some of the first computations performed on practical CFD problems. Spradley et al [1981] presented a General Interpolants Method (GIM) to analyse complex three-dimensional flow fields described by the inviscid Euler equations as well as the time-averaged Navier-Stokes Equations. The code combined the techniques of finite-element (for the geometry definition) with finite-difference (to solve the resulting equations). The solution of the equations were obtained using a MacCormack predictor-corrector type scheme and was found to be the most time consuming of all the modules. By re-ordering the index over which the calculations were performed the solver was adapted for use on the CDC STAR-100. A number of different problems were considered and a sixfold improvement in speed was achieved over the same code on a CDC 7600 scalar machine. When a pipeline CYBER 203 was used a further improvement of two was achieved.

Kordulla [1984] also reported on the vectorisation of a MacCormack based CFD code for the CRAY-1S machine. The problem studied was flow past a hemisphere-cylinder configuration at a 5° angle of attack and a Reynolds number of 212,500 referenced with the radius of the sphere. The computational grids used were 31x20x31 and 42x20x31. Since the vectorisation of the explicit steps were straightforward the emphasis was on the vectorisation of the implicit steps in the predictor-corrector scheme. The results indicated that the computation times on the IBM 3081K were about eight times slower than on the CRAY-1S (scalar processor). When the CRAY auto-vectoriser was switched on the ratio increased to 10 and for the manually vectorised code the ratio was further increased to 31. Although the CRAY vectorising compiler has improved considerably since then, this example helps to illustrate the limitations in relying on a vectorising compiler. There is clearly a need for user-interaction.

Borrel et al [1985] also used the MacCormack scheme to simulate three-dimensional flow past a wing. The solution is determined using the Euler equations to obtain pressure and velocity components. A similar vectorisation approach was taken to Kordulla [1984] where loop indices are re-ordered and data dependencies are suppressed. A 51x50x19 computation grid was used and the vectorisation on a CRAY-1S resulted in a fourfold increase in speed. The problem of simulating an interacting jet with supersonic flow was also carried out on a 40x30x35 grid and similar improvements in speed were obtained.

Koppenol [1985] demonstrated the large reductions in CPU time which can be achieved when taking software written for the CYBER 180/855 scalar machine and porting it onto a CYBER 205. Impressive reductions in speed are quoted for the solution of a two-dimensional fluid flow problem. Although the results appear biased they do at least give a practical estimation of the difference between the machines.

Schwamborn [1984] used a finite-difference formulation to solve a laminar three-dimensional boundary-layer on the surface of a wing-like spheroid. The discretisation resulted in the solution of a set of block tridiagonal matrix systems. A hotspot analysis reveals that the execution of two routines is responsible for up to 98% of the total computation time, therefore, the effort in vectorisation is concentrated here. At best a 40% improvement was achieved because most of the computations being performed were inherently scalar. This demonstrates the need to carefully reconsider the sequence of computation steps and if possible, to re-structure them to good effect. It may be necessary to use a different solver which has a higher level of vectorisation. Schwamborn states that "the only way to write a three-dimensional, boundary-layer code with high vectorisation is to use a difference scheme using only data in one plane". One assumes that this was with reference to a CRAY-based architecture, but it is unlikely that this will apply to all pipeline machines. Other architectures such as the CYBER 205 would work most efficiently with long vectors. Moreover, it is the authors belief that an explicit whole-field solver would be better suited for a three-dimensional problem since the vector operations would be of maximum possible length.

On some three-dimensional problems it becomes impractical to carry out the simulation using a scalar processor, instead results are presented for the vectorised computation only (Rizzi and Therre [1985]). This approach to presenting results is informative to an engineer since it becomes possible to determine how quickly a problem can be solved.

Thus far the numerical algorithms have been restricted to the solution of tridiagonal systems. Much work has been done on the solution of a large sparse matrix system, this system is not necessarily tridiagonal and is often encountered when using a discretisation scheme to represent the domain of interest. The growth and popularity of the pipeline vector processor as an architecture to solve computer intensive CFD problems can be partly attributed to the availability of explicit numerical algorithms which are readily vectorised. Examples of such algorithms are the JUR and conjugate gradient (Hestenes and Stiefel [1952]) algorithms. A large number of the results quoted for the use of such algorithms have been based on the solution of the Poisson equation. The discretisation of the Poisson equation using a five point finite-difference technique results in a linear system of equations, these make up a sparse pentadiagonal matrix (A) in two dimensions.

The conjugate gradient algorithm is based mainly on matrix-vector and vector-vector operations and is therefore ideal for vector processing. However, the algorithm has been reported to be slow in some cases (Concus, Golub and O'leary

[1975]). To overcome this problem a preconditioning matrix (P) is introduced into the formulation, the purpose of the preconditioning matrix is to lower the condition number of the original matrix and the right-hand-side vector **b**, hence the matrix system becomes

$$\mathrm{P}^{-1}\mathrm{A}\mathbf{x} = \mathrm{P}^{-1}\mathbf{b}$$

The choice of the matrix P raises interesting points, for example, will it destroy the structure and other desirable properties present in the original matrix? Will it be detrimental to the convergence rate of the original conjugate gradient method? How expensive is the generation of the matrix P relative to the total computation time and will the matrix formulation for P be such that efficient vectorisation is possible? It is found that the solution of a tridiagonal matrix system (an intermediate step in the preconditioned algorithm) poses some problems when attempting to vectorise this step. Various approaches have been taken to overcome this problem. One suggestion is the use of the cyclic reduction algorithm (Rodrigue and Wolitzer [1981] and Jordan [1981]). Alternatively, any other tridiagonal solver presented thus far could be used.

Dubois, Greenbaum and Rodrigue [1979] suggest the use of a truncated Neumann expansion to represent the inverse of the original matrix as the preconditioning matrix. Despite full vectorisation there was a significant increase in the number of conjugate gradient iterations.

Van der Vorst [1982] suggested a truncated Incomplete Cholesky Conjugate Gradient algorithm (ICCG) where the inverse of the matrix (1-E) is given by

$$(1-E)^{-1} \approx (1-E^2)(1+E)$$

In the two cases studied, the truncated ICCG algorithm was more efficient than the truncated Neumann expansion. Also, the increase in the number of iterations was minimal and as a result the vectorised version of the truncated ICCG was competitive with the scalar ICCG algorithm. However, in a later study van der Vorst [1986] showed that for some problems the number of iterations can increase significantly to make the vectorised ICCG algorithm less competitive. The improvements in using the vectorised truncated ICCG over the scalar ICCG algorithm on a CRAY-1 and CRAY X-MP were up to 50%, with over a twofold increase on the CYBER 205.

The simplest preconditioning matrix is the Jacobi or diagonal preconditioner (JCG). Radicati and Vitaletti [1987] compare the solution times of the JCG and the ICCG algorithms on an IBM 3090-VF machine. The problem was a three-dimensional elliptic partial differential equation with mixed boundary conditions and is solved on a 40^3 grid. In the case of the ICCG algorithm the solution of the intermediate matrix system is solved once and stored. Although this results in a higher cost per iteration this is offset by the reduced number of overall conjugate gradient iterations. In each case comparisons were made between the vectorised and scalar ICCG and JCG algorithms. The compressed diagonal storage method was used since this produces vector lengths of order 40^3. In scalar mode the ICCG is superior to the JCG algorithm but the opposite is true in vector mode. This is mainly due to the essentially scalar computations which are used to solve the intermediate matrix system. Despite this, speed-up factors of 2 are reported in favour of the vector ICCG and up to 6 in favour of the vector JCG algorithm.

Block preconditioning can also be used as part of the conjugate gradient algorithm (Meurant [1984] and Concus, Golub and Meurant [1985]). A vectorised Cholesky decomposition is used as a block preconditioner to solve three test problems, the inner products were coded in CAL (Cray Assembler Language). The ICCG algorithm is implemented for comparison, and computations were performed on the CRAY-1S and CRAY X-MP machines. The best improvements were obtained using the block preconditioning algorithm rather than the ICCG, but the times recorded did not include the time to generate the preconditioning matrix. Furthermore, in one test case the approximation of the inverse was poor enough to cause a severe degradation in the performance of the vectorised algorithms.

Kightley and Jones [1985] consider the solution of large three-dimensional turbulent flow simulations using SIMPLE. The emphasis is on the solution of the pressure-correction equation which is solved using the conjugate gradient algorithm with various preconditioners. These preconditioners include the Jacobi, standard incomplete Cholesky, truncated incomplete Cholesky (van der Vorst [1982]) and a block factorisation. In the solution of the 'trivial' Poisson equation the elaborate preconditioners are not worth the extra expense and the JCG algorithm is considered to be the best. However, as the complexity of the problem increases the ICCG algorithm is the most efficient even though the block preconditioner is more robust.

Later, Kightley and Thompson [1987] carry out a comparison of preconditioned conjugate gradient algorithms with different multigrid methods. The conjugate gradient algorithms considered are the JCG, standard ICCG and truncated ICCG. The multigrid algorithms used are those described in Wesseling et al [1982] and Hemker et al [1983, 1984, 1985] and are denoted in brackets by a pseudo-name. These include the incomplete LU factorisation (MGD1), incomplete block factorisation (MGD5), the point red-black SOR (MG001) and the line-zebra SOR (MG003) algorithms. Results were presented for the solution of the Poisson equation with a discretised uniform 128x128 grid on a CRAY-1S. A speed-up factor of 3.3 and 4 were obtained in favour of the vectorised MGD1 and MG001 algorithms, respectively. A case is found where the truncated ICCG is less efficient than the standard ICCG algorithm (Kightley and Jones [1985]). The general conclusion was that the conjugate gradient based methods were efficient for low accuracy solutions but the multigrid methods were more appropriate when a much higher accuracy is desired in the solution.

Kincaid et al [1986] consider the application of the conjugate gradient (CG) and chebychev (SI) methods as a means of accelerating some popular iterative algorithms. The CG acceleration was substantially faster for the solution of Poisson's equation on a 20x20 grid using a scalar processor. (Scalar simulations were performed on a CYBER 170/750 and all vector simulations performed on a CYBER 205). Using a 64x64 grid there was little to choose between the vectorised red-black SOR-CG, Jacobi-CG and Richardson-CG. Even though the latter two required more CG iterations these algorithms were easier to implement and were recommended for general use.

Elaborate ordering schemes for the CG algorithm have been examined by Melhem and Gannon [1987]. For ill-conditioned systems of equations the column-wise two colour ICCG algorithm is shown to be more efficient than the natural ordered JCG algorithm.

Kapitza and Eppel [1987] describe an incomplete Crout factorisation for the conjugate gradient algorithm which is used to solve a three-dimensional Poisson equation. This is referred to as the Idealised Generalised conjugate gradient (IGCG) algorithm. The simulation was performed on a CYBER 205 and the performance compared to a number of iterative relaxation algorithms. Their unit of measure was the work unit (which is the time taken to carry out one iteration of the algorithm, WU) and speed-up factors of 10 over popular iterative algorithms such as the red-black SOR were not uncommon. However, it should be realised that the computation involved in a single WU of the CG algorithm is not the same as that of an iterative algorithm.

Gentzsch [1987] proposed a fully vectorised SOR variant for a general second order elliptic partial differential equation. The motivation for this was that there are overheads associated with the use of a red-black ordering, these would be quite significant on some vector processing architectures, for example the CRAY-2 and IBM 3090VF. The original unknowns are transformed to give a discretised approximation, instead of being described by the traditional five-point molecule with connections north-east-south-west (N-E-S-W), it is now described by NE-SE-SW-NW (figure 1.2.2). The new variant was tested on the solution of Poisson's equation using a 127x127 grid and was found to be twice as fast as the red-black algorithm on both the CRAY-2 and IBM 3090VF machines.

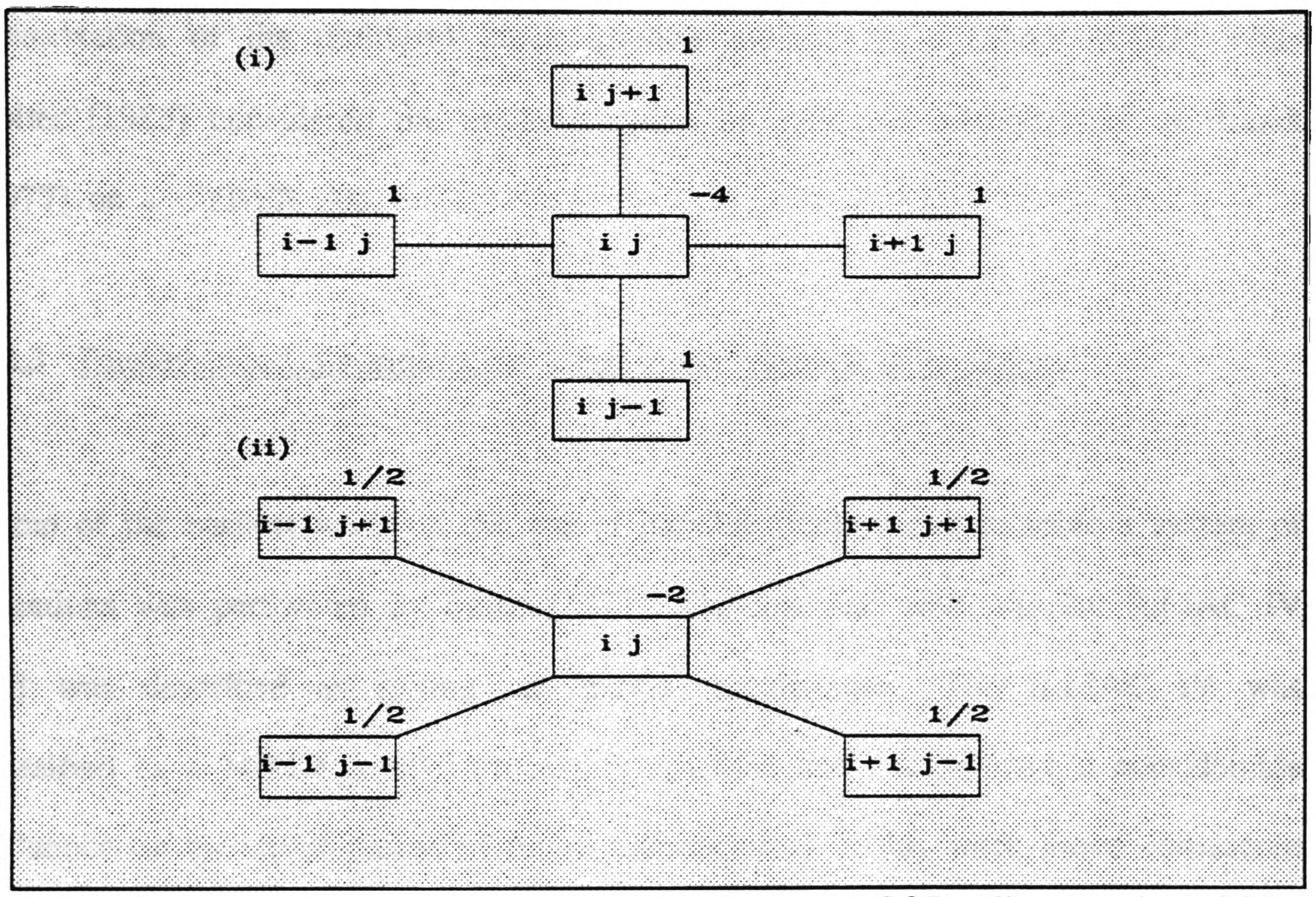


FIGURE 1.2.2 Computation molecule for (i) natural SOR (ii) vectorised SOR

The implementation and comparison of multigrid methods for pipeline architectures has been briefly mentioned. Hemker, Wesseling and Zeeuw [1984] compare two different preconditioning matrices on the CRAY-1 and CYBER 205 machines. The preconditioners were an incomplete LU factorisation (ILU) and a zebra SOR algorithm. They concluded that the zebra SOR algorithm was more efficient than the ILU factorisation on the CRAY-1, but the opposite was true when they were implemented on the CYBER 205. Vanka and Misengades [1987] suggested the vectorisation of the multigrid block implicit method on a CRAY X-MP, while Holter [1985] considered the implementation of multigrid methods due to Brandt [1977] on a CYBER 205.

1.2.3 Parallel-based algorithms for large sparse systems of equations

Some of the earliest work on the use of parallel architectures to solve a system of equations was performed by Stone [1973]. The machine used was the ILLIAC IV and was described as having an 'exotic' architecture. (The ILLIAC IV was classified as a MIMD parallel processing machine and was to have a considerable influence on the development of future architectures). In his work Stone considered the implementation of a tridiagonal solver using LU decomposition by recursive doubling. Unfortunately, the only results presented were based on the number of arithmetic operations.

The popularity of the cyclic reduction algorithm is such that it has been implemented on the ICL DAP (Whiteway [1979]). The ICL DAP is made up of a 64x64 array of processing elements, all of which simultaneously carry out the same instruction on a different data set. However, the implementation of the cyclic reduction algorithm on such an architecture leads to an inefficient utilisation of some of the processing elements. A hybrid algorithm is used, this consists of a cyclic reduction step followed by a Jacobi iteration process. In this application the cyclic reduction is being used as a preconditioner where the condition number of the matrix is being reduced.

The partition algorithm (Wang [1981]) is based on the notion of divide and conquer and like the modified Cramer's rule it has a higher operation count than the cyclic reduction. Although the algorithm has been clearly written for use in a pipeline or parallel processing environment, no results were presented.

Seager [1986] compared the performances of the JCG and ICCG algorithms using the four processor CRAY X-MP. The parallel processing is in the form of multitasking and microtasking. In multitasking there is a queue of tasks which are scheduled by the operating system and given to a processor which becomes free for computation. Microtasking involves the vectorisation of inner loops and the execution of the outer loops over the four processors. Microtasking has a finer level of parallelism than multitasking. In the solution of the Poisson equation using a 168x168 grid it is observed that the overheads associated with multitasking are significantly higher than those for microtasking. For the ICCG algorithm factors of 2.7 and 3.9 were obtained when multitasking and microtasking was used, respectively. For the JCG algorithm the improvements are lower with 2.2 and 3.1 when multitasking and microtasking was used, respectively.

1.3 Discussion

A literature review of publications using a pipeline processor to solve partial differential equations has been carried out. One observation which arises is that there is no single 'best' algorithm. This is not surprising since there are a number of different factors which can have a significant effect on the performance of an algorithm.

The comparison between different algorithms is highly problem dependent. The convergence rate of some algorithms tend to decrease noticably as the diagonal dominance of the matrix becomes weaker. Therefore, one suggestion could be to solve a number of different matrix systems with varying diagonal dominance factors, this would help to present a more complete picture.

Another problem involves the implementation of the algorithm on different pipeline architectures. Despite the fact that a scalar algorithm is universal to all scalar machines this is not the case for the same vectorised algorithm. The vectorisation techniques used to restructure the scalar algorithm may be different and so lead to a performance specific to that vectorisation technique. In addition, the use of software tools such as compilers and low-level run-time vector libraries which have varying levels of sophistication can make comparisons even more difficult. Finally, the fact that different pipeline architectures have different characteristics means that it is unlikely any single algorithm can claim to be the most efficient on all pipeline architectures.

The survey clearly shows that the computation effort is concentrated on the solution of linear systems of equations typified by the discretisation of the Poisson equation. In some cases almost total vectorisation of the code is possible for some algorithms and this leads to substantial reductions in computation times. A high proportion of the code is vectorised because there is a relatively small overhead associated with the setting up of coefficients and source terms. However, will this be the case in fluid flow simulations where there are many more factors to be considered?

It is known that the problems discussed with regard to the implementation of algorithms on pipeline architectures will still apply to CFD computations. The solution of a linear system of equations still forms a major component in the solution procedure, however, the essentially scalar computations become more significant. These involve the generation of more complex diffusion and convection coefficients as well as complicated source terms.

1.4 Outline of present work

A fundamental description of various parallel processing architectures is presented, and attention is then focused on the pipeline vector processor and how it fits into various classes. All the computations in this work are carried out on the VA-1 pipeline processor, this is attached to a MASSCOMP 5400 machine (MASSCOMP [1984]); Therefore, a detailed characterisation of this machine is given. A measure of the expected speed-up is determined using Amdahl's law, this has proved useful and is used throughout this work to assess the performance of the vectorised code.

Different solution procedures are reviewed with regard to the solution of the Navier-Stokes equations in Chapter 3. These solution procedures involve SIMPLE and its derivatives. The SIMPLE procedure is chosen for implementation because of its suitability to the problems to be solved. In addition, a whole-field strategy is adopted since this will enable vector operations of maximum possible length.

In Chapter 4 a number of different algorithms such as the Thomas, cyclic reduction, JCG, JUR, SOR and red-black SOR are applied to the solution of the Laplace equation on a unit square, for a number of different grids. The algorithms are then vectorised in various ways, the Thomas and SOR being restructured to remove the recursion present in the scalar formulation. The expected improvement factors are predicted using Amdahl's law. This identifies the point-by-point and conjugate gradient solvers as the most efficient vectorised algorithms.

The complexity of the problems solved are extended to fluid flow simulations involving the solution of pressure and momentum components (Chapter 5). The test cases involve the solution of the two-dimensional lid-driven cavity problem and the flow in a suddenly expanding duct. The effect of just vectorising the pressure-correction equation solver in the SIMPLE procedure leads to modest improvements in speed, the limiting factor being the scalar computations. Further vectorisation is carried out on the rest of the SIMPLE procedure and this leads to a more substantial reduction in the computation time.

In Chapter 6 the effect of introducing scalar transport equations such as enthalpy and k-ε turbulence representations are investigated. Here the test cases include the natural convection in a square cavity problem which introduces the solution of the enthalpy equation for Rayleigh numbers up to 10^6, and the k and ε equations for a Rayleigh number of 10^7. The second case is two-dimensional, turbulent, L-shaped flow in a duct. Although there is a reduction in the total contribution of the pressure-correction solution, the vectorisation of the scalar equations still leads to worthwhile reductions in time.

A multigrid solution strategy based on the ACM method of Settari and Aziz [1973] is presented in Chapter 7 following a review of multigrid methods. The ACM method is used to solve the pressure-correction equation and is applied to the four test cases described in Chapters 5 and 6. The improvements in computational speed are more notable in the cases where there is a dominant pressure field. The best performance of the scalar algorithms was achieved with up to four levels of the ACM method. Whereas, the same algorithm vectorised is most effective with just two levels. It is likely that the number of levels used by the vectorised algorithm will increase as the grid size is increased.

Finally, conclusions and suggestions for future development of the present work are presented in Chapter 8.

CHAPTER TWO

2.0 CLASSIFICATION OF ARCHITECTURES

2.1 Introduction

This chapter describes the general classification of a computer according to its architecture. Attention is focused on the pipeline vector processor category, and in particular to the MASSCOMP 5400 computer with an attached pipeline vector processor (VA-1). The potential of such a vector pipeline processor is investigated. A means of predicting the expected speed-ups in using such a processor is also outlined. This strategy is to be used at a later stage for the consolidation of quoted speed-ups for a CFD code.

2.2 Classification of architectures

The classification of computer architectures into an accurate and universal form is not an easy task. To date, there have been three different approaches presented. These are due to Flynn [1966, 1972], Shore [1973] and Hockney and Jesshope [1981]. All three have their merits but no single approach has emerged as the universally accepted classification scheme.

There are many reasons for this, the most significant of which is the broad spectrum of parallel architectures which have been proposed. Some of these architectures have come into being because of their obvious potential (for example, pipelining), others remain essentially theoretical (for example, the MISD machine proposed by Flynn [1966]). Another problem with attempting to classify these architectures is that in some cases the more useful architectures do not fall into a single category. They may fall into many categories, or none at all, hence requiring a separate category. The three different approaches to classifying these architectures are now presented.

2.3 Classification due to Flynn

The classification due to Flynn [1966] provides a broad characterisation of the different computer architectures. However, the categories defined are based on the flow of data or instructions (referred to as a 'stream'), rather than on the structure of the machines. Whether the instruction or data streams are single or multiple will determine one of four possible categories.

2.3.1 Single Instruction stream - Single Data stream (SISD)

This class of machine accepts a single stream of instructions, each of which acts upon a single stream of data items. A pipeline processor can be used to increase the rate at which instructions are processed, therefore machines with pipeline processors of this type are classed as SISD machines. SISD machines are also collectively called standard von Neumann machines.

2.3.2 Single Instruction stream - Multiple Data stream (SIMD)

This class of machine also accepts a single stream of instructions, however, each instruction acts upon a multiple stream of data items. The multiple stream of data can also be regarded as a vector of data, where each vector element represents a single stream of data items. The multiple stream of data can be achieved either through an array of processors or a pipeline processor. There are many examples of SIMD machines including processor arrays such as the ICL Distributed Array Processor (DAP) and ILLIAC IV, and pipelined vector machines such as the CRAY-1 and the CYBER 205.

2.3.3 Multiple Instruction stream - Single Data stream (MISD)

In this class of machines there are many different instructions being performed on single data items. To date, there are no practical examples of this class.

2.3.4 Multiple Instruction stream - Multiple Data stream (MIMD)

This final class is representative of true multiprocessor machines. In this class each processor accepts its own instruction stream and acts upon its own stream of data. Gorsline [1980] suggests that the pipeline processor falls into this class since it performs many instructions on a multiple scalar stream of data. Examples of MIMD machines include the Denelcor Heterogeneous Element Processor (HEP) and an array of transputer processors.

2.4 Classification due to Shore

The classification of machines according to how they are organised was proposed by Shore [1973]. Six different machine types (I - VI) are described, each machine type defined using four basic parts - a control unit (CU), a processing unit (PU), an instruction memory (IM) and a data memory (DM). What differentiates the six machine types is the particular way in which the basic parts (including multiples) are arranged.

2.4.1 Machine I

This arrangement describes the conventional von Neumann machine (figure 2.4.1-1). The single DM read produces all bits from a single word for processing in parallel by the PU, this is referred to as a horizontal word slice. However, since the PU may contain multiple functional units and may also be pipelined, machines such as the CRAY-1 can also be included in this class.

2.4.2 Machine II

This arrangement is very similar to that of machine I. The major difference is that the single DM read produces a single bit from all words (figure 2.4.2-1). Again, all bits are processed in parallel by the PU, this is referred to as a vertical bit slice. The more words that need to be processed, the more significant the speed advantage of this machine. Examples of this machine type include STARAN and the ICL DAP.

2.4.3 Machine III

This arrangement provides both horizontal and vertical PU's and so allows access to both bit and word slices (figure 2.4.3-1). This machine type is a combination of machines I and II and therefore has the benefits of both. An example of this machine type is the Orthogonal Computer of Shooman.

2.4.4 Machine IV

This arrangement is a natural extension to that of machine I. Here, the PU and DM are replicated, and these are under the control of a single CU (figure 2.4.4-1). Although there is no direct communication between PU's, this architecture can easily be extended. An example of this machine type is the PEPE (Parallel Element Processor Ensemble) machine.

2.4.5 Machine V

This arrangement is an improvement to machine IV. It allows PU's to communicate with its two neighbours and is sometimes referred to as the connected array class (figure 2.4.5-1). An example of this machine type is the ILLIAC IV machine.

2.4.6 Machine VI

This final arrangement has a single component containing the PU and DM (figure 2.4.6-1). Here the processing logic is distributed throughout the memory. Examples of this type of machine are associative processors.

2.5 Classification due to Hockney and Jesshope

As part of this classification a comprehensive notation is introduced to aid with the description of different architectures. Hockney and Jesshope [1981] define a

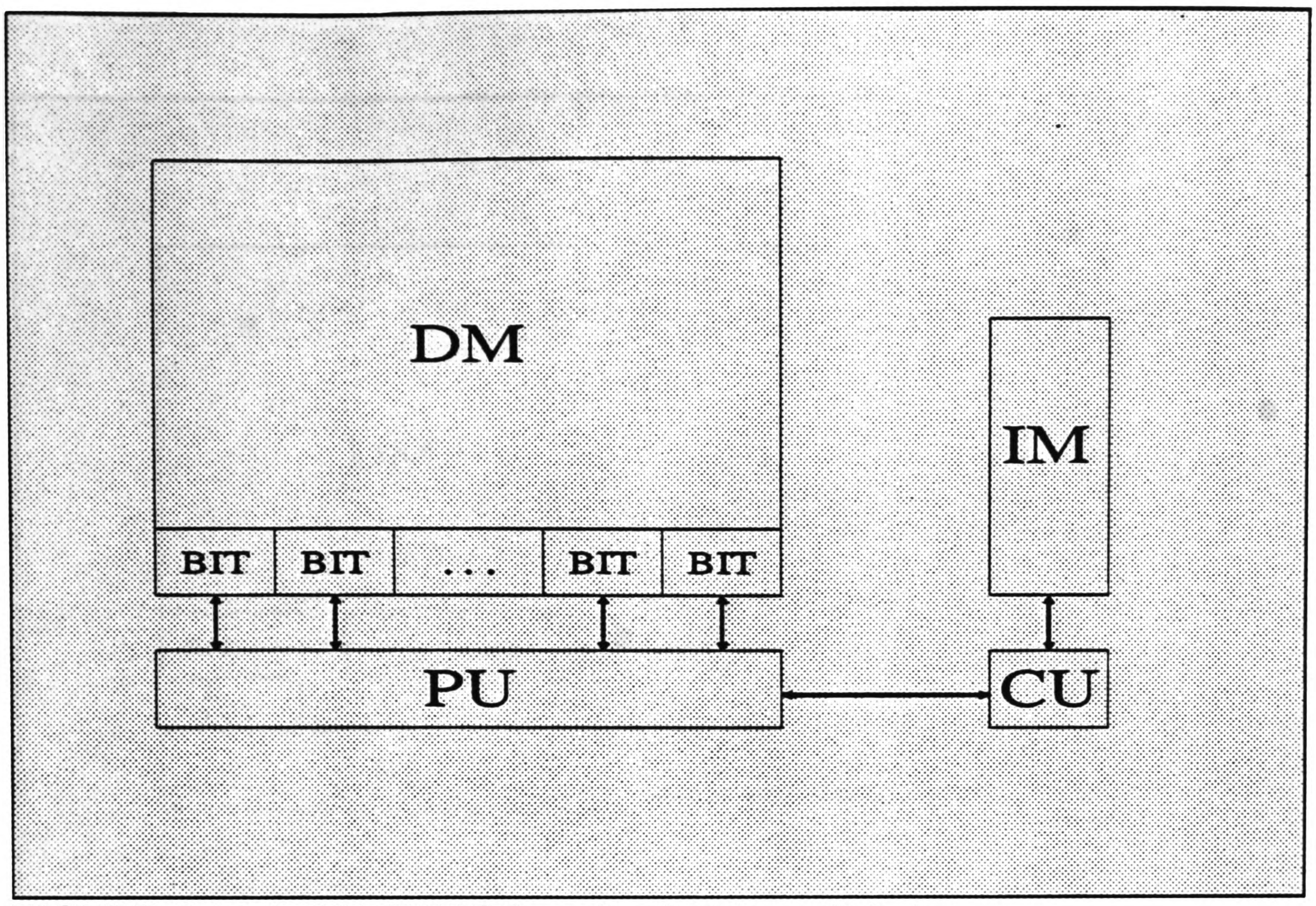


FIGURE 2.4.1-1 MACHINE I:word-serial bit-parallel class

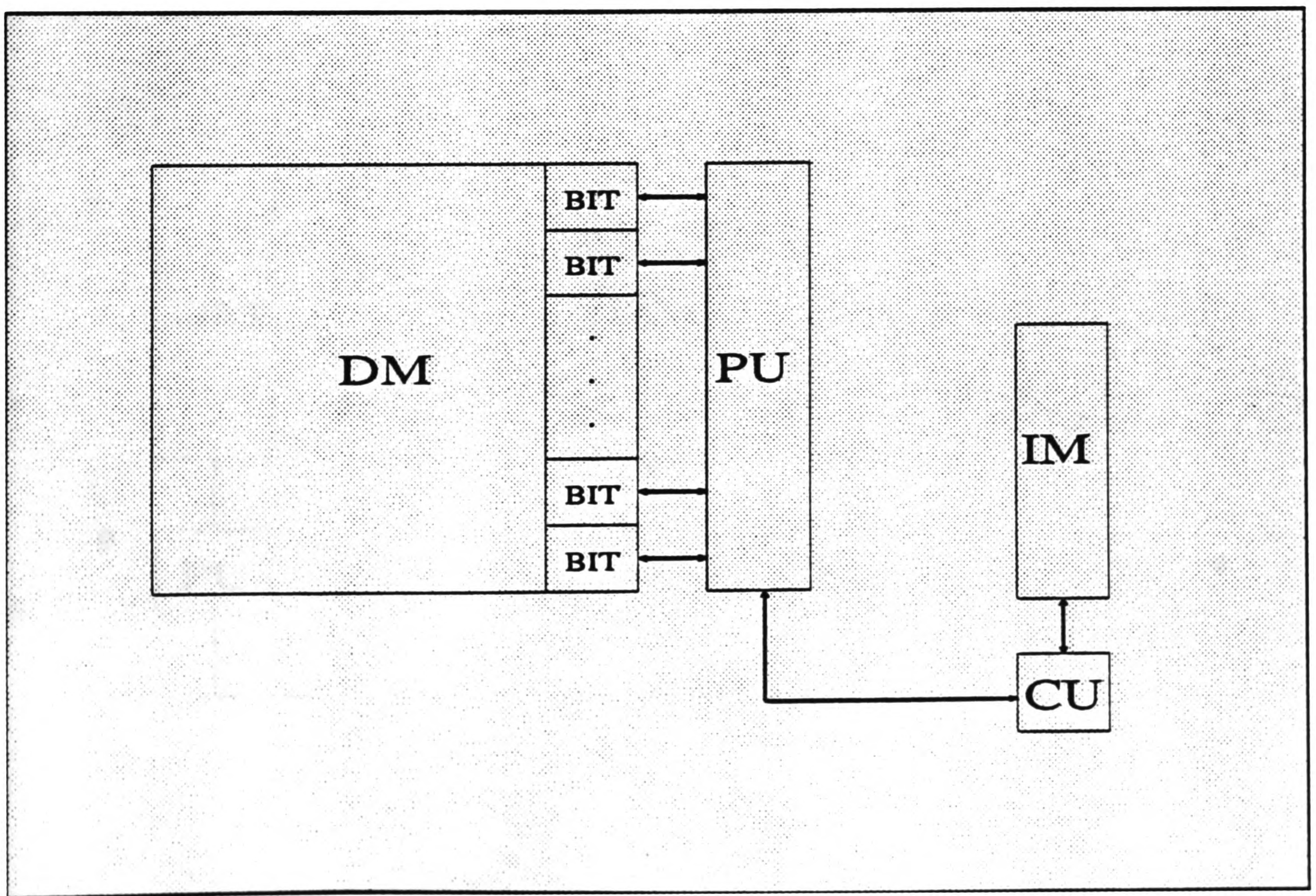


FIGURE 2.4.2-1 MACHINE II:word-parallel bit-serial class

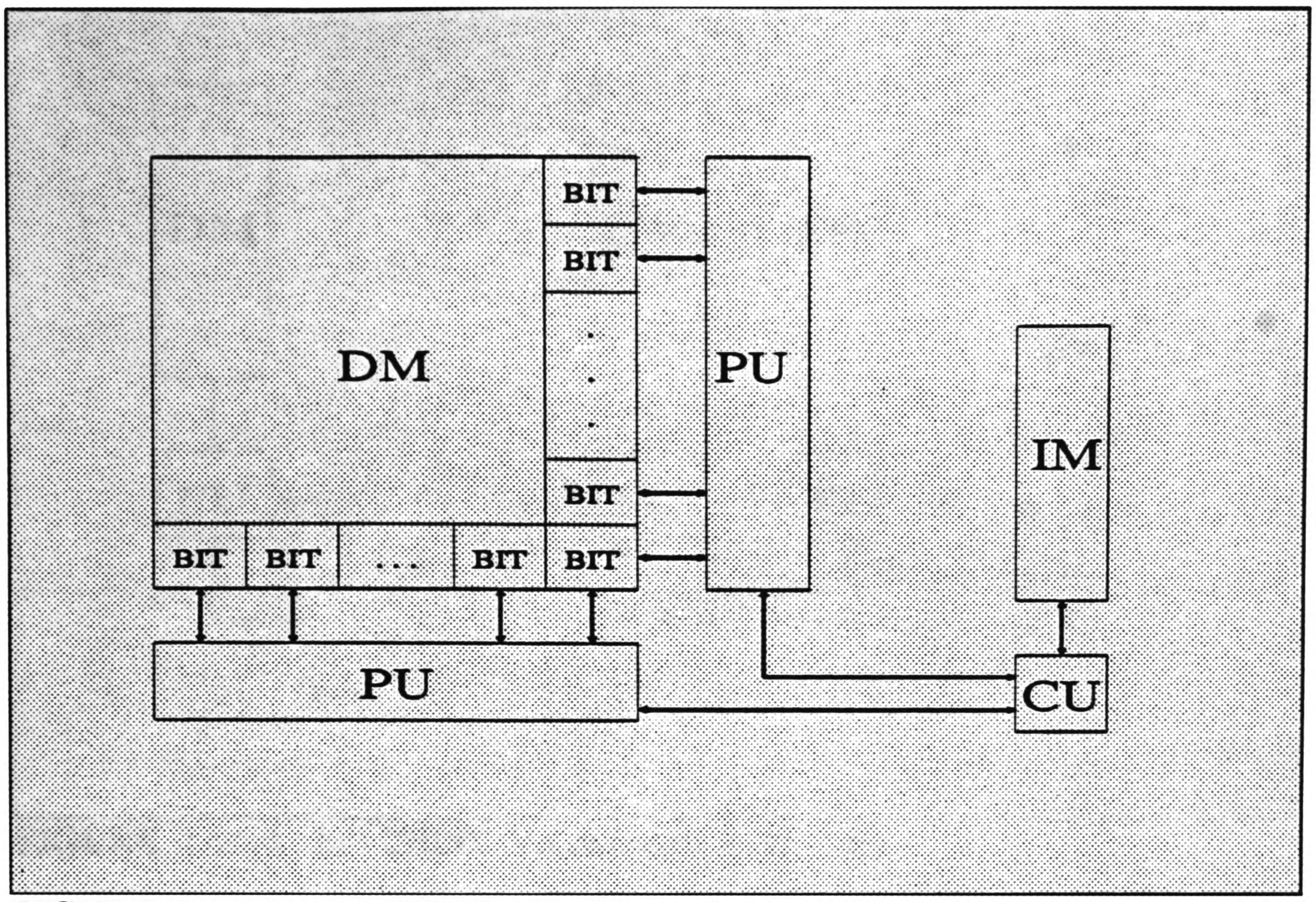


FIGURE 2.4.3-1 MACHINE III:orthogonal class

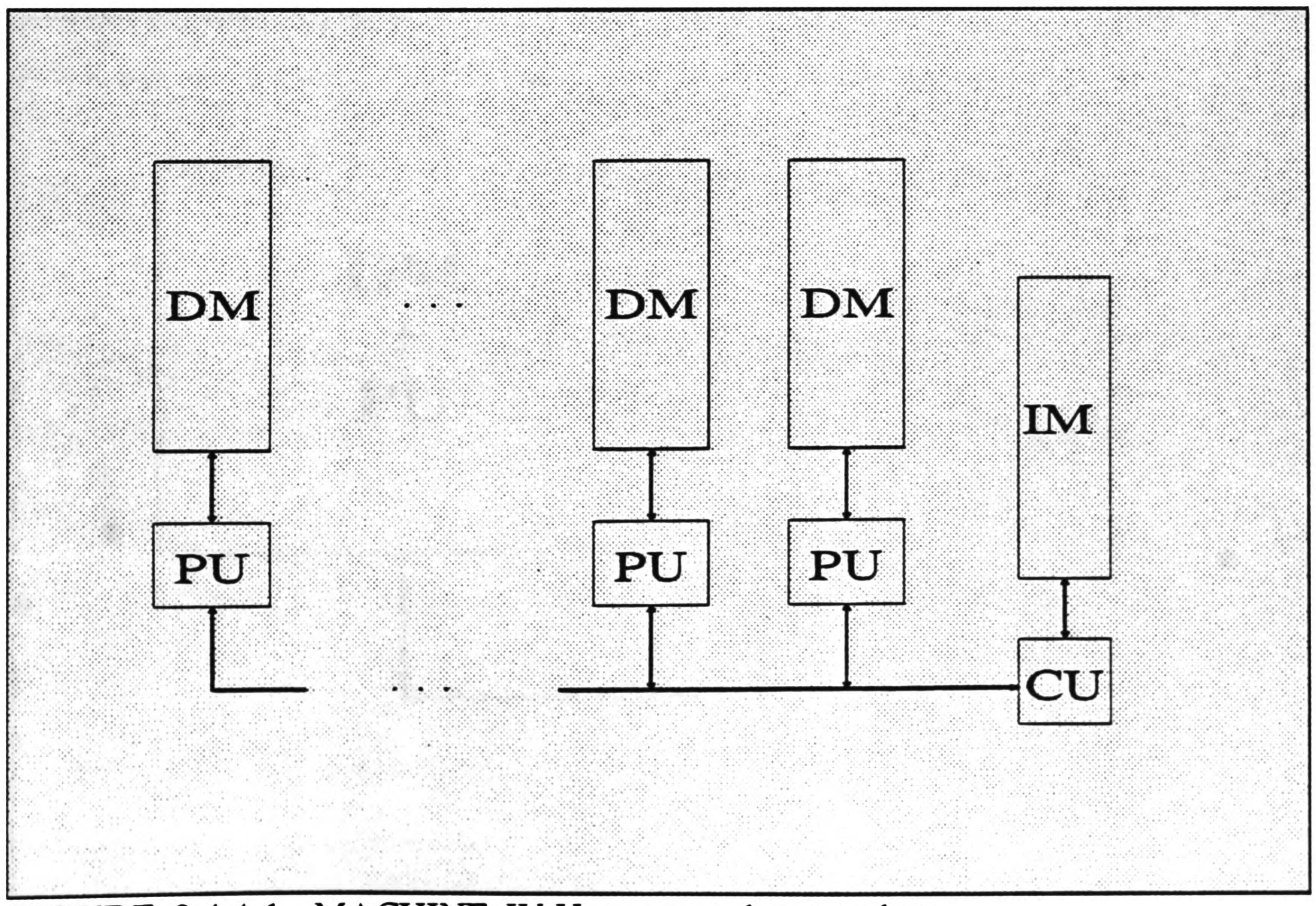


FIGURE 2.4.4-1 MACHINE IV:Unconnected array class

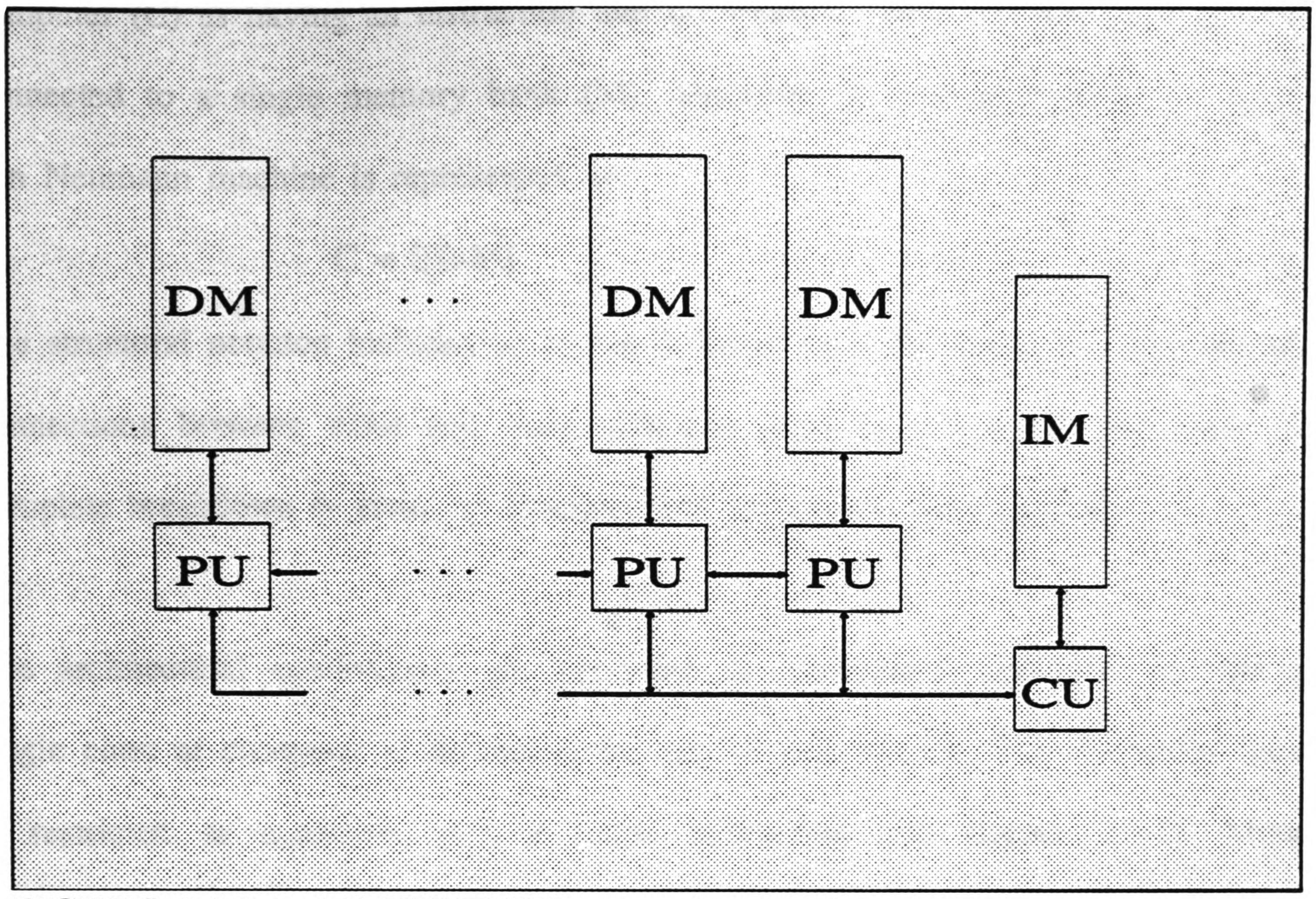


FIGURE 2.4.5-1 MACHINE V:Connected array class

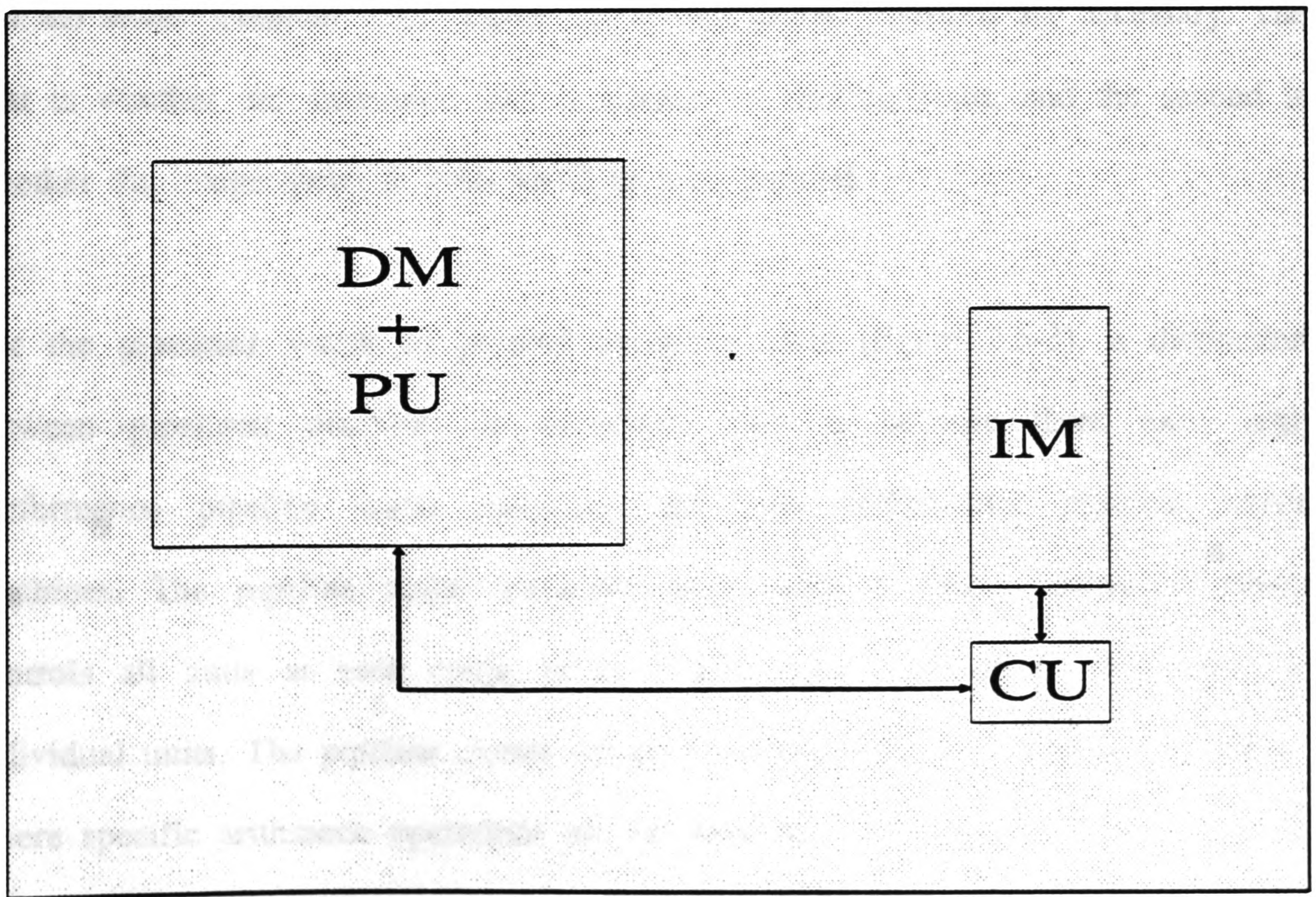


FIGURE 2.4.6-1 MACHINE VI:Logic in memory class computer (C) as having an instruction unit (I) which controls an execution unit (E) connected to a single memory bank (M). Therefore in notational form the scalar von Neumann machine is represented by

$$C = I[E\text{-}M]$$

The structural notation includes up to ten different rules for units, six rules for the connections between units and three different types of control of the units. A complete breakdown is given by Hockney and Jesshope [1981] pp32-42.

The architectural subdivisions are presented as hierarchical structures so that a single class of computer is defined at the end of each branch. The discussion here is restricted to machines with a single instruction unit (figure 2.5-1). More specifically, machines with a single instruction unit - single unpipelined execution units (serial processors), and a single instruction unit - multiple execution units (pipelined vector or parallel processors).

For the serial computer class (figure 2.5-2) two further divisions are necessary. The first is whether the arithmetic unit is integer- or floating-point, and the second is whether the integer-point is 1-bit serial or n-bit parallel.

For the pipelined vector or parallel computer class (figure 2.5-3), a distinction between pipelined machines is necessary. This is because there exist high performance pipeline scalar machines and high performance pipeline vector machines. The pipeline scalar machines have either a single instruction which controls all units at each cycle, or a system where instructions are issued to individual units. The pipeline vector computers are divided into two classes. Those where specific arithmetic operations are executed are referred to as special-purpose pipelines, and those where more than one arithmetic operation can be executed are referred to as general-purpose pipelines.

The final subdivision of multiple execution units is the processor array class of computers. These can be either floating-point or few-bit execution units. Further divisions describe the way in which the processors are connected.

Flynn's approach provides a useful, broad, easy-to-remember classification of architectures. However, it does have its drawbacks. For example, the interpretation of the term 'stream' can be such that the pipeline processor is placed in all four categories. It may be classed as SISD because it processes a single stream of vector data , or SIMD if every element of the vector is regarded as an individual stream of data. It can be classed as MISD or MIMD if the pipeline arithmetic unit performs in parallel on a scalar or vector stream of data. Flynn placed the pipeline processor together with processor arrays despite the completely different architectures.

The classifications due to Flynn and Shore are very similar. Machine I and the SISD class are equivalent, and machines II, III, IV and V provide a detailed breakdown of the SIMD class. Not surprisingly, there is no obvious class for the pipeline processor.

The classification of Hockney and Jesshope provides a detailed breakdown of computer architecture based on functional units. Although more precise, (for example it has a clear classification of the pipeline processor), it does have the drawback of being less memorable.

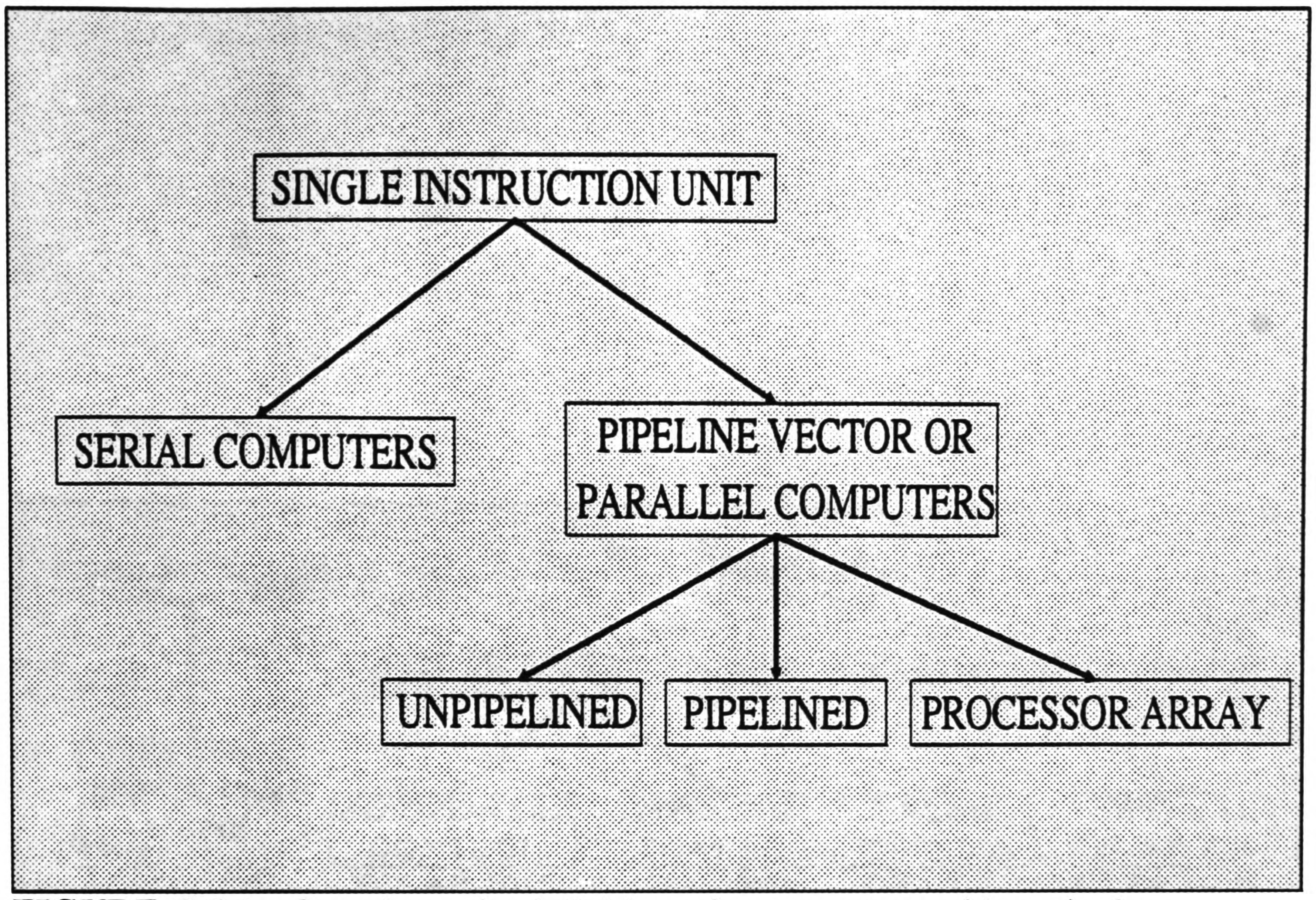


FIGURE 2.5-1 Overview of subdivisions for computers with a single instruction unit

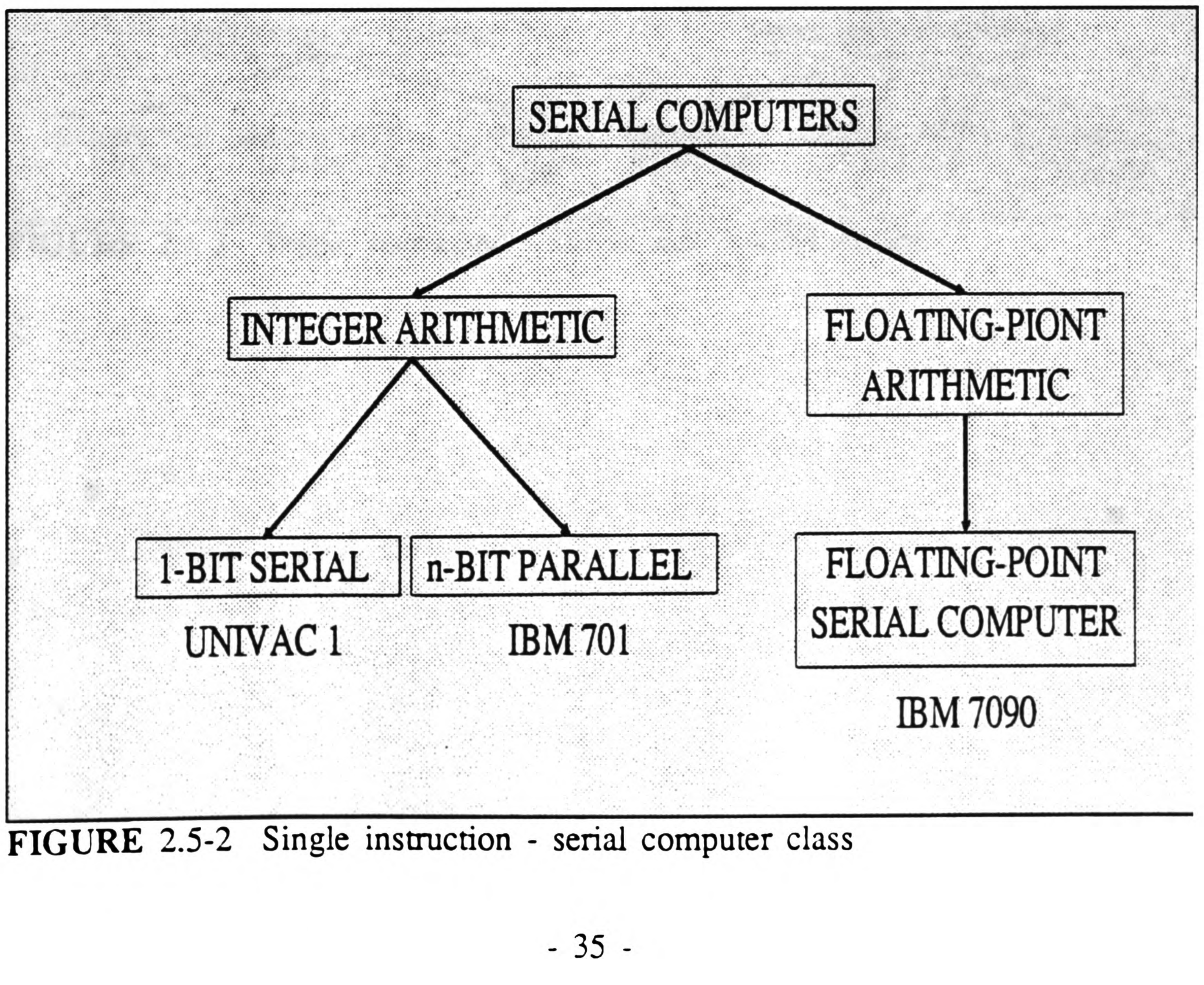


FIGURE 2.5-2 Single instruction - serial computer class

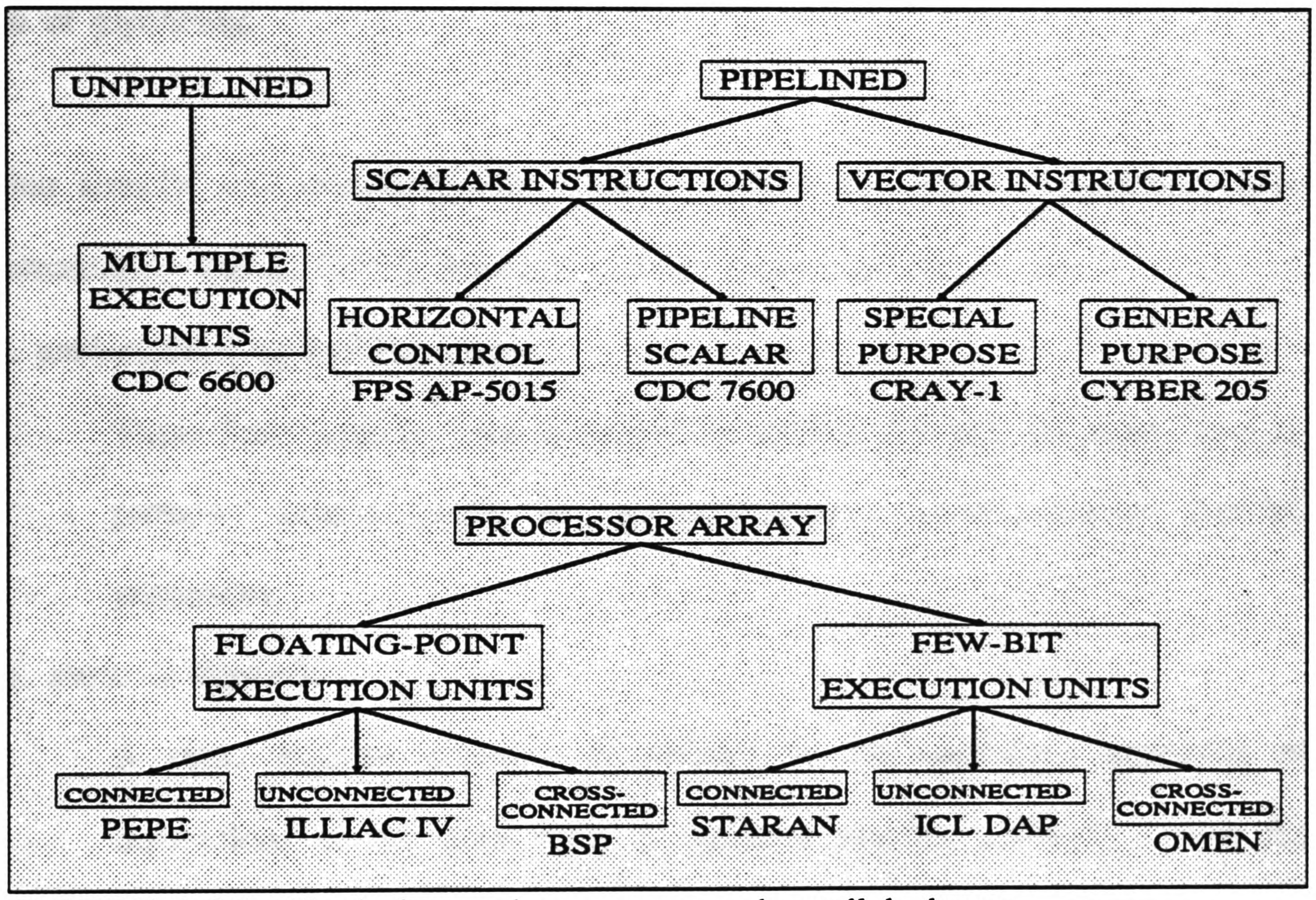


FIGURE 2.5-3 Single instruction - vector and parallel classes

2.6 Classification of pipeline processors

The notion of using a pipeline process to improve the efficiency of a system has existed for many years. It has been widely used in automated industrial plants, in particular the car industry. This has since been extended to enhance computer performance. The CDC 7600 was amongst the first of such computers to utilise the idea of pipelining.

It has already been mentioned that different pipeline processor configurations exist (Ramamorthy and Li [1977], Handler [1977]). Three such classes are:

i. unifunctional or multifunctional

   These have already been described in section 2.5 and are either special-purpose (unifunctional) or general-purpose (multifunctional) pipeline processors.

ii. static or dynamic

   A static pipeline processor is defined by the continuous execution of instructions of the same type. A dynamic pipeline processor allows the simultaneous existence of several functional configurations.

iii. scalar or vector

   Processing a sequence of scalar operations under the control of a loop defines a pipeline scalar processor, and similarly for processing vector operations defines a pipeline vector processor.

All future references to a pipeline processor will imply a pipeline vector processor.

2.7 How a pipeline processor attains its speed

A pipeline consists of a number of processing stages, where each stage is responsible for a specific task in an arithmetic operation. Information is transferred between adjacent stages under the control of a common clock. Consider the problem of performing the arithmetic operation

$$c_i = a_i + b_i \qquad i=1,...,4$$

where it takes four stages to complete a single addition. Figure 2.7-1 shows the benefit in using a pipeline processor over the conventional scalar processor. By overlapping the arithmetic operations a result is obtained after the fourth clock cycle, and thereafter a single result is obtained every clock cycle (total of 7 clock cycles). In the case of a scalar processor a result is obtained every fourth clock cycle (total of 16 clock cycles).

In general, an arithmetic operation which takes l stages can process vectors of length n in a time given by

$$T_l = l + (n-1) \qquad (2.7\text{-}1)$$

where T_l is the time in clock periods. Here, l clock cycles are required to obtain the first result and n-1 cycles to complete the remaining n-1 results. Using a scalar processor the time taken to complete the arithmetic operation is given by

$$T_1 = nl \qquad (2.7\text{-}2)$$

We can now define the speed-up S_l of a pipeline processor with l stages over the conventional scalar processor as

$$S_l = \frac{T_1}{T_l} = \frac{nl}{l + (n-1)} \tag{2.7.3}$$

Thus the theoretical speed-up (S_{max}) approaches l for a large vector length. This speed-up is never reached for many reasons. These include a penalty time incurred in initialising the pipeline processor and delay times between clock cycles.

2.8 Memory-to-memory and register-to-register pipeline processors

The difference between these two architectural configurations depends on where the operands are retrieved from within the pipeline processor. If all the source operands and results are retrieved directly from the main memory then this describes the memory-to-memory architecture. Here it is necessary to specify the base address, offsets, increments and vector lengths which define the vectors to be used. Examples of machines with this configuration include the STAR-100 and CYBER 205.

If the source operands and results are retrieved indirectly from the main memory and through registers, then this describes the register-to-register architecture. Examples of machines with this configuration include the CRAY family and the Fujitsu VP-400.

Thus far different pipeline architectures have been considered. Attention is now focused on a pipeline processor which has been used as part of this research.

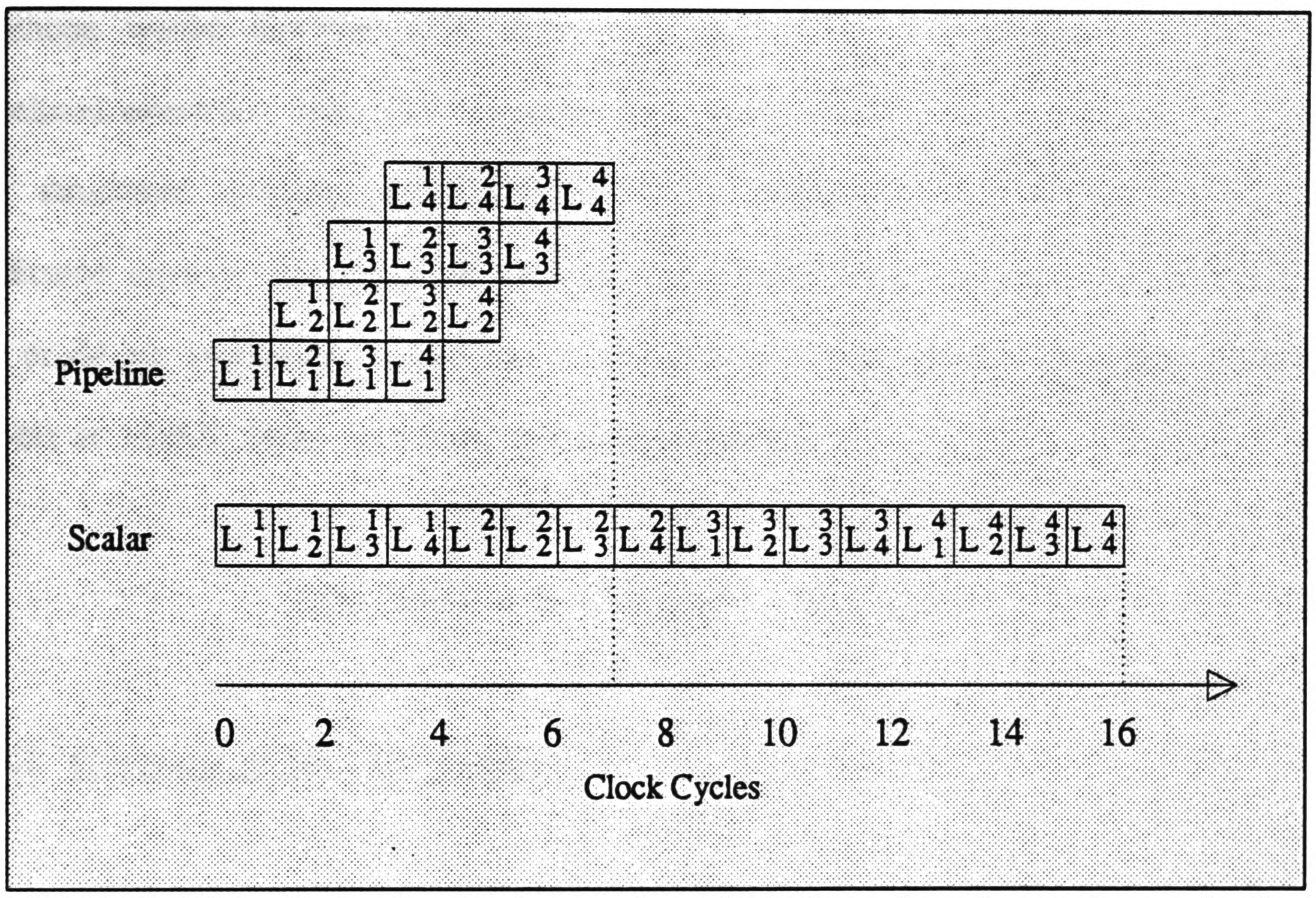


FIGURE 2.7-1 Reduction in the number of clock cycles when using a pipeline processor

2.9 The MASSCOMP MC5400 system

The MASSCOMP MC5400 machine (MASSCOMP [1984]) was a product of the Massachusetts Computer Corporation based in Westford, Massachusetts. In accordance with the main classifications discussed in sections 2.3, 2.4 and 2.5, the MC5400 can be described as SIMD by Flynn or machine I by Shore. Hockney and Jesshope would describe such a machine as a single instruction unit with a pipelined execution unit. The execution unit operates on vector instructions. Using the structured notation, the MC5400 with a vectorised instruction unit has a dedicated pipeline floating-point adder and multiplier which can execute either 32-bit or 64-bit operations. It also has a configurable vector memory of either 32,000 32-bit or 16,000 64-bit locations, this is summarised as

$$C(MC5400) = Iv\ [Fp_{32,64}(+,*)\ -\ M_{32K*32,16K*64}]$$

2.9.1 Overview of MC5400 system

Figure 2.9.1-1 gives an overview of the MC5400. In its simplest form it consists of a triple-bus architecture. The Multibus Adapter direct memory access (DMA) transfers at a rate up to 6Mb/sec between the memory bus and the Multibus. The memory bus operates at a rate of 12Mb/sec, the Multibus at a rate of 6Mb/sec and the STD+bus at a rate of 2Mb/sec. The MC5400 CPU is based on the Motorola MC68020 which runs at 16.7MHz. The MC5400 has an enhanced UNIX operating system which is compatible with system V and Berkeley 4.2. The host CPU is connected to a Vector Accelerator board, VA-1 (Davies [1987]).

2.9.2 Overview of VA-1 board

The vector accelerator is an auxiliary processor dedicated to the host processor. Figure 2.9.2-1 shows the programmer's conceptual overview of the VA-1 board and its connection to the host processor. The connection between the host and the board is through the MASSCOMP Memory Interconnect bus (MI). Up to seven VA-1 boards may be connected to the MI bus, and the boards can operate independently or in parallel.

The vector accelerator board consists of the following units:-

i. The VA memory

This is a scratch pad consisting of approximately 32,000 32-bit locations. The memory can serve as the source and/or destination of arithmetic operations.

ii. The VA controller

This has two main functions. Firstly, it handles virtual to physical memory management, and secondly, it controls and schedules the DMA and MATH processors. It does this in the following way:- It decodes the information in a ring buffer packet and conveys this information to the relevant processor. It determines the completion of a given instruction and updates the associated packet. It also monitors synchronisation between the DMA and MATH processors.

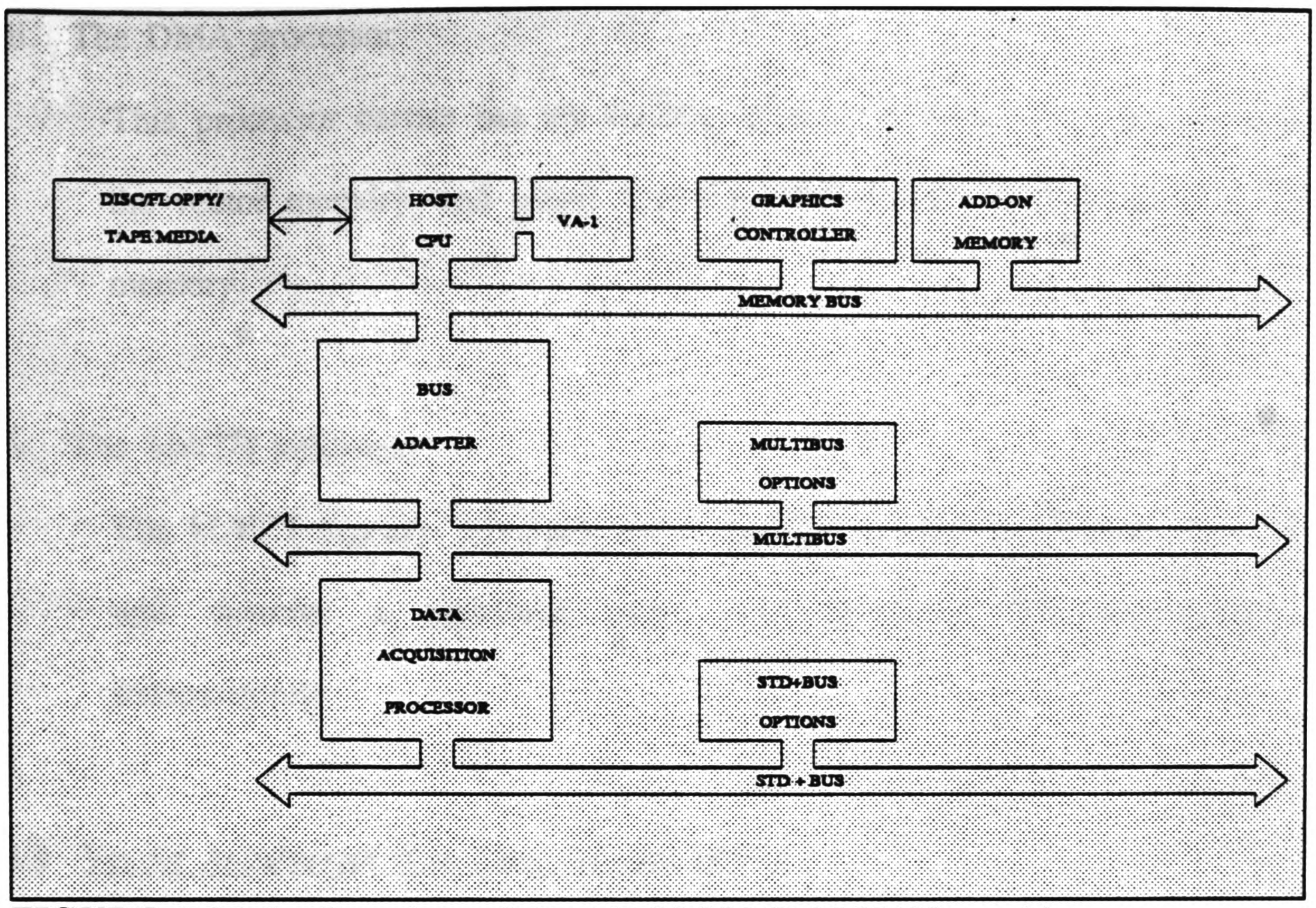


FIGURE 2.9.1-1 Overview of the MASSCOMP MC5400 machine

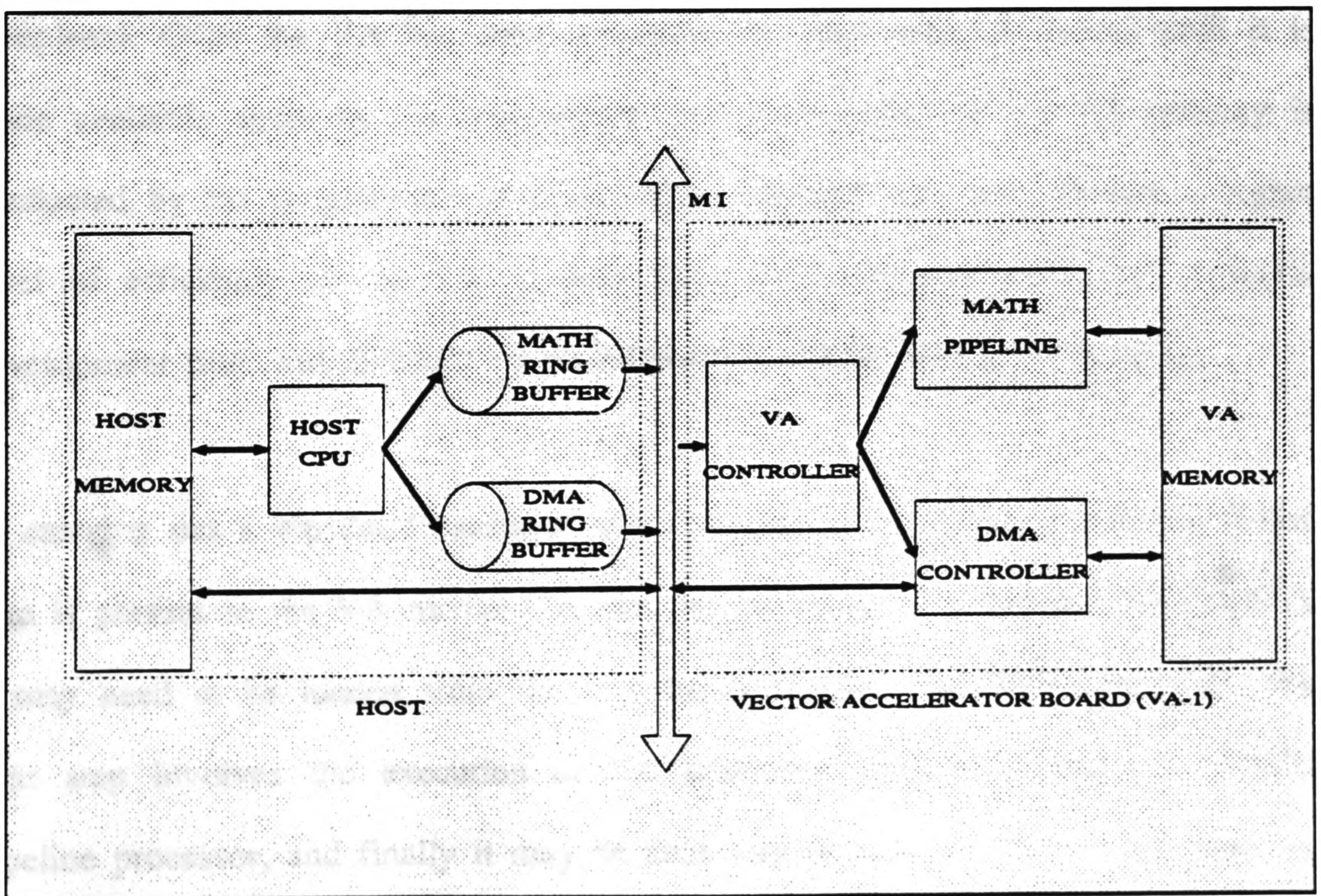


FIGURE 2.9.2-1 Conceptual view of the VA-1 vector accelerator board

iii. The DMA processor

This processor carries out the loading of data from the host memory into the vector memory, and stores data from the vector memory into the host memory.

iv. The MATH processor

The MATH pipeline processor has a memory-to-memory arrangement. The unit features a pipelined adder and multiplier that can operate independently.

2.9.3 Vector Accelerator Run Time Library (RTL)

The run time library is a set of 'low-level' subroutines which can be called from within a FORTRAN or C program. Initialisation of the VA is carried out at the start of a program execution and terminated after all vector instructions have been completed. Once the VA has been initialised no other user has access until it is made available again by the programmer. All management of the VA memory is conducted by the programmer at a low-level. Although this may achieve a higher level of efficiency of the VA, it does have the drawbacks of a high program development time and difficulty in maintaining flexibility within the program.

In carrying out a pipelined execution the first step is to ensure that the necessary data is present in the VA memory. It may be the result of a previous execution or it may need to be loaded from the host memory using the DMA processor. The next step involves the execution of the arithmetic operation using the MATH pipeline processor, and finally it may be necessary to store the result back onto the host memory. Between each vector instruction the responsibility lies with the programmer to ensure correct synchronisation.

The RTL contains an extensive range of arithmetic operations. These include scalar outputs for example, the dot product of vectors; Monadic vector operations such as negation; Diadic vector-vector and scalar-vector operations such as addition and multiplication; Finally, tertiary operations involving vectors and scalars, for example, (vector+vector)*scalar. Gather and scatter routines allow the processing of data where non-linear, irregular increments of vector elements are needed. Mathematical operations such as square root, reciprocation and trigonometric functions are also available.

A set of high-level subroutines have been written (Ierotheou [1987]). These routines represent a subset of the low-level RTL routines and include arithmetic operations used extensively within a CFD code. Indeed, Cross et al [1989] have used these high-level routines to assist in the rapid solution of enthalpy-based solidification problems. The high-level routines were written to address three areas essential to the 'survival' of the vectorised CFD code and to vector processing on the MC5400. These areas are:

i. The significant reduction in the time taken to develop CFD code suitable for vector processing. This is achieved by removing the burden of having to manage the VA memory at a low-level. The typical programming needed to carry out the simple addition of two vectors using both the low-level and high-level routines is given in Appendix 2.9.3. Using the high level routines not only makes the code compact and easy to follow, but also significantly easier to use.

ii. The solution of large problems is not restricted by the maximum memory available of 32K. This is achieved quite readily by splitting the vectors into 'slices' or 'chunks' which do fit into the vector memory, so making full use of the vector processing power. Thus for the addition of two vectors with lengths greater than 16,000, the vectors are partitioned into slices of 16,000 (Appendix 2.9.3). The only drawback to this approach is the overhead incurred in loading the relevant slices into the vector memory and then storing back the partial result. The net result in performance of these routines is discussed in section 2.9.4.

iii. The transferring of the vectorised CFD code onto other machines with vector processing capabilities. This addresses the question of portability of the code. This is a straightforward procedure, the only requirement would be the writing of the equivalent high-level routines for the ported machine. Since nearly all of the more powerful supercomputers use FORTRAN expressions to utilise their vector processing power, the writing of these routines would be a trivial task.

2.9.4 Performance of the MC5400

2.9.4.1 Measurement using $n_{1/2}$ and r_∞

Hockney [1977] introduces two parameters to describe the hardware performance of a pipeline processor. These parameters represent the vector length required to achieve half the maximum performance ($n_{1/2}$), and the maximum computation rate (r_∞). To determine these parameters is straightforward. Timings (t) are recorded for the multiplication of two vectors, and this is repeated for a number of different vector lengths (n). Plotting the CPU time with vector length, the resulting straight line is given by

$$t = an + b \qquad (2.9.4.1\text{-}1)$$

where a represents the slope of the line and b represents the intercept of the line with the t-axis (figure 2.9.4.1-1). The parameter $n_{1/2}$ is given by the modulus of the intercept of the line with the n-axis (i.e. b/a), and r_∞ is given by the reciprocal of the slope of the line (i.e. $1/a$).

Figure 2.9.4.1-2 shows the variation of CPU time with vector length when the scalar processor was used to carry out the multiplication operation. The straight-line graph has the following equation

$$t = 1.52\text{x}10^{-5}\ n \qquad (2.9.4.1\text{-}2)$$

From this $n_{1/2}$=0, this is not surprising since there is no start-up time in carrying out a scalar operation. The slope gives r^{-1}_∞ and hence r_∞=0.08 Mflops.

Figure 2.9.4.1-3 shows the variation of CPU time with vector length when the high-level VA routine was used to carry out the multiplication. The straight-line graph has the following equation

$$t = 2.87\text{x}10^{-6}\ n + 1.1\text{x}10^{-4} \qquad (2.9.4.1\text{-}3)$$

From this $n_{1/2}$=60 and r_∞=0.348 Mflops (the rate at which the high-level RTL routine performs). Here $n_{1/2}$ is not zero since there is a start-up time and a load/store penalty incurred in using the VA. Despite this, the execution of the high-level RTL routine is at least four times faster than the scalar equivalent operation.

If we consider the time taken to carry out the load/store as negligible, then the CPU time to execute just the multiplication is far less, and is shown in figure 2.9.4.1-4. The straight-line graph has the following equation

$$t = 1.56\text{x}10^{-7}\ n + 7.6\text{x}10^{-5} \qquad (2.9.4.1\text{-}4)$$

Here $n_{1/2}$=500 and r_∞=4.5 Mflops. The low-level RTL routine has at least an order of magnitude of improvement over the high-level RTL routine, and a factor of at least fifty improvement over the scalar execution.

It is interesting to note that although both the high-level and low-level RTL routines have similar start-up times the $n_{1/2}$ and r_∞ parameters are significantly different. A reason for this is that an increase in n causes a marked increase in the load/store time. This is enough to make the execution rate r_∞ of the low-level RTL routine much higher than the same high-level routine. A direct consequence of this is that $n_{1/2}$ will also be larger. Thus, a much higher vector length operation is needed to use the low-level routine effectively.

Table 2.9.4.1-1 gives the linear relationship for other arithmetic operations when using the scalar and pipeline processors. It is interesting to note that improvements in speed lie between a factor of 4 and a factor of 170.

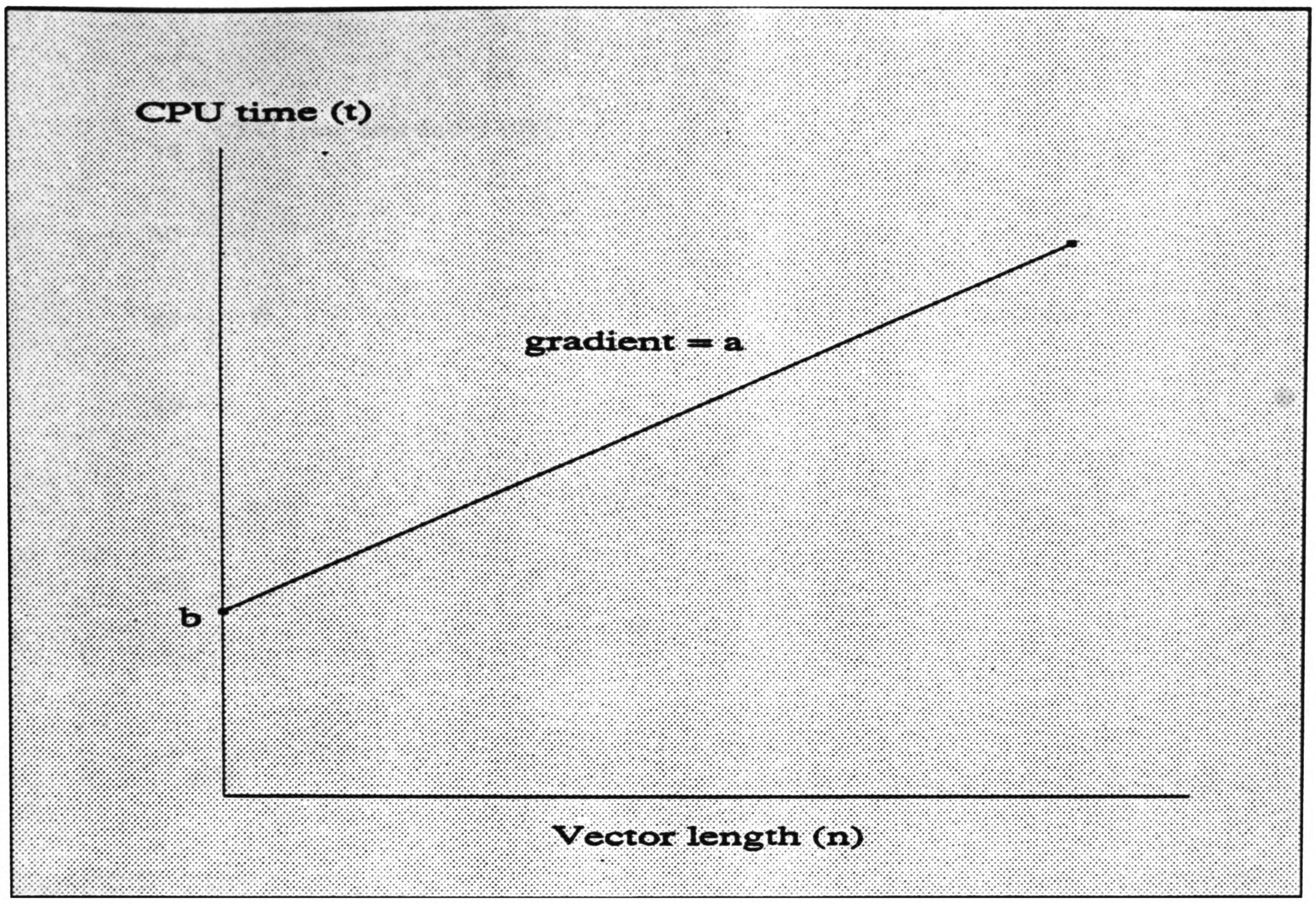


FIGURE 2.9.4.1-1 Relationship between vector size and CPU time

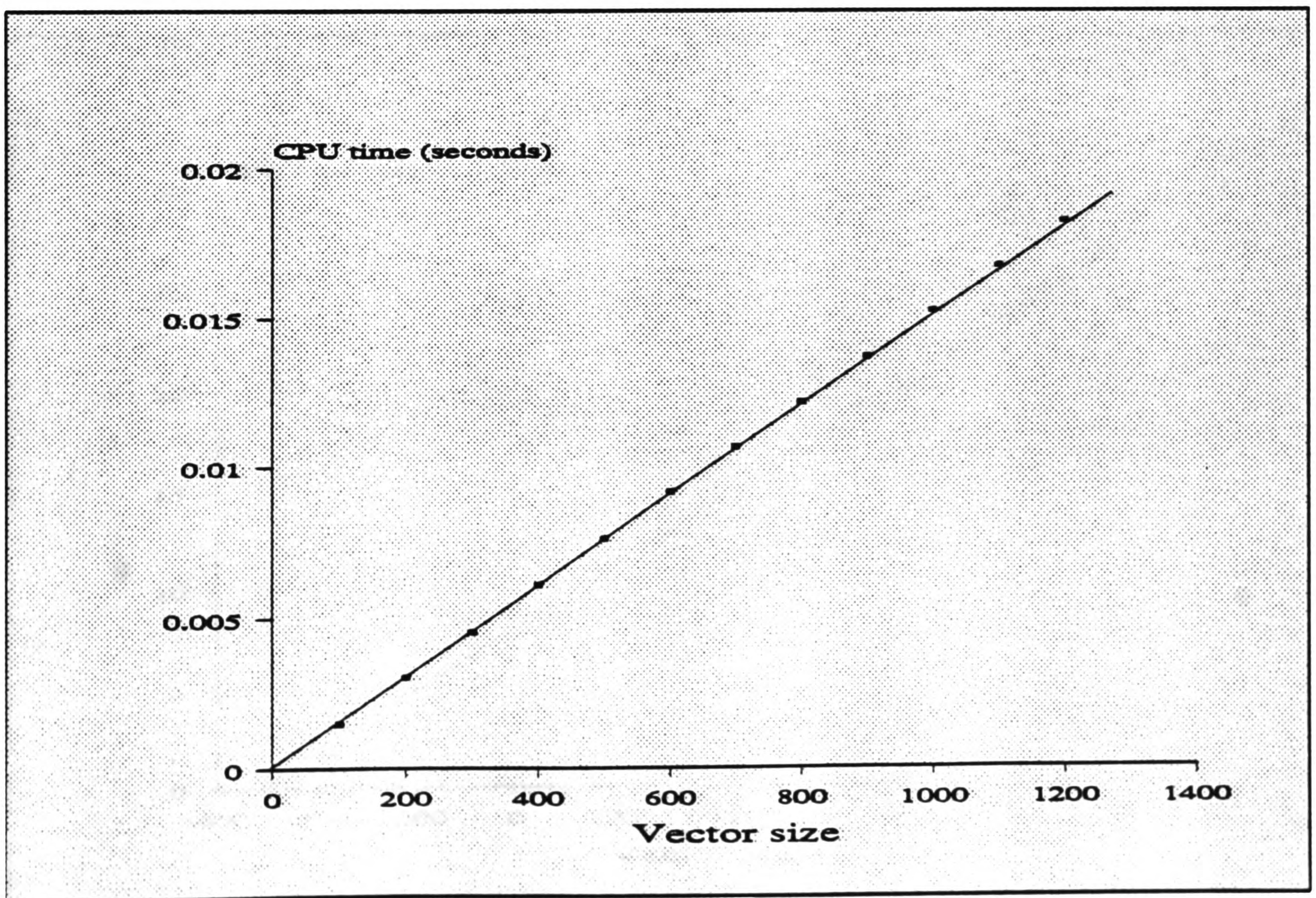


FIGURE 2.9.4.1-2 Measurement of $n_{1/2}$ and r_∞ using the scalar processor

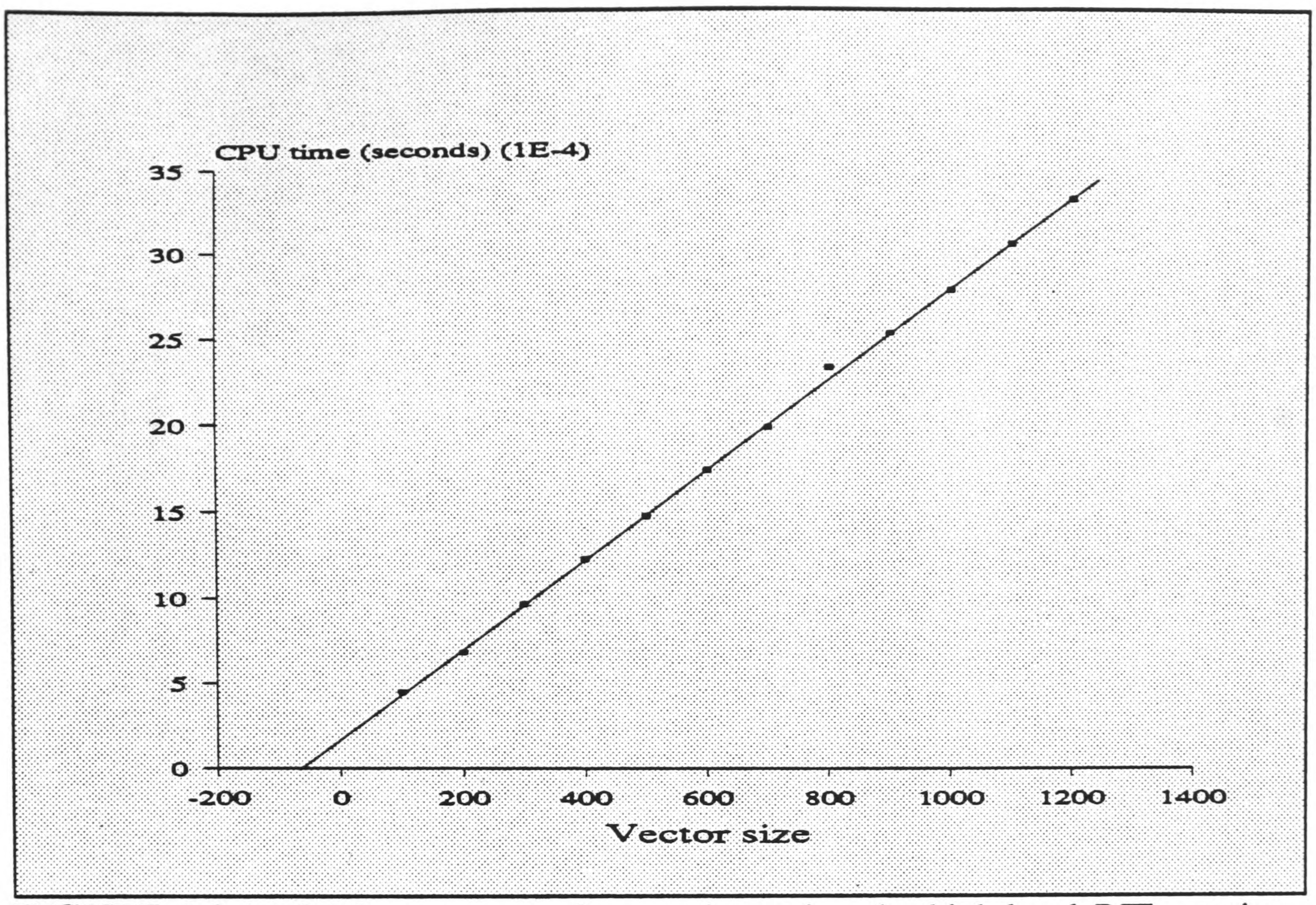


FIGURE 2.9.4.1-3 Measurement of $n_{1/2}$ and r_∞ using the high-level RTL routine

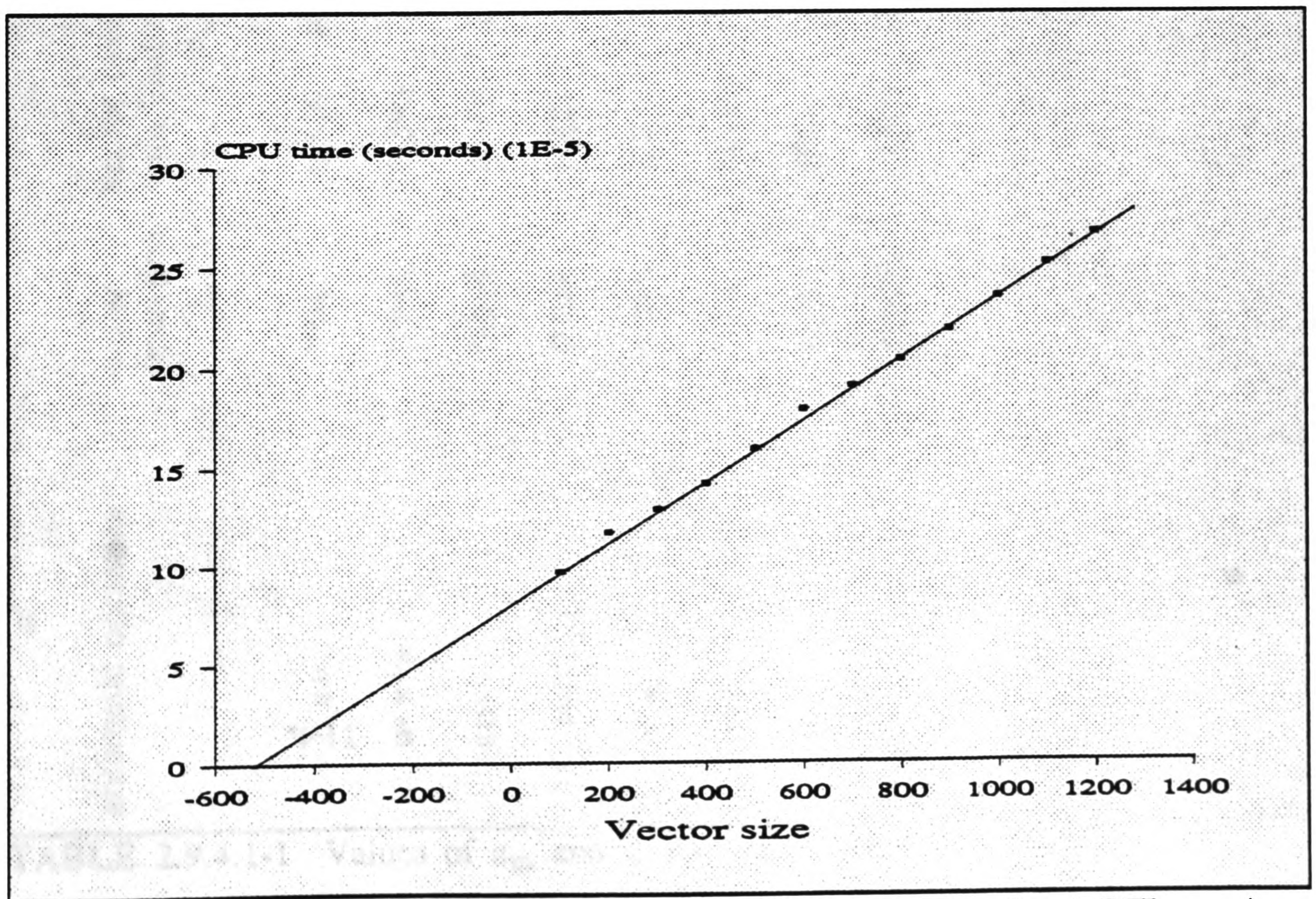


FIGURE 2.9.4.1-4 Measurement of $n_{1/2}$ and r_∞ using the low-level RTL routine

arithmetic operation	scalar operation	vector operation with load/store		vector operation without load/store		speed-up without load/store	speed-up with load/store
	r_∞	r_∞	$n_{1/2}$	r_∞	$n_{1/2}$		
$\sum_{i=1}^{n} u_i v_i$	0.041	0.549	109	7.1	705	173.2	13.4
u+v*w	0.045	0.290	73	6.6	722	146.7	6.4
au+v	0.049	0.380	89	4.5	485	91.8	7.8
$1/u_i$ $i=1(1)n$	0.066	0.424	98	1.4	270	21.2	6.4
u*v	0.080	0.348	60	4.5	500	56.3	4.35
u+v	0.080	0.348	60	4.5	500	56.3	4.35
-u	0.081	0.565	93	7.5	660	92.6	7.0
$\max_{i=1(1)n} \lvert u_i \rvert$	0.084	0.909	145	4.1	410	48.8	10.8

TABLE 2.9.4.1-1 Values of $n_{1/2}$ and r_∞ for some arithmetic operations

2.9.4.2 LINPACK performance

Another means of measuring the performance of the MC5400 is in the solution of a dense system of equations using the LINPACK software (Dongarra et al [1979]). The software was run in a FORTRAN environment using single precision arithmetic. The results (tables 2.9.4.2-1 and 2.9.4.2-2) show that the scalar processor performs at 0.085 Mflops and the pipeline processor at a rate of 0.25 Mflops. (These results are consistent with those quoted in section 2.9.4.1 for the high-level routines). The LINPACK results reflect the use of the high-level routines rather than the low-level routines. This is because all vectorised computations involved load/store operations which were unavoidable, since only the Basic Linear Algorithm Subroutines called BLAS (Lawson et al [1979]) were allowed modification. According to the LINPACK results, the scalar processor is over 140 times slower than a CRAY-1S, and the combination of the scalar and pipeline processors are up to 50 times slower than the CRAY-1S.

2.10 Expected gains in vectorisation of a program

Amdahl [1967] considers the situation where there are two processors, for example a scalar and a vector processor, with different execution rates. The scalar processor has an execution rate r_s and the pipeline processor a rate r_v. Further consider a program which has a total of I instructions. Then the scalar processor will execute these instructions in a time T_s given by

$$T_s = \frac{I}{r_s} \qquad (2.10\text{-}1)$$

```
   norm.re          resid           machep           x(1)            x(n)
1.54914737E+00   3.69101763E-05   1.19209290E-07   9.99986172E-01
9.99992490E-01


  times are reported for matrices of order   100
     sgefa        sgesl         total        mflops          units          ratio
 times for array with leading dimension of 201
  7.817E+00  2.333E-01  8.050E+00  8.530E-02  2.345E+01  1.437E+02
  7.850E+00  2.500E-01  8.100E+00  8.477E-02  2.359E+01  1.446E+02
  7.833E+00  2.333E-01  8.067E+00  8.512E-02  2.350E+01  1.440E+02
  7.832E+00  2.333E-01  8.065E+00  8.514E-02  2.349E+01  1.440E+02

 times for array with leading dimension of 200
  7.833E+00  2.333E-01  8.067E+00  8.512E-02  2.350E+01  1.440E+02
  7.817E+00  2.667E-01  8.083E+00  8.495E-02  2.354E+01  1.443E+02
  7.817E+00  2.333E-01  8.050E+00  8.530E-02  2.345E+01  1.438E+02
  7.822E+00  2.333E-01  8.055E+00  8.525E-02  2.346E+01  1.438E+02
```

TABLE 2.9.4.2-1 LINPACK results using the scalar processor

```
   norm.re          resid           machep           x(1)            x(n)
1.54914737E+00   3.69101763E-05   1.19209290E-07   9.99986172E-01
9.99992490E-01


  times are reported for matrices of order   100
     sgefa        sgesl         total        mflops          units          ratio
 times for array with leading dimension of 201
  2.683E+00  1.000E-01  2.783E+00  2.467E-01  8.107E+00  4.970E+01
  2.633E+00  8.333E-02  2.717E+00  2.528E-01  7.913E+00  4.851E+01
  2.700E+00  1.000E-01  2.800E+00  2.452E-01  8.155E+00  5.000E+01
  2.630E+00  9.000E-02  2.720E+00  2.525E-01  7.922E+00  4.857E+01

 times for array with leading dimension of 200
  2.633E+00  8.333E-02  2.717E+00  2.528E-01  7.913E+00  4.851E+01
  2.600E+00  1.000E-01  2.700E+00  2.543E-01  7.864E+00  4.821E+01
  2.600E+00  8.333E-02  2.683E+00  2.559E-01  7.816E+00  4.792E+01
  2.603E+00  8.833E-02  2.692E+00  2.551E-01  7.840E+00  4.807E+01
```

TABLE 2.9.4.2-2 LINPACK results using the scalar and pipeline processor

Using both the scalar and pipeline processors, the execution of the same I instructions can be carried out in a time T_{vs} given by

$$T_{vs} = I\left(\frac{f_s}{r_s} + \frac{f_v}{r_v}\right) \tag{2.10-2}$$

where f_s and f_v represent the fraction of instructions executed by the scalar and pipeline processors, respectively. Hence we have also

$$f_s + f_v = 1 \tag{2.10-3}$$

If we now define a speed-up factor (S) as the ratio of scalar to vector execution times

$$S = \frac{T_s}{T_{vs}} = \frac{I}{r_s} \cdot \frac{1}{I}\left(\frac{f_s}{r_s} + \frac{f_v}{r_v}\right)^{-1}$$

$$= \frac{1}{r_s}\left(\frac{r_s r_v}{f_s r_v + f_v r_s}\right) = \frac{r_v}{(1-f_v)r_v + f_v r_s}$$

$$= \frac{1}{(1-f_v) + f_v r_s / r_v}$$

$$S = \frac{1}{(1-f_v) + f_v/\tau} \tag{2.10-4}$$

where τ is the ratio of vector to scalar execution rates given by

$$\tau = \frac{r_v}{r_s} \tag{2.10-5}$$

Figure 2.10-1 shows the variation of S with f_v for two extreme ratio values $\tau=2$,

500. It also shows the graphs for the ratios of low-level routine speed to the scalar speed (τ=173.2) and high-level routine speed to the scalar speed (τ=4.35). It follows that all speed-up factors quoted from now on which compare the equivalent scalar and vector codes must lie within the shaded area.

2.11 Closure

Three different classification schemes for parallel architectures have been discussed. For a broad overview of these architectures Flynn's classification has emerged as the most popular. However, for a more complete breakdown Hockney's classification is recommended, especially as it clearly classes the pipeline processor.

It has been shown how pipeline vector processors attain their high performance relative to the scalar host processor. This has been typified by the MASSCOMP MC5400 machine. The questions of portability of the CFD code and general ease-of-use of the vector processor have been addressed. The result of which means that vectorised CFD codes developed on the MC5400 can be ported onto other machines with vector processing capabilities.

A means of measuring the effectiveness of vectorisation for any machine has been described. This facility can be used to predict the expected gains in speed of a vectorised CFD code.

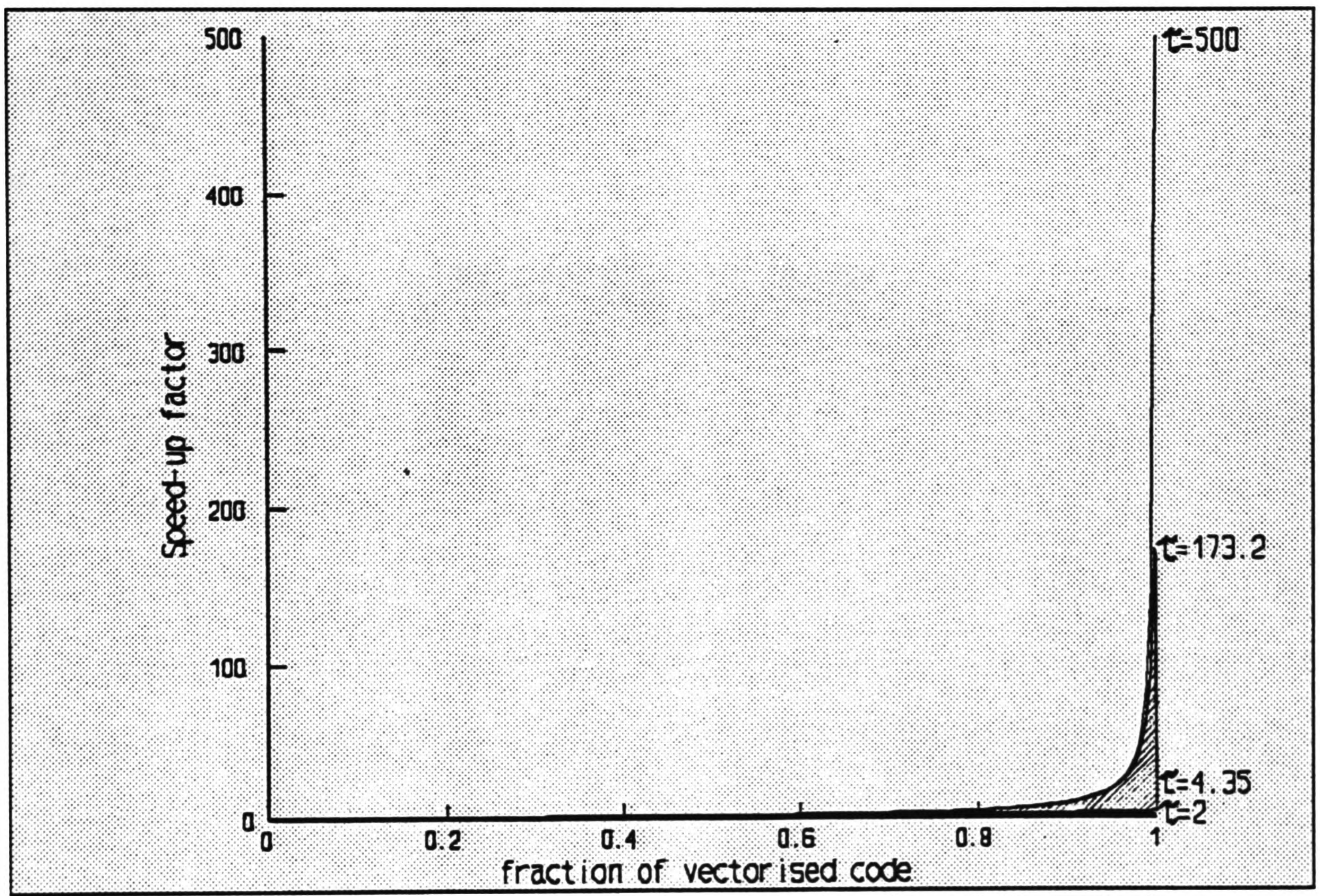


FIGURE 2.10-1 The speed-up achieved when a fraction of the code is vectorised for different execution ratios (τ)

CHAPTER THREE

3.0 SOLUTION PROCEDURES

3.1 Introduction

This chapter considers the general mathematical representation of the governing conservation laws and also a means of solving the resulting equations using a solution procedure. These equations are typically coupled and non-linear. The conservation laws can be expressed as a system of partial differential equations. These equations are presented, without loss of generality, to include laminar and turbulent flows, as well as steady state and transient situations. A number of different solution procedures exist which can be used to solve the resulting discretised equations. To date, the SIMPLE-based solution procedures (Patankar and Spalding [1972]) are amongst the most popular.

3.2 The governing differential equations

3.2.1 General conservation equation

The governing partial differential equations are a mathematical representation of the physical conservation laws of momentum, mass, enthalpy and other conserved fluid properties. With u denoting the x-direction velocity, the differential equation governing the conservation of momentum for a Newtonian fluid can be expressed as

$$\frac{\partial(\rho u)}{\partial t} + \mathrm{div}(\rho u u) = \mathrm{div}(\mu \,\mathrm{grad} u) + S_u - \frac{\partial p}{\partial x} \qquad (3.2.1\text{-}1)$$

Similar equations can be written for the v and w velocity components in the y and z-directions, respectively. The differential equation governing the conservation of momentum for a fluid is of particular importance and is expressed as

$$\frac{\partial \rho}{\partial t} + \mathrm{div}(\rho \mathbf{u}) = 0 \qquad (3.2.1\text{-}2)$$

This equation is also known as the continuity equation. As the name suggests, the net mass flux entering the system must balance that leaving the system to ensure continuity.

The differential equation governing the conservation of enthalpy in its general form has many contributions, however assuming Fick's law of diffusion this can be written in a compact form as

$$\frac{\partial}{\partial t}(\rho h) + \mathrm{div}(\rho \mathbf{u} h) = \mathrm{div}(K \mathrm{grad} T) + S_h \qquad (3.2.1\text{-}3)$$

It is apparent that equations (3.2.1-1) to (3.2.1-3) have the same basic structure and this is true of all conserved equations. These equations can be represented by a single general conservation equation given by

$$\frac{\partial}{\partial t}(\rho \phi) + \mathrm{div}(\rho \mathbf{u} \phi) = \mathrm{div}(\Gamma_\phi \mathrm{grad} \phi) + S_\phi \qquad (3.2.1\text{-}4)$$

Here ϕ is the dependent variable. The term $\frac{\partial}{\partial t}(\rho\phi)$ represents the transient or time rate of change, $\mathrm{div}(\rho \mathbf{u} \phi)$ represents the transportation of ϕ by convection, $\mathrm{div}(\Gamma_\phi \mathrm{grad}\phi)$ represents the contribution through diffusion of ϕ (Γ_ϕ is a diffusion coefficient) and finally, S_ϕ is a 'source' or 'sink' expression which contains any other contributions which do not fit easily into the other terms.

3.2.2 The discretisation of the general conservation equation

The solution of the differential equations typified by equation (3.2.1-4) is achieved by constructing a set of algebraic linear equations. The solution of such a set of equations provides a discrete representation of the continuous solution of the differential equation.

For a given problem the 'discretisation' of a domain can be carried out in a number of ways. Currently, the most popular numerical methods include the finite-element, finite-difference, and control-volume approaches. Roach [1982] provides a detailed discussion of these and other types of numerical methods. The numerical method employed in this study is the control-volume approach.

The domain of interest is subdivided using a finite number of control-volume rectangles which do not overlap. The grid is made up of orthogonal intersecting grid lines (figure 3.2.2-1). The intersection of these grid lines are called nodes, and all scalar variables are evaluated at these points. The u-velocity components are evaluated midway between two adjacent horizontal grid nodes, and similarly the v-velocity components are evaluated midway between two adjacent vertical grid nodes.

In this study the 'staggered-grid' approach of Harlow and Welch [1965] is adopted. The advantages of using a staggered-grid include the availability of velocities at the control-volume faces to evaluate flux values directly, and the simple evaluation of the pressure gradients as part of the discretisation of the

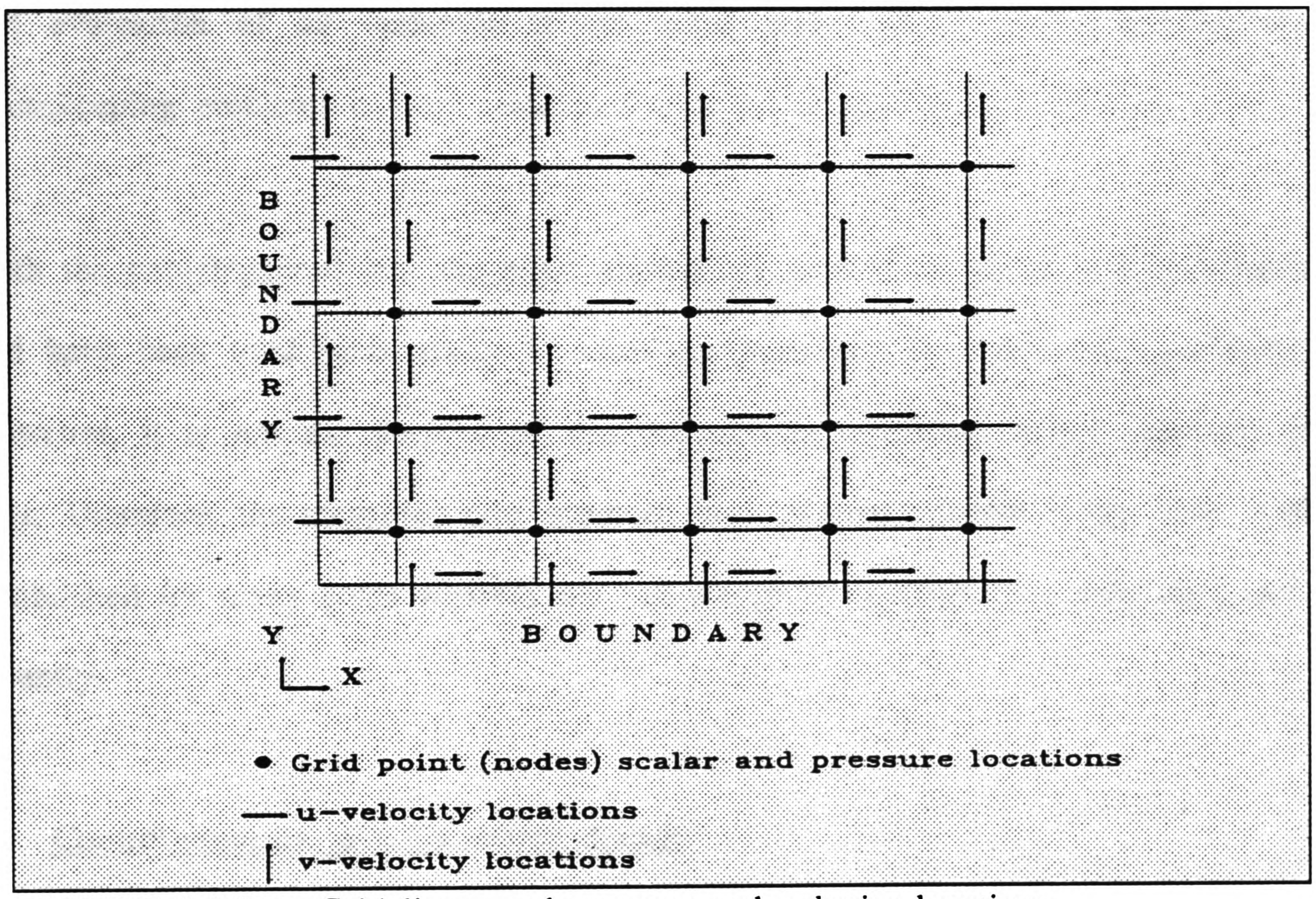


FIGURE 3.2.2-1 Grid lines, node centres and velocity locations momentum equations. The advantages of such an approach are appreciated when deriving the control-volume equations in section 3.4.

It should be noted that recently many workers such as Rhie and Chow [1983], Prakash and Patankar [1985], Jones et al [1985], Schneider and Raw [1987] and Lonesdale and Webster [1989] have successfully implemented a non-staggered or collocation method. The obvious attraction of collocation methods is the storage and evaluation of all variables at the same grid nodes, thus eliminating the housekeeping problems usually associated with the staggered-grid approach.

Both the non-staggered and staggered approaches have their merits and drawbacks and have been used successfully to solve computational fluid dynamics problems. However, it is not the purpose of this research to determine which approach is best. Therefore, the staggered-grid approach was adopted because it has been well established for many years whereas the non-staggered approach has been used only recently.

3.3 Control-volumes in a discretised domain

In the staggered-grid approach there are two main forms of control-volumes, those at general internal nodes and those situated near a boundary location. The general internal control-volumes for each of the dependent variables, namely the staggered velocity components u, v and the scalar and pressure variables ϕ are shown in figure 3.3-1. There are three distinct types of control-volumes which exist for near-boundary nodes and these are shown in figure 3.3-2. One of the drawbacks in adopting a staggered-grid approach is that in order to make the control-volume

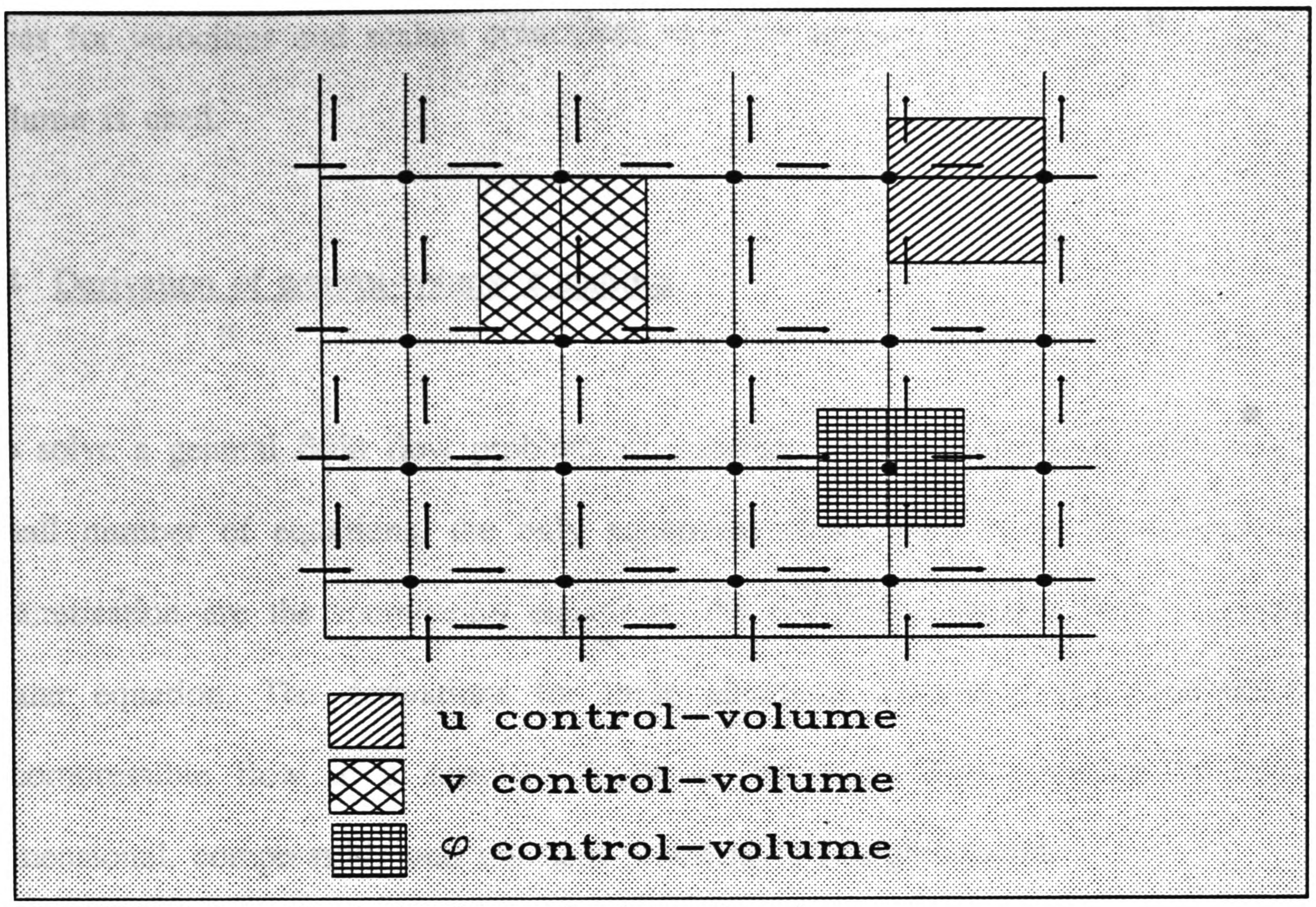


FIGURE 3.3-1 Control-volumes for internal nodes

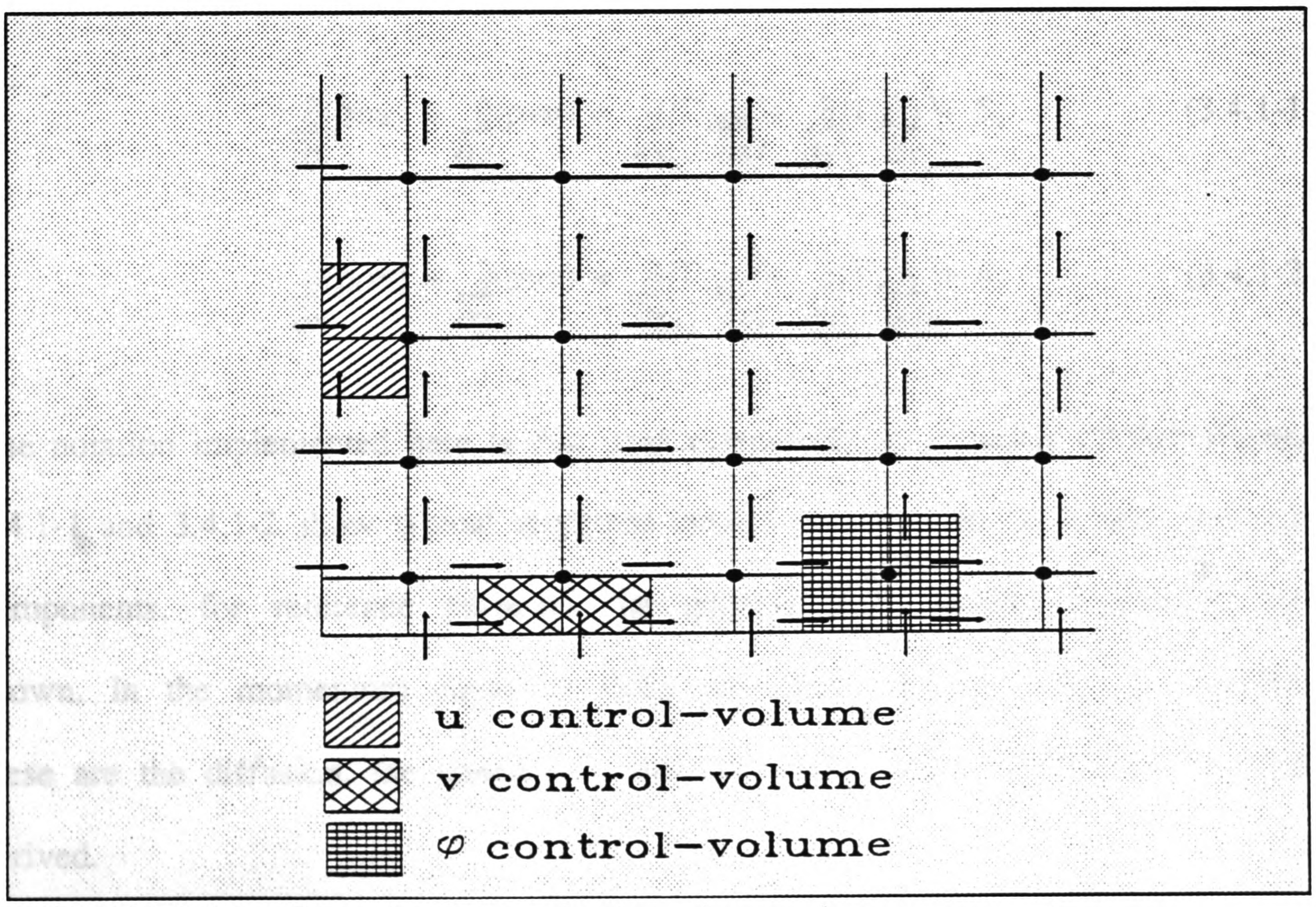


FIGURE 3.3-2 Control-volumes for boundary nodes faces for velocities and scalars coincident with the boundary nodes, a half control-volume is used.

3.4 Derivation of control-volume equations

To solve a general fluid flow problem all variables can be determined with only a small number of equations, the only equations which need to be considered for discretisation are the momentum equations, the continuity equation and a general scalar equation. The derivations for these equations assume a two-dimensional, incompressible flow in a cartesian frame of reference. The extension to three-dimensional, compressible and transient problems is straightforward.

3.4.1 The momentum equations

The momentum equations for the u and v-velocity components can be expressed as

$$\frac{\partial(\rho uu)}{\partial x} + \frac{\partial(\rho vu)}{\partial y} = \frac{\partial}{\partial x}\left(\Gamma_u \frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\Gamma_u \frac{\partial u}{\partial y}\right) + S_u \qquad (3.4.1\text{-}1)$$

$$\frac{\partial(\rho uv)}{\partial x} + \frac{\partial(\rho vv)}{\partial y} = \frac{\partial}{\partial x}\left(\Gamma_v \frac{\partial v}{\partial x}\right) + \frac{\partial}{\partial y}\left(\Gamma_v \frac{\partial v}{\partial y}\right) + S_v \qquad (3.4.1\text{-}2)$$

The notation implemented here is that used extensively by Patankar [1980]. Figures 3.4.1-1 and 3.4.1-2 show typical staggered control-volumes for the u and v-velocity components, for reference purposes the general scalar control-volume is also shown. In the momentum equations there are three separate terms to consider, these are the diffusion, the convection and the source term. These terms are now derived.

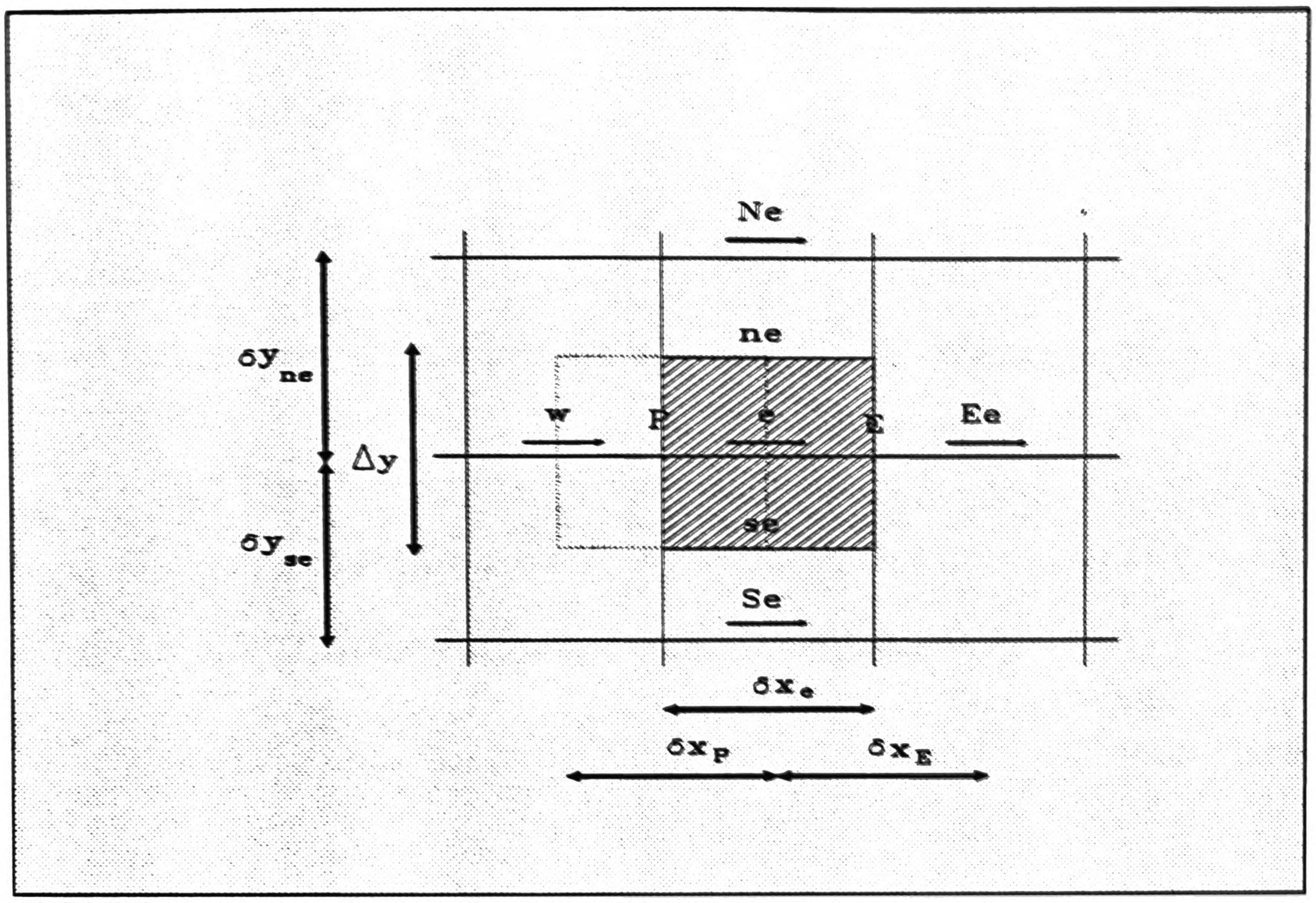


FIGURE 3.4.1-1 u-velocity control-volume

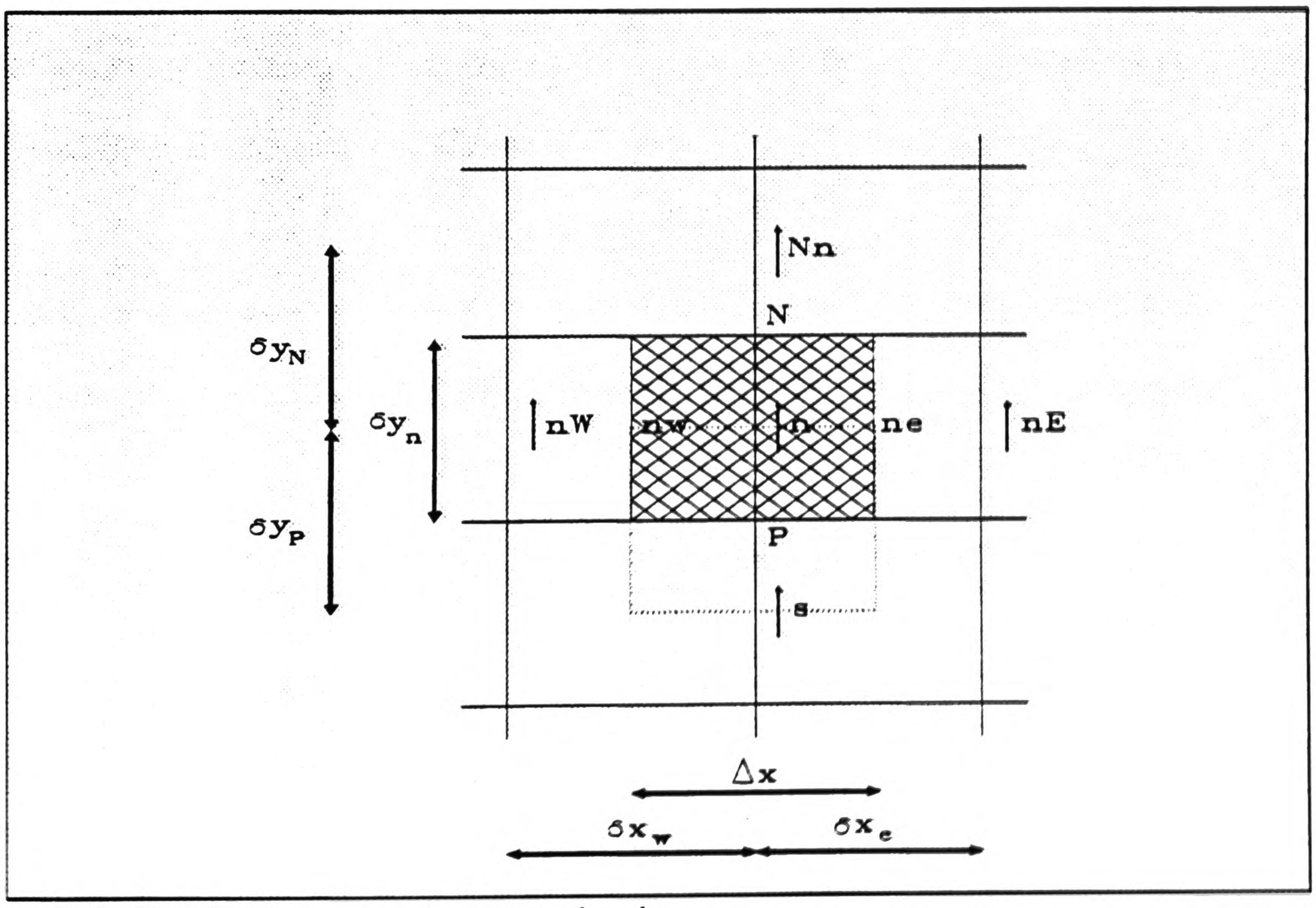


FIGURE 3.4.1-2 v-velocity control-volume

3.4.2 The diffusion term

The solution of the u-velocity diffusion component is obtained by integration of equation (3.4.1-1) over a control-volume e. This yields the following expression

$$\int_{x_P}^{x_E}\int_{y_{se}}^{y_{ne}} -\left\{\frac{\partial}{\partial x}\left(\Gamma_u\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(\Gamma_u\frac{\partial u}{\partial y}\right)\right\}dydx$$

$$= (D_E+D_P+D_{ne}+D_{se})u_e - (D_E u_{Ee}+D_P u_w+D_{ne}u_{Ne}+D_{se}u_{Se})$$

$$D_E = \frac{\Gamma_E\Delta y}{\delta x_E}$$

$$D_P = \frac{\Gamma_P\Delta y}{\delta x_P}$$

$$D_{ne} = \frac{\Gamma_{ne}\delta x_e}{\delta y_{ne}} \qquad (3.4.2\text{-}1)$$

$$D_{se} = \frac{\Gamma_{se}\delta x_e}{\delta y_{se}}$$

where the D's represent the flux due to diffusion across a given face. The evaluation of quantities at locations E, P, ne and se are determined using linear interpolation if they are not known.

A similar derivation process is used to determine the diffusion term for the v-velocity component over a control-volume n, this is given by the integration

$$\int_{x_{nw}}^{x_{ne}}\int_{y_S}^{y_N} -\left\{\frac{\partial}{\partial x}\left(\Gamma_v\frac{\partial v}{\partial x}\right)+\frac{\partial}{\partial y}\left(\Gamma_v\frac{\partial v}{\partial y}\right)\right\}dydx$$

$$= (D_{ne}+D_{nw}+D_N+D_P)v_n - (D_{ne}v_{nE}+D_{nw}v_{nW}+D_N v_{Nn}+D_P v_s)$$

$$D_{ne} = \frac{\Gamma_{ne}\delta y_n}{\delta x_e}$$

$$D_{nw} = \frac{\Gamma_{nw}\delta y_n}{\delta x_w}$$

(3.4.2-2)

$$D_N = \frac{\Gamma_N \Delta x}{\delta y_N}$$

$$D_P = \frac{\Gamma_P \Delta x}{\delta y_P}$$

3.4.3 The convection term

The solution of the u-velocity convection component is obtained by integration of equation (3.4.1-1) over a control-volume e. This yields the following expression

$$\int_{x_P}^{x_E}\int_{y_{se}}^{y_{ne}} \left[\frac{\partial(\rho uu)}{\partial x} + \frac{\partial(\rho vu)}{\partial y} \right] dydx$$

$$= (C_E+C_P+C_{ne}+C_{se})u_e - (C_E u_{Ee}+C_P u_w+C_{ne}u_{Ne}+C_{se}u_{Se})$$

$$C_E = \text{Max}\{-(\rho u)_E \Delta y,\ 0\}$$

$$C_P = \text{Max}\{(\rho u)_P \Delta y,\ 0\}$$

$$C_{ne} = \text{Max}\{-(\rho v)_{ne} \delta x_e,\ 0\}$$

$$C_{se} = \text{Max}\{(\rho v)_{se} \delta x_e,\ 0\}$$

(3.4.3-1)

In this study an upwind differencing scheme is used to determine the convection coefficients C. The C's describe the flux due to convection across a given control-volume face. A similar derivation can be carried out for the convection term of a v-velocity component over a control-volume n, and is given by

$$\int_{x_{nw}}^{x_{ne}}\int_{y_P}^{y_N} \left[\frac{\partial(\rho uv)}{\partial x} + \frac{\partial(\rho vv)}{\partial y} \right] dydx$$

$$= (C_{ne}+C_{nw}+C_N+C_P)v_n - (C_{ne}v_{nE}+C_{nw}v_{nW}+C_N v_{Nn}+C_P v_s)$$

$$C_{ne} = \text{Max}\{-(\rho u)_{ne} \delta y_n,\ 0\}$$

$$C_{nw} = \text{Max}\{(\rho u)_{nw} \delta y_n,\ 0\}$$

(3.4.3-1)

$$C_N = \mathrm{Max}\{-(\rho v)_N \Delta x,\ 0\}$$

$$C_P = \mathrm{Max}\{(\rho v)_P \Delta x,\ 0\}$$

For convection dominated flows, the method of approximating the convective term has been of interest for many years. It has been reported by many that the central differencing scheme is a poor approximation for such flows, for example Timin and Esmail [1983] and Patel [1987]. Other schemes have been devised such as the hybrid scheme of Spalding [1972], the skew-diffusion scheme of Raithby [1976], the Quadratic Upstream Interpolation for Convective Kinematics (QUICK) scheme of Leonard [1979], the Corner UPwInDing (CUPID) scheme of Patel et al [1988] and the Curvature Compensated Convective Transport (CCCT) scheme of Gaskell and Lau [1988]. The upwind difference scheme is by no means the 'best' scheme, but for the problems considered in this study the scheme performs well enough. One of the more elaborate schemes could have been used instead of the upwind difference scheme, however, in this study the investigation of the various schemes was not considered.

3.4.4 The source term

The final term in the u-momentum equation is obtained by integrating the source term over the control-volume e. This gives

$$\int_{x_P}^{x_E}\int_{y_{se}}^{y_{ne}} S_u \, dydx = S_u \delta x_e \Delta y \tag{3.4.4-1}$$

Using the linearisation procedure suggested by Patankar [1980] equation (3.4.4-1) becomes

$$\int_{x_P}^{x_E}\int_{y_{se}}^{y_{ne}} S_u \, dydx = (S_e^{u*} - (S_e^u)' u_e^*) + (S_e^u)' u_e$$

$$(S_e^u)' = \begin{cases} \dfrac{dS_e^u}{du_e} & \text{if } \dfrac{dS_e^u}{du_e} < 0 \\ 0 & \text{otherwise} \end{cases} \qquad (3.4.4\text{-}2)$$

where $(S_e^u)'$ is the gradient of the source term over the control-volume e, and the asterisk indicates known approximations. Similarly for the v-velocity component, the integration of the source term over the control-volume n is given by

$$\int_{x_{nw}}^{x_{ne}}\int_{y_P}^{y_N} S_v \, dydx = S_v \Delta x \delta y_n$$

$$= (S_n^{v*} - (S_n^v)' v_n^*) + (S_n^v)' v_n$$

$$(S_n^v)' = \begin{cases} \dfrac{dS_n^v}{dv_n} & \text{if } \dfrac{dS_n^v}{dv_n} < 0 \\ 0 & \text{otherwise} \end{cases} \qquad (3.4.4\text{-}3)$$

3.4.5 Continuity equation

The continuity equation ensures mass conservation of the fluid and is given by

$$\frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} = 0 \qquad (3.4.5\text{-}1)$$

Integrating the continuity equation over a control-volume P (figure 3.4.5-1) gives

$$\int_{x_w}^{x_e}\int_{y_s}^{y_n} \left[\frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} \right] dydx = 0$$

$$\Rightarrow ((\rho u)_e - (\rho u)_w)\Delta y + ((\rho v)_n - (\rho v)_s)\Delta x = 0 \qquad (3.4.5\text{-}2)$$

Using the velocity-correction formulae of Patanakar [1980]

$$u_e = u_e^* + d_e(p_P' - p_E') \qquad (3.4.5\text{-}3a)$$

$$v_n = v_n^* + d_n(p_P' - p_N') \qquad (3.4.5\text{-}3b)$$

where u^* is the velocity to be corrected, p_P' is the correction to pressure at P and d_e is given by

$$d_e = \frac{A_e}{a_e} \qquad (3.4.5\text{-}4)$$

where A_e is the area of the control-volume face and a_e is the diffusion+convection coefficient at e. Substituting (3.4.5-3) into (3.4.5-2) gives

$$((\rho u)_e - (\rho u)_w)\Delta y + ((\rho v)_n - (\rho v)_s)\Delta x - (\rho d)_e p_E'\Delta y - (\rho d)_w p_W'\Delta y - (\rho d)_n p_N'\Delta x$$
$$- (\rho d)_s p_S'\Delta x + ((\rho d)_e\Delta y + (\rho d)_w\Delta y + (\rho d)_n\Delta x + (\rho d)_s\Delta x)p_P' = 0 \qquad (3.4.5\text{-}5)$$

3.4.6 The scalar equation

The general scalar ϕ equation is representative of variables such as enthalpy (h), kinetic energy (k) and dissipation rate (ε) as used in turbulence modelling. The equation has the form

$$\frac{\partial(\rho u\phi)}{\partial x} + \frac{\partial(\rho v\phi)}{\partial y} = \frac{\partial}{\partial x}\left(\Gamma_\phi\frac{\partial\phi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\Gamma_\phi\frac{\partial\phi}{\partial y}\right) + S_\phi \qquad (3.4.6\text{-}1)$$

and is integrated over the control-volume P in a manner similar to that for the continuity equation (figure 3.4.5-1). Like the momentum equations, the scalar equation has a diffusion, convection and source term, and these are now presented.

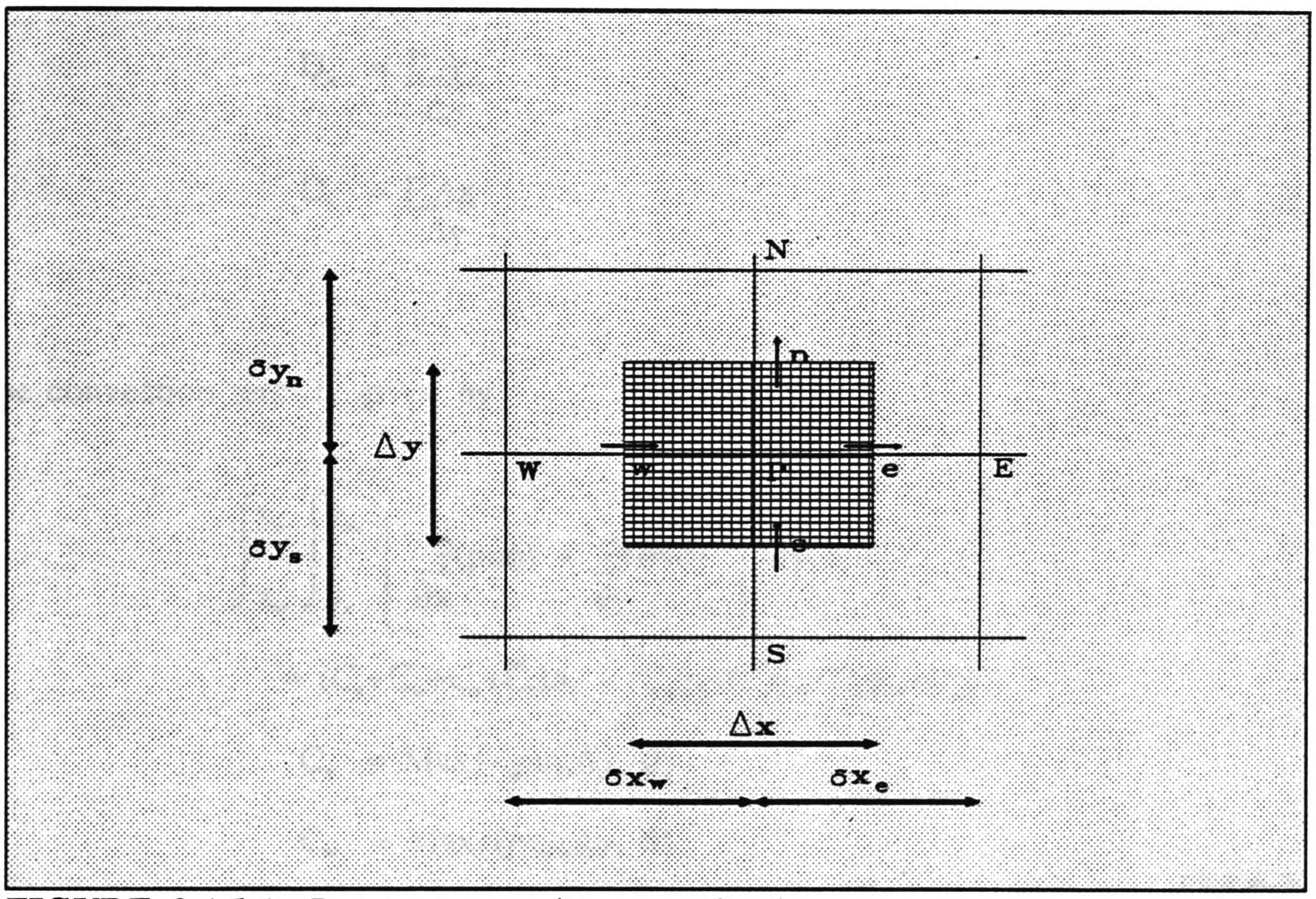


FIGURE 3.4.5-1 Pressure-correction control-volume

The diffusion term is given by

$$\int_{x_w}^{x_e}\int_{y_s}^{y_n} -\left\{\frac{\partial}{\partial x}\left(\Gamma_\phi\frac{\partial\phi}{\partial x}\right)+\frac{\partial}{\partial y}\left(\Gamma_\phi\frac{\partial\phi}{\partial y}\right)\right\} dydx$$

$$= (D_e+D_w+D_n+D_s)\phi_P - (D_e\phi_E+D_w\phi_W+D_n\phi_N+D_s\phi_S)$$

$$D_e = \frac{\Gamma_e\Delta y}{\Delta x}$$

$$D_w = \frac{\Gamma_w\Delta y}{\Delta x}$$

$$D_n = \frac{\Gamma_n\Delta x}{\Delta y} \qquad (3.4.6\text{-}2)$$

$$D_s = \frac{\Gamma_s\Delta x}{\Delta y}$$

The convection term is given by

$$\int_{x_w}^{x_e}\int_{y_s}^{y_n} \left\{\frac{\partial(\rho u\phi)}{\partial x}+\frac{\partial(\rho v\phi)}{\partial y}\right\} dydx$$

$$= (C_e+C_w+C_n+C_s)\phi_P - (C_e\phi_E+C_w\phi_W+C_n\phi_N+C_s\phi_S)$$

$$C_e = \mathrm{Max}\{-(\rho u)_e\Delta y,\ 0\}$$

$$C_w = \mathrm{Max}\{(\rho u)_w\Delta y,\ 0\}$$

$$C_n = \mathrm{Max}\{-(\rho v)_n\Delta x,\ 0\} \qquad (3.4.6\text{-}3)$$

$$C_s = \mathrm{Max}\{(\rho v)_s\Delta x,\ 0\}$$

Finally, the integration of the source term is given by

$$\int_{x_w}^{x_e}\int_{y_s}^{y_n} S_\phi\ dydx = S_\phi\Delta x\Delta y$$

$$= (S_P^* - (S_P^*)'\phi_P^*) + (S_P^*)'\phi_P$$

$$(S_P^*)' = \begin{cases} \dfrac{dS_P^*}{d\phi_P} & \text{if } \dfrac{dS_P^*}{d\phi_P} < 0 \\ 0 & \text{otherwise} \end{cases} \qquad (3.4.6\text{-}4)$$

3.4.7 The final control-volume equations

The discretised diffusion, convection and source terms when substituted into the general equation (3.2.1-4) gives an overall control-volume equation of the form

$$A_P\phi_P = \Sigma A_{nb}\phi_{nb} + S \qquad (3.4.7\text{-}1)$$

where A_{nb} represents the coefficients for the four neighbouring nodes based on convection and diffusion contributions and S represents the source term contributions.

It is important that the physics inherent in the original differential equations is preserved when using the control-volume approach. Conservation must be satisfied not only for each individual control-volume but also for the entire domain. Therefore, as well as ensuring continuity at each control-volume the flux continuity between two adjoining faces must be continuous. Ensuring that all coefficients are positive and satisfy the inequality

$$A_P \geq \Sigma A_{nb} \qquad (3.4.7\text{-}2)$$

then the discretised equations will also satisfy the boundedness property. To prevent violation of this inequality through the linearisation of the source term, the gradient is always negative. This condition gives rise to a diagonally dominant system of equations which can be solved using either direct or iterative algorithms.

3.5 Solution procedures

A solution procedure provides a strategy for solving the system of coupled, non-linear discretised equations. Probably the most widely used solution procedure is based on the original work of Patankar and Spalding [1972], and is referred to as SIMPLE (Semi-Implicit Method for Pressure Linked Equations). As the name suggests, the method solves the control-volume equations by de-coupling the pressure-linked equations using iteration. Many variations of the SIMPLE procedure exist, some of the more widely used variations include CTS SIMPLE (Raithby et al [1979, 1980]); SIMPLER (Patankar [1980, 1981]); SIMPLEST (Spalding [1980]); SIMPLEC (Van Doormaal and Raithby [1984]); PISO (Issa [1986]); FIMOSE (Latimer and Pollard [1985]) and IMPLE (Wang et al [1989]).

3.5.1 The SIMPLE solution procedure

The main computational steps for the SIMPLE procedure are given below:

(1) Guess the initial velocity and pressure fields u, v and p^*

(2) Solve the momentum equations (section 3.4.1) to give an approximation to the velocity fields u^* and v^*. These solutions are based on the guessed pressure field.

(3) Solve for a pressure-correction field p' (formulated from the continuity equation, section 3.4.5).

(4) The pressure-correction field is used to correct the velocity approximations using the velocity-correction formulae (3.4.5-3).

(5) The pressure-correction field is also used to correct the pressure field.

(6) At this stage any scalar variables are solved (section 3.4.6).

(7) Check to see if the solutions obtained for all variables satisfy the convergence criteria, if not, steps (2)-(6) are repeated until convergence has been achieved.

In general, mass errors exist when solving the velocity fields in step (2). These errors define the source term in the continuity equation and ideally should be zero. The purpose of the correction to the velocity fields in step (5) is to eliminate the continuity errors. Thus the SIMPLE procedure attains convergence through a series of iterations, where at the end of each iteration, the velocity fields satisfy continuity.

Since there usually exists a strong coupling between the differential equations, it is often necessary to exercise some form of relaxation to achieve a converged solution. The relaxation employed here is a linear under-relaxation of the form

$$\phi^{new} = \alpha_\phi \phi^{new} + (1-\alpha_\phi)\phi^{old} \qquad (3.5.1\text{-}1)$$

and is applied to all variables except pressure. The pressure field is under-relaxed as

$$p^{new} = p^{old} + \alpha_p p' \qquad (3.5.1\text{-}2)$$

where the relaxation parameters α_ϕ and α_P are positive and usually less than 1.

3.5.2 <u>The SIMPLEC solution procedure</u>

In deriving the velocity-correction formula (3.4.5-3a) for the u velocity component an assumption was made that the term $\Sigma a_{nb}u'_{nb}$ was negligible (Patankar [1980]). However, Van Doormaal and Raithby [1984] realised that ignoring this term while the left-hand-side of the equation retains a term of comparable magnitude makes the formulation inconsistent. To introduce a consistent approximation, the term $\Sigma a_{nb}u'_{nb}$ is not neglected, but a term of similar magnitude $\Sigma a_{nb}u'_{e}$ is subtracted from both sides. The momentum equation for the u velocity component now becomes

$$(a_e - \Sigma a_{nb})u'_e = \Sigma a_{nb}(u'_{nb} - u'_e) + A_e(p'_P - p'_E) \qquad (3.5.2\text{-}1)$$

and the term $\Sigma a_{nb}(u'_{nb} - u'_e)$ is neglected making SIMPLE <u>C</u>onsistent, hence the name SIMPLEC. The coefficient d_e in the velocity-correction formula is no longer given by (3.4.5-4) but is modified to

$$d_e = \frac{A_e}{a_e - \Sigma a_{nb}} \qquad (3.5.2\text{-}2)$$

(The momentum equation for the v velocity component is modified in a similar manner). Therefore, the sequence of steps in SIMPLEC are identical to the steps in SIMPLE with the following modifications:

(a) The d's are now defined by expressions such as (3.5.2-2) and are used in equations (3.4.5-3) and (3.4.5-5)

(b) There is no relaxation of the pressure-correction field when updating the pressure field (3.5.1-2) i.e. $\alpha_P=1$.

3.5.3 The CTS SIMPLE (Consistent Time Step) solution procedure

There is a single difference between this solution procedure and SIMPLE, this is the way the relaxation parameters α_P and α_ϕ are defined. In CTS SIMPLE these parameters are defined as

$$\alpha_P = \frac{1}{1 + E} \qquad (3.5.3\text{-}1)$$

$$\alpha_\phi = \frac{E}{1 + E} \qquad (3.5.3\text{-}2)$$

where E represents a distorted time step. This is done so that α_P is consistent with the time step used in the momentum equations.

3.5.4 The SIMPLER solution procedure

One of the drawbacks of SIMPLE is that the pressure-correction field although over-estimated, is of the right order of magnitude to effectively correct the velocity fields but not the pressure field. Therefore SIMPLE can be modified by introducing a separate equation to evaluate the pressure field. This equation is based on pseudo-velocities (Patankar [1980]), for example the u velocity component has pseudo-velocities such as

$$\hat{u}_e = \frac{\sum a_{nb} u'_{nb} + b}{a_e} \qquad (3.5.4\text{-}1)$$

The pressure field equation is similar to the pressure-correction equation (3.4.5-5) except that the starred velocities are replaced by pseudo-velocities.

The main computational steps of SIMPLER are given below:

(1) Guess the initial velocity and pressure fields.

(2) Generate the pseudo-velocity fields $\hat{u}$ and $\hat{v}$ typified by (3.5.4-1).

(3) Solve for the pressure field p^* (similar to the p' equation (3.4.5-5), except pseudo-velocities are used to define the mass source term).

(4) Solve the momentum equations (section 3.4.1) to give an approximation to the velocity fields u^* and v^*. These solutions are based on the pressure field from step (3).

(5) Solve for a pressure-correction field p' (formulated from the continuity equation, section 3.4.5).

(6) The pressure-correction field is used to correct the velocity approximations using the velocity-correction formulae (3.4.5-3).

(7) At this stage any scalar variables are solved (section 3.4.6).

(8) Check to see if the solutions obtained for all variables satisfy the convergence criteria, if not, steps (2)-(7) are repeated until convergence has been achieved.

3.5.5 <u>The IMPLE solution procedure</u>

Wang et al [1989] describe a "better procedure than SIMPLER". The essence of IMPLE is to update the velocity fields using a formulation based on density-correction rather than the conventional pressure-correction. In general, the starred velocities do not satisfy the continuity equation, and this is described by a mass error term b given by

$$b = ((\rho u^*)_e - (\rho u^*)_w)\Delta y + ((\rho v^*)_n - (\rho v^*)_s)\Delta x \qquad (3.5.5\text{-}1)$$

A 'time-dependent' term A is introduced such that the continuity equation is satisfied, and is given by

$$A - b = 0 \qquad (3.5.5\text{-}2)$$

$$A = \frac{\rho_P' dV}{dt} \qquad (3.5.5\text{-}3)$$

$$\Rightarrow \quad \rho_P' = \frac{b\,dt}{dV} \qquad (3.5.5\text{-}4)$$

where ρ_P' is the density correction, dV is the control-volume and dt is the time step in the iteration procedure. This now defines a density correction field. The velocity-correction formulae are also modified and are given by

$$u_e = u_e^{*} + \frac{(\rho_P' - \rho_E')}{\rho_e^{*}} | u_e^{*} | \qquad (3.5.5\text{-}5a)$$

$$v_n = v_n^{*} + \frac{(\rho_P' - \rho_N')}{\rho_n^{*}} | v_n^{*} | \qquad (3.5.5\text{-}5b)$$

The sequence of steps in IMPLE are similar to those in SIMPLER with the exception of steps (5) and (6), these now become

(5) Calculate the density-correction field using (3.5.5-1) and (3.5.5-4)

(6) Correct the velocity approximations using the velocity-correction formulae (3.5.5-5)

3.5.6 The PISO solution procedure

The PISO solution procedure achieves the solution by a series of time-marching steps. Each time step consists of one predictor and one corrector step for the pressure field, and one predictor and two corrector steps for the velocity fields.

The momentum equation for the u velocity component in section 3.4.1 can be written as

$$a_e u_e^{*} = \Sigma a_{nb} u_{nb}^{*} + A_e(p_P^{i} - p_E^{i}) + b_e^{i} \qquad (3.5.6\text{-}1)$$

where * represents the predictor approximations and ^{i} represents the converged solution at the previous time step. This equation is updated by the first corrector approximation (denoted by **) and is given by

$$a_e u_e^{**} = \Sigma a_{nb} u_{nb}^{*} + A_e(p_P^{*} - p_E^{*}) + b_e^{i} \qquad (3.5.6\text{-}2)$$

and the correction equation is obtained by subtracting (3.5.6-1) from (3.5.6-2) to give

$$u_e^{**} = \hat{u}_e + d_e(p_P^{*} - p_E^{*}) \qquad (3.5.6\text{-}3a)$$

$$\hat{u}_e = u_e^{*} - d_e(p_P^{i} - p_E^{i}) \qquad (3.5.6\text{-}3b)$$

A similar set of prediction and correction equations are defined for the v velocity components. The approximations u^{**} and v^{**} are used to formulate the pressure equation from the continuity equation. This is the predictor step for the pressure field.

If the momentum equation for the u velocity component is now expressed as

$$a_e u_e^{***} = \Sigma a_{nb} u_{nb}^{**} + A_e(p_P^{**} - p_E^{**}) + b_e^{i} \qquad (3.5.6\text{-}4)$$

where *** denotes the second corrector approximation, then the second corrector equation is defined by subtracting (3.5.6-4) from (3.5.6-2), this gives

$$u_e^{***} = \hat{u}_e + d_e(p_P^{**} - p_E^{**}) \qquad (3.5.6\text{-}5a)$$

$$\hat{u}_e = u_e^{*} + \frac{[\sum a_{nb}(u_{nb}^{**} - u_{nb}^{*})]}{a_e} - d_e(p_P^{*} - p_E^{*}) \qquad (3.5.6\text{-}5b)$$

The approximations u^{***} and v^{***} are used to formulate the pressure equation as before, and this is then the corrector step for the pressure field.

The main computational steps for PISO are given below:

(1) Using the previous time step solution u^i, v^i and p^i the momentum equations (3.5.6-1) are solved to give u^* and v^*.

(2) Calculate the coefficients for the pressure equation and hence solve for the pressure field p^*.

(3) Correct the velocity fields (i.e. equation (3.5.6-3)) to give u^{**} and v^{**}.

(4) Using the corrected velocities, calculate the coefficients and then solve for the corrected pressure field p^{**}.

(5) Correct the velocity fields (i.e. equation (3.5.6-5)) to give u^{***} and v^{***}.

3.5.7 The FIMOSE solution procedure

The basic concept of FIMOSE (Fully Implicit Method for Operator-Split Equations) is to de-couple the pressure-velocity link so that variables are dealt with one at a time. It is interesting to note that in FIMOSE no pressure-correction equation is used.

Using a mass flow rate formulation, a velocity block-correction can be defined such that

$$\bar{u}' = \frac{\dot{m}_{in} - \sum \rho u_i^* A_i}{\rho.(\text{flow area})} \qquad (3.5.7\text{-}1)$$

where $\bar{u}'$ is the velocity correction added to each velocity along a constant grid line, $\dot{m}_{in}$ is the mass flow rate at the inlet, u_i^* is the current velocity approximation and A_i is the area of control-volume i. The change in the pressure gradient (Δp) is given by

$$\Delta p = \frac{\dot{m}_{in} - \sum \rho u_i^* A_i}{\sum \rho d_i A_i} \qquad (3.5.7\text{-}2)$$

and this is added to all grid nodes to maintain the correct mass flow rate.

The main computational steps for the FIMOSE solution procedure are given below:

(1) Guess a velocity field which satisfies the continuity equation, and also a pressure field.

(2) Calculate the momentum coefficients and solve for the velocity approximations u^* and v^* (section 3.4.1).

(3) To ensure continuity is satisfied, the velocities are corrected using equations (3.5.7-1) and (3.5.7-2).

(4) Calculate pseudo-velocities (3.5.4-1) and hence solve for the pressure field p^* (3.4.5-5.).

(5) Using the pressure field p^* solve the momentum equations again to give u^{**} and v^{**}.

(6) Apply the velocity corrections as defined by equations (3.5.7-1) and (3.5.7-3), this ensures a divergence-free velocity field.

(7) Re-calculate pseudo-velocities (3.5.4-1) and hence determine the pressure field p^{**}.

(8) Using the new pressure field solve the momentum equations again to give u^{***} and v^{***}.

(9) Apply the velocity corrections as defined by equations (3.5.7-1) and (3.5.7-3), this ensures a divergence-free velocity field.

(10) If convergence has not been achieved use u^{***}, v^{***} and p^{***} as the latest approximations and return to step (2).

3.5.8 The SIMPLEST solution procedure

This procedure can be applied to all of the above, however, there is no guarantee that it will improve the performance of the solution procedure it is applied to. Spalding [1980] recognised that as the discretised grid was made finer the performance of SIMPLE deteriorated. This was traced to the dominance of the convection terms in the momentum equations, and placing these terms into the right-hand-side with other source terms generally improved the convergence rate of SIMPLE. To acknowledge this modification the procedure is referred to as SIMPLEST (SIMPLE-ShorTened).

3.6 Implementation of the SIMPLE family

The choice of implementation of SIMPLE is of particular interest in this study. An efficient implementation is desired for the scalar processor, but also an implementation is needed which will allow for vectorisation. Two different implementations which have been extensively used are described below. Patankar [1980] does not suggest any specific implementation for SIMPLE, moreover the choice of implementation is left to the programmer.

3.6.1 The NEAT approach

The NEAT (Nearly Exact Adjustment of Terms) approach was suggested by Spalding [1976] and is basically a line technique where all variables u, v, p and ϕ

are solved for in turn for a given constant grid line, and the most recent information from the neighbouring grid lines can be used. Therefore equation (3.4.7-1) is re-arranged for a constant x grid line and is given by

$$A_P\phi_P - A_N\phi_N - A_S\phi_S = A_E\phi_E - A_W\phi_W + S \qquad (3.6.1\text{-}1)$$

The resulting tridiagonal matrix is then solved. In addition to solving all the variables on a line, a block correction (Settari and Aziz [1973]) is carried out to enhance convergence. When all lines have been visited in turn then one SIMPLE iteration has been carried out.

The NEAT approach has been used by Pun and Spalding [1976] as part of the CHAMPION series codes and also by Patankar [1981].

3.6.2 The whole-field pressure-correction approach

In the whole-field pressure-correction approach the variables u, v, p and ϕ are solved in turn for the entire calculation domain. Therefore information about the neighbouring grid nodes are needed in this formulation. In this approach the pressure-correction field is solved to a much higher degree of convergence within each SIMPLE iteration. Chapter 4 describes how such an approach can lead to the solution of either a tridiagonal matrix system of equations defined by (3.6.1-1), or a pentadiagonal system of equations defined by (3.4.7-1). Therefore, this approach is more flexible than the NEAT approach. It will be shown in Chapter 4 that the system of equations resulting from the whole-field approach are solved more efficiently on the VA-1 processor than those resulting from the NEAT approach.

Therefore, the whole-field pressure-correction approach is preferred in the implementation of the SIMPLE procedure.

3.7 Choice of solution procedure

It is extremely difficult to suggest the best solution procedure of those presented. Attempts to compare some of these procedures have been carried out by many and is still an area of great interest.

Many similarities between these procedures exist, for example, Van Doormaal and Raithby [1984] state that when the linearised term $(S_e^u)'$ (3.4.4-2) is zero SIMPLEC becomes identical to CTS SIMPLE. Latimer and Pollard [1985] state that PISO is similar in many ways to FIMOSE. Also, the first prediction and correction steps of PISO are identical to SIMPLE, and that PISO and SIMPLER are very similar.

Jang et al [1986] carried out a comparison of SIMPLEC, SIMPLER and PISO. In their conclusions they state that if the coupling between the momentum and scalar equations is weak or non-existent, then PISO is more efficient and stable than either SIMPLEC or SIMPLER. However, for problems where the coupling is strong, then SIMPLEC and SIMPLER have a similar performance and are more efficient than PISO.

Wang et al [1989] show that IMPLE can be more efficient than SIMPLE and has a performance similar to SIMPLER. However, Ierotheou et al [1988] have shown that in some cases SIMPLE can be more efficient than SIMPLER. For example, the solution of the 'cavity with moving-lid' problem (Chapter 5) was obtained using the SIMPLE and SIMPLER procedures. The results show that the SIMPLE procedure is up to five times more efficient than SIMPLER. Both procedures were implemented using the whole-field approach, in addition, the SIMPLEST procedure was also applied to both SIMPLE and SIMPLER. The SIMPLE procedure modified to include SIMPLEST is referred to as SIMPLEST$_1$ and the SIMPLER procedure modified to include SIMPLEST is referred to as SIMPLEST$_2$. Figure 3.7-1 shows the CPU time taken to solve the problem with a Reynolds number Re=100 for a number of uniform (*n*x*n*) grids.

No single procedure has emerged as the 'best' and there are many reasons for this. These include the characteristics of the problem being solved, more specifically, the coupling of the equations, the boundary conditions and grid size all have a bearing on which procedure performs most efficiently. The particular implementation of the solution procedure, whether it is the NEAT or whole-field approach, will have a significant effect on the performance. In this study SIMPLE is implemented with a whole-field approach. While it is recognised that it is by no means the most efficient procedure it is found to be sufficient for the test problems studied and also allows for a straightforward extension to the other solution procedures if necessary. Furthermore, the whole-field approach will allow for an efficient implementation on the VA-1 processor.

3.8 Closure

In this chapter the control-volume approach was presented and used to describe the discrete representation of the governing partial differential equations. The control-volume formulations for the momentum, continuity and scalar equations were

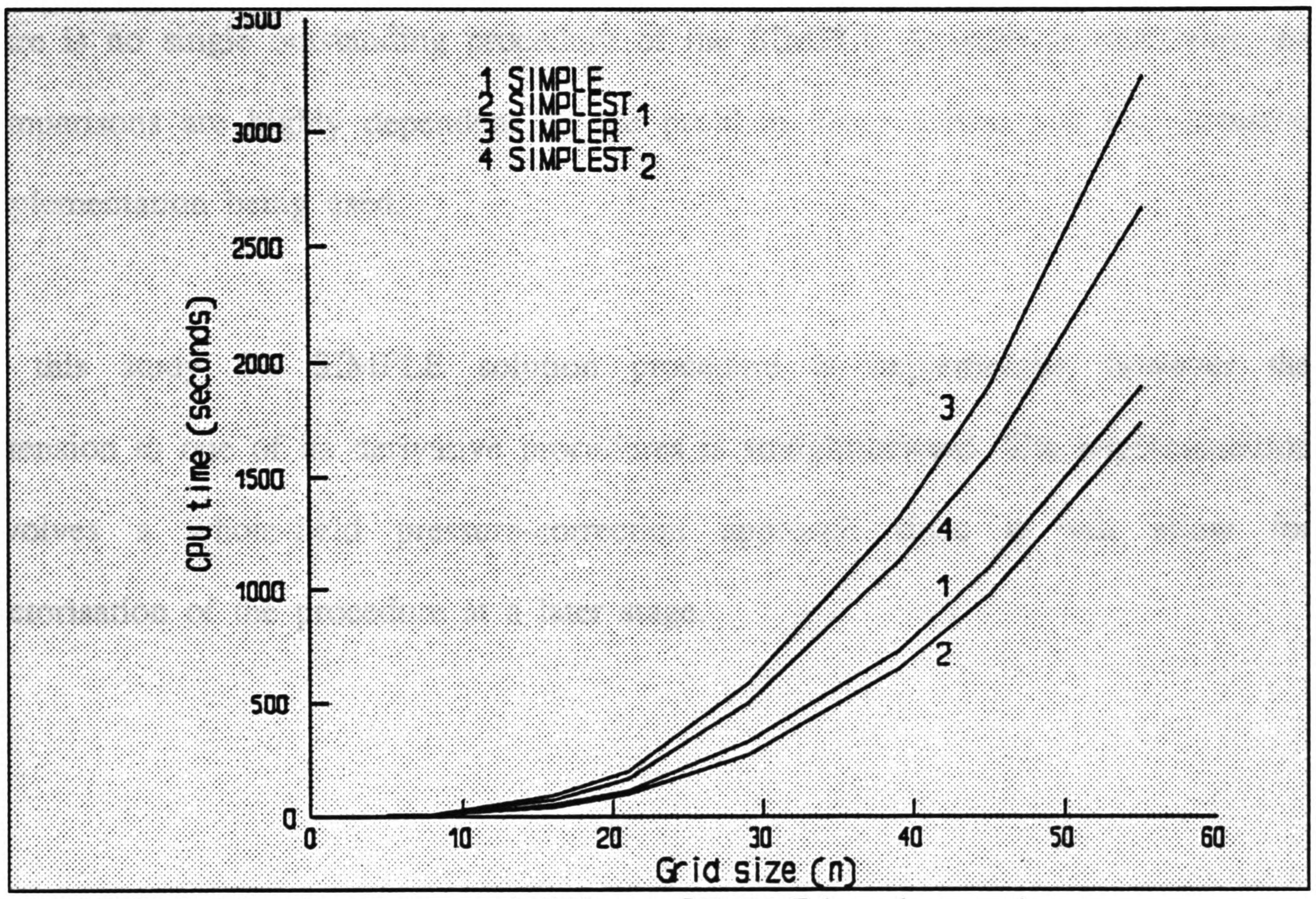


FIGURE 3.7-1 Comparison of different SIMPLE-based procedures derived for two-dimensional, steady state, incompressible flows. In the formulations a staggered-grid was used to discretise the momentum equations, and the general convective terms were represented using an upwind difference scheme.

A number of different solution procedures have also been described which can be used to solve the resulting discretised equations. These solution procedures are all based on the original SIMPLE procedure. Comparison studies have shown that there is no single outstanding procedure of the SIMPLE derivatives, moreover, the comparisons are highly dependent on the problem being solved and the particular implementation being used.

In this study the SIMPLE solution procedure is implemented, however the extension to one of its derivative procedures is straightforward. The implementation involves a whole-field pressure-correction approach since it will allow for vectorisation of the procedure at a later stage.

CHAPTER FOUR

4.0 SOLUTION OF LINEAR SYSTEMS OF EQUATIONS

4.1 Introduction

An important consideration in the implementation of the SIMPLE procedure is the intermediate solution of the resulting linear algebraic equations typified by equation (3.4.7-1). In this chapter a small subset of the many linear equation algorithms which can be used to solve these equations are described. The descriptions include details for both scalar and vector implementations.

These solvers are compared in both their scalar and vector implementations for the solution of a very trivial heat conduction problem represented by the Poisson equation.

4.2 The Poisson equation

This type of equation involves no convection component and requires the solution of only a single variable ϕ. In two-dimensions the Poisson equation can be written as

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = S_\phi \qquad (4.2\text{-}1)$$

This is an example of an elliptic partial differential equation and arises naturally in many problems including the steady state distribution of heat in a plane region and steady state problems involving incompressible fluids.

A unique solution to the Poisson equation is determined by the boundary conditions and the source term S_ϕ. When the source term is zero this is a special case of the Poisson equation and is called the Laplace equation. The problem solved here is the Laplace equation in a unit square domain, with both Neuman and Dirichlet boundary conditions (figure 4.2-1).

A grid is imposed onto the domain and a central-difference approximation is used to represent the partial derivatives, these are given by

$$\frac{\partial^2\phi}{\partial x^2} = \frac{\phi_{i+1j} - 2\phi_{ij} + \phi_{i-1j}}{(\Delta x)^2} - \frac{(\Delta x)^2}{12}\frac{\partial^4\phi}{\partial x^4} \tag{4.2-2a}$$

$$\frac{\partial^2\phi}{\partial y^2} = \frac{\phi_{ij+1} - 2\phi_{ij} + \phi_{ij-1}}{(\Delta y)^2} - \frac{(\Delta y)^2}{12}\frac{\partial^4\phi}{\partial y^4} \tag{4.2-2b}$$

$$\Delta x = x_{ij} - x_{i-1j} \tag{4.2-2c}$$

$$\Delta y = y_{ij} - y_{ij-1} \tag{4.2-2d}$$

substituting equation (4.2-2) into (4.2-1)

$$\frac{\phi_{i+1j} - 2\phi_{ij} + \phi_{i-1j}}{(\Delta x)^2} + \frac{\phi_{ij+1} - 2\phi_{ij} + \phi_{ij-1}}{(\Delta y)^2} = 0 \tag{4.2-3}$$

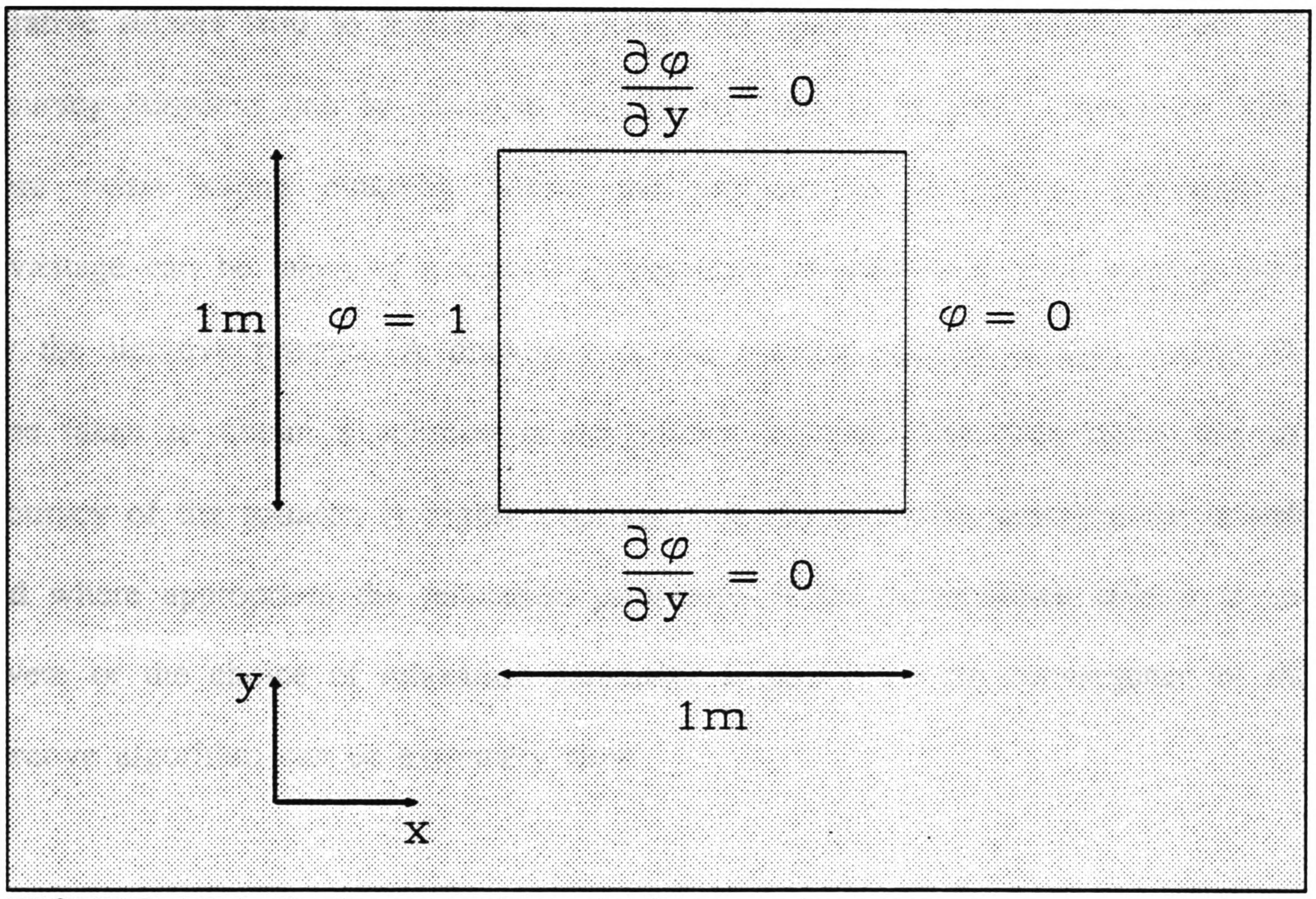
assuming a uniform grid is used, then

$$\phi_{ij-1} + \phi_{i-1j} - 4\phi_{ij} + \phi_{i+1j} + \phi_{ij+1} = 0 \tag{4.2-4}$$

$$\Rightarrow a_{ij-1}\phi_{ij-1} + a_{i-1j}\phi_{i-1j} - a_{ij}\phi_{ij} + a_{i+1j}\phi_{i+1j} + a_{ij+1}\phi_{ij+1} = 0 \tag{4.2-5}$$

$$A\phi = \mathbf{b} \tag{4.2-6}$$

where the a's denote the coefficients at the respective nodes. The resulting matrix of equations (A) is symmetric, positive definite and pentadiagonal in structure. This can now be solved using a linear equation solver.



$\frac{\partial \varphi}{\partial y} = 0$
1m
$\varphi = 1$
$\varphi = 0$
$\frac{\partial \varphi}{\partial y} = 0$
y
1m
x

FIGURE 4.2-1 Laplace equation solved on a unit square

4.3 Linear equation solvers

Linear equation solvers (or algorithms) may be classified broadly into direct and iterative solvers. Both approaches have their merits, however in general, iterative solvers are more favourable for large-order systems of equations.

Iterative solvers tend to require less storage for sparse matrix systems. Also, low accuracy solutions can be obtained rapidly and this is of great importance when there exists strong coupling within the problem to be solved. Furthermore, advantage can be taken of a known approximate solution as a good starting point for the iterative process. A drawback of the iterative solvers is that usually the time taken to obtain a solution is not known a priori. The time taken and the accuracy of the solution is highly dependent on the tolerance, convergence criteria and where appropriate the relaxation parameter. If the convergence criteria is too severe or the choice of relaxation parameter is poor then the convergence of the iterative algorithm can be extremely slow.

It is an impossible task to attempt to discuss all linear equation solvers, instead a small selection of the more popular ones are considered for discussion. This is followed by a description of a subset of these solvers and their implementation details for scalar and pipeline processors.

Of the direct algorithms the Thomas algorithm [1949] is probably the most extensively used tridiagonal solver to date. It has been used by many authors such as Spalding [1972] who refers to the solver as TDMA (TriDiagonal Matrix

Algorithm), Smith [1969], Conte and De Boor [1980] and Roache [1982]. Other tridiagonal solvers do exist, for example, the cyclic reduction algorithm (Hockney [1965]) which does not include any recursive steps within the algorithm and becomes more competitive when implemented on vector and parallel architectures. The algorithm has increased in popularity because of this property (Lambiotte and Voigt [1975], Masden and Rodrigue [1976] and Boris and Winsor [1982]). Other direct algorithms include Gaussian elimination with pivoting, LU factorisations such as Crout, Doolittle and Choleski (Burden, Faires and Reynolds [1981]) and the Strongly Implicit Procedure (SIP) of Stone [1968], in particular the modifications suggested by Schneider and Zedan [1981].

One class of iterative algorithms are described as gradient algorithms. The process of solving a set of n simultaneous equations can be visualised as finding the position of a minimum for an error function in an n-dimensional space. The method of conjugate gradients (Hestenes and Stiefel [1952]) typifies such gradient algorithms. One of the useful properties of such algorithms allows a solution to be obtained in up to n steps.

If the conjugate algorithm is used in an iterative sense for the solution of a banded matrix system then a satisfactory solution can be obtained in significantly fewer than n steps. Here the conjugate gradient algorithm can be far more efficient than the Gaussian elimination algorithm. If however, the conjugate gradient algorithm is used as a direct algorithm the solution is obtained after n steps (assuming exact arithmetic is used). In this case, for a fully populated matrix the conjugate gradient algorithm would execute almost six times more computation than the Gaussian elimination algorithm (Jennings [1985]).

The convergence rate of the conjugate gradient algorithm can be improved by applying a preconditioning matrix to the original matrix system. This technique has been adopted by many authors who have observed the sometimes slow convergence rate of the conjugate gradient algorithm. Some of the works that have helped to promote the popularity of the conjugate gradient algorithm are due to Meijerink and van der Vorst [1977], Kershaw [1978], Hageman and Young [1981], van der Vorst [1982, 1986], Concus, Golub and Meurant [1985], Sonneveld et al [1985], Kightley and Jones [1985], Kightley and Thompson [1987], Lai and Liddell [1987], Kincaid et al [1986], Melhem and Gannon [1987] and Kapitza and Eppel [1987].

Other iterative algorithms based on point-by-point and line-by-line techniques are also used extensively and are sometimes referred to as classical or stationary algorithms. Amongst this family of algorithms are the Jacobi with under-relaxation (JUR), Gauss-Seidel and successive over-relaxation (SOR) algorithms. Varga [1962] gives an excellent account of these and many other such algorithms. In the notation which follows the mnemonic for a line-by-line algorithm is preceded by the letter L, otherwise it is assumed to be a point-by-point algorithm. For example, JUR describes the point-by-point Jacobi with under-relaxation and LJUR describes the line-by-line Jacobi with under-relaxation.

Many different variations exist for the SOR algorithm which are based on a pre-defined ordering scheme and have been described by O'leary [1984] and Adams and Jordan [1986]; These schemes are of interest for two reasons. Firstly, for the matrices which result from a central-difference formulation as in section 4.2, it has been shown that the 'red-black' ordering for the SOR algorithm (RBSOR) and the 'natural' order SOR algorithm have the same asymptotic rate of convergence (Young [1971]). Secondly, the RBSOR algorithm lends itself more readily to vector processing architectures. Fujino [1989] has implemented a number of different ordering schemes on a vector processor and these include a natural, red-black (two colour) and a rainbow (seven-colour) ordering. These orderings are described in section 4.7.

Finally, for the iterative algorithms described above, the idea of using a series of coarser meshes to solve the original fine mesh has received much attention. This concept of multigrids can be used to enhance the convergence rate of an existing algorithm. The impact of using one such multigrid method to solve computational fluid dynamics problems is discussed in a later chapter.

4.4 Linear equation solvers used in this study

Of the many algorithms which exist only a subset are considered for implementation in this work. The algorithms are classed as either tridiagonal or pentadiagonal iterative algorithms. For the solution of the Poisson equation the algorithms were implemented for execution on both the scalar and pipeline processors. The algorithms executed on the scalar processor are referred to as scalar algorithms and those executed on the pipeline processor are referred to as vector algorithms. The vector algorithms are in fact the scalar algorithms re-structured for implementation on a pipeline processor.

4.5 Tridiagonal algorithms

Two of the many tridiagonal algorithms were considered for the solution of the Poisson equation, these were the Thomas algorithm [1949] and the cyclic reduction algorithm (Hockney [1965]). The Thomas algorithm was chosen because it is a very robust, efficient algorithm which has been used extensively in the past, in particular in the CHAMPION series codes (Spalding [1972]). The cyclic reduction algorithm was chosen because although it is not as efficient as the Thomas algorithm when executed on a scalar processor, it does lend itself more readily to parallel and vector pipeline type architectures. This advantage has been demonstrated recently by Whiteway [1979] and Hockney and Jesshope [1981] on parallel architectures. Although these algorithms are direct they have been implemented within an iterative framework using a line-by-line technique.

When using a line-by-line technique, an approximation of the solution field either side of the current line is used. Thus the solution of line i is found based on the approximations of lines i-1 and i+1 (figure 4.5-1), and is represented by the equation

$$a_{ij+1}\phi_{ij+1} - a_{ij}\phi_{ij} + a_{ij-1}\phi_{ij-1} = - a_{i+1j}\phi_{i+1j} - a_{i-1j}\phi_{i-1j} + b_{ij} \qquad (4.5\text{-}1)$$

$$A\phi = \mathbf{b} \qquad (4.5\text{-}2)$$

where

$$A = \begin{bmatrix} \cdot & \cdot & & \\ & \cdot & \cdot & \\ & \cdot & \cdot & a_{ij+1} \\ & \cdot & -a_{ij} & & \cdot \\ & a_{ij-1} & & \cdot \\ & & \cdot & \cdot \\ & & & \cdot & \cdot \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ -a_{i-1j}\phi_{i-1j}-a_{i+1j}\phi_{i+1j}+b_{ij} \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

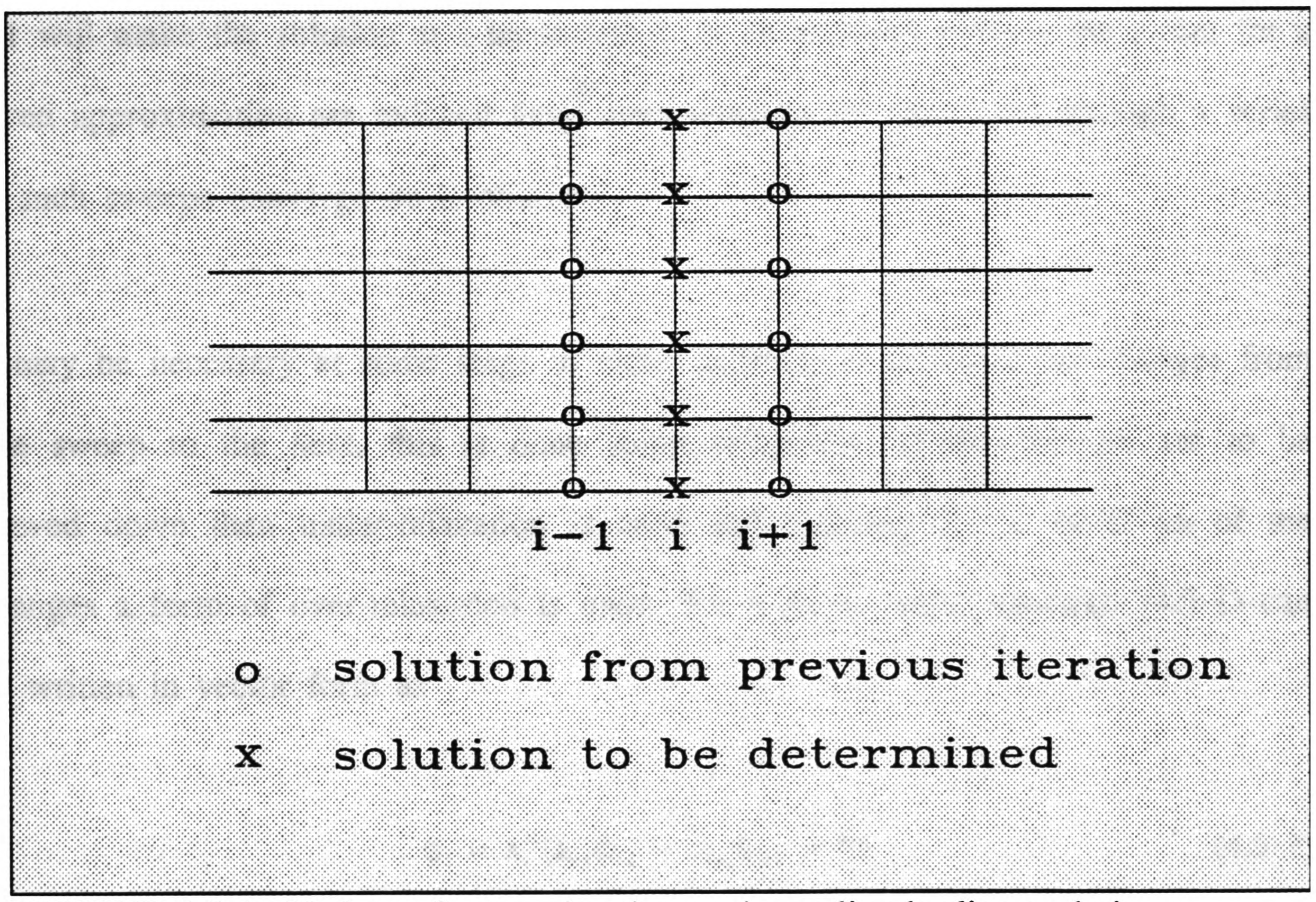

i−1
i
i+1
o solution from previous iteration
x solution to be determined

FIGURE 4.5-1 Update of approximations using a line-by-line technique

This results in the solution of a tridiagonal matrix system. The procedure is repeated for all lines. When all lines have been visited in turn by 'sweeping' from left to right one sweep has been completed. In general, the solution field obtained at this stage will not be the correct solution. This is not surprising since the solution for a given line i is obtained using the approximations at line i+1, where the values at line i+1 are from the previous sweep. The subsequent solution at line i+1 will make the solution at i inconsistent. In an iterative process the errors for a given approximation are reduced from one sweep to the next, and through a series of such sweeps convergence is obtained.

It may be necessary at some stage to either increase or decrease the changes from one sweep to the next, this is done using *relaxation.* If the changes are to be slowed down then under-relaxation is used and similarly for a speed up of the changes a form of over-relaxation is used. For a given line i, equation (4.5-1) can be written in vector form as

$$\phi_i = a_i^{-1}(a_{i-1}\phi_{i-1} + a_{i+1}\phi_{i+1} + \mathbf{b}) \qquad (4.5\text{-}3)$$

where ϕ_i represents a vector of all node approximations on line i. If the approximation from the previous sweep ϕ_i^* is added and then subtracted from equation (4.5-3) this gives

$$\phi_i = \phi_i^* + a_i^{-1}(a_{i-1}\phi_{i-1} + a_{i+1}\phi_{i+1} + \mathbf{b} - a_i\phi_i^*) \qquad (4.5\text{-}4)$$

where the term $a_i^{-1}(a_{i-1}\phi_{i-1} + a_{i+1}\phi_{i+1} + \mathbf{b} - a_i\phi_i^*)$ represents the change from one sweep to the next. This change can be controlled by introducing a linear relaxation factor α, where the improvement in the approximation is now given by

$$\phi_i = \phi_i^* + \alpha a_i^{-1}(a_{i-1}\phi_{i-1} + a_{i+1}\phi_{i+1} + \mathbf{b} - a_i\phi_i^*) \qquad (4.5\text{-}5)$$

When α lies between 0 and 1 this has the effect of under-relaxation and if α is greater than 1 this has an over-relaxation effect on the approximation. The range of α is dependent on the algorithm being used and furthermore the choice of α in this range can be crucial to the performance of the algorithm. Unfortunately, there are currently no general rules for the optimal choice of α. There are many reasons for this such as the problem being solved, the number and distribution of the grid nodes and the algorithm implemented.

The iterative algorithm described by equation (4.5-5) is called the line-by-line SOR (LSOR) algorithm. In this algorithm α lies in the range $0<\alpha<2$, therefore, either under-relaxation ($0<\alpha<1$) or over-relaxation ($1<\alpha<2$) can be exercised. With little modification equation (4.5-5) can represent the line-by-line JUR (LJUR) algorithm, this is given by

$$\phi_i = \phi_i^* + \alpha a_i^{-1}(a_{i-1}\phi_{i-1}^* + a_{i+1}\phi_{i+1} + \mathbf{b} - a_i\phi_i^*) \qquad (4.5\text{-}6)$$

where ϕ_{i-1}^* is the approximation from the previous sweep. The relaxation parameter α lies in the range $0<\alpha<1$, therefore only under-relaxation can be exercised using this algorithm.

Another consideration for line-by-line algorithms is the choice of sweeping direction. This can vary within a two-dimensional domain from left to right, right to left, bottom to top, top to bottom or any combination of these. The advantage to using a combination of these sweeping directions is that boundary effects can be conveyed throughout the domain at a faster rate than using a single sweeping direction. However, this is not always the case. In this study the line algorithms are implemented with the sweeping direction from left to right. The Thomas and cyclic reduction algorithms were used to solve for the resulting tridiagonal matrix systems of the general form

$$A\phi = \mathbf{b} \tag{4.5-7}$$

where

$$A = \begin{bmatrix} d_1 & u_1 & & & & \\ l_2 & d_2 & u_2 & & & \\ & l_3 & d_3 & u_3 & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & \cdot \\ & & & & \cdot & \cdot \; u_{m-1} \\ & & & & l_m & d_m \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \cdot \\ \cdot \\ \cdot \\ b_m \end{bmatrix} \qquad \phi = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \cdot \\ \cdot \\ \cdot \\ \phi_m \end{bmatrix}$$

4.5.1 Thomas algorithm

The Thomas algorithm is a special case of Gaussian elimination and consists of a forward elimination followed by a back substitution. It is this simplicity which makes the algorithm very efficient when implemented on a conventional scalar processor. The main steps for the solution of a tridiagonal system of equations of order $m>1$ (4.5-7) are now presented

ALGORITHM 4.5.1-1 : Scalar Thomas algorithm

```
Forward elimination
(1)    $r_j = l_j/d_{j-1}$              ⎫
(2)    $d_j = d_j - r_j u_{j-1}$         ⎬  $j=2(1)m$
(3)    $b_j = b_j - r_j b_{j-1}$         ⎭
Backward substitution
(4)    $\phi_m = b_m/d_m$
(5)    $\phi_j = (b_j - u_j\phi_{j+1})/d_j$     $j=m-1(-1)1$
```

The implementation of such an algorithm on a pipeline processor is possible, however the major obstacle to overcome is the implicit nature of the algorithm. Fortunately, any implicit statement can be replaced by an iterative explicit step, however, the algorithm then becomes iterative rather than direct and may require more storage and computation as a result. The implicit nature of the Thomas algorithm is typified by the recursion present in all but one of its steps, i.e step (4). The explicit Thomas algorithm is then defined by replacing all implicit steps by iterative explicit ones. In the following description * denotes an approximation from the previous iteration, B and D are temporary storage vectors and MAXIT is the maximum number of iterations allowed before termination.

ALGORITHM 4.5.1-2 : Scalar explicit Thomas algorithm

```
Forward elimination
(1)    iter = 1
       set $D_j^* = d_j$              $j=1(1)m$
           $B_j^* = b_j$              $j=1(1)m$
(2)    $r_j = l_j/D_{j-1}^*$          $j=2(1)m$
(3)    $D_j = d_j - r_j u_{j-1}$      $j=2(1)m$
```

```
(4)   Check for convergence of $D_j$ with $D^*_j$. If not converged
      $D^*_j = D_j$                 $j=1(1)m$
      goto step (2)

(5)   $B_j = b_j - r_j B^*_{j-1}$        $j=2(1)m$

(6)   Check for convergence of $B_j$ with $B^*_j$. If not converged
      $B^*_j = B_j$                 $j=1(1)m$
      goto step (5)

 Backward substitution
(7)   set $\phi^*_j = B_j$

(8)   $\phi_j = (B_j - u_j\phi^*_{j+1})/D_j$   $j=1(1)m-1$

(9)   Check for convergence of $\phi_j$ with $\phi^*_j$. If not converged and
      iter≤MAXIT
      $\phi^*_j = \phi_j$                 $j=1(1)m$
      iter = iter + 1
      goto step (8)
```

To describe the algorithm in a form suitable for vectorisation it is convenient to introduce multiplication and division operations between vectors as

$$\mathbf{a\#b} = [a_1b_1,\ a_2b_2,\ldots,a_{nm}b_{nm}]^T \qquad (4.5.1\text{-}1a)$$

$$\mathbf{a\backslash b} = [a_1/b_1,\ a_2/b_2,\ldots,a_{nm}/b_{nm}]^T \qquad (4.5.1\text{-}1b)$$

and shifted vector operations as

$$\mathbf{a}(n) = [a_{n+1},\ a_{n+2},\ldots,\ a_{nm},\ \underbrace{0,\ldots,0}_{\leftarrow n\rightarrow}]^T \qquad (4.5.1\text{-}2a)$$

$$\mathbf{a}(-n) = [\underbrace{0,\ldots,0}_{\leftarrow n\rightarrow},\ a_1,\ a_2,\ldots,\ a_{nm-n}]^T \qquad (4.5.1\text{-}2b)$$

The main computation steps are now presented for the vector Thomas algorithm.

ALGORITHM 4.5.1-3a : Vector Thomas algorithm

```
Forward elimination
(1)   iter = 1
      set $\mathbf{D}^* = d$

          $\mathbf{B}^* = \mathbf{b}$

(2)   $\mathbf{r} = \Lambda\mathbf{D}^*(-1)$

(3)   $\mathbf{D} = d - \mathbf{r}\#u(-1)$

(4)   Check for convergence of D with D*. If not converged
      $\mathbf{D}^* = \mathbf{D}$
      goto step (2)

(5)   $\mathbf{B} = \mathbf{b} - \mathbf{r}\#\mathbf{B}^*(-1)$

(6)   Check for convergence of B with B*. If not converged
      $\mathbf{B}^* = \mathbf{B}$
      goto step (5)

Backward substitution
(7)   set $\phi^* = \mathbf{B}$

(8)   $\phi = (\mathbf{B} - u\#\phi^*(1))\backslash\mathbf{D}$

(9)   Check for convergence of $\phi$ with $\phi^*$. If not converged and
      iter≤MAXIT
      $\phi^* = \phi$
      iter = iter + 1
      goto step (8)
```

All recursive steps have been replaced with explicit steps coupled with iteration, however this has led to three separate checks for convergence (steps (4), (6) and (9)). This is an overhead far too expensive to include in the algorithm and ideally a single check should be made on the approximation of the solution ϕ^*. It has been discovered that checks for convergence can be deferred until the completion of all explicit steps (1), (2), (3), (5), (7), and (8). This results in a more efficient implementation of the vector Thomas algorithm.

ALGORITHM 4.5.1-3b : Vector Thomas algorithm (revised)

```
Forward elimination
(1)   iter = 1
      set $\mathbf{D}^* = \boldsymbol{d}$
          $\mathbf{B}^* = \mathbf{b}$
(2)   $\mathbf{r} = \boldsymbol{l}\backslash\mathbf{D}^*(-1)$
(3)   $\mathbf{D} = \boldsymbol{d}$ - r#$u$(-1)
(4)   $\mathbf{B} = \mathbf{b}$ - r#$\mathbf{B}^*$(-1)
Backward substitution
(5)   when iter = 1     set $\phi^* = \mathbf{B}$
(6)   $\phi = (\mathbf{B} - u\#\phi^*(1))\backslash\mathbf{D}$
(7)   Check for convergence of $\phi$ with $\phi^*$. If not converged and
      iter$\leq$MAXIT
      $\phi^* = \phi$, $\mathbf{B}^* = \mathbf{B}$, $\mathbf{D}^* = \mathbf{D}$
      iter = iter + 1
      goto step (2)
```

4.5.2 Cyclic reduction algorithm

The cyclic reduction algorithm was originally devised for use on parallel architectures. The idea is analogous to that used in the method of cascade sums. Given the recurrence relation

$$\phi_{j+1} = a_j\phi_j + b_j \qquad (4.5.2\text{-}1)$$

which relates neighbouring terms ϕ_j and ϕ_{j+1}, it is possible to combine adjacent terms of the relation so that there exists a new relation between ϕ_j and ϕ_{j+2}. This itself is a recurrence relation and the process can be repeated to give another relation between ϕ_j and ϕ_{j+4} etc. This is repeated until ϕ_j is related to ϕ_{j+m} where m

is the order of the matrix being solved. Here the last term is related to the first and hence the solution can be determined. At each level of the process the number of recurrence relations are reduced by a factor of two.

Taking the row j of a tridiagonal matrix system

$$l_j\phi_{j-1} + d_j\phi_j + u_j\phi_{j+1} = b_j \qquad (4.5.2\text{-}2)$$

this recurrence relation is general for all rows $1 \le j \le m$ where $l_1 = u_m = 0$. If this row is now normalised with respect to d_j this gives

$$L_j^{(1)}\phi_{j-1} + \phi_j + U_j^{(1)}\phi_{j+1} = B_j^{(1)}$$

where

$$\begin{aligned} L_j^{(1)} &= l_j/d_j \\ U_j^{(1)} &= u_j/d_j \\ B_j^{(1)} &= b_j/d_j \end{aligned} \qquad (4.5.2\text{-}3)$$

Similarly the equations for rows j-1 and j+1 can be written as

$$L_{j-1}^{(1)}\phi_{j-2} + \phi_{j-1} + U_{j-1}^{(1)}\phi_j = B_{j-1}^{(1)} \qquad (4.5.2\text{-}4)$$

$$L_{j+1}^{(1)}\phi_j + \phi_{j+1} + U_{j+1}^{(1)}\phi_{j+2} = B_{j+1}^{(1)} \qquad (4.5.2\text{-}5)$$

Equations (4.5.2-4) and (4.5.2-5) are used to substitute for ϕ_{j-1} and ϕ_{j+1} in equation (4.5.2-3), this yields

$$\begin{aligned} - L_j^{(1)}L_{j-1}^{(1)}\phi_{j-2} + (1-L_j^{(1)}U_{j-1}^{(1)} - L_{j+1}^{(1)}U_j^{(1)})\phi_j - U_j^{(1)}U_{j+1}^{(1)}\phi_{j+2} \\ = B_j^{(1)} - L_j^{(1)}B_{j-1}^{(1)} - U_j^{(1)}U_{j+1}^{(1)} \end{aligned} \qquad (4.5.2\text{-}6)$$

This can be normalised with respect to $(1 - L_j^{(1)}U_{j-1}^{(1)} - L_{j+1}^{(1)}U_j^{(1)})$ to give

$$L_j^{(2)}\phi_{j-2} + \phi_j + U_j^{(2)}\phi_{j+2} = B_j^{(2)}$$

where

$$\begin{aligned}
D_j^{(2)} &= 1 - L_j^{(1)}U_{j-1}^{(1)} - L_{j+1}^{(1)}U_j^{(1)} \\
L_j^{(2)} &= -L_j^{(1)}L_{j-1}^{(1)} / D_j^{(2)} \\
U_j^{(2)} &= -U_j^{(1)}U_{j+1}^{(1)} / D_j^{(2)} \\
B_j^{(2)} &= (B_j^{(1)} - L_j^{(1)}B_{j-1}^{(1)} - U_j^{(1)}B_{j+1}^{(1)}) / D_j^{(2)}
\end{aligned} \qquad (4.5.2\text{-}7)$$

Equation (4.5.2-7) can then be written in terms of ϕ_{j-4}, ϕ_j and ϕ_{j+4}. This process can now be described for a general level p

$$L_j^{(p)}\phi_{j-k} + \phi_j + U_j^{(p)}\phi_{j+k} = B_j^{(p)}$$

where

$$\begin{aligned}
D_j^{(p)} &= 1 - L_j^{(p-1)}U_{j-k}^{(p-1)} - L_{j+k}^{(p-1)}U_j^{(p-1)} \\
L_j^{(p)} &= -L_j^{(p-1)}L_{j-k}^{(p-1)} / D_j^{(p)} \\
U_j^{(p)} &= -U_j^{(p-1)}U_{j+k}^{(p-1)} / D_j^{(p)} \\
B_j^{(p)} &= (B_j^{(p-1)} - L_j^{(p-1)}B_{j-k}^{(p-1)} - U_j^{(p-1)}B_{j+k}^{(p-1)}) / D_j^{(p)} \\
k &= 2^{p-2}
\end{aligned} \qquad (4.5.2\text{-}8)$$

The effect of using the cyclic reduction algorithm is shown in figure 4.5.2-1 for a matrix system m=8. The solution is obtained after $\log_2 m$ reduction levels. The main steps of the algorithm are now presented.

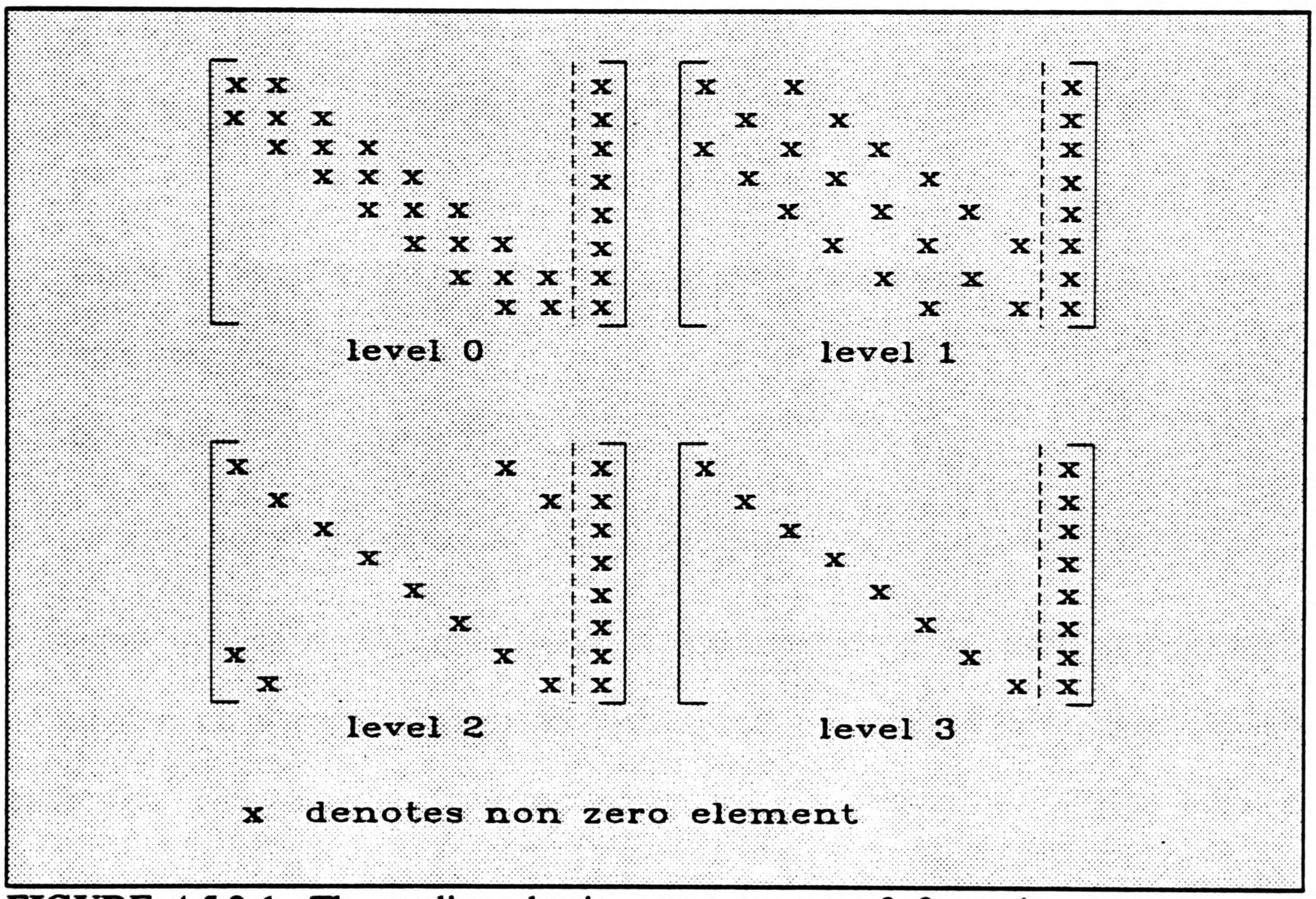


FIGURE 4.5.2-1 The cyclic reduction process on an 8x8 matrix system

ALGORITHM 4.5.2-1 : Scalar cyclic reduction algorithm

```
(1)  set k=1, step=1

 Normalise rows with respect to main diagonal
(2)  $l_j = l_j/d_j$                          j=1(1)$m$

     $u_j = u_j/d_j$                          j=1(1)$m$

     $b_j = b_j/d_j$                          j=1(1)$m$

 Carry out reduction
(3)  $d_j = 1 - l_j u_{j-k} - l_{j+k} u_j$    j=1(1)$m$

(4)  $b_j = b_j - l_j b_{j+k} - u_j b_{j-k}$  j=1(1)$m$

(5)  $l_j = -l_j l_{j-k}$                     j=1(1)$m$

(6)  $u_j = -u_j u_{j+k}$                     j=1(1)$m$

(7)  k = 2k

(8)  Check to see if step$\geq\log_2 m$. If check is not satisfied then
     step = step + 1
     goto step (2)

(9)  $\phi_j = b_j/d_j$                       j=1(1)$m$
```

The interest in such an algorithm lies in the fact that it can be easily implemented for execution on a pipeline processor because all computation steps have explicit formulations. The algorithm implemented for the VA-1 processor will be referred to as the vector cyclic reduction algorithm and is now described in a form suitable for vectorisation.

ALGORITHM 4.5.2-2 : Vector cyclic reduction algorithm

```
(1)  set k=1, step=1

 Normalise rows with respect to main diagonal
(2)  $l = l \backslash d$

     $u = u \backslash d$

     $b = b \backslash d$
```

```
Carry out reduction
(3)   $\boldsymbol{d} = \mathbf{1} - \boldsymbol{l}\#\boldsymbol{u}(-k) - \boldsymbol{l}(k)\#\boldsymbol{u}$

(4)   $\mathbf{b} = \mathbf{b} - \boldsymbol{l}\#\mathbf{b}(k) - \boldsymbol{u}\#\mathbf{b}(-k)$

(5)   $\boldsymbol{l} = -\boldsymbol{l}\#\boldsymbol{l}(-k)$

(6)   $\boldsymbol{u} = -\boldsymbol{u}\#\boldsymbol{u}(k)$

(7)   $k = 2k$

(8)   Check to see if $step \geq \log_2 m$. If check is not satisfied then
      step = step + 1
      goto step (2)

(9)   $\phi = \mathbf{b} \backslash \boldsymbol{d}$
```

4.6 Results for the Laplace equation using tridiagonal algorithms

Since the vector Thomas algorithm is an iterative algorithm a convergence criteria must be defined to terminate the iterative process. For the Laplace equation a simple criteria is used based on an absolute difference of approximations, i.e.

$$|\phi_j^* - \phi_j| < \zeta \qquad \forall\, j \tag{4.6-1}$$

where ϕ_j^* represents the approximation from the previous iteration, ϕ_j the approximation from the current iteration and ζ is a pre-defined tolerance, in this case 10^{-5}. The LSOR algorithm is used to solve the Laplace equation with a near optimum relaxation parameter α=1.82.

The grid used was uniform and the number of nodes varied from 5x5 to 45x45. Table 4.6-1 shows the variation of CPU time with the grid size for both the Thomas and the cyclic reduction algorithms. Also shown are the speed-up factors

(S) defined by the ratio of scalar to vector CPU times and the proportion of code vectorised. The comparison of scalar and vector Thomas algorithms indicates that the scalar algorithm is superior (figure 4.6-1). This is highlighted by the speed-up factors less than unity and this situation is termed 'slow-down'. The scalar algorithm is a factor of 2 faster than the vector algorithm and this has occurred for two main reasons:

(1) The vector algorithm being an iterative process carries out many times more computations than the scalar algorithm, and

(2) The VA-1 processor is not being utilised efficiently since the vector operations of length *m* are significantly lower than the $n_{1/2}$ parameter.

Since the number of operations in the vector algorithm are not the same as those in the scalar algorithm this prohibits an analysis of the performance using Amdahl's law. However, it can be inferred that as the grid size is increased the vector algorithm will become more competitive. This is because as an iterative method it will be more efficient and also the VA-1 will be used more efficiently. Furthermore, a graph showing the variation of speed-up with grid size (figure 4.6-2) shows that the peak speed-up has not yet been reached.

The comparison of scalar and vector cyclic reduction algorithms is shown graphically (figure 4.6-3) and indicates that the vector algorithm is significantly faster than the scalar equivalent. The vector algorithm is over five times faster and there is still further improvement expected as the grid size is increased (figure 4.6-4). The reason for this is that the VA-1 would be used more efficiently as it performs best with long vectors. The fraction of effort in using the VA-1 for the 45x45 grid problem is f_v=0.996 (table 4.6-1), using Amdahl's equation (2.10-4) the

GRID SIZE	5	11	15	21	25	31	35	41	45
SCALAR THOMAS	.33	1.65	3.25	6.50	9.40	14.98	19.53	34.00	50.91
VECTOR THOMAS	3.96	11.56	18.08	26.16	32.45	42.51	50.51	74.96	101.81
SPEED-UP S	.1	.1	.2	.2	.3	.4	.4	.5	.5
VECTOR FRACTION f_v	.9987	.9986	.9984	.9981	.9979	.9977	.9977	.9980	.9979
SCALAR CYCLIC	.96	5.61	11.19	29.16	42.43	66.16	103.50	180.80	274.16
VECTOR CYCLIC	1.73	4.88	6.91	11.98	14.53	18.18	25.26	37.40	51.03
SPEED-UP S	.6	1.2	1.6	2.4	2.9	3.6	4.1	4.8	5.4
VECTOR FRACTION f_v	.9968	.9966	.9958	.9958	.9954	.9945	.9961	.9960	.9958

TABLE 4.6-1 Results for the Thomas and cyclic reduction algorithms

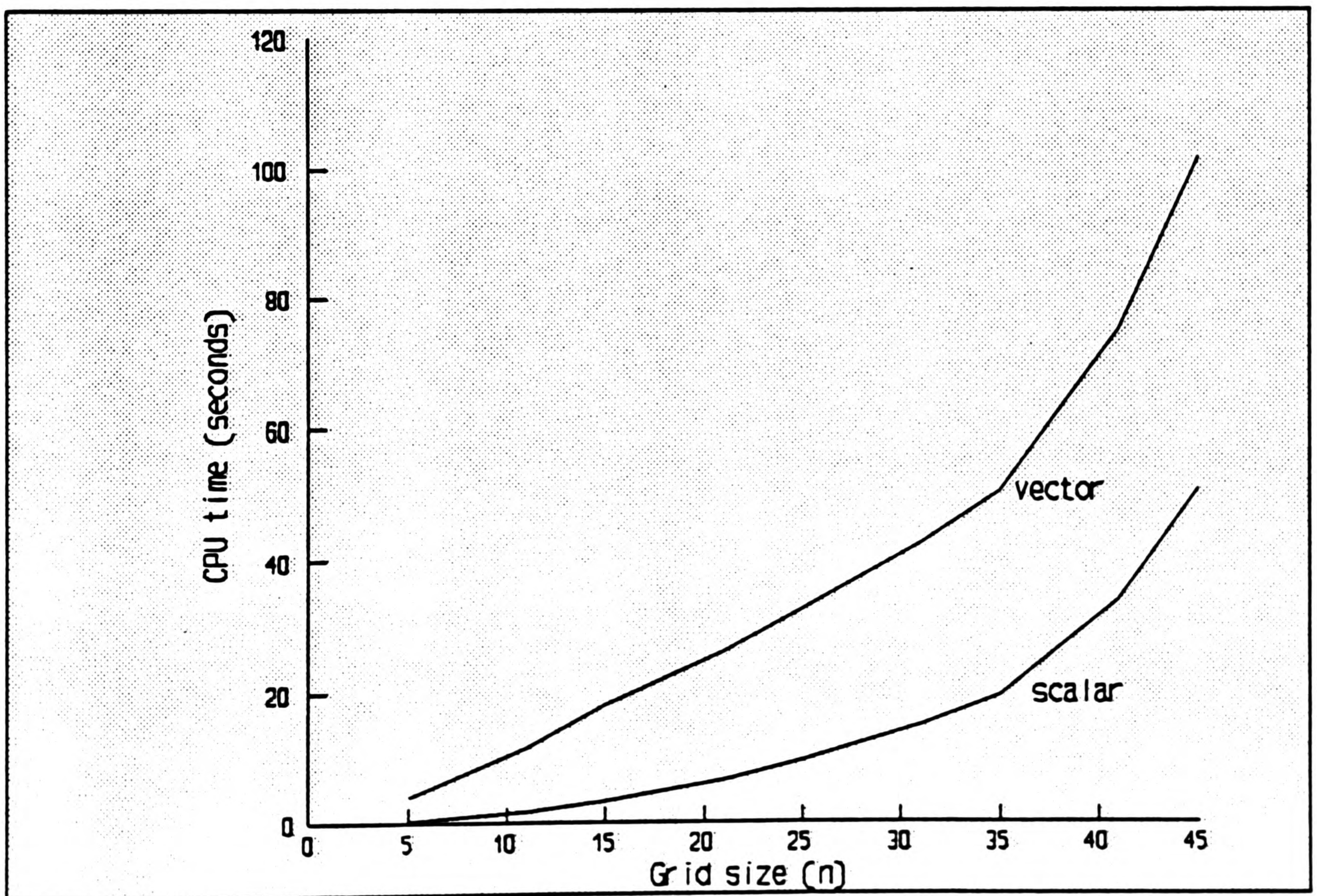


FIGURE 4.6-1 Results of the scalar and vector Thomas algorithm when used to solve the Laplace equation

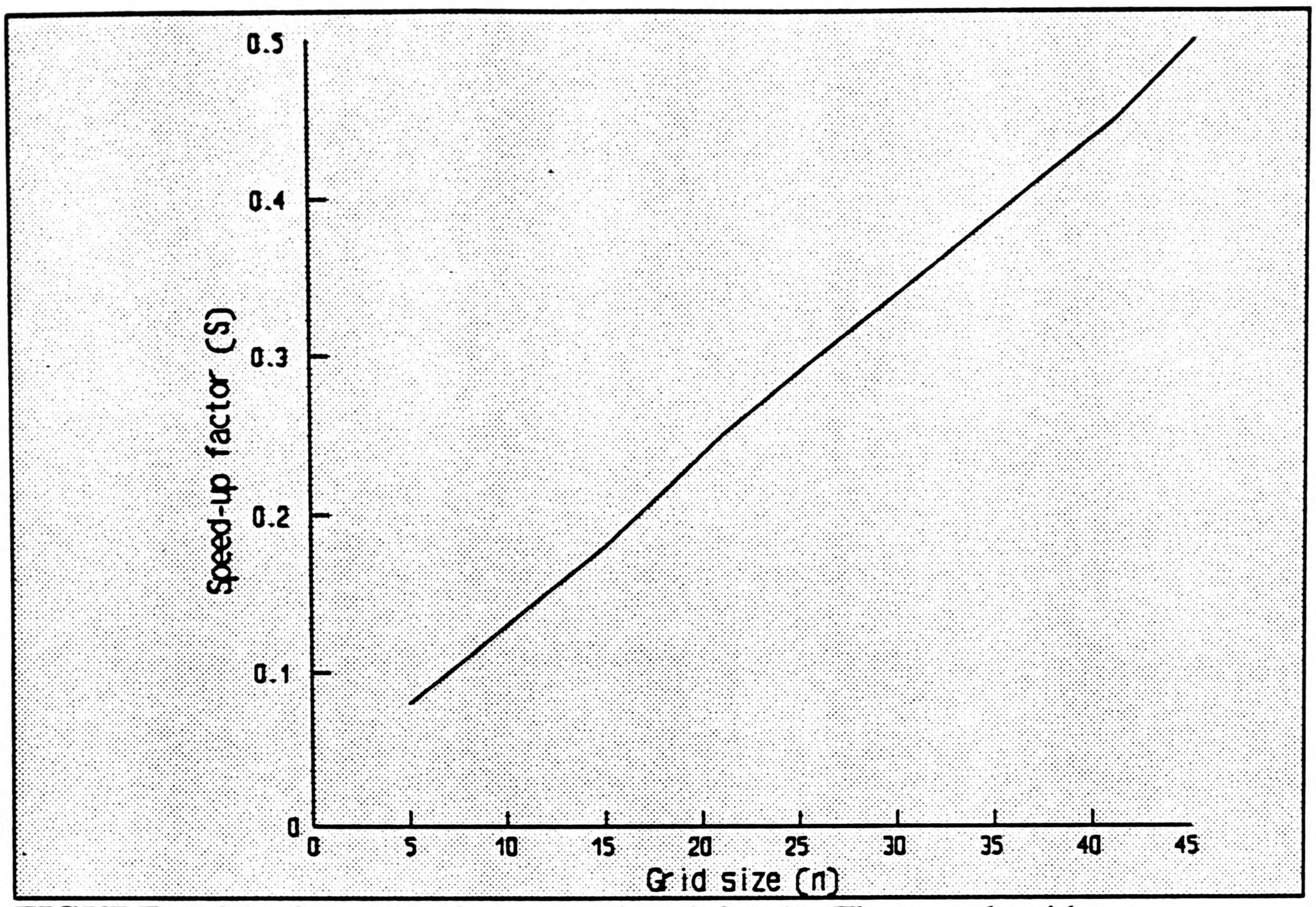


FIGURE 4.6-2 Speed-up factors achieved for the Thomas algorithm

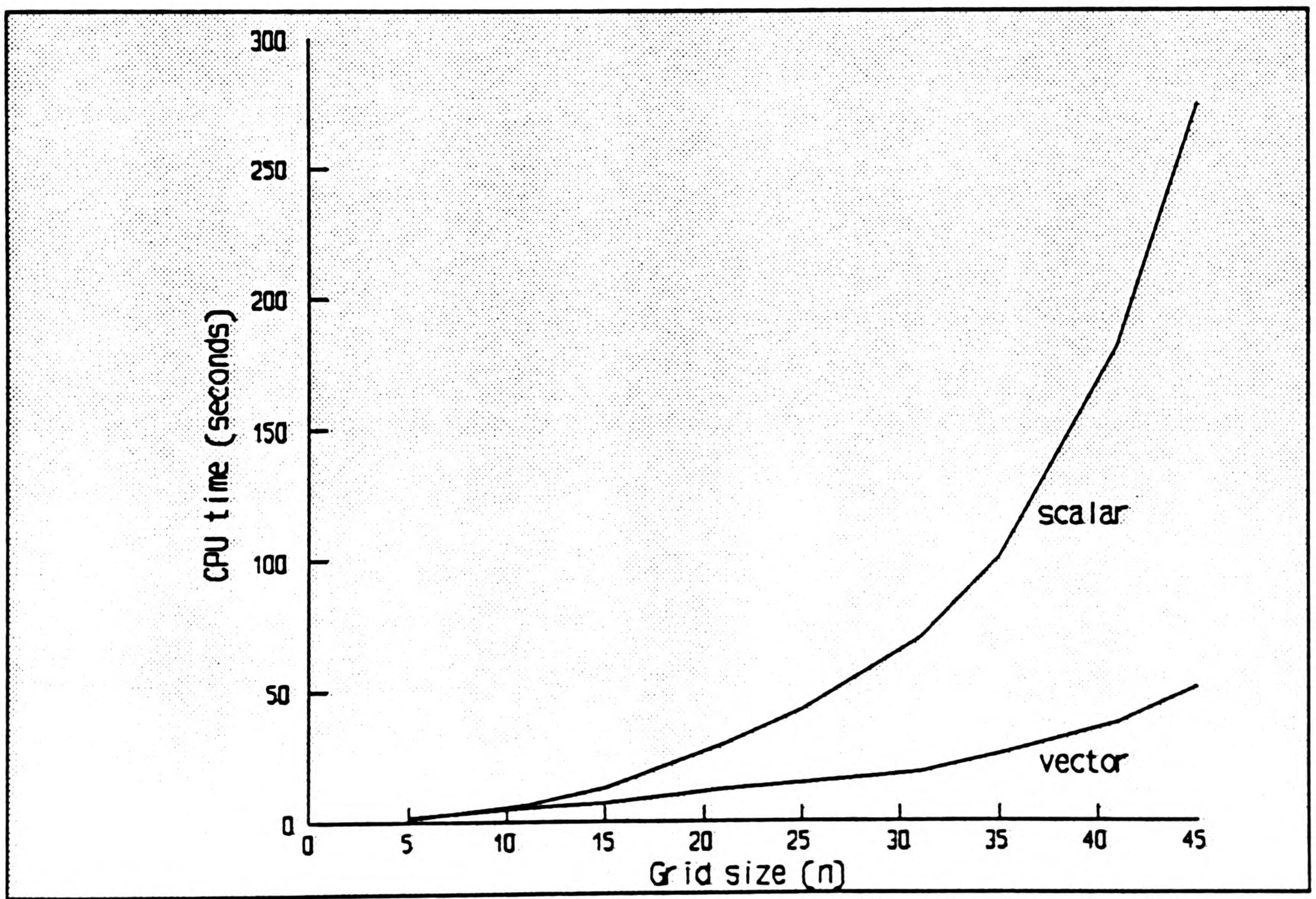


FIGURE 4.6-3 Results of the scalar and vector cyclic reduction algorithm when used to solve the Laplace equation

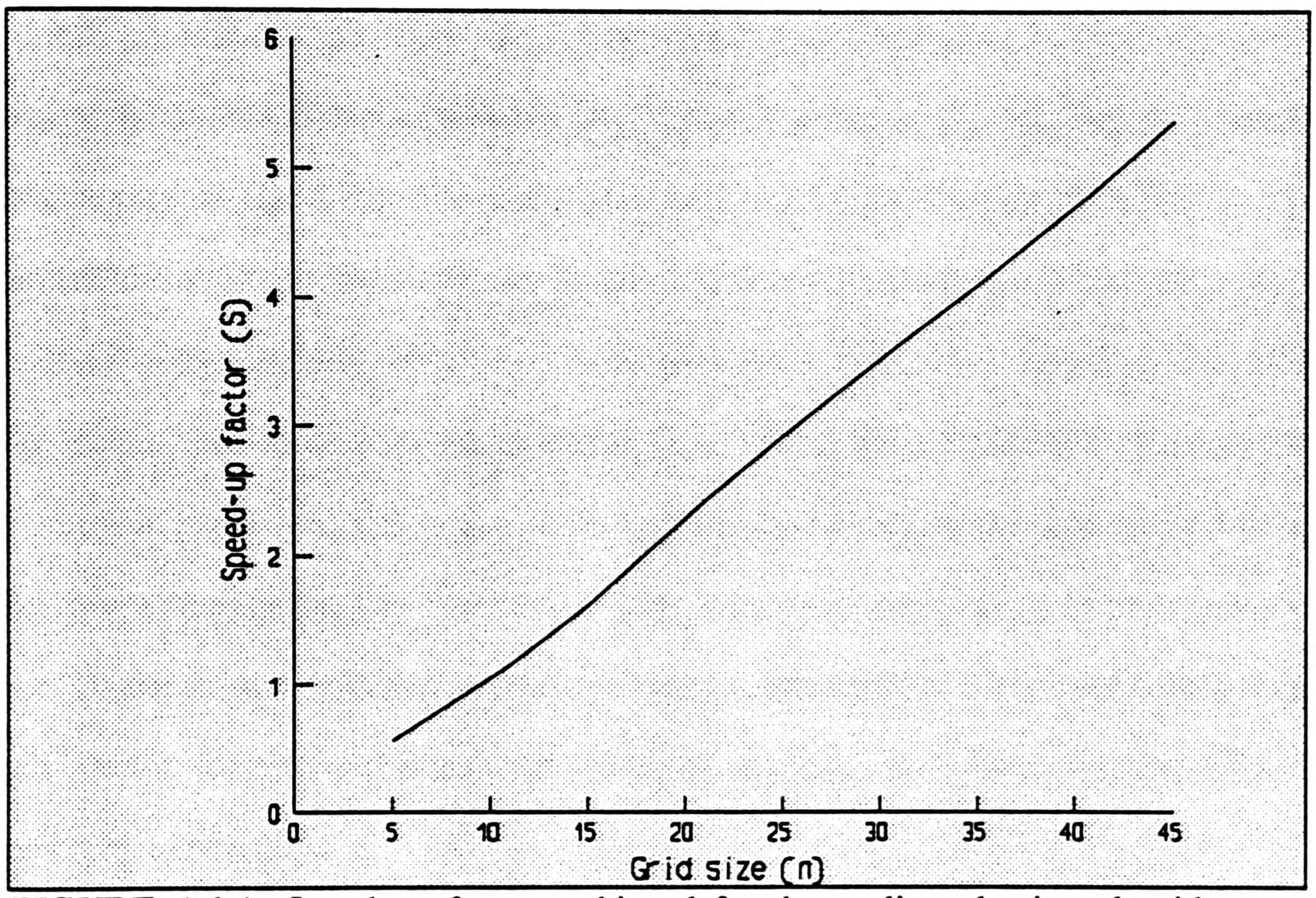


FIGURE 4.6-4 Speed-up factors achieved for the cyclic reduction algorithm expected speed-up would lie between 4.29 and 102.56. The speed-up of up to 5.5 is consistent with what is expected.

The results suggest that the scalar Thomas algorithm is over five times more efficient than the scalar cyclic reduction algorithm for the solution of the Laplace equation. However, the vector cyclic reduction algorithm is a factor of two faster than the vector Thomas algorithm.

In deciding which is the 'best' algorithm overall it is interesting to note that there is no significant difference between the scalar Thomas algorithm and the vector cyclic reduction algorithm. For small grid sizes the scalar Thomas algorithm is the more efficient of the two, but for larger grid sizes it is expected that the cyclic reduction algorithm will be better. However, for a significant speed-up the grid sizes would have to be of a much higher order. Thus, for the solution of equations such as the Laplace equation the vectorised tridiagonal algorithms are not a viable route when implemented on the VA-1. This would not necessarily be the case for other machines with pipeline processors which perform efficiently when executing small vector length operations, this class of machine is typified by the CRAY supercomputer family.

4.7 Pentadiagonal algorithms

In this study four pentadiagonal algorithms were considered for the solution of the Laplace equation. These were the JUR, SOR, RBSOR and the conjugate gradient algorithm with a Jacobi preconditioner (JCG). These algorithms were not very popular before pipeline and parallel architectures were introduced. They tended to be too computationally expensive and preference was given to the line-by-line iterative algorithms, in particular the LSOR which was more efficient on storage and in many cases computation. However, point-by-point and search algorithms are enjoying a revival and are now as popular as the line algorithms. This is mainly due to the new supercomputer class of machines, these provide speed through vectorisation and larger data memory which are necessary for some of these algorithms. Interest in algorithms such as the JUR and SOR have been further stimulated by the coupling with multigrid methods (Chapter 7).

4.7.1 The point-by-point JUR algorithm

This is the most fundamental of all point-by-point algorithms. When used to solve a discretised domain each node is visited and updated in a systematic manner. Here the order is chosen so that each node on a horizontal row is updated in turn, this is then repeated for all rows. When all nodes have been updated this defines one 'iteration'. An approximation ϕ_{ij} is updated by using the four neighbouring nodes ϕ_{i-1j}, ϕ_{i+1j}, ϕ_{ij-1} and ϕ_{ij+1}, all of these approximations are taken from the previous iteration (figure 4.7.1-1). The updated nodes are referred to as 'new' approximations and those from the previous iteration are referred to as 'old' approximations and denoted by $^{\bullet}$. Thus the new approximation ϕ_{ij} is determined by

$$a_{ij}\phi_{ij} = a_{i-1j}\phi^{\bullet}_{i-1j} + a_{i+1j}\phi^{\bullet}_{i+1j} + a_{ij-1}\phi^{\bullet}_{ij-1} + a_{ij+1}\phi^{\bullet}_{ij+1} + b_{ij} \qquad \begin{array}{l} i=1(1)n \\ j=1(1)m \end{array} \quad (4.7.1\text{-}1)$$

$$\Rightarrow \phi_{ij} = a^{-1}_{ij}\{a_{i-1j}\phi^{\bullet}_{i-1j} + a_{i+1j}\phi^{\bullet}_{i+1j} + a_{ij-1}\phi^{\bullet}_{ij-1} + a_{ij+1}\phi^{\bullet}_{ij+1} + b_{ij}\} \quad \begin{array}{l} i=1(1)n \\ j=1(1)m \end{array} \quad (4.7.1\text{-}2)$$

This defines the point-by-point Jacobi algorithm. Relaxation can be introduced into the Jacobi algorithm in a manner similar to that used for the line algorithms, thus

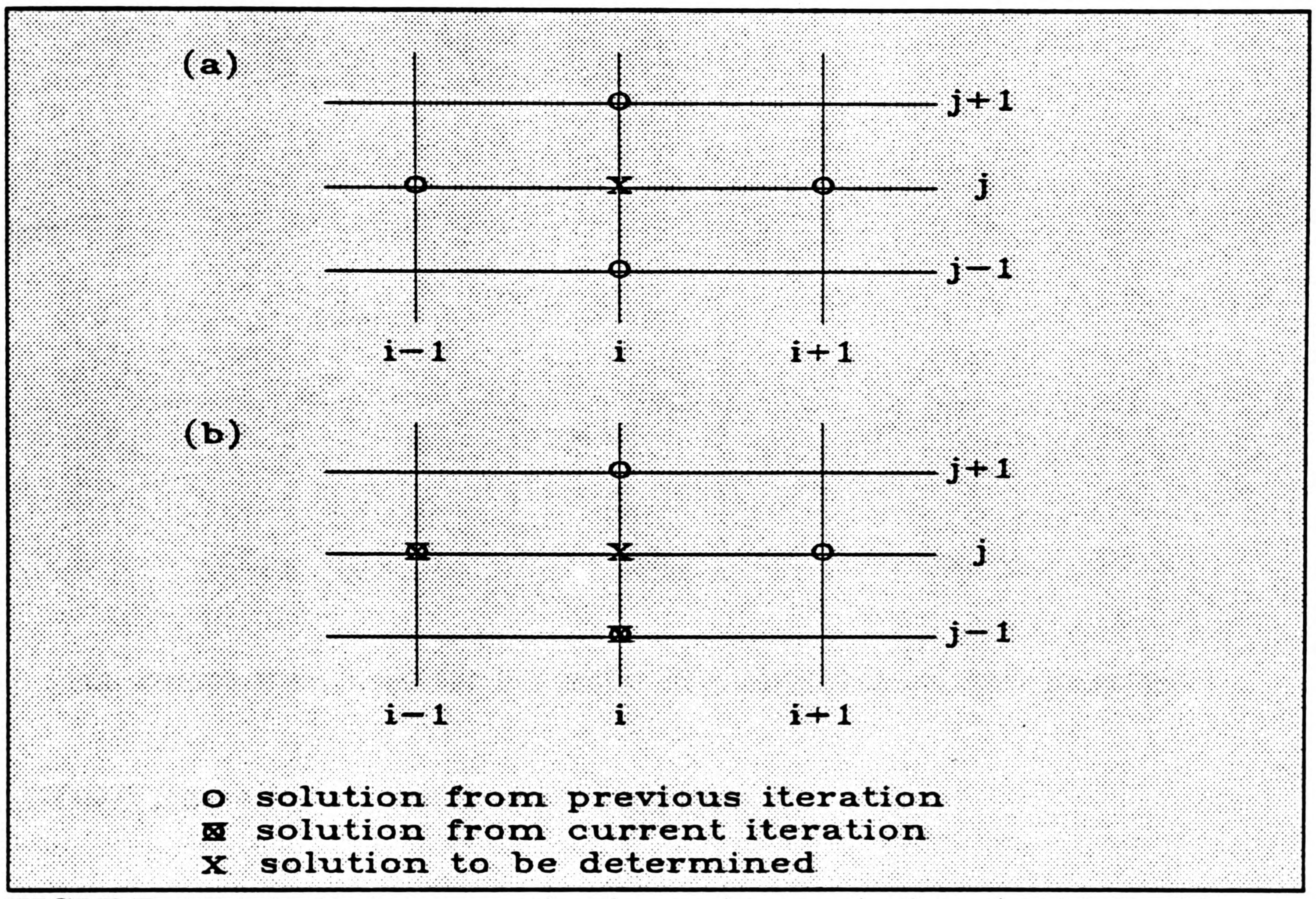

(a)
j+1
j
j−1
i−1
i
i+1
(b)
j+1
j
j−1
i−1
i
i+1
o solution from previous iteration
⊠ solution from current iteration
X solution to be determined

FIGURE 4.7.1-1 Update approximations using a point-by-point (a) Jacobi (b) Gauss-Seidel technique

$$\phi_{ij} = \phi^{*}_{ij} + \alpha a^{-1}_{ij}\{a_{i-1j}\phi^{*}_{i-1j} + a_{i+1j}\phi^{*}_{i+1j} + a_{ij-1}\phi^{*}_{ij-1} + a_{ij+1}\phi^{*}_{ij+1} + b_{ij} - a_{ij}\phi^{*}_{ij}\} \quad \begin{array}{l} i=1(1)n \\ j=1(1)m \end{array} \qquad (4.7.1\text{-}3)$$

This is now the point-by-point JUR algorithm where the relaxation parameter α lies between 0 and 1. When α=1 equation (4.7.1-3) reverts to the Jacobi algorithm. A pentadiagonal matrix system results from this formulation and has a general form

$$A\phi = \mathbf{b} \qquad (4.7.1\text{-}4)$$

where

$$A = \begin{bmatrix} d_{11} & ul_{21} & & u2_{12} & \\ l1_{11} & d_{21} & ul_{31} & & . \\ & l1_{21} & d_{31} & ul_{41} & . \\ l2_{21} & . & . & . & u2_{nm} \\ & . & . & . & . \\ & & . & . & . \; ul_{nm-1} \\ & & l2_{nm-1} & l1_{n-1m} & d_{nm} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \\ . \\ . \\ . \\ b_{nm} \end{bmatrix} \quad \phi = \begin{bmatrix} \phi_{11} \\ \phi_{21} \\ \phi_{31} \\ . \\ . \\ . \\ \phi_{nm} \end{bmatrix}$$

The main computation steps for the scalar JUR algorithm are now presented.

ALGORITHM 4.7.1-1 : Scalar JUR algorithm

```
(1)   set iter=1

(2)   $\phi_{ij} = \phi^{*}_{ij} + \alpha d^{-1}_{ij}\{l1_{i-1j}\phi^{*}_{i-1j} + ul_{i+1j}\phi^{*}_{i+1j} + l2_{ij-1}\phi^{*}_{ij-1} + u2_{ij+1}\phi^{*}_{ij+1} + b_{ij}-d_{ij}\phi^{*}_{ij}\}$
                                                   $i=1(1)n$
                                                   $j=1(1)m$

(3)   check for convergence of $\phi_{ij}$ with $\phi^{*}_{ij}$. If not converged and
      iter$\leq$MAXIT
      $\phi^{*}_{ij} = \phi_{ij}$
      iter = iter + 1
      goto step (2)
```

Since all steps of the algorithm are explicit the vector algorithm would be the same. However, for an efficient implementation on the VA-1 processor the nested loop in step (2) is replaced by a single loop. The approximation field ϕ and the coefficients *d*, *l1*, *l2*, *u1*, *u2* are represented by contiguous vectors. This is illustrated in figure 4.7.1-2 for a grid where $m=n=3$. Therefore in the description of the vector algorithm the operations are of length *nm*.

ALGORITHM 4.7.1-2 : Vector JUR algorithm

```
(1)   set iter=1
(2)   $\phi = \phi^* + \alpha\{l1\#\phi^*(-1) + u1\#\phi^*(1) + l2\#\phi^*(-n) + u2\#\phi^*(n) + b - d\#\phi^*\}\backslash d$
(3)   check for convergence of $\phi$ with $\phi^*$. If not converged and iter≤MAXIT
          $\phi^* = \phi$
          iter = iter + 1
          goto step (2)
```

4.7.2 The point-by-point SOR algorithm

In the JUR algorithm all neighbouring approximations were assumed from the previous iteration. However, the approximations at nodes ϕ_{i-1j} and ϕ_{ij-1} are known for the current iteration, it would be better to use these latest approximations rather than their old values from the previous iteration (figure 4.7.1-1), this is the essence of the Gauss-Seidel algorithm and can be described by the iteration

$$a_{ij}\phi_{ij} = a_{i-1j}\phi_{i-1j} + a_{i+1j}\phi^*_{i+1j} + a_{ij-1}\phi_{ij-1} + a_{ij+1}\phi^*_{ij+1} + b_{ij} \qquad \begin{array}{l} i=1(1)n \\ j=1(1)m \end{array} \qquad (4.7.2\text{-}1)$$

$$\Rightarrow \phi_{ij} = a^{-1}_{ij}\{a_{i-1j}\phi_{i-1j} + a_{i+1j}\phi^*_{i+1j} + a_{ij-1}\phi_{ij-1} + a_{ij+1}\phi^*_{ij+1} + b_{ij}\} \qquad \begin{array}{l} i=1(1)n \\ j=1(1)m \end{array} \qquad (4.7.2\text{-}2)$$

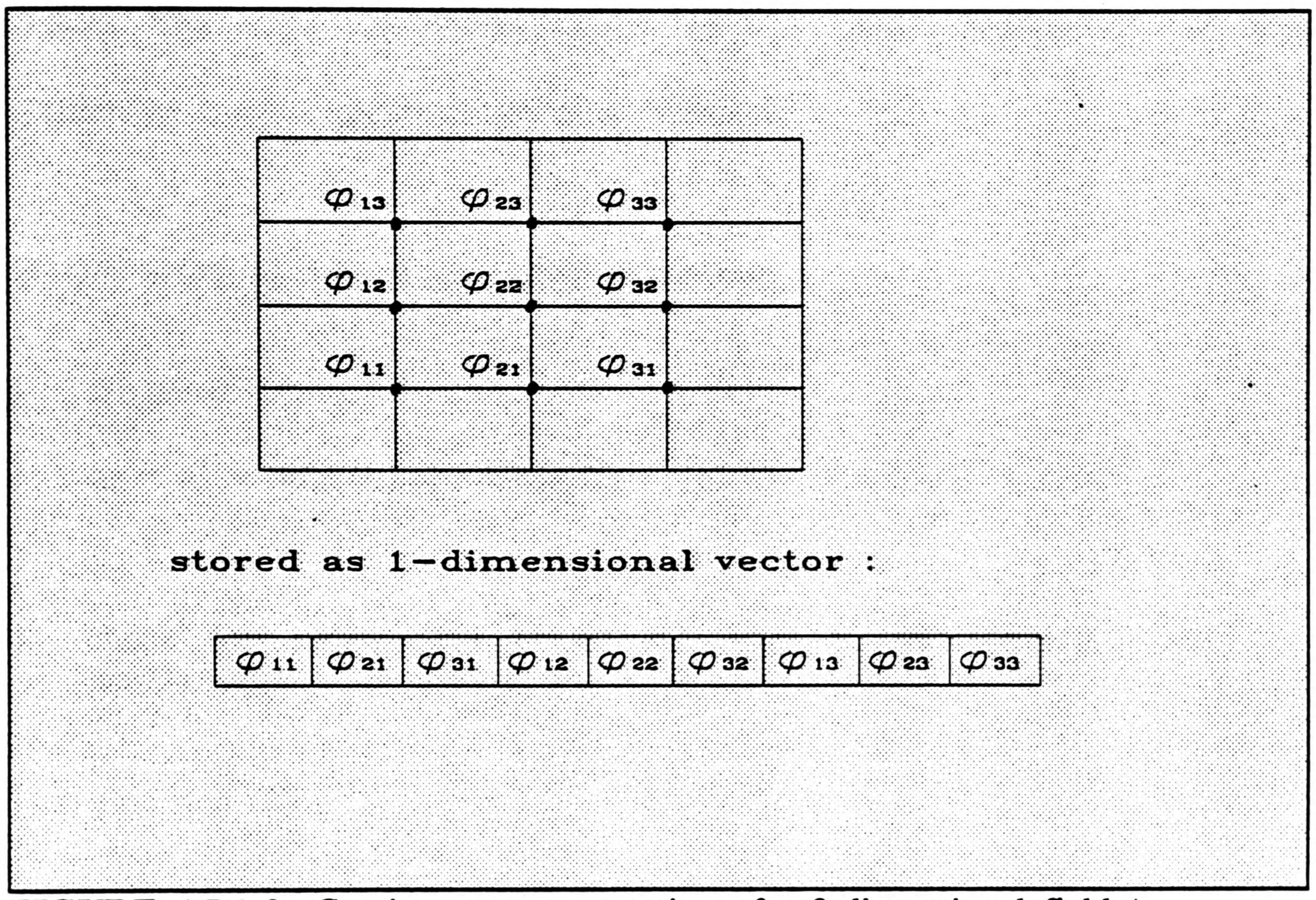


FIGURE 4.7.1-2 Contiguous representation of a 2-dimensional field ϕ

Relaxation can be introduced into the algorithm in the usual way

$$\phi_{ij} = \phi^{*}_{ij} + \alpha a^{-1}_{ij}\{a_{i-1j}\phi_{i-1j} + a_{i+1j}\phi^{*}_{i+1j} + a_{ij-1}\phi_{ij-1} + a_{ij+1}\phi^{*}_{ij+1} + b_{ij} - a_{ij}\phi^{*}_{ij}\} \quad \begin{array}{l} i=1(1)n \\ j=1(1)m \end{array} \quad (4.7.2\text{-}3)$$

This now defines the point-by-point SOR algorithm. To ensure convergence α must lie in the range $0<\alpha<2$. For an under-relaxed approximation to the solution α must lie in the range $0<\alpha<1$, for an over-relaxed approximation α lies in the range $1<\alpha<2$ and if $\alpha=1$ then the algorithm reverts to the Gauss-Seidel algorithm (4.7.2-2). The main computation steps for the implementation of the SOR algorithm on a scalar processor are now presented.

ALGORITHM 4.7.2-1 : Scalar SOR algorithm

```
(1)  set iter=1
(2)  $\phi_{ij} = \phi^{*}_{ij} + \alpha d^{-1}_{ij}\{l1_{i-1j}\phi_{i-1j} + u1_{i+1j}\phi^{*}_{i+1j} + l2_{ij-1}\phi_{ij-1} + u2_{ij+1}\phi^{*}_{ij+1} + b_{ij} - d_{ij}\phi^{*}_{ij}\}$
                                                    $i=1(1)n$
                                                    $j=1(1)m$
(3)  check for convergence of $\phi_{ij}$ with $\phi^{*}_{ij}$. If not converged and
     $iter \le MAXIT$
     $\phi^{*}_{ij} = \phi_{ij}$
     iter = iter + 1
     goto step (2)
```

Vectorisation is prohibited by the inherently scalar formulation of the algorithm, whereby approximations for the current iteration reside on both sides of the equation, for example, ϕ_{ij} is on the left-hand-side while ϕ_{i-1j} and ϕ_{ij-1} are on the right-hand-side. This data dependency can be broken by re-writing the step as a series of explicit iterative ones, this is a technique similar to that used for the Thomas algorithm. Hence equation (4.7.2-3) can be written as

$$a_{ij}\phi_{ij} - \alpha(a_{i-1j}\phi_{i-1j} + a_{ij-1}\phi_{ij-1}) = (1-\alpha)a_{ij}\phi^{*}_{ij} + \alpha(a_{i+1j}\phi^{*}_{i+1j} + a_{ij+1}\phi^{*}_{ij+1} + b_{ij}) \quad \begin{array}{l} i=1(1)n \\ j=1(1)m \end{array} \quad (4.7.2\text{-}4)$$

If we define

$$p_{ij} = \phi_{ij} \quad (4.7.2\text{-}5)$$

$$q_{ij} = (1-\alpha)a_{ij}\phi^{*}_{ij} + \alpha(a_{i+1j}\phi^{*}_{i+1j} + a_{ij+1}\phi^{*}_{ij+1} + b_{ij}) \quad (4.7.2\text{-}6)$$

$$\Rightarrow a_{ij}p_{ij} - \alpha(a_{i-1j}p_{i-1j} + a_{ij-1}p_{ij-1}) = q_{ij} \quad (4.7.2\text{-}7)$$

this can be expressed iteratively as

$$p^{(k+1)}_{ij} = a_i^{-1}(q_{ij} + \alpha(a_{i-1j}p^{(k)}_{i-1j} + a_{ij-1}p^{(k)}_{ij-1})) \quad (4.7.2\text{-}8)$$

Thus the vectorised point-by-point SOR algorithm has a nested iteration structure. In the description of this algorithm the vector operations are of length *nm*.

ALGORITHM 4.7.2-2 : Vector SOR algorithm

```
(1)  set iter=1

(2)  set k=1
         $\mathbf{p}^{(0)} = \phi^{*}$

(3)  $\mathbf{q} = (1-\alpha)d\#\phi^{*} + \alpha(u1\#\phi^{*}(1) + u2\#\phi^{*}(n) + \mathbf{b})$

(4)  $\mathbf{p}^{(k+1)} = (\mathbf{q} + \alpha(l1\#\mathbf{p}^{(k)}(-1) + l2\#\mathbf{p}^{(k)}(-n)))$

(5)  check for convergence of $\mathbf{p}^{(k+1)}$ with $\mathbf{p}^{(k)}$. If not converged and
     $k \leq MAXIT$
     $\mathbf{p}^{(k)} = \mathbf{p}^{(k+1)}$
     $k = k + 1$
     goto step (4)

(6)  set $\phi = \mathbf{p}^{(k+1)}$

(7)  check for convergence of $\phi$ with $\phi^{*}$. If not converged and
     $iter \leq MAXIT$
     $\phi^{*} = \phi$
     $iter = iter + 1$
     goto step (2)
```

4.7.3 The RBSOR algorithm

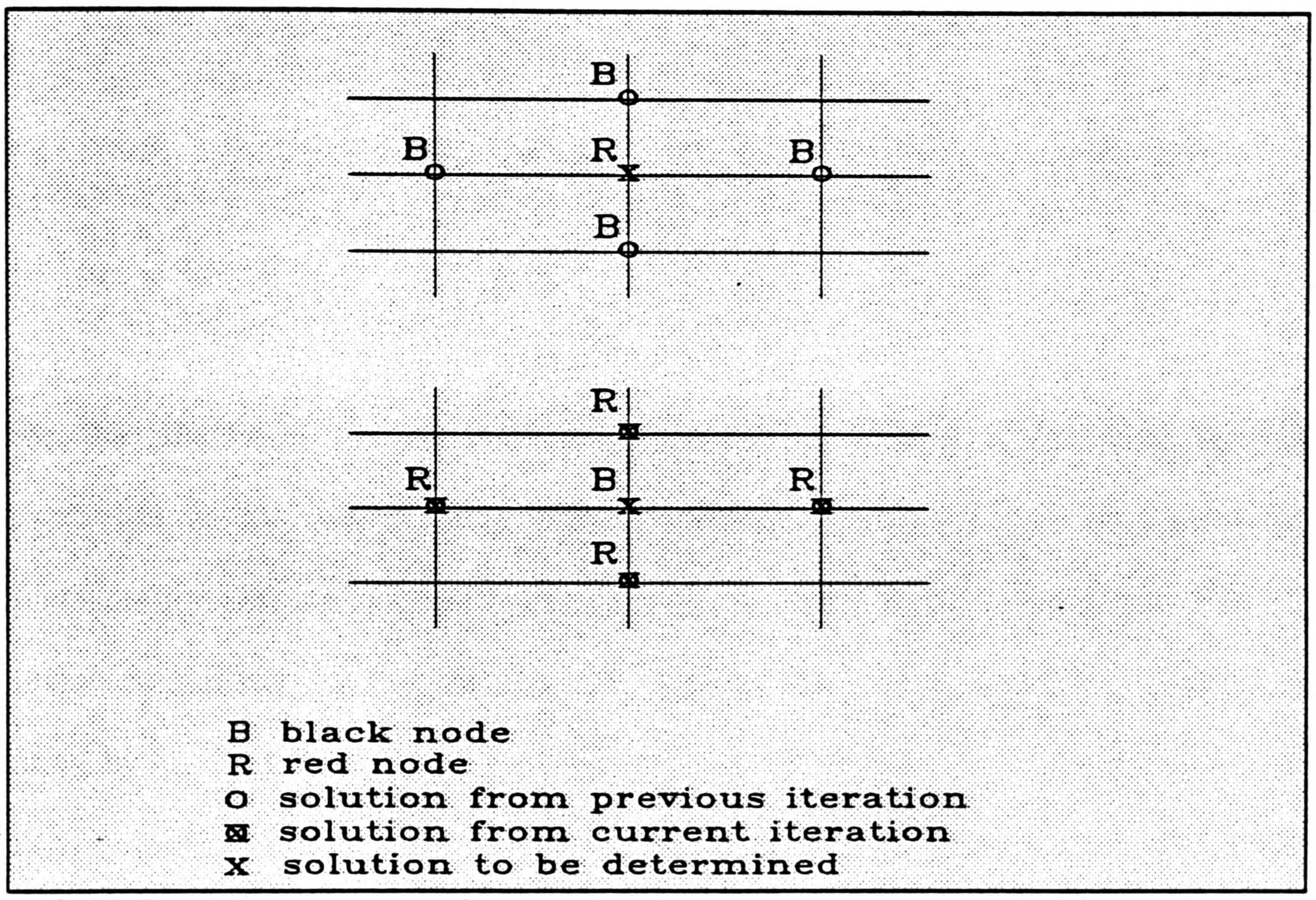
In both the JUR and SOR algorithms the order in which all nodes were updated was based on a natural ordering. However, there are many other orderings which could have been adopted (O'leary [1984] and Adams and Jordan [1986]). These ordering schemes have been applied to the SOR algorithm the most popular being the red-black ordering. This has many other names such as 'chessboard', 'odd-even', or '2-colour' ordering. The order in which nodes are updated differs from the natural ordering in that all nodes of one colour are updated first, followed by the updating of the second colour.

A single RBSOR iteration consists of visiting all red nodes in a natural order, followed by visiting all black nodes in the same manner. When visiting all red nodes the only information required is based on the neighbouring black nodes which are approximations from the previous iteration. Similarly, when updating the black nodes only information regarding the red nodes is required, these are approximations from the current iteration (figure 4.7.3-1). Thus a single RBSOR iteration can be defined as comprising an update of red nodes

$$\phi_{ij} = \phi^{\bullet}_{ij} + \alpha a^{-1}_{ij}\{a_{i-1j}\phi^{\bullet}_{i-1j} + a_{i+1j}\phi^{\bullet}_{i+1j} + a_{ij-1}\phi^{\bullet}_{ij-1} + a_{ij+1}\phi^{\bullet}_{ij+1} + b_{ij} - a_{ij}\phi^{\bullet}_{ij}\} \quad \begin{array}{l} i=1(1)n \\ j=1(2)m \end{array} \quad (4.7.3\text{-}1a)$$

followed by an update of black nodes

$$\phi_{ij} = \phi^{\bullet}_{ij} + \alpha a^{-1}_{ij}\{a_{i-1j}\phi_{i-1j} + a_{i+1j}\phi_{i+1j} + a_{ij-1}\phi_{ij-1} + a_{ij+1}\phi_{ij+1} + b_{ij} - a_{ij}\phi^{\bullet}_{ij}\} \quad \begin{array}{l} i=1(1)n \\ j=2(2)m \end{array} \quad (4.7.3\text{-}1b)$$



B
B
R
B
B
R
R
B
R
R
B black node
R red node
O solution from previous iteration
⊠ solution from current iteration
X solution to be determined

FIGURE 4.7.3-1 Update of approximations using the red-black SOR algorithm

ALGORITHM 4.7.3-1 : Scalar RBSOR algorithm

```
(1)  set iter=1

 red node update
(2)  $\phi_{ij} = \phi^*_{ij} + \alpha a^{-1}_{ij}\{a_{i-1j}\phi^*_{i-1j} + a_{i+1j}\phi^*_{i+1j} + a_{ij-1}\phi^*_{ij-1} + a_{ij+1}\phi^*_{ij+1} + b_{ij} - a_{ij}\phi^*_{ij}\}$
                                                                  i=1(1)n
                                                                  j=1(2)m

 black node update
(3)  $\phi_{ij} = \phi^*_{ij} + \alpha a^{-1}_{ij}\{a_{i-1j}\phi_{i-1j} + a_{i+1j}\phi_{i+1j} + a_{ij-1}\phi_{ij-1} + a_{ij+1}\phi_{ij+1} + b_{ij} - a_{ij}\phi^*_{ij}\}$
                                                                  i=1(1)n
                                                                  j=2(2)m

(4)  check for convergence of $\phi_{ij}$ with $\phi^*_{ij}$. If not converged and
     iter≤MAXIT
     $\phi^*_{ij} = \phi_{ij}$
     iter = iter + 1
     goto step (2)
```

Since the algorithm has no data dependencies but still has the same rate of convergence as the natural order SOR algorithm (Young [1971]) this makes the implementation of the algorithm on a pipeline architecture worthwhile. For an efficient vectorised implementation, all vectors are partitioned into subvectors defined by

$$\mathbf{a}^R = [a_1, a_2, \ldots, a_r]^T \qquad (4.7.3\text{-}2a)$$

$$\mathbf{a}^B = [a_1, a_2, \ldots, a_b]^T \qquad (4.7.3\text{-}2b)$$

where

$$r = \begin{cases} nm\text{-}1 & \text{for } m \text{ even} \\ nm & \text{for } m \text{ odd} \end{cases} \qquad (4.7.3\text{-}3a)$$

$$b = \begin{cases} nm & \text{for } m \text{ even} \\ nm\text{-}1 & \text{for } m \text{ odd} \end{cases} \qquad (4.7.3\text{-}3b)$$

In the above definition *n* must be odd so that all vectors can be easily referenced using a stride of 2. If *n* is even then a uniform stride cannot be used and vectors of pointers are needed to access successive red and black nodes (figure 4.7.3-2). Assuming that n is odd the main steps of the vector algorithm are now given.

ALGORITHM 4.7.3-2 : Vector RBSOR algorithm

```
(1)  set iter=1

  red node update
(2)  $\phi^R = \phi^{*R} + \alpha[\mathbf{\mathit{l1}}^R\#\{\phi^*(-1)\}^R + \mathbf{\mathit{u1}}^R\#\{\phi^*(1)\}^R + \mathbf{\mathit{l2}}^R\#\{\phi^*(-n)\}^R +$
                         $\mathbf{\mathit{u2}}^R\#\{\phi^*(n)\}^R + b^R - \mathbf{\mathit{d}}^R\#\phi^{*R}]\backslash \mathbf{\mathit{d}}^R$

  black node update
(3)  $\phi^B = \phi^{*B} + \alpha[\mathbf{\mathit{l1}}^B\#\{\phi(-1)\}^B + \mathbf{\mathit{u1}}^B\#\{\phi(1)\}^B + \mathbf{\mathit{l2}}^B\#\{\phi(-n)\}^B +$
                         $\mathbf{\mathit{u2}}^B\#\{\phi(n)\}^B + b^B - \mathbf{\mathit{d}}^B\#\phi^{*B}\}\backslash \mathbf{\mathit{d}}^B$

(4)  check for convergence of $\phi$ with $\phi^*$. If not converged and iter$\leq$MAXIT
     $\phi^* = \phi$
     iter = iter + 1
     goto step (2)
```

4.7.4 The conjugate gradient algorithm with a Jacobi preconditioner (JCG)

The popularity of conjugate gradient algorithm has grown over the last decade particularly with research into numerous preconditioners. Amongst the most widely used are the Jacobi or diagonal preconditioner, block factorisations and the incomplete cholesky (ICCG) factorisation (see for example Kightley and Jones [1985] and van der Vorst et al [1982, 1986]).

The attraction of the conjugate gradient algorithm is that a large proportion of the steps involve matrix and vector operations which are ideal for implementation on pipeline processors. The choice of preconditioner is a difficult task as this depends largely on the problem being solved. Here a simple Jacobi preconditioner is

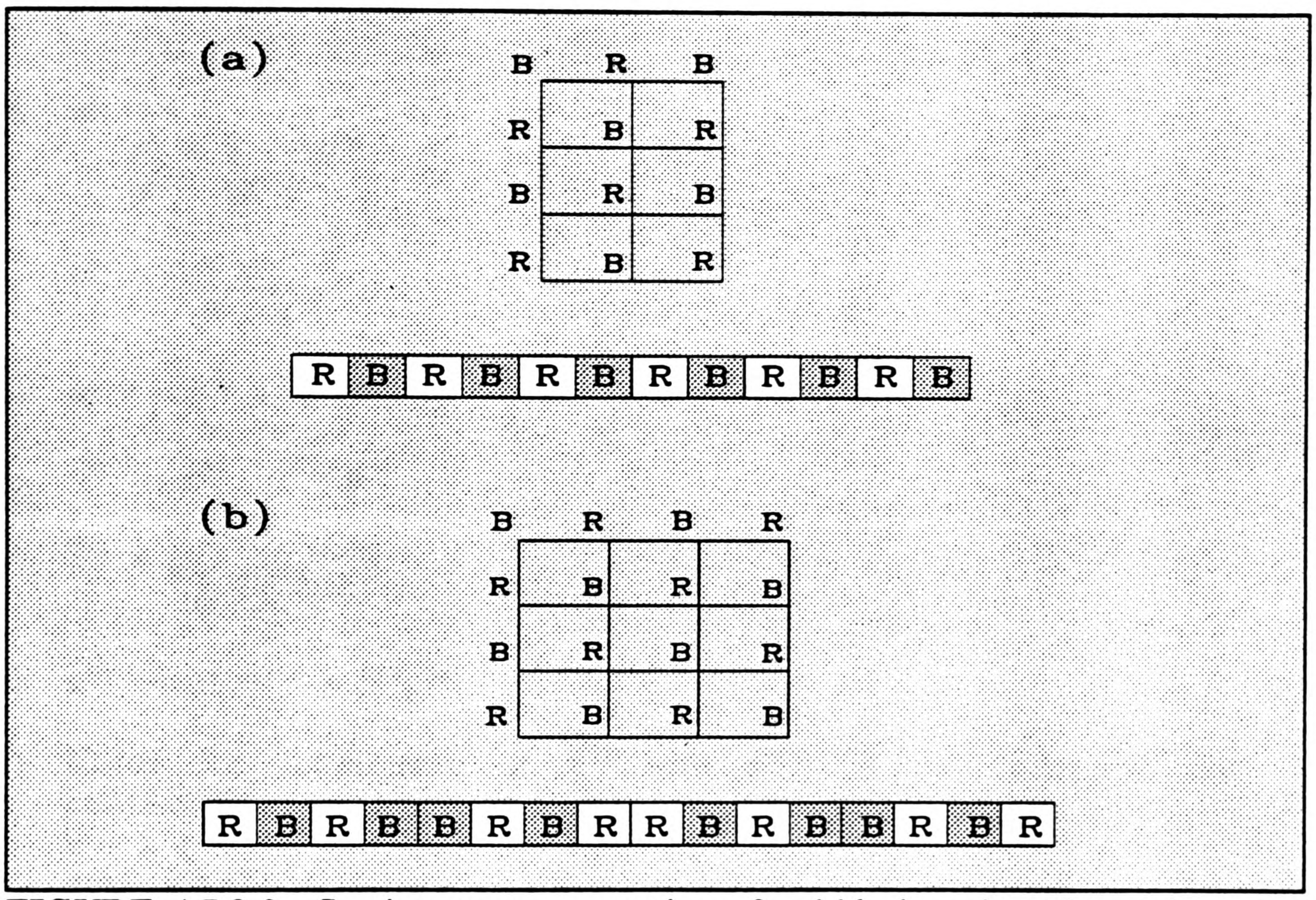


FIGURE 4.7.3-2 Contiguous representation of red-black nodes when *n* is (a) odd (b) even implemented since it has been shown to be as competitive as the more elaborate preconditioners for some problems (Kightley and Jones [1985] and Kincaid et al [1986]).

Given the matrix system defined by (4.5-7) and also an initial approximation to the solution $\phi^{(0)}$ and a preconditioning matrix M, then the preconditioned conjugate gradient algorithm is defined by the following iterative steps

```
(1)  set k=1
         $\mathbf{r}^{(0)} = \mathbf{b} - A\phi^{(0)}$
         $\mathbf{p}^{(0)} = M^{-1}\mathbf{r}^{(0)}$

(2)  $\alpha^{(k)} = \dfrac{(\mathbf{r}^{(k)},M^{-1}\mathbf{r}^{(k)})}{(\mathbf{p}^{(k)},A\mathbf{p}^{(k)})}$

(3)  $\phi^{(k+1)} = \phi^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$
                                                                    (4.7.4-1)
(4)  $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)}A\mathbf{p}^{(k)}$

(5)  $\beta^{(k)} = \dfrac{(\mathbf{r}^{(k+1)},M^{-1}\mathbf{r}^{(k+1)})}{(\mathbf{r}^{(k)},M^{-1}\mathbf{r}^{(k)})}$

(6)  $\mathbf{p}^{(k+1)} = M^{-1}\mathbf{r}^{(k+1)} + \beta^{(k)}\mathbf{p}^{(k)}$

(7)  Check for convergence of $\phi^{(k)}$ with $\phi^{(k+1)}$. If not converged and
     $k \le MAXIT$
     k = k + 1
     goto step (2)
```

A preconditioning matrix is required to accelerate the convergence rate of the standard conjugate gradient algorithm. The effect of the preconditioner is to represent the matrix A with a smaller condition number. In the case where the preconditioning matrix is the main diagonal $M=D_A$ of A, this resulting algorithm is equivalent to a polynomial acceleration of the basic Jacobi algorithm (Hageman and Young [1981]) and hence the name. In this algorithm $M^{-1}\mathbf{r}$ is not carried out within the iterative process. By scaling the original matrix A so that it has a unit main diagonal and still preserving its symmetric sparse structure, the algorithm may be simplified. This algorithm is inherently explicit and therefore the scalar and vector algorithms are equivalent. The main computational steps of the JCG algorithm are given below.

ALGORITHM 4.7.4-1 : Scalar/vector JCG algorithm

```
 Scale the system represented by equation (4.5-7) by evaluating
(1)   $\tilde{A} = D_A^{-1/2} A D_A^{-1/2}$
      $\tilde{\mathbf{b}} = D_A^{-1/2}\mathbf{b}$
      $\tilde{\phi} = D_A^{1/2}\phi$

 For the preconditioning matrix M=I,
(2.1) set k=1
      $\mathbf{r}^{(0)} = \tilde{\mathbf{b}} - \tilde{A}\tilde{\phi}^{(0)}$
      $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$

(2.2) $\alpha^{(k)} = \dfrac{(\mathbf{r}^{(k)},\mathbf{r}^{(k)})}{(\mathbf{p}^{(k)},\tilde{A}\mathbf{p}^{(k)})}$

(2.3) $\tilde{\phi}^{(k+1)} = \tilde{\phi}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$

(2.4) $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)}\tilde{A}\mathbf{p}^{(k)}$

(2.5) $\beta^{(k)} = \dfrac{(\mathbf{r}^{(k+1)},\mathbf{r}^{(k+1)})}{(\mathbf{r}^{(k)},\mathbf{r}^{(k)})}$

(2.6) $\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta^{(k)}\mathbf{p}^{(k)}$

(2.7) Check for convergence of $\tilde{\phi}^{(k)}$ with $\tilde{\phi}^{(k+1)}$. If not converged and
      k≤MAXIT
      $k = k + 1$
      goto step (2.2)

 Scale back the solution
(3)   $\tilde{\phi} = D_A^{-1/2}\phi$
```

4.8 Results for the Laplace equation using pentadiagonal algorithms

All of these algorithms contain an iterative process and as such a convergence criteria is needed to terminate the iterations. Here a simple criteria based on the absolute difference approximations is used,

$$|\phi_j^* - \phi_j| < \zeta \qquad \forall\, j \tag{4.8-1}$$

where the pre-defined tolerance ζ is set to 10^{-5}. In each of the relaxation algorithms the relaxation parameters were set to a near optimum value, in the JUR algorithm α=1.0 and in the SOR and RBSOR α=1.8. A uniform grid was used and the number of nodes varied from 5x5 to 45x45.

Table 4.8-1 shows the variation of CPU time with grid size and the speed-up factors (S) for each algorithm considered. The comparison between the scalar and vector JUR algorithms is shown in figure 4.8-1 and indicates that the vector algorithm is far superior. The speed-up factors increase as the grid is increased and range from a factor of 2 for a 5x5 grid up to 90 for a 45x45 grid. A factor of 90 appears to be the maximum speed-up that can be obtained (figure 4.8-2).

The comparison between the scalar and vector SOR algorithms is shown in figure 4.8-3, this indicates that the vector SOR is generally more efficient. However, for small grid sizes up to 12x12 the scalar algorithm is marginally faster. This is highlighted by the speed-up factor graph (figure 4.8-4). This is because for a small grid size the nested iteration in the vector algorithm is more computationally expensive than the single iteration process in the scalar algorithm. This overhead becomes insignificant as the grid sizes are increased because the pipeline processor is being used more efficiently. A maximum speed-up of 9 can be achieved when using the vector algorithm over its scalar counterpart.

GRID SIZE	5	11	15	21	25	31	35	41	45
SCALAR JUR	.7	12.25	37.65	129.83	243.9	525.8	811.5	1422.38	1975.18
VECTOR JUR	.26	.53	.81	1.88	3.22	6.38	9.48	16.05	22.07
SPEED-UP S	2.7	23.1	46.5	69.1	75.7	82.4	85.6	88.6	89.5
VECTOR FRACTION f_v	.9981	.9956	.9955	.9966	.9975	.9982	.9989	.9991	.9979
SCALAR SOR	.13	1.8	6.1	21.9	42.1	94.2	147.9	265.9	374.4
VECTOR SOR	1.43	2.1	2.51	3.92	6.2	12.3	17.88	29.40	40.08
SPEED-UP S	.1	.9	2.9	5.6	6.8	7.7	8.3	9.0	9.3
VECTOR FRACTION f_v	.9992	.9975	.9984	.9992	.9994	.9996	.9997	.9998	.9998
SCALAR RBSOR	.15	1.8	6.1	21.9	42.4	94.8	148.8	266.8	377.2
VECTOR RBSOR	.23	.18	.35	.47	.73	1.43	2.12	3.61	5.1
SPEED-UP S	.7	10.0	17.4	46.6	58.1	66.3	70.2	73.9	74.0
VECTOR FRACTION f_v	.9971	.9956	.9914	.9879	.9895	.9923	.9936	.9951	.9960
SCALAR JCG	.08	.85	1.96	5.42	9.16	17.23	24.53	38.75	51.1
VECTOR JCG	.18	.19	.18	.18	.22	.33	.43	.63	.83
SPEED-UP S	.4	4.5	10.9	30.1	41.6	52.2	57.0	61.5	61.6
VECTOR FRACTION f_v	.9996	.9991	.9988	.9983	.9985	.9989	.9990	.9991	.9993

TABLE 4.8-1 Results for the JUR, SOR, RBSOR and JCG algorithms

The comparison between the scalar and vector RBSOR algorithms is shown in figure 4.8-5, this indicates that the vector algorithm is superior. However, for small grid sizes up to 6x6 the scalar algorithm is marginally faster. The reason for this is the relatively short vector lengths used in the vector computations which are of length $n^2/2$. Figure 4.8-6 shows that a speed-up factor of up to 74 can be achieved for the vectorisation of this algorithm.

The comparison of the scalar and vector JCG algorithms is shown in figure 4.8-7. The results show that the vector algorithm is superior to the scalar algorithm when the grid size is large, and only marginally worse in the case where the grid size is small. The variation of speed-up with grid size shows that a factor of up to 62 can be achieved when the vector JCG algorithm is compared to its scalar equivalent (figure 4.8-8).

Using Amdahl's law, an analysis can be carried out to determine what theoretical speed-up factors can be achieved. The analysis can be carried out on all of the algorithms considered with the exception of the SOR algorithm since the vector and scalar algorithms have different structures. Practically the entire code can be vectorised for this problem shown by $f_v \approx 1.0$ in table 4.8-1, and therefore the analysis is trivial and the expected speed-up factors will lie between 4 and 170.

There are two reasons for the high speed-up factors which have actually been achieved. The first has already been mentioned and deals with the 'ideal' situation where near full vectorisation has been achieved. The second is the efficient utilisation of the architecture where vector operations were typically of length n^2 for the JUR and JCG algorithms and $n^2/2$ for the RBSOR algorithm.

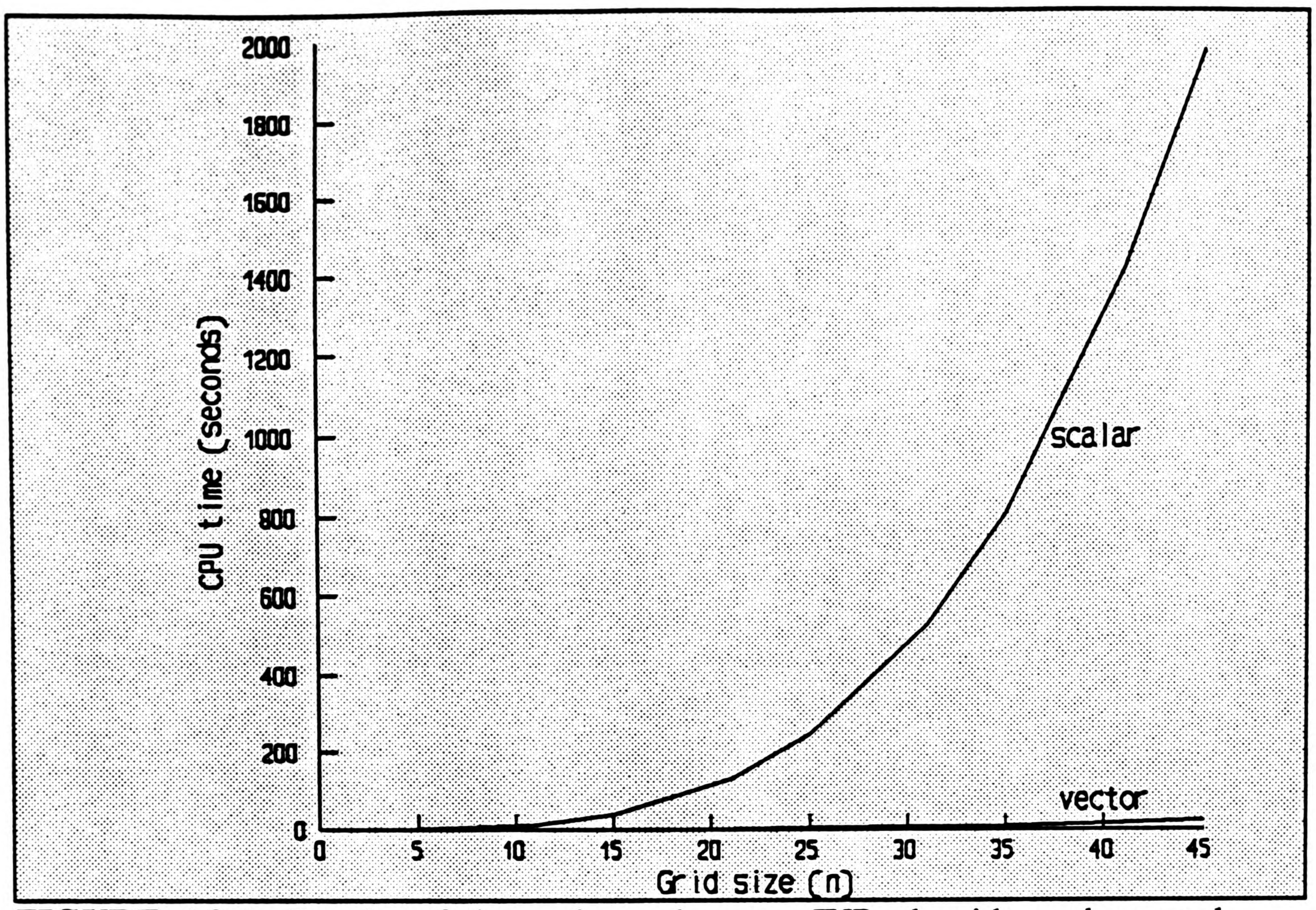


FIGURE 4.8-1 Results of the scalar and vector JUR algorithm when used to solve the Laplace equation

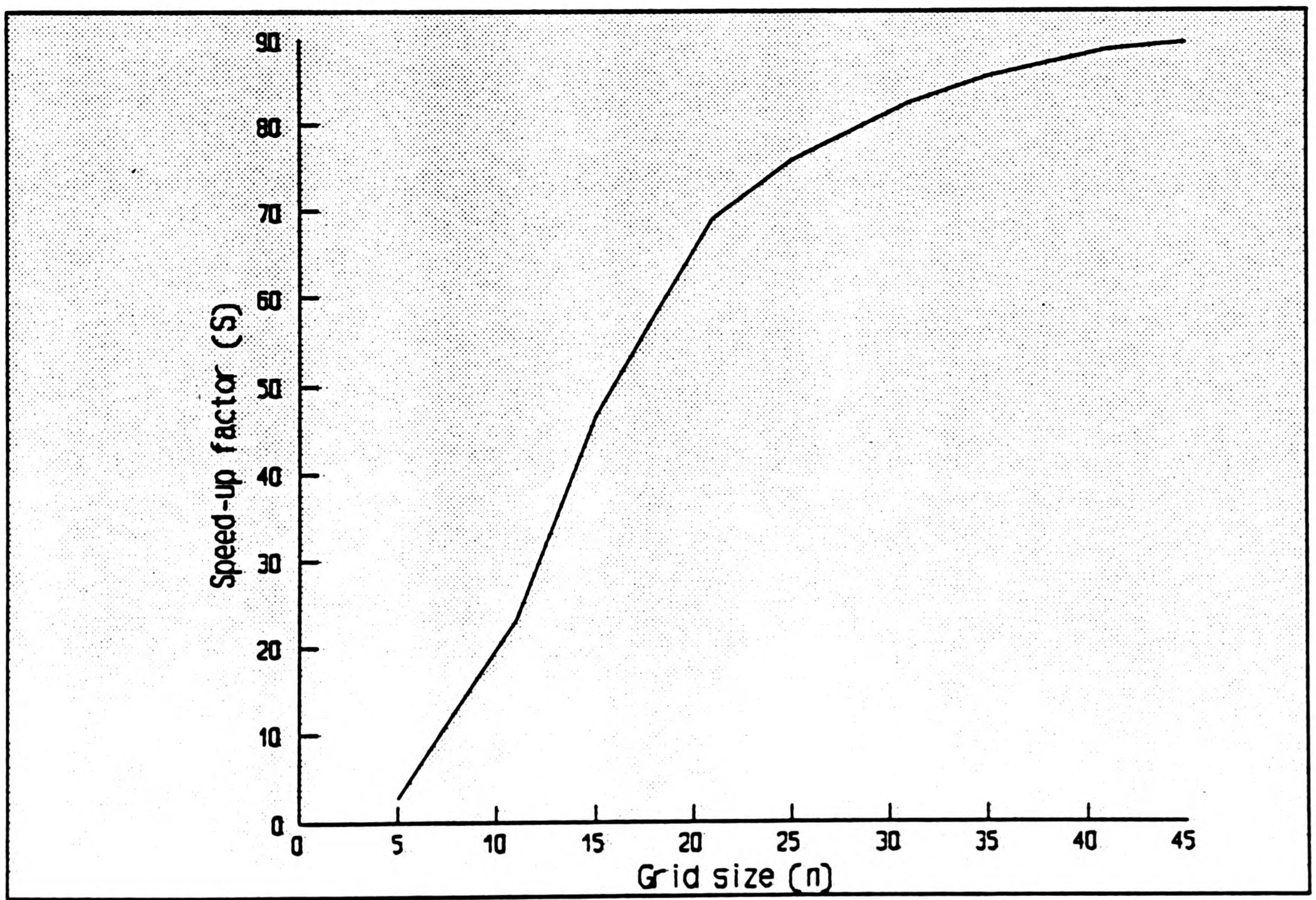


FIGURE 4.8-2 Speed-up factors achieved for the JUR algorithm

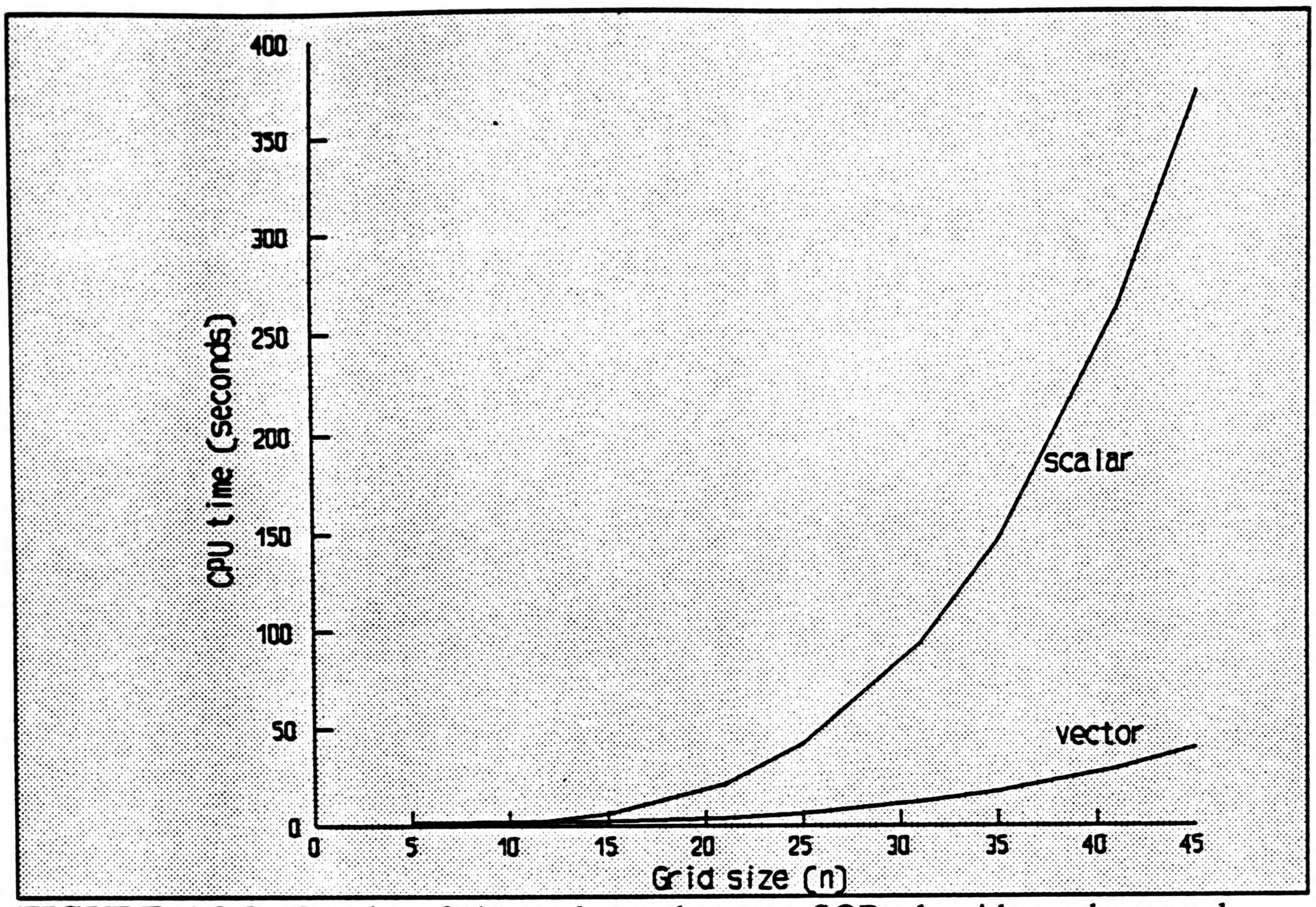


FIGURE 4.8-3 Results of the scalar and vector SOR algorithm when used to solve the Laplace equation

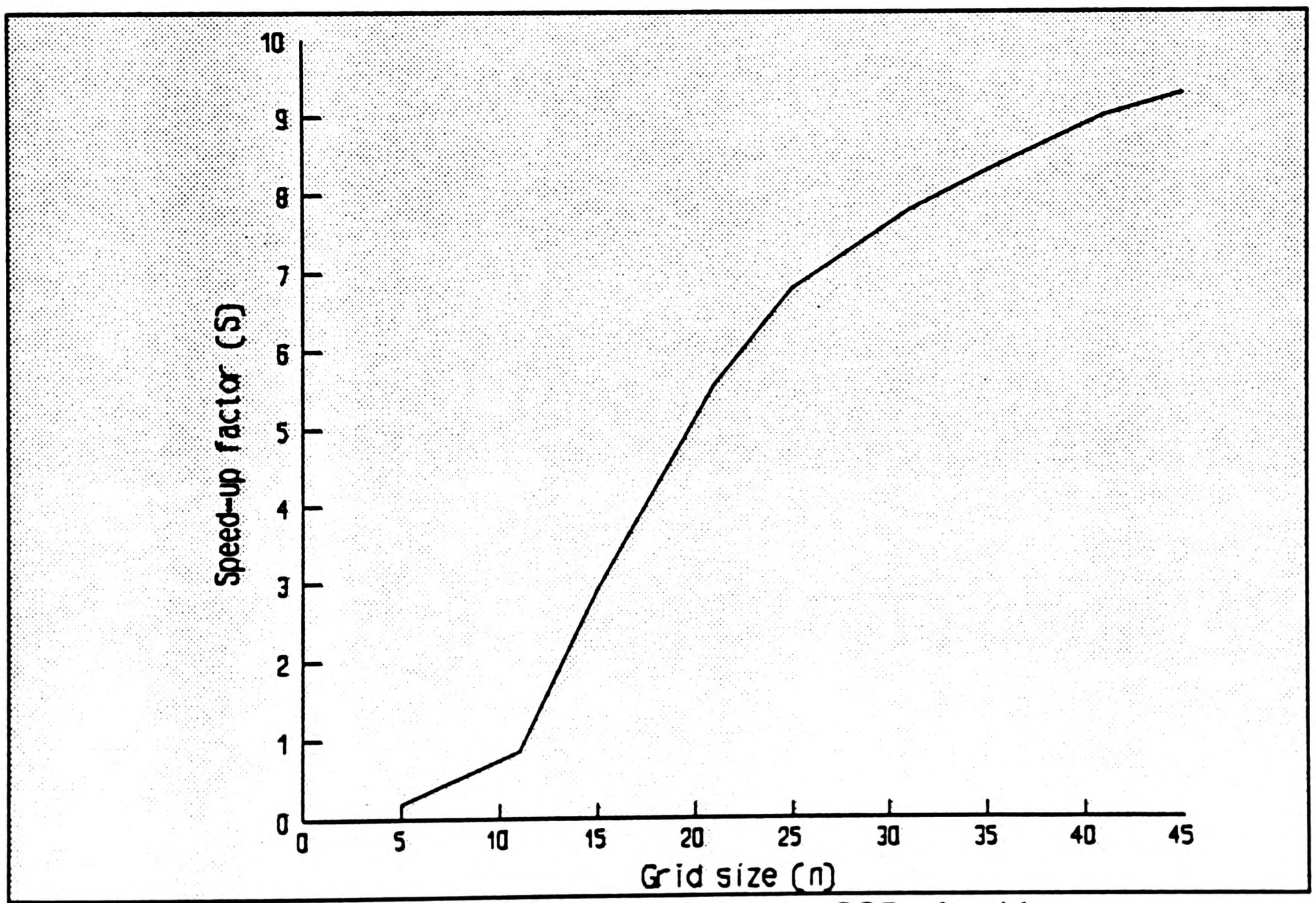


FIGURE 4.8-4 Speed-up factors achieved for the SOR algorithm

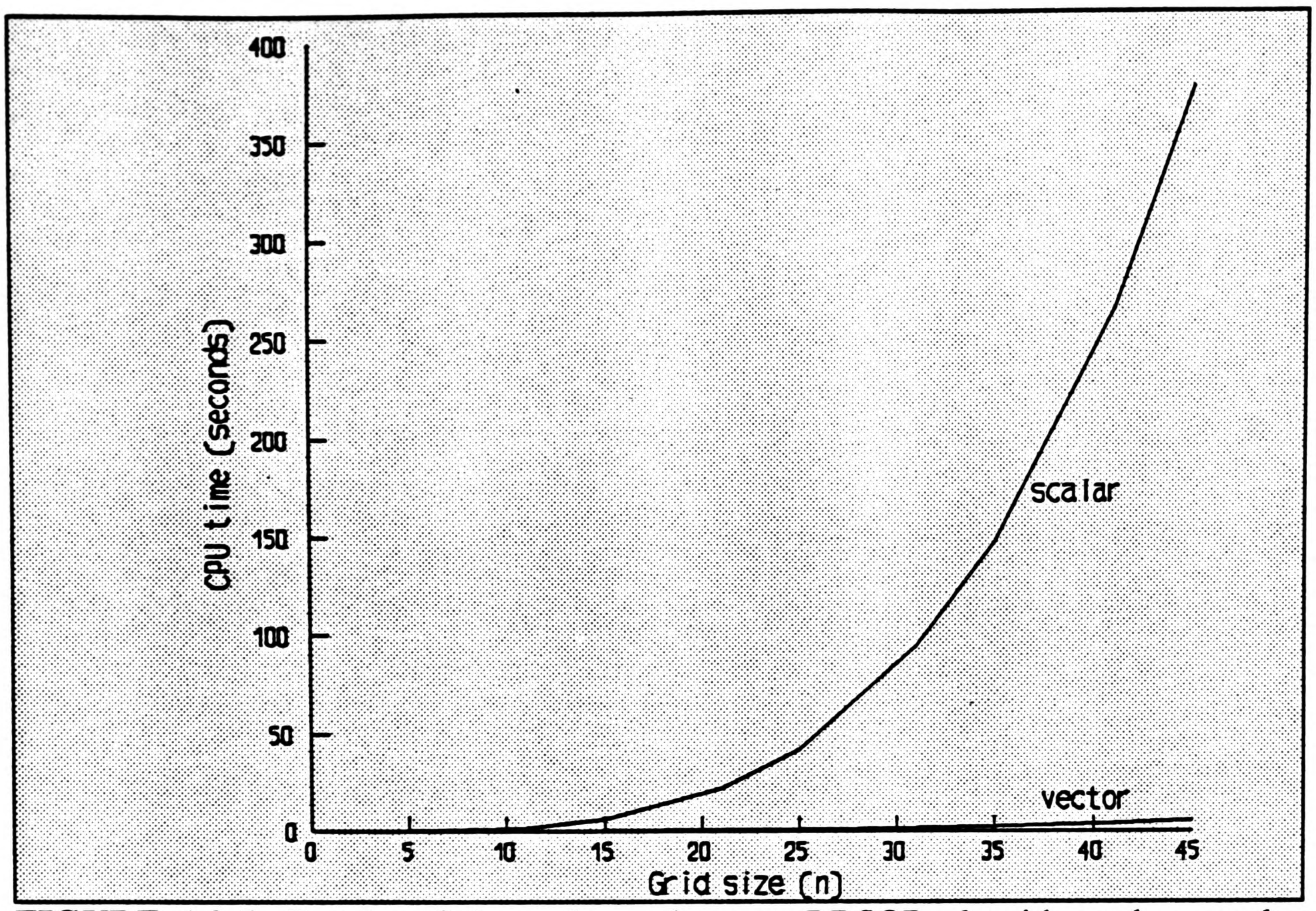


FIGURE 4.8-5 Results of the scalar and vector RBSOR algorithm when used to solve the Laplace equation

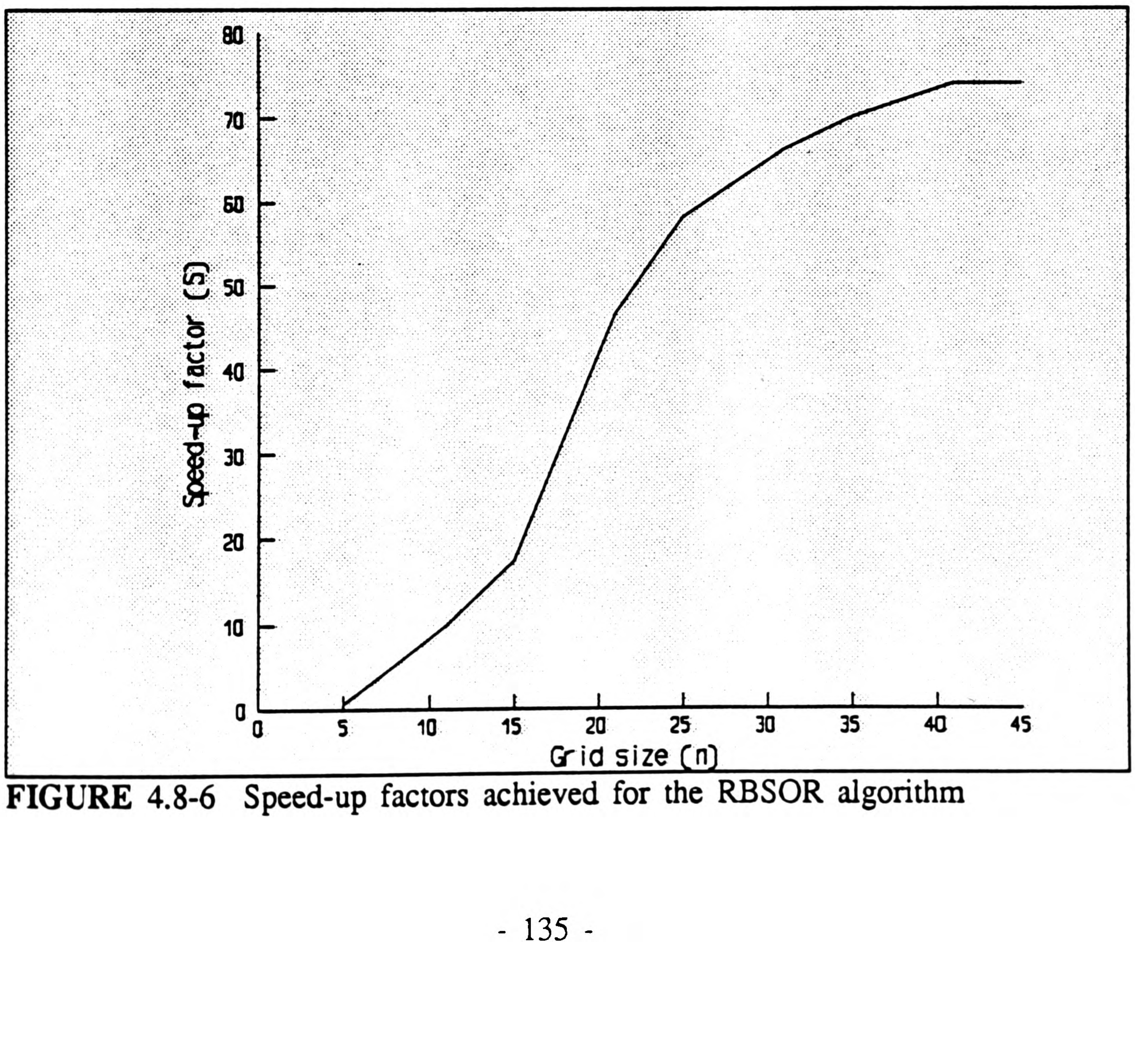


FIGURE 4.8-6 Speed-up factors achieved for the RBSOR algorithm

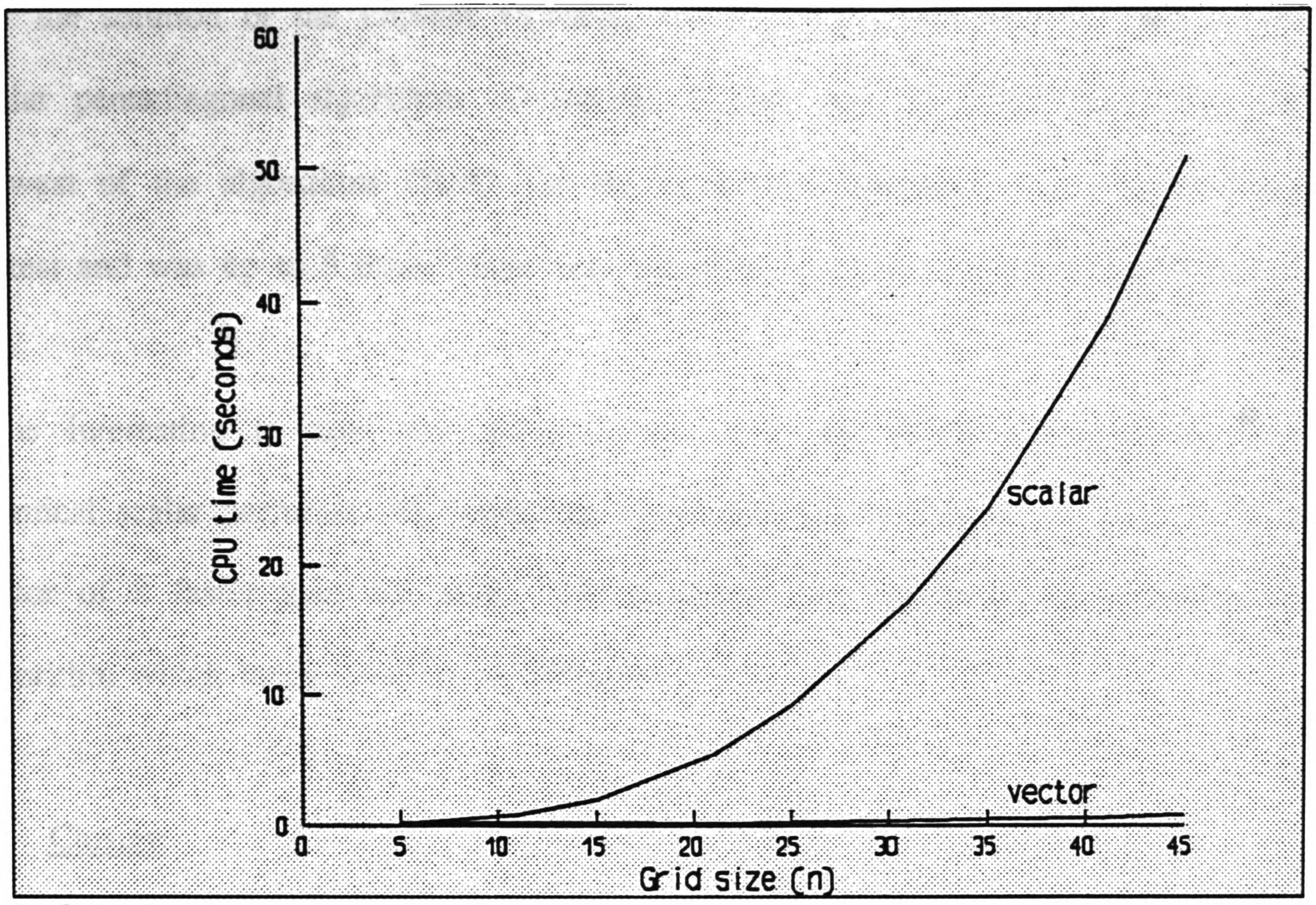


FIGURE 4.8-7 Results of the scalar and vector JCG algorithm when used to solve the Laplace equation

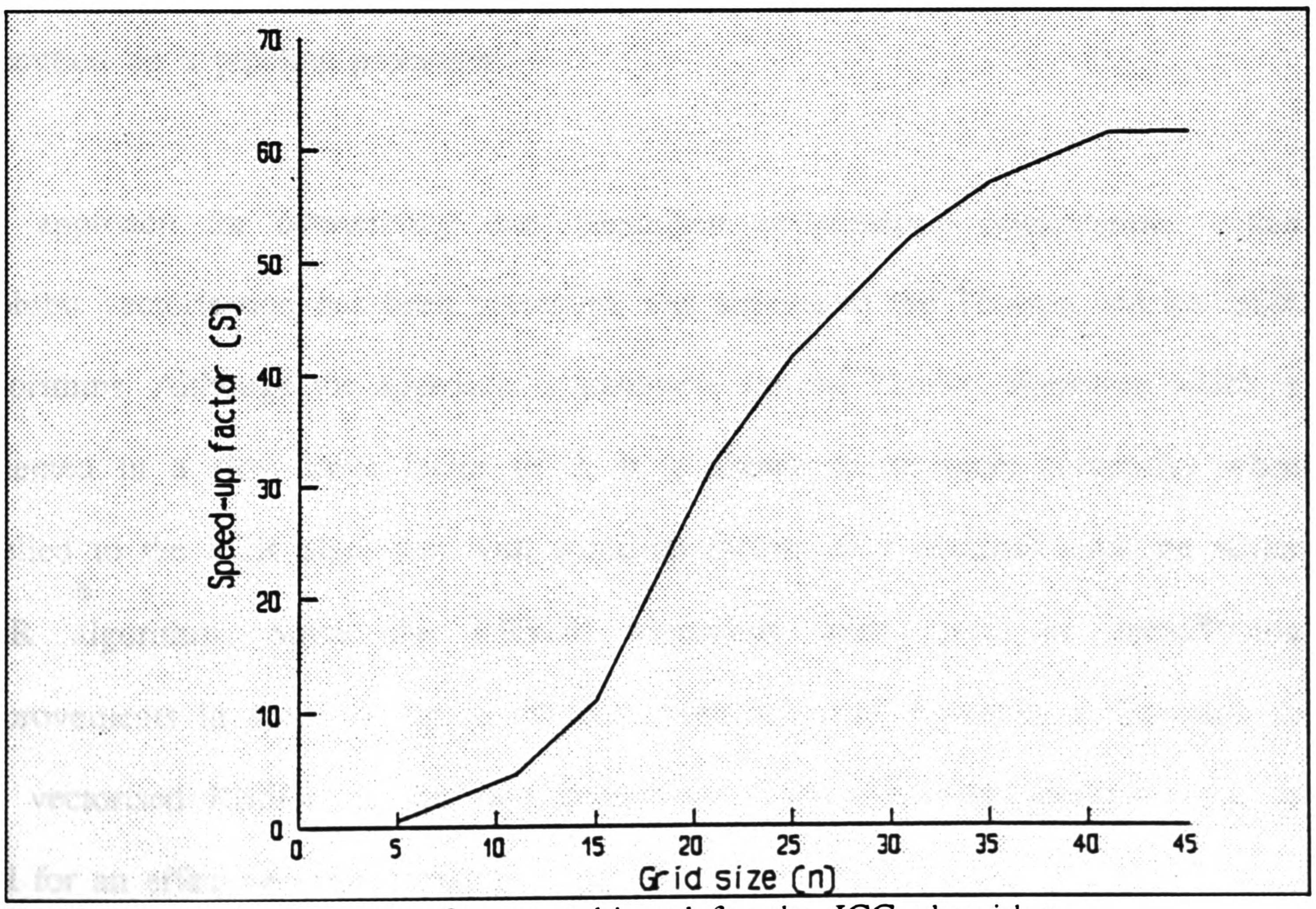


FIGURE 4.8-8 Speed-up factors achieved for the JCG algorithm

For the solution of the Laplace equation, the JCG algorithm was the best of the scalar pentadiagonal algorithms considered, it was over 30 times faster than the slowest of the algorithms (JUR). Of the vectorised algorithms the JCG was the fastest and was up to 6 times faster than the slowest vector algorithm the (SOR).

It is interesting to note that although the SOR and RBSOR algorithms have identical scalar performances, when the algorithms are vectorised the RBSOR is a factor of 8 faster than the SOR. This illustrates the importance of choosing an algorithm which lends itself to vectorisation.

4.9 Closure

A selection of the vast number of linear equation solvers have been considered for the solution of the Laplace equation on a unit square with mixed boundary conditions. The discussion focused on the implementation of these algorithms for execution on a pipeline processor.

An approach for overcoming data dependent or recursive computations which prohibit vectorisation has been presented, and applied to the Thomas and the SOR algorithms. Although the approach is unsuitable for the Thomas algorithm which is reflected in a slow-down factor of 2, it is shown to be more successful when applied to the SOR algorithm with speed-up factors of 9 obtained over the scalar SOR algorithm. Since the RBSOR algorithm lends itself to vectorisation improvements in speed of up to 70 have been achieved, however, a drawback to the vectorised RBSOR is that one dimension of the discretised domain must be odd for an effective implementation.

The Thomas algorithm is the best tridiagonal algorithm. The vectorised tridiagonal algorithms do not show any improvements on the scalar Thomas algorithm because the vector operations are too small to make efficient use of the pipeline processor. The JCG algorithm is the fastest pentadiagonal scalar algorithm and the JUR the slowest. However, the picture changes significantly when the pentadiagonal algorithms are vectorised. Although the vector JCG still performs best, the vector JUR becomes more competitive. The very high improvements in speed can be attributed partly to the vector operations of length n^2 and partly to the almost complete vectorisation of the scalar code.

It is interesting to note that on such a 'trivial' problem the JCG algorithm performs by far the best. Indeed, many publications which show the spectacular speed of the JCG algorithm are with reference to the Poisson equation. On the basis of these results, one could argue that the JCG algorithm should always be used for the solution of a linear system of equations. However, the discussions in later chapters show the dangers of choosing any one single algorithm, moreover, the discussions include problems with the JCG algorithm itself.

CHAPTER FIVE

5.0 VECTORISATION OF THE SIMPLE SOLUTION PROCEDURE

5.1 Introduction

In Chapter 4 spectacular improvements in speed were obtained from the vectorisation of algorithms which were used in the solution of the Laplace equation. However, the extension to problems which involve the solution of at least three coupled non-linear equations instead of a single linear equation, together with the added complication of a larger proportion of essentially scalar code is of interest. These coupled systems result from the mathematical description of fluid flow problems, and ideally, we would like to achieve the same order of improvement in speed as that obtained for the linear problem.

In this chapter two test problems are investigated which typify the problems encountered in CFD, these problems are often quoted as standard test cases for validating CFD codes. The first problem is a closed system comprising a square cavity containing a fluid with a moving lid. The second problem consists of a square duct with a restricted inlet, this problem is more commonly referred to as the 'backward facing step' or 'sudden expansion' problem. Both test cases involve the solution of a two-dimensional velocity field and a pressure field. The SIMPLE procedure is used to solve the resulting coupled system of equations.

5.2 Scalar algorithms

Within a SIMPLE iteration the solution of the momentum equations are not solved to a high degree of convergence and hence an iterative algorithm is appropriate. For the solution of the momentum equations a LJUR algorithm is used instead of a LSOR algorithm. Although the LJUR algorithm generally has a slower rate of convergence (Varga [1962]) it is more suitable and the reasons for this are twofold. Firstly, the changes made to the velocity solution fields by the LJUR algorithm are small and this means that less relaxation is needed, whereas a higher level of relaxation would be needed for the LSOR algorithm. Secondly, the changes made to the solution by the LJUR algorithm result in a more stable process for this particular implementation of the SIMPLE procedure. Here a single LJUR sweep is carried out for the solution of the u-momentum equation followed by a single LJUR sweep for the v-momentum equation.

The solution of the pressure-correction equation is of particular interest because the resulting system of equations are analogous to that of the Poisson equation. In deciding which algorithms to use the results from Chapter 4 serve as an indicator. The algorithms considered in this study are the LSOR algorithm, the JCG algorithm and the JUR algorithm. The latter is chosen for its impressive vectorising performance and not for its scalar performance.

The level of convergence chosen for the solution of the pressure-correction equation is greater than that used for the momentum equations, here a converged solution is said to have been obtained when the following criteria is satisfied

$$\frac{|| p'^{(new)} - p'^{(old)} ||_2}{\mathrm{Max}\{10^{-10}, || p'^{(new)} ||_2 \}} \leq \zeta \qquad (5.2\text{-}1)$$

Through trial and error the tolerance ζ is set $1.0\text{x}10^{-3}$, $2.5\text{x}10^{-4}$, $1.0\text{x}10^{-4}$ for the LSOR, JCG and JUR algorithms respectively. This ensures that the quality of the solution obtained from the three different algorithms is consistent.

5.3 PROBLEM 1: Square cavity with moving lid problem

In this problem rotation of the fluid is caused by the moving lid on top of the square cavity (figure 5.3-1). This results in no predominant flow direction and the differential equations describing this situation are highly elliptic.

The steady state solution of such a problem has become a popular example for testing and validating numerical algorithms. Its geometric simplicity and highly elliptic character have attracted many workers to provide numerical solutions for this and many of its variations. Some of the more notable works are those of Burggraf [1966], Bozman and Dalton [1973], de Vahl Davis and Malinson [1976] and Ghia et al [1982] all of which used a stream function - vorticity formulation to solve the resulting equations.

5.3.1 Physical and geometrical specification

The boundary conditions for the cavity are shown in figure 5.3-1. The moving wall has velocity components $u=1ms^{-1}$ and $v=0ms^{-1}$. All other walls have a no-slip velocity condition and thus $u=v=0ms^{-1}$. The initial velocity and pressure fields were set up so that $u=v=0ms^{-1}$ and $p=0Nm^{-2}$. The flow is taken to be laminar and steady state simulations are performed for Reynolds numbers Re=100 and Re=400. The domain is discretised using a uniform 32x32 grid.

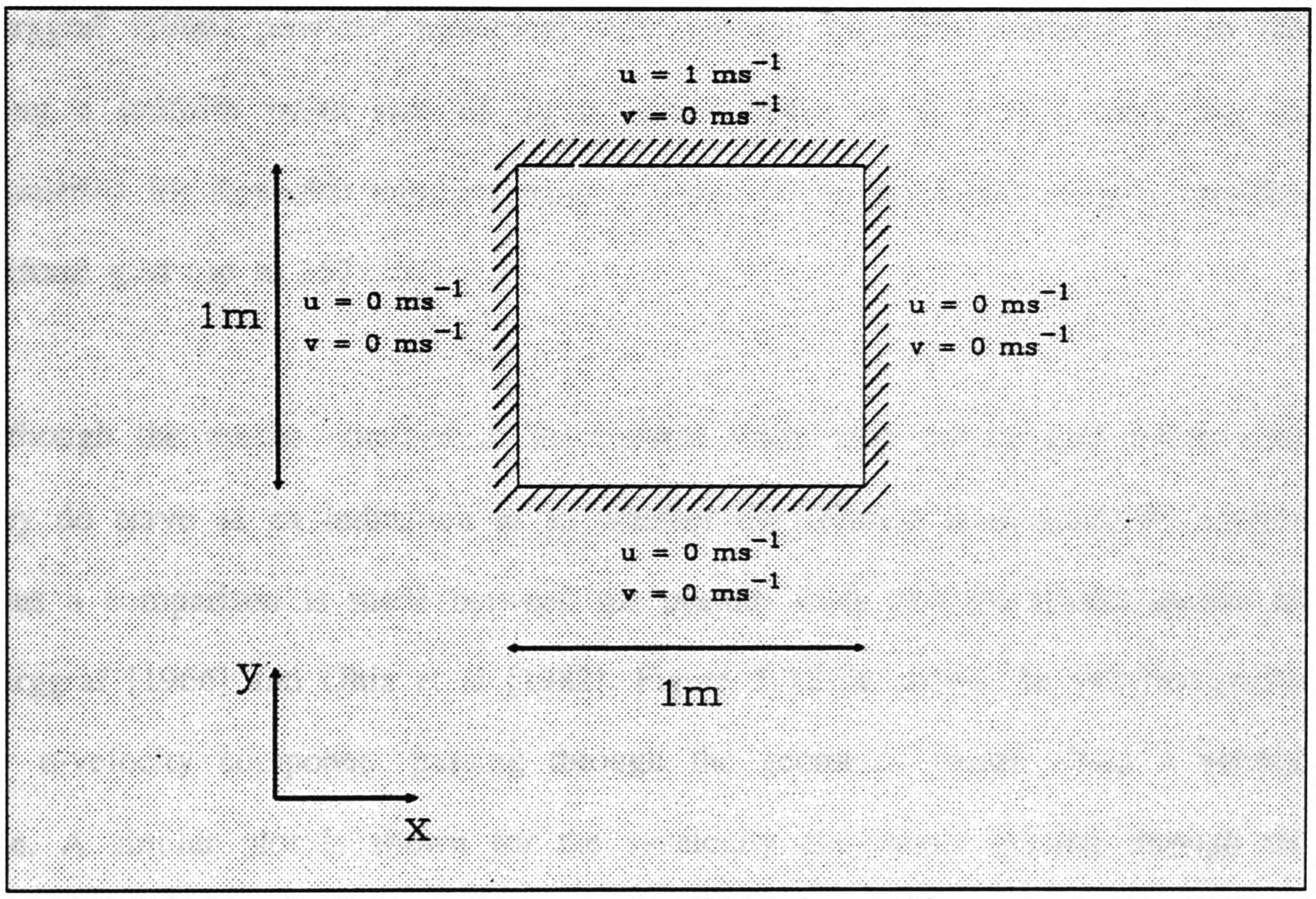


FIGURE 5.3-1 Definition of the moving lid cavity problem

5.3.2 Results using scalar algorithms

The resulting velocity vector plot for Re=100 is shown in figure 5.3.2-1a. Contour plots of the velocity components and the pressure field are also shown for completeness (figures 5.3.2-1b to 5.3.2-1d). A similar set of results are also presented for the case where Re=400 (figure 5.3.2-2). In both cases the results were obtained after 100 SIMPLE iterations.

Burggraf [1966] provided numerical solutions for Reynolds numbers up to 400 using a uniform mesh size of up to 40x40. Ghia et al [1982] carry out the simulation for Reynolds numbers ranging from 100 up to 10,000 using a very fine uniform grid (up to 257x257).

Although the results obtained in the present study may not be grid independent they do serve as an indication as to whether the solution is qualitatively correct. Thus a comparison is made between the present study and the results quoted by Burggraf [1966] and Ghia et al [1982]. Figure 5.3.2-3a shows the velocity profile for u-velocity component passing through the geometric centre along a vertical line. A similar plot is shown for the v-velocity component passing through the geometric centre along a horizontal line (figure 5.3.2-3b). For Re=100 there is excellent agreement with the results of Burggraf [1966] and Ghia et al [1982]. This indicates that for such a Reynolds number the 32x32 uniform grid employed here is satisfactory. For a Reynolds number of 400 the results begin to differ from the grid independent results of Ghia et al [1982] but are still comparable to the results of Burggraf [1966] (figure 5.3.2-4). This is not unexpected, and the results are indicative of the magnitude of errors which can be expected when using

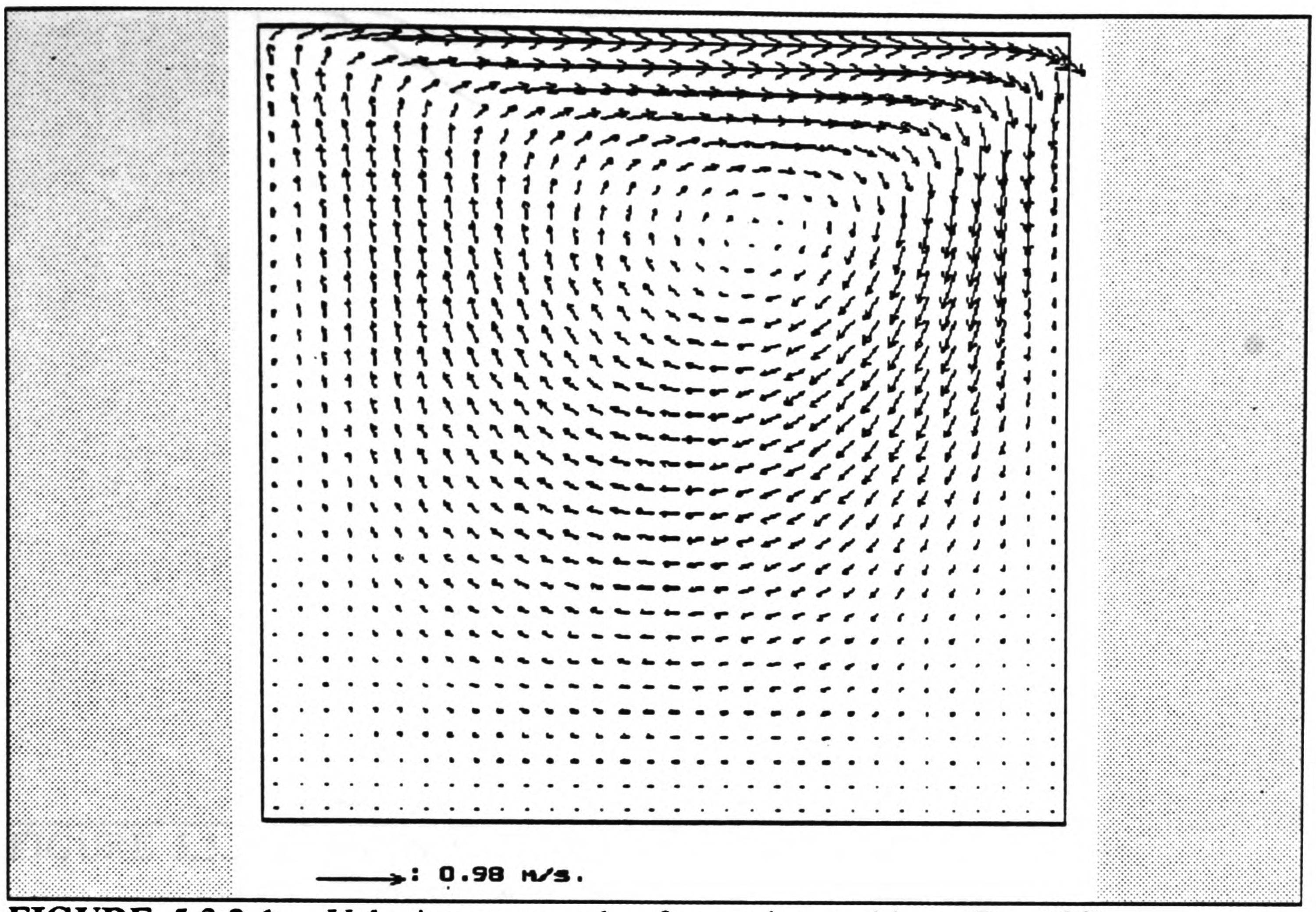


FIGURE 5.3.2-1a Velocity vector plot for cavity problem (Re=100)

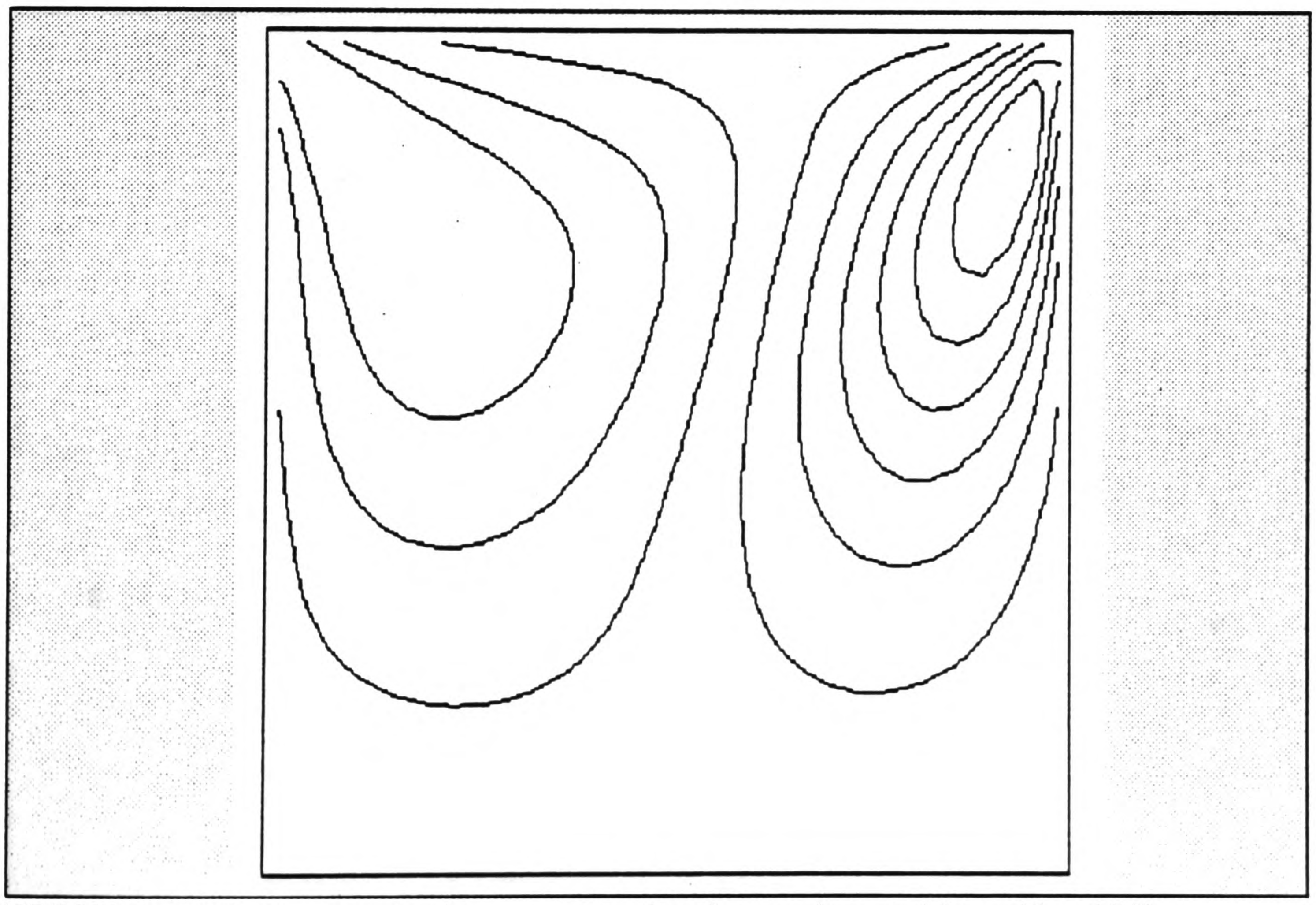


FIGURE 5.3.2-1b u-velocity contour plot for cavity problem (Re=100). Contours at -0.247 (0.0719) 0.472

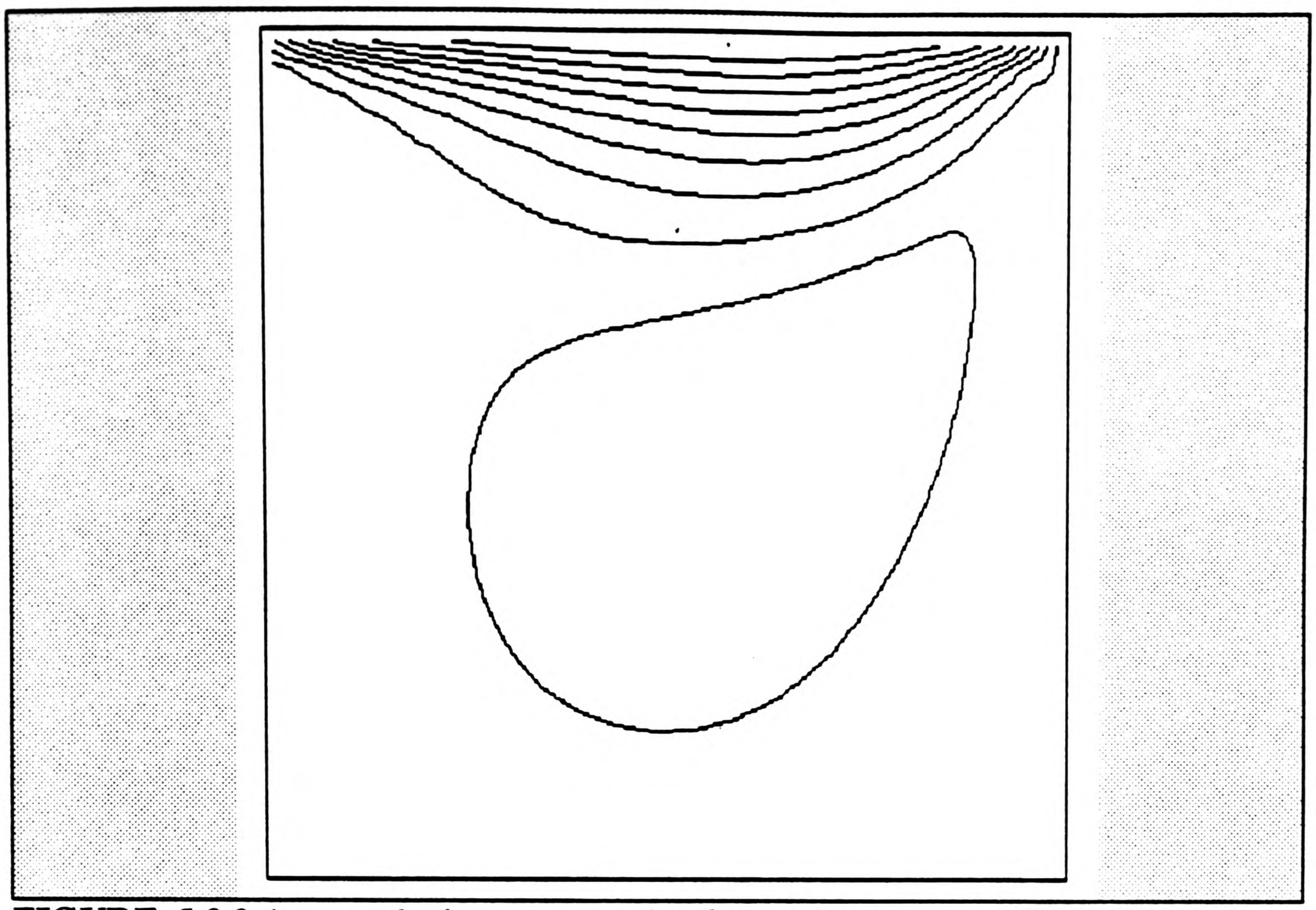


FIGURE 5.3.2-1c v-velocity contour plot for cavity problem (Re=100).
Contours at -0.209 (0.1104) 0.895

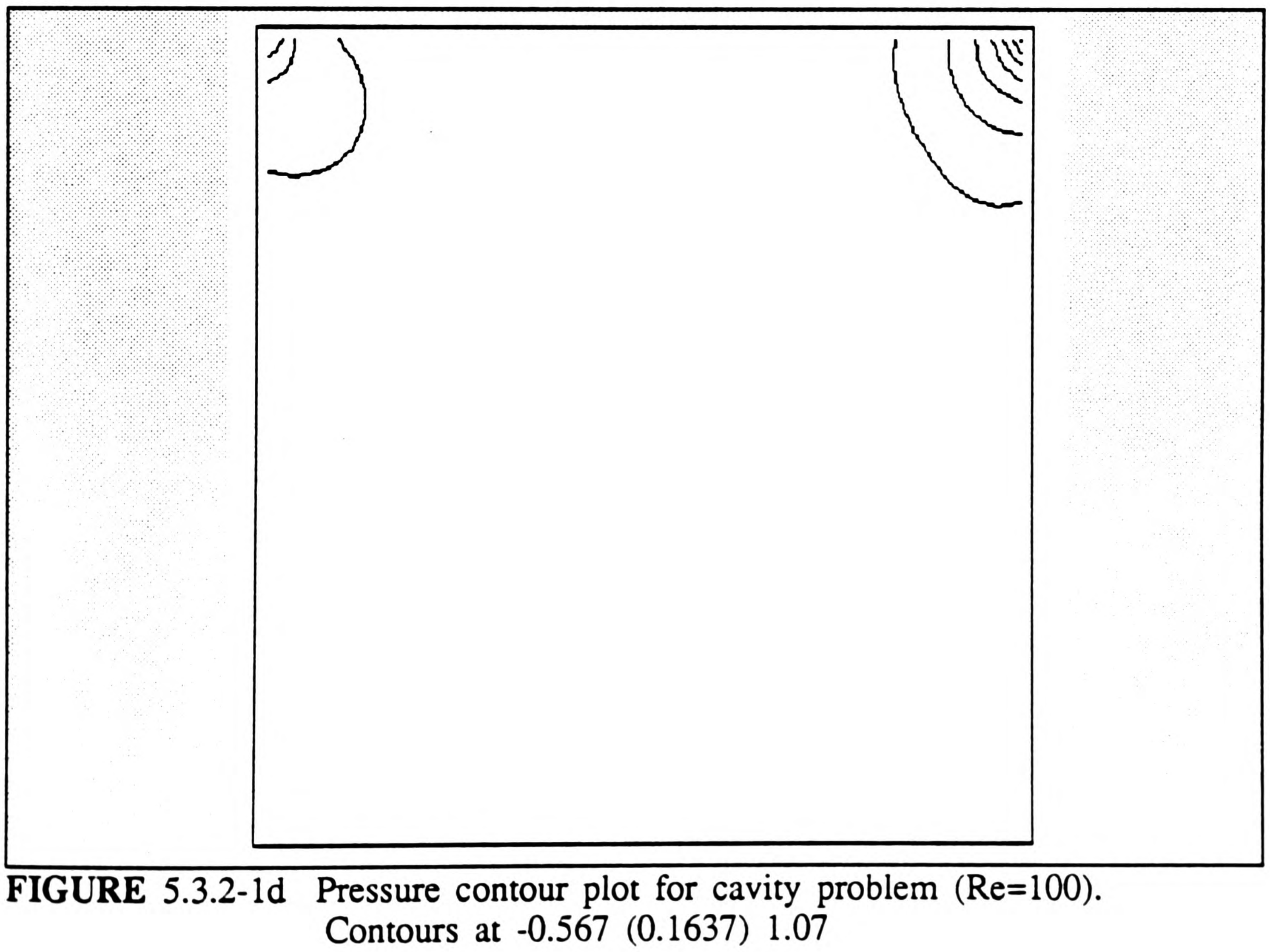


FIGURE 5.3.2-1d Pressure contour plot for cavity problem (Re=100).
Contours at -0.567 (0.1637) 1.07

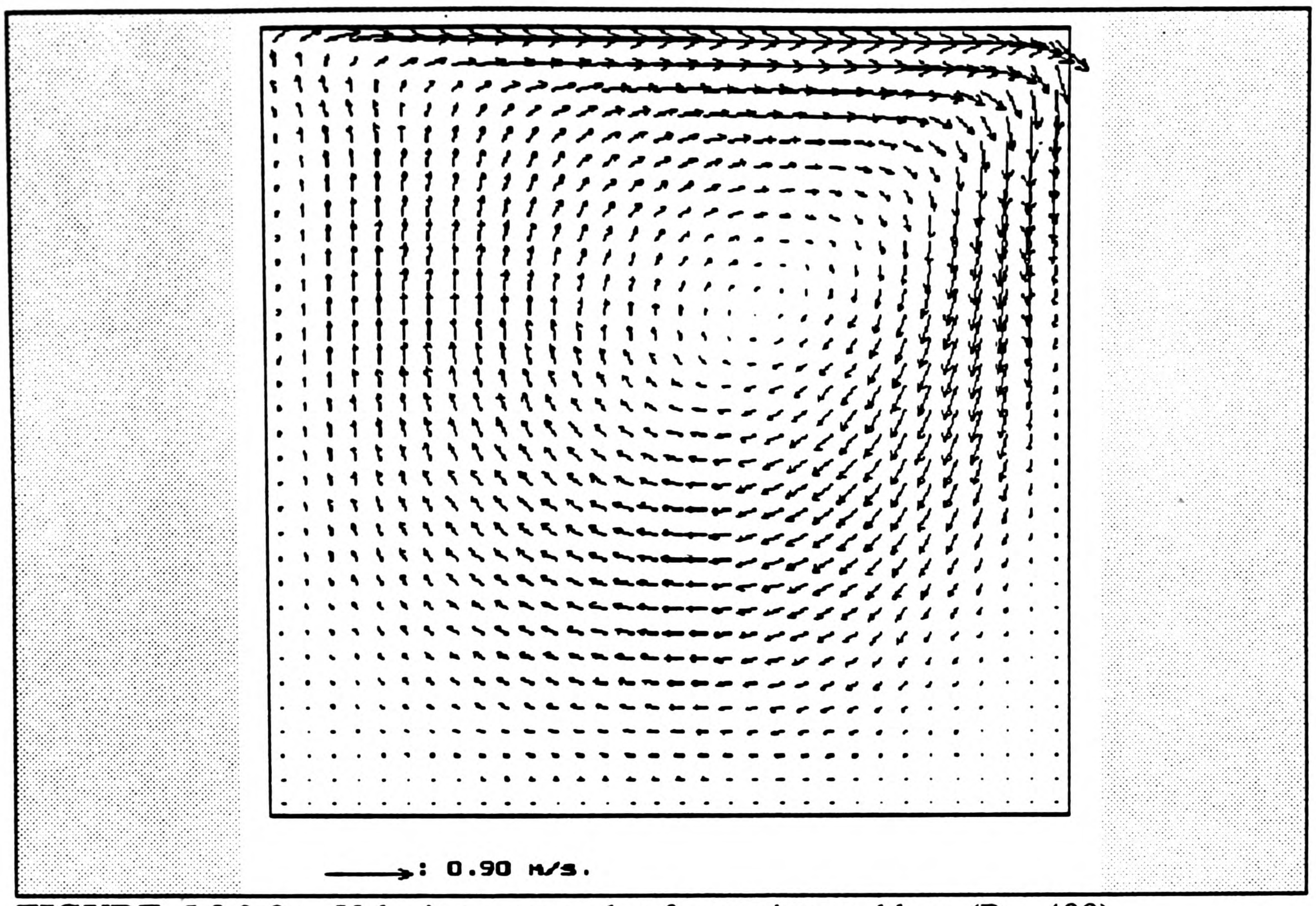


FIGURE 5.3.2-2a Velocity vector plot for cavity problem (Re=400)

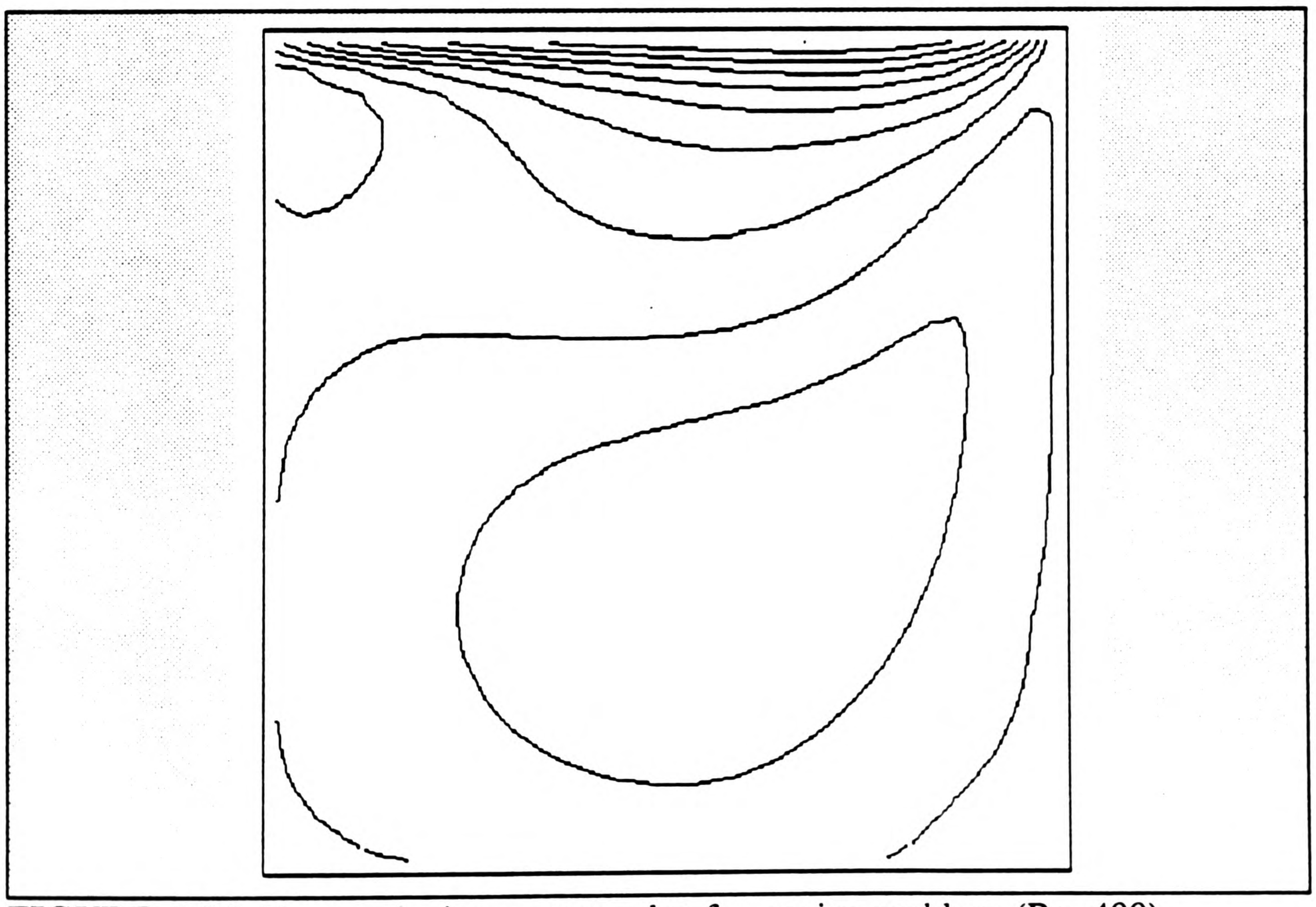


FIGURE 5.3.2-2b u-velocity contour plot for cavity problem (Re=400). Contours at -0.21 (0.1036) 0.826

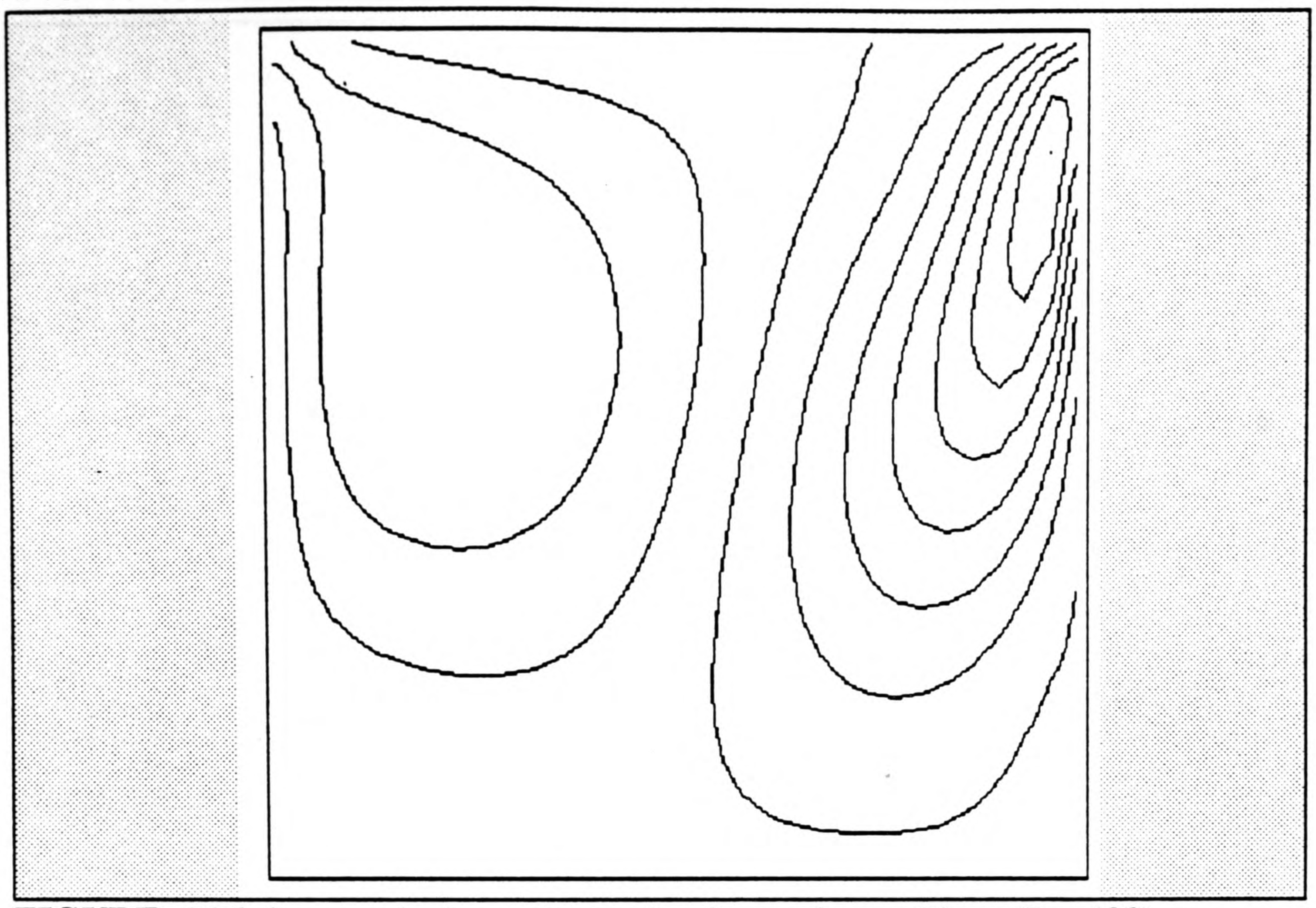


FIGURE 5.3.2-2c v-velocity contour plot for cavity problem (Re=400). Contours at -0.213 (0.0728) 0.515

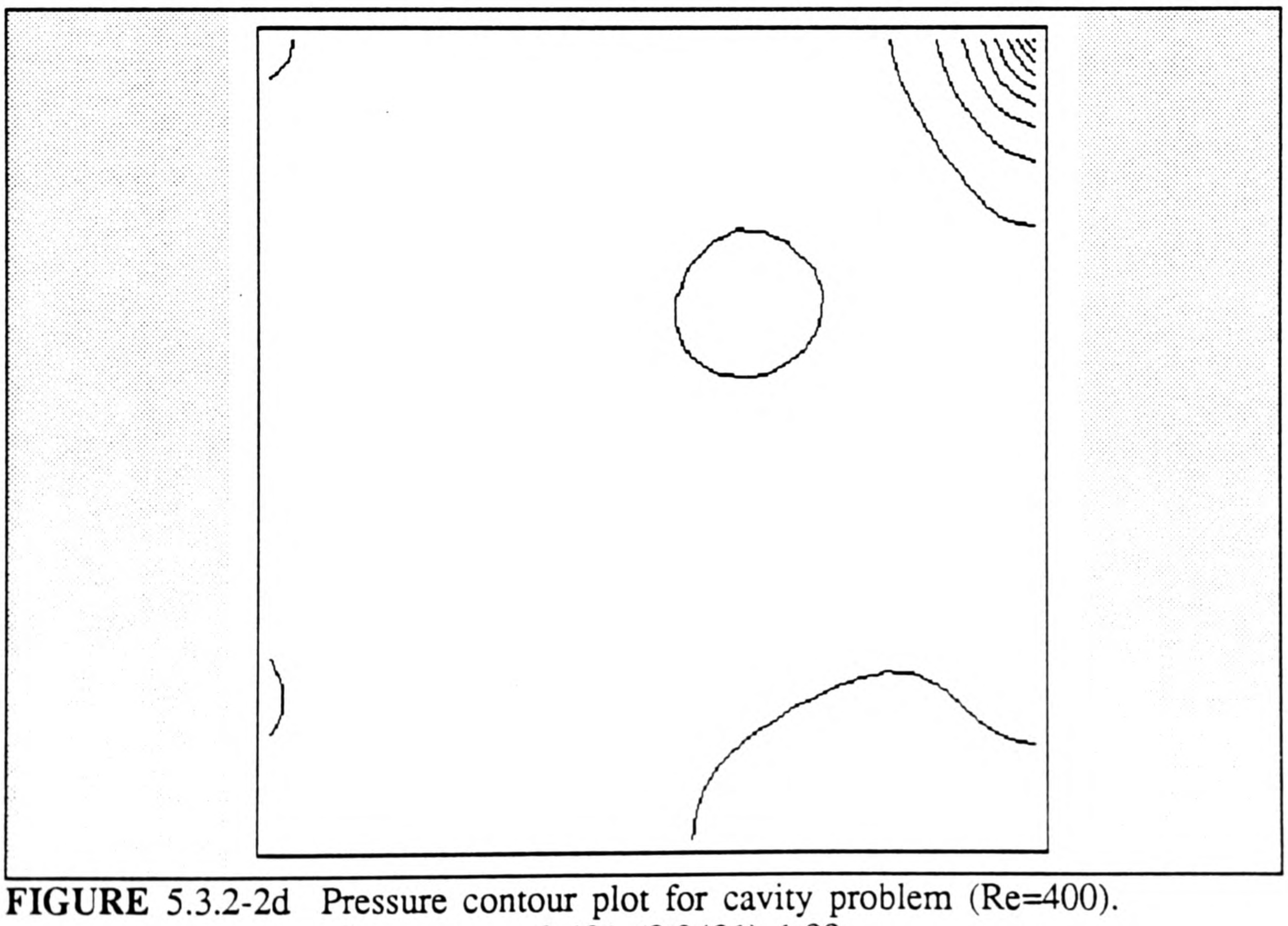


FIGURE 5.3.2-2d Pressure contour plot for cavity problem (Re=400). Contours at -0.431 (0.2421) 1.99

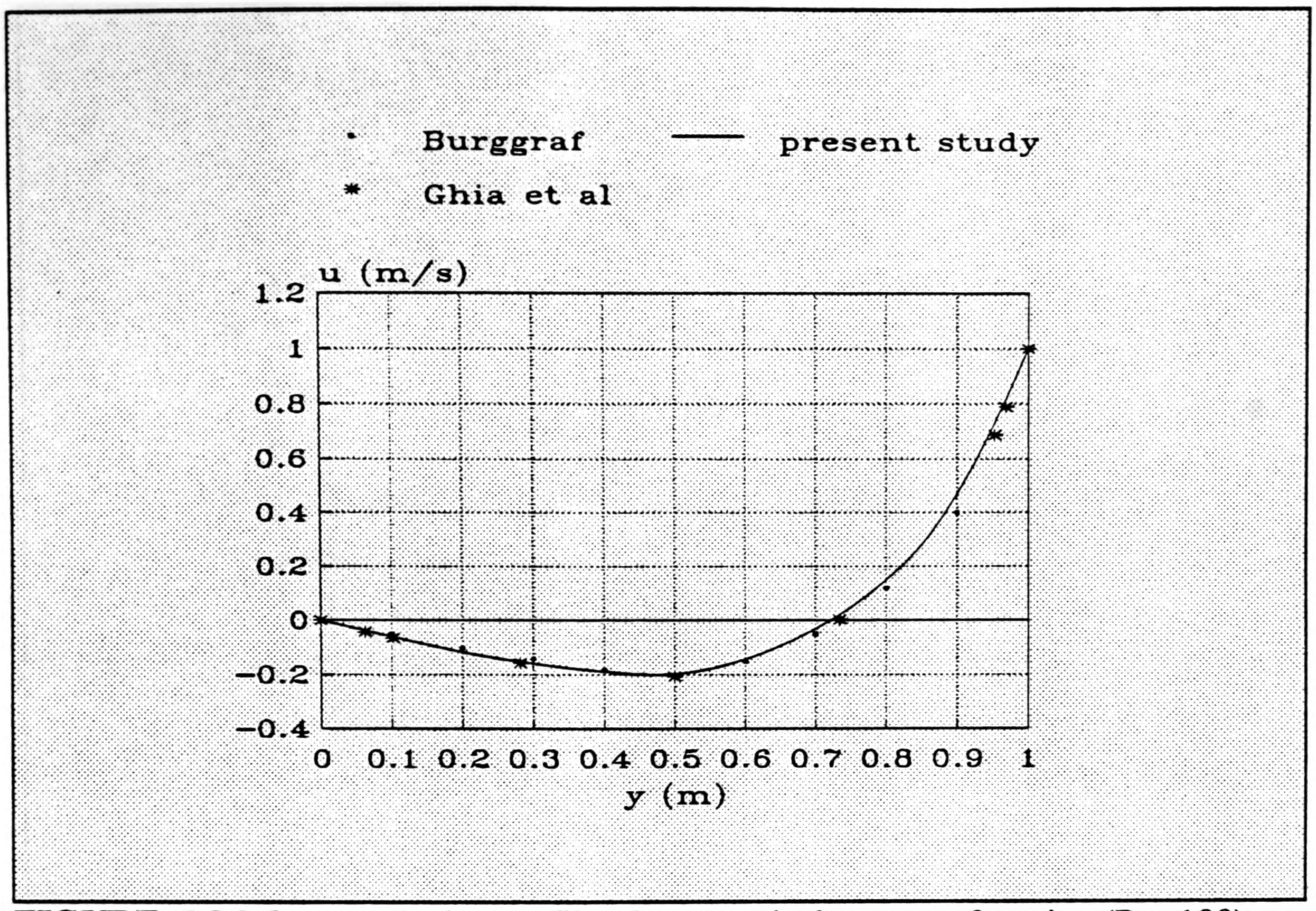


FIGURE 5.3.2-3a u-velocity profile along vertical centre of cavity (Re=100)

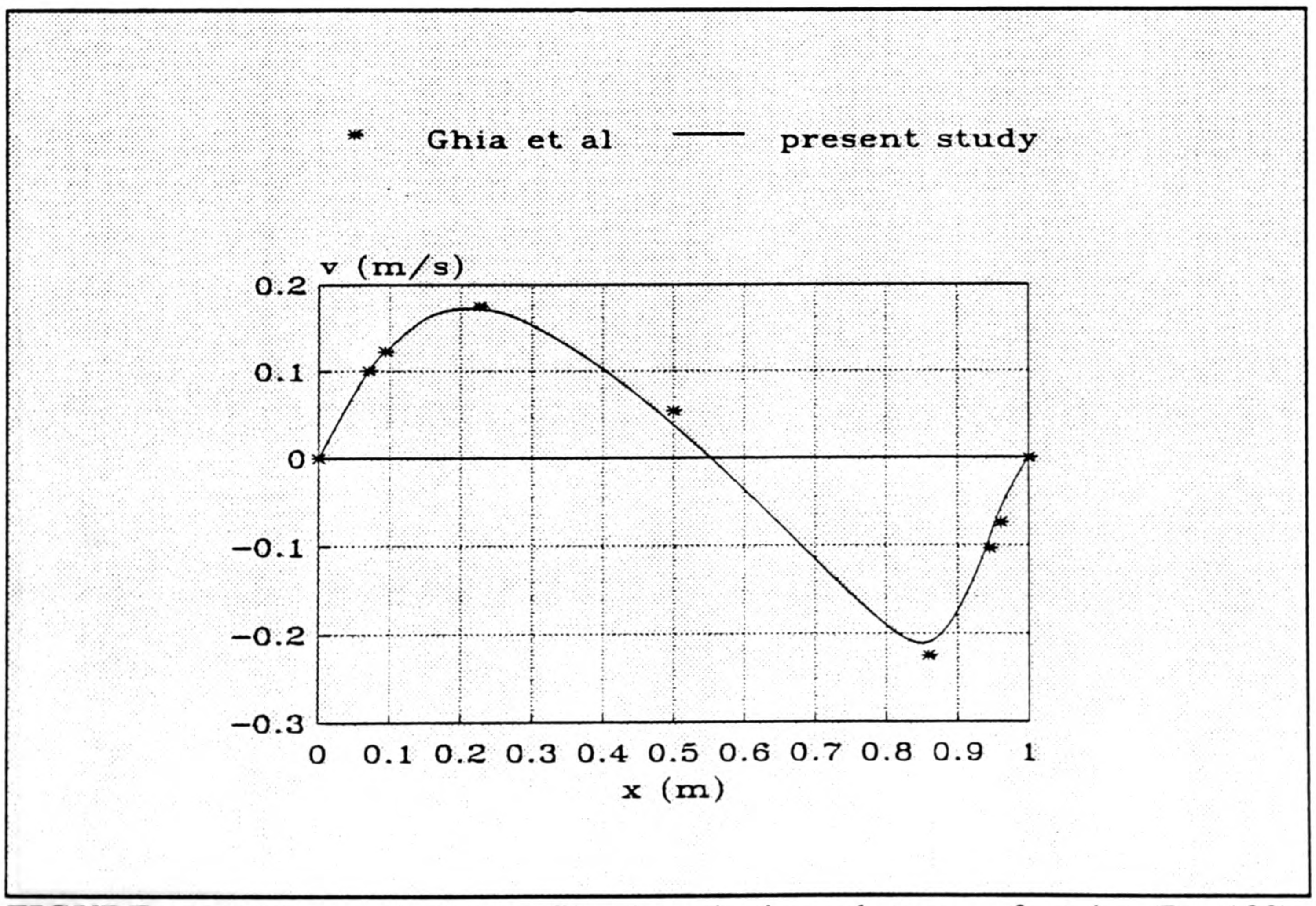


FIGURE 5.3.2-3b v-velocity profile along horizontal centre of cavity (Re=100)

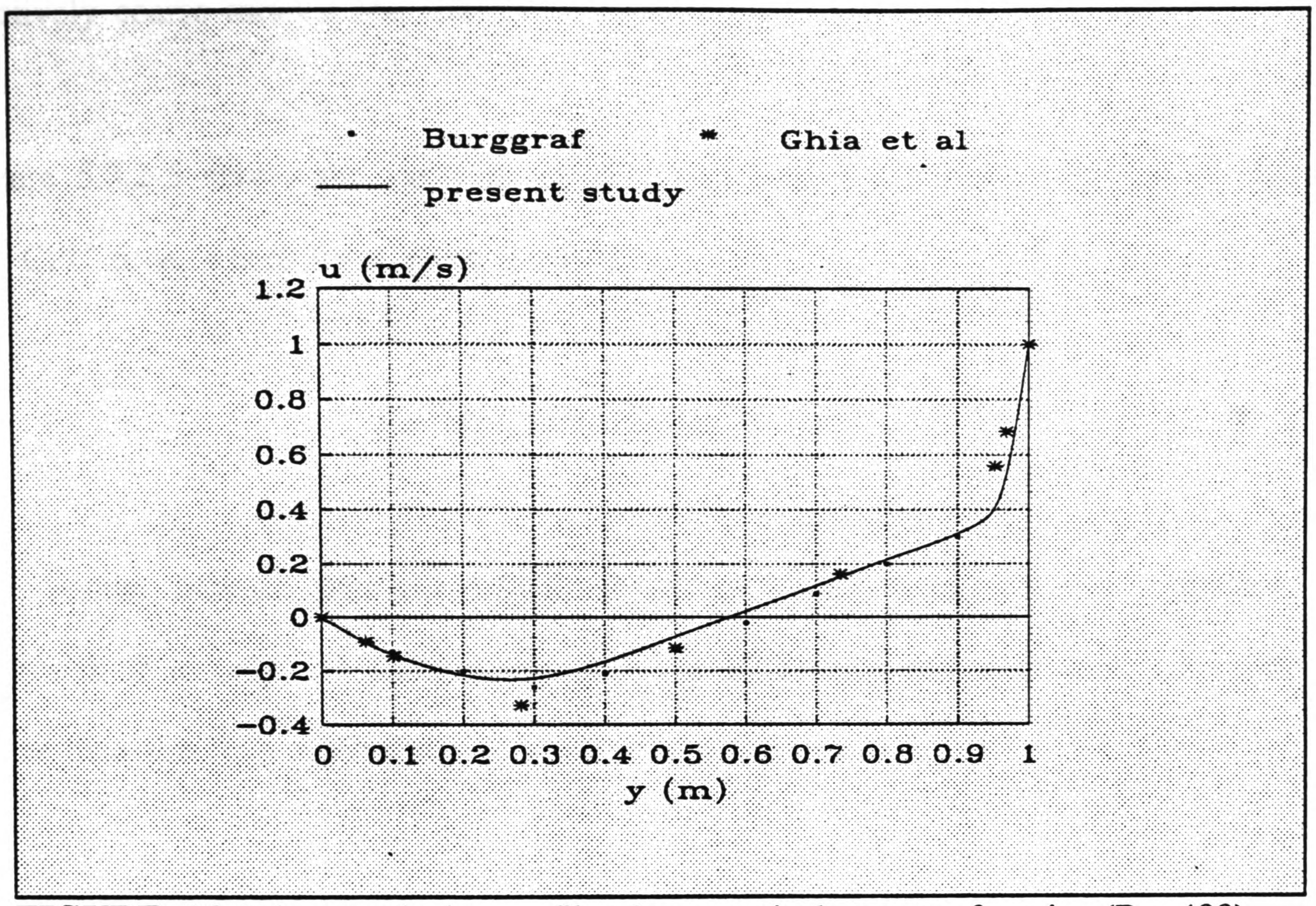


FIGURE 5.3.2-4a u-velocity profile along vertical centre of cavity (Re=400)

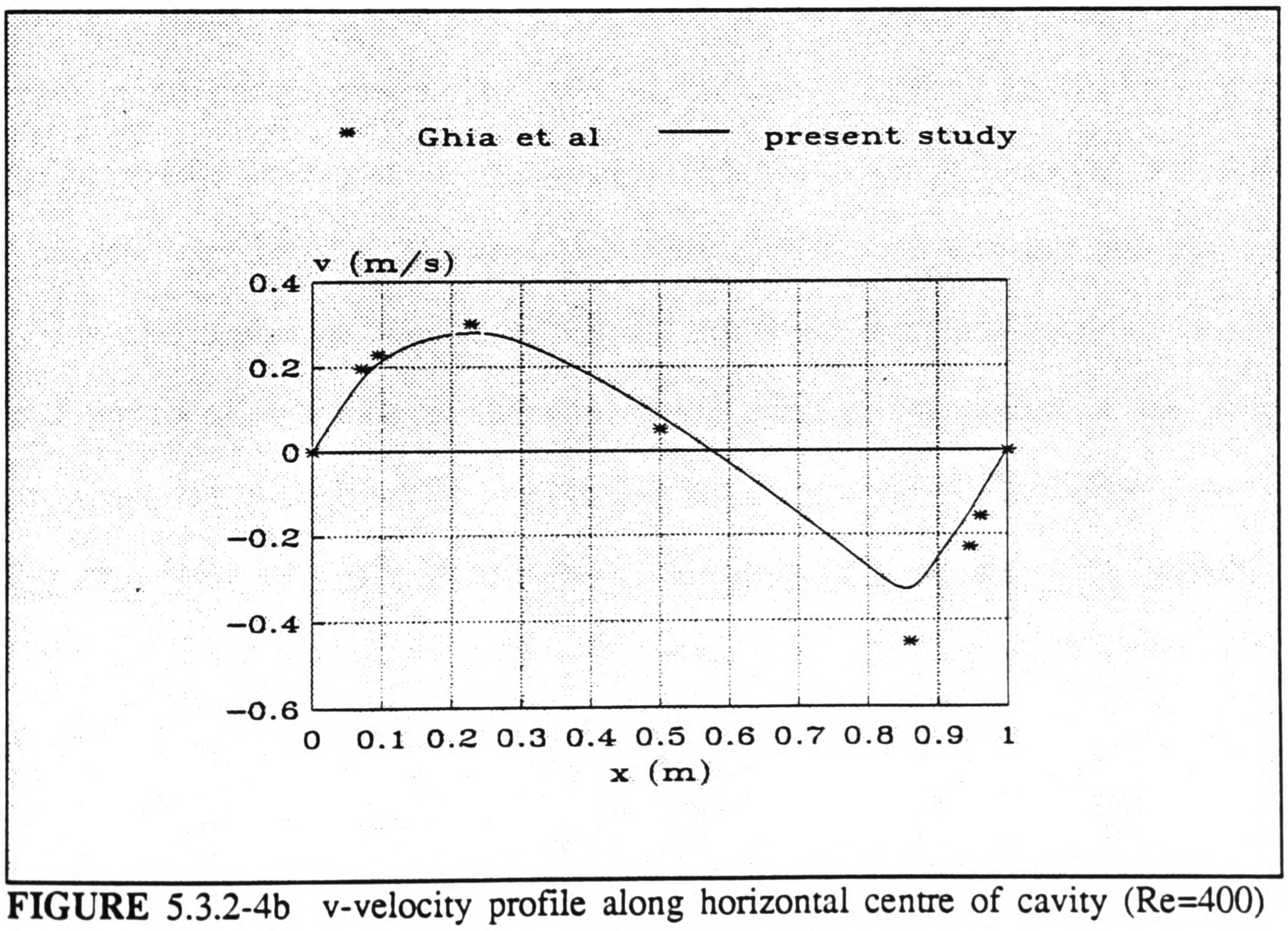


FIGURE 5.3.2-4b v-velocity profile along horizontal centre of cavity (Re=400) relatively coarse meshes and low-order differencing schemes.

Figure 5.3.2-5 shows the variation of the logarithm of maximum residual r, with the CPU time, where for a given iteration

$$r = \text{Maximum residual}\{u, v, p'\} \qquad (5.3.2\text{-}1)$$

The results indicate that the LSOR algorithm is the most efficient algorithm. Although it is only 1.3 times faster than the JCG algorithm it is over 3.2 times faster than the JUR algorithm.

5.4 PROBLEM 2: Sudden expansion problem

In this problem the fluid enters a restricted opening between two parallel plates (figure 5.4-1). The resulting flow includes a recirculation region which forms behind the closed section of the plates. This is not surprising since a sudden increase in the cross-sectional flow area can result in a reversal of the fluid flow in the immediate vicinity of the step change. Like the cavity problem this test case has become very popular. Macagno and Hung [1967] were amongst the first to report detailed numerical simulations for this problem. Since then many variations have been considered such as the size of the opening, the profile of the inlet velocity and the inlet Reynolds number (Back and Roschke [1972], Iribarne et al [1972] and Pollard [1980]). Experimental measurements have also been recorded by Denham and Patrick [1974] for the variation involving a single plane duct expansion.

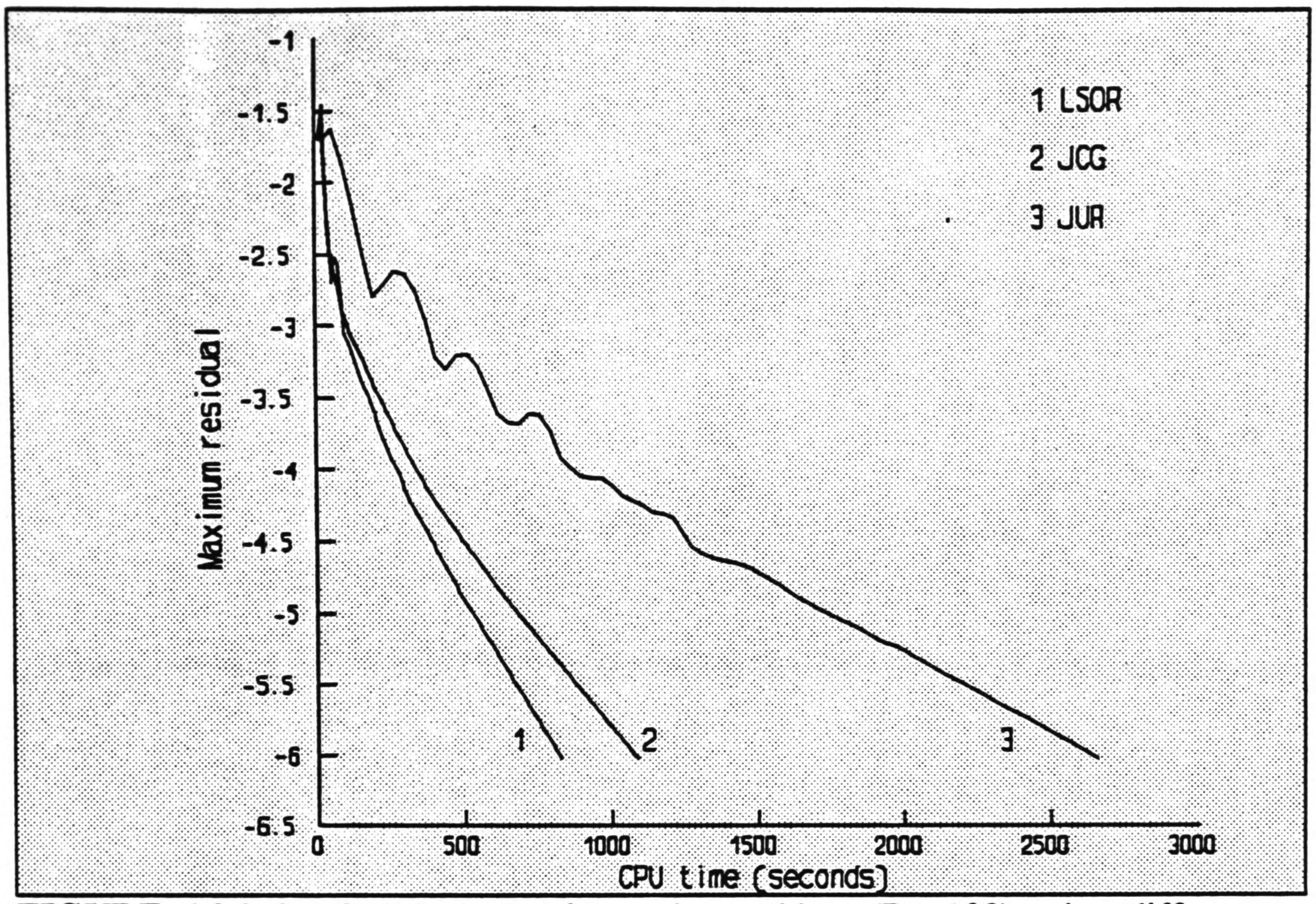


FIGURE 5.3.2-5a Residual plot for cavity problem (Re=100) using different scalar algorithms to solve the pressure-correction equation

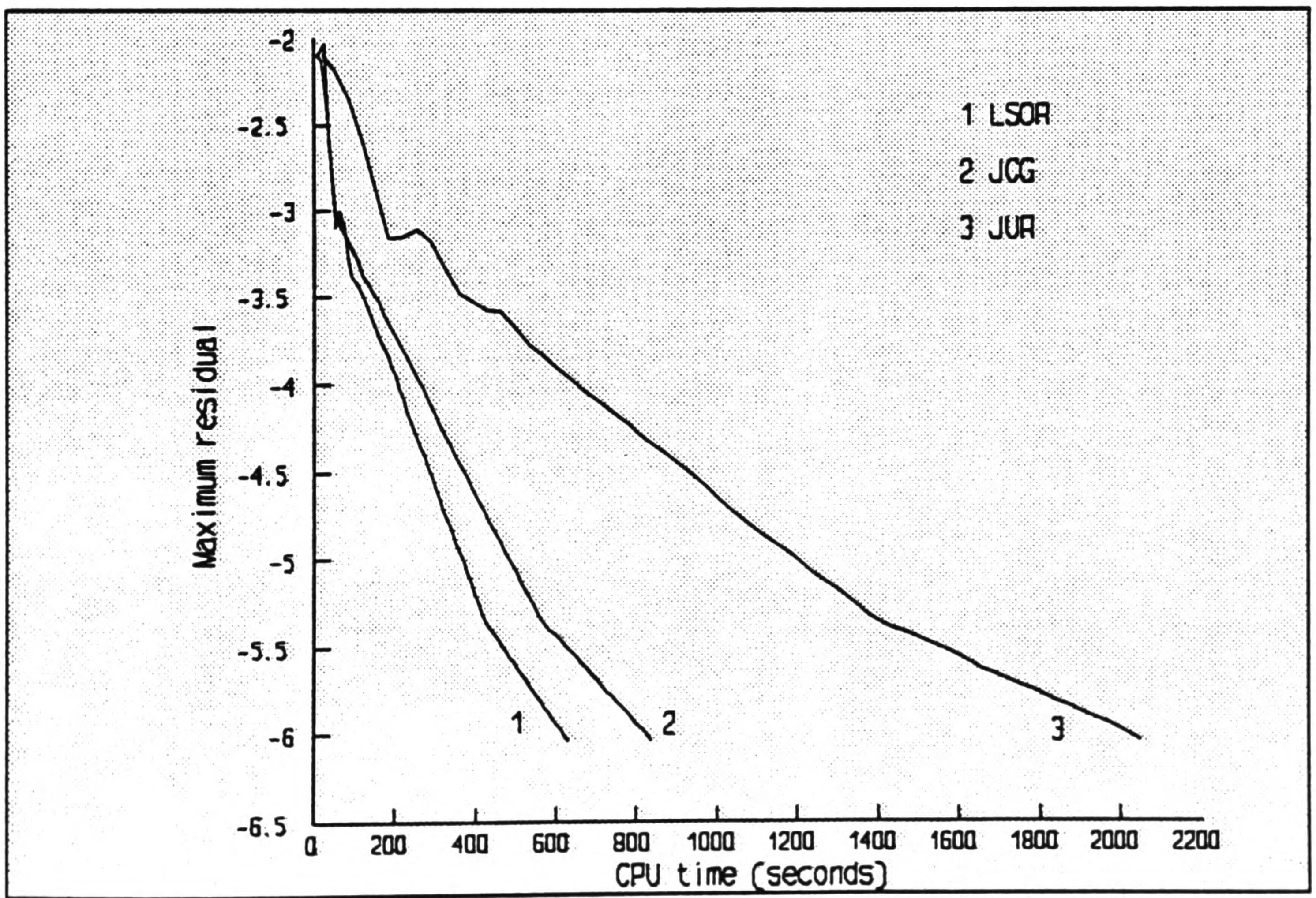


FIGURE 5.3.2-5b Residual plot for cavity problem (Re=400) using different scalar algorithms to solve the pressure-correction equation

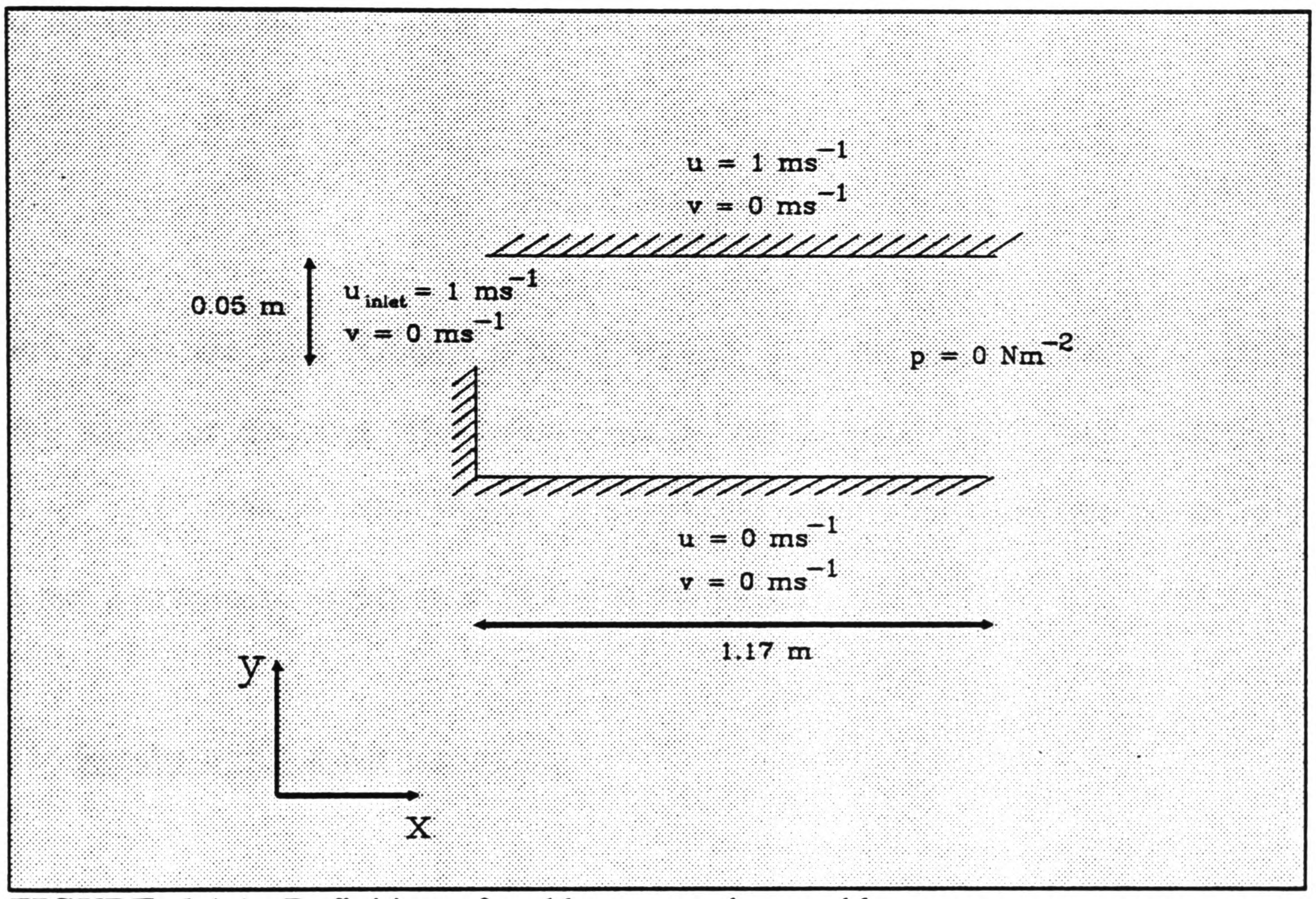


FIGURE 5.4-1 Definition of sudden expansion problem

5.4.1 Physical and geometrical specification

The boundary conditions for the sudden expansion problem are shown in figure 5.4-1. The inlet u-velocity has a parabolic profile with a mean inlet velocity $u_{inlet}=1.0ms^{-1}$ and a v-velocity $v=0ms^{-1}$. All walls are assumed to have a no-slip velocity condition and the outlet pressure is fixed at zero. The ratio of the channel length to the inlet is 23.4 and the expansion ratio of the inlet to the total width is 2. The flow is taken to be laminar and a steady state solution is obtained for an inlet Reynolds number $Re_{inlet}=50$, which is defined as

$$Re_{inlet} = \frac{u_{inlet}\ \rho\ L}{\mu} \qquad (5.4.1\text{-}1)$$

where u_{inlet} is the inlet velocity, ρ is the density, μ is the absolute viscosity and L is the width of the inlet. The initial velocity and pressure fields were $u=0.5ms^{-1}$, $v=0ms^{-1}$ and $p=0Nm^{-2}$. The domain is discretised using a non-uniform 64x16 grid and has a maximum aspect ratio of 6.4 at the outlet.

5.4.2 Results using scalar algorithms

The resulting velocity vector plot is shown in figure 5.4.2-1a, the recirculation region immediately behind the closed section of the square duct can be seen. Velocity and pressure contour plots are also shown (figures 5.4.2-1b to 5.4.2-1d), these show the two singularity points at the abrupt expansion and at the re-attachment point.

A plot of the maximum logarithm residual defined by equation (5.3.2-1) with CPU

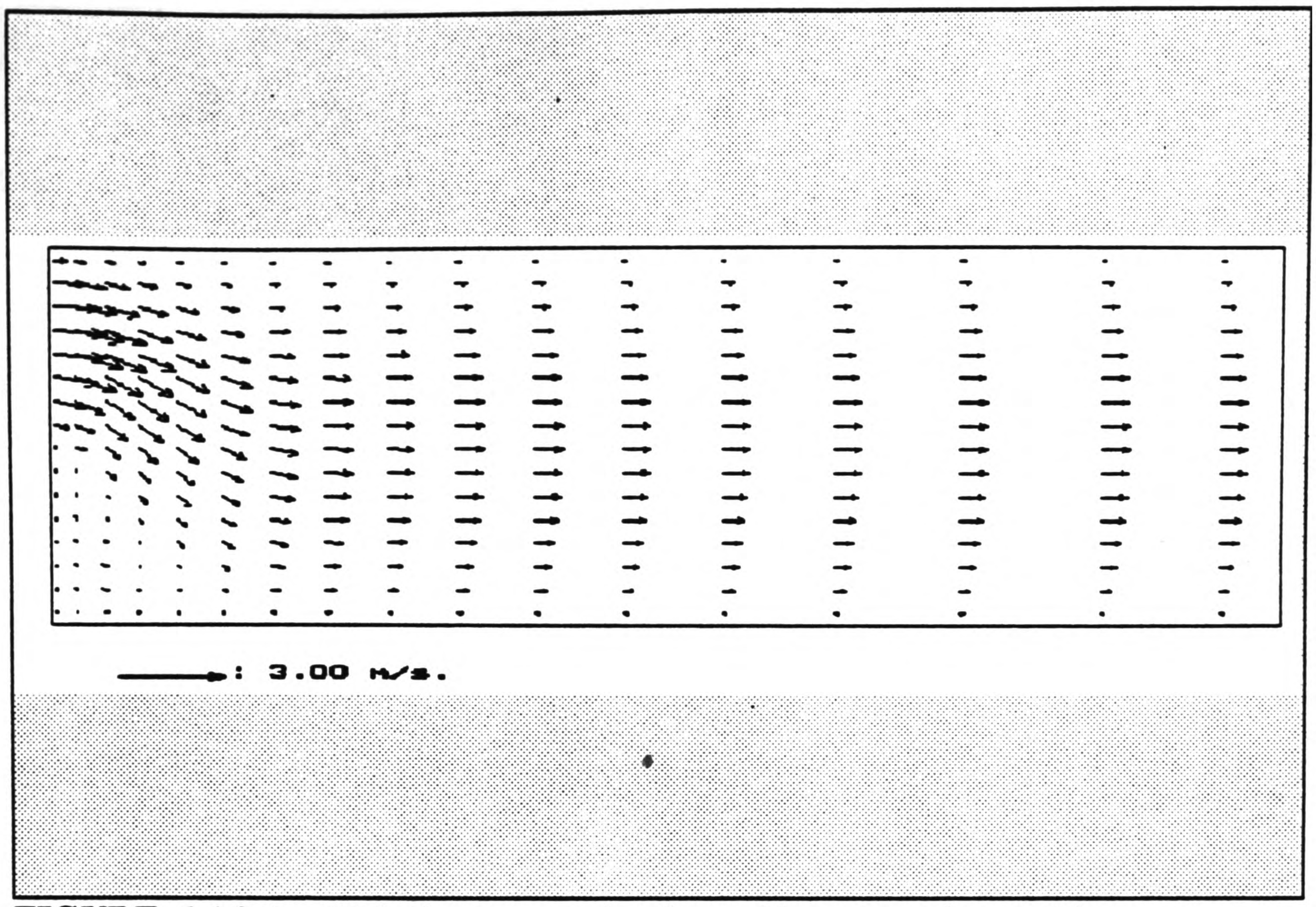


FIGURE 5.4.2-1a Velocity vector plot for sudden expansion problem

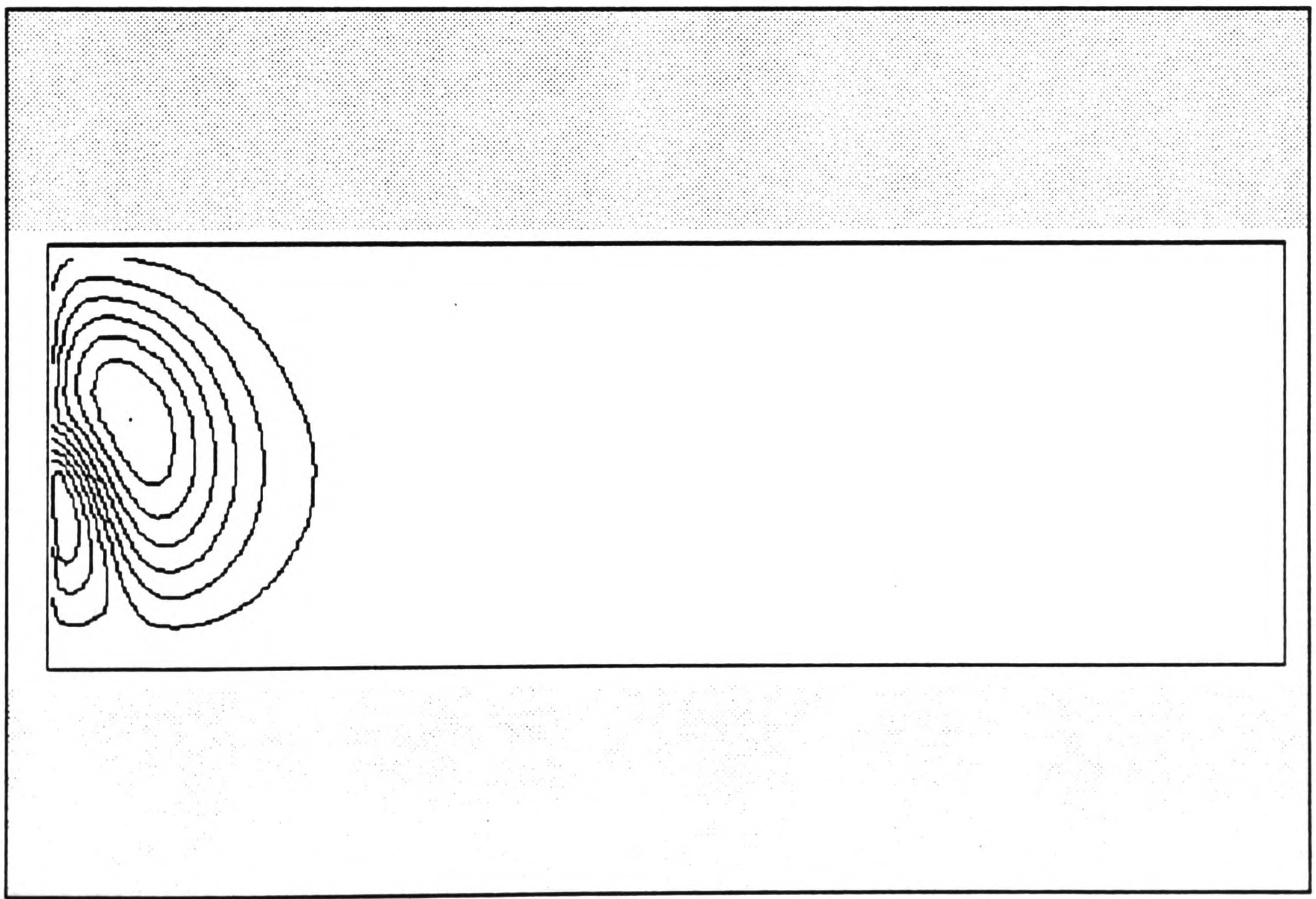


FIGURE 5.4.2-1b u-velocity contour plot for sudden expansion problem. Contours at -0.167 (0.1637) 1.47

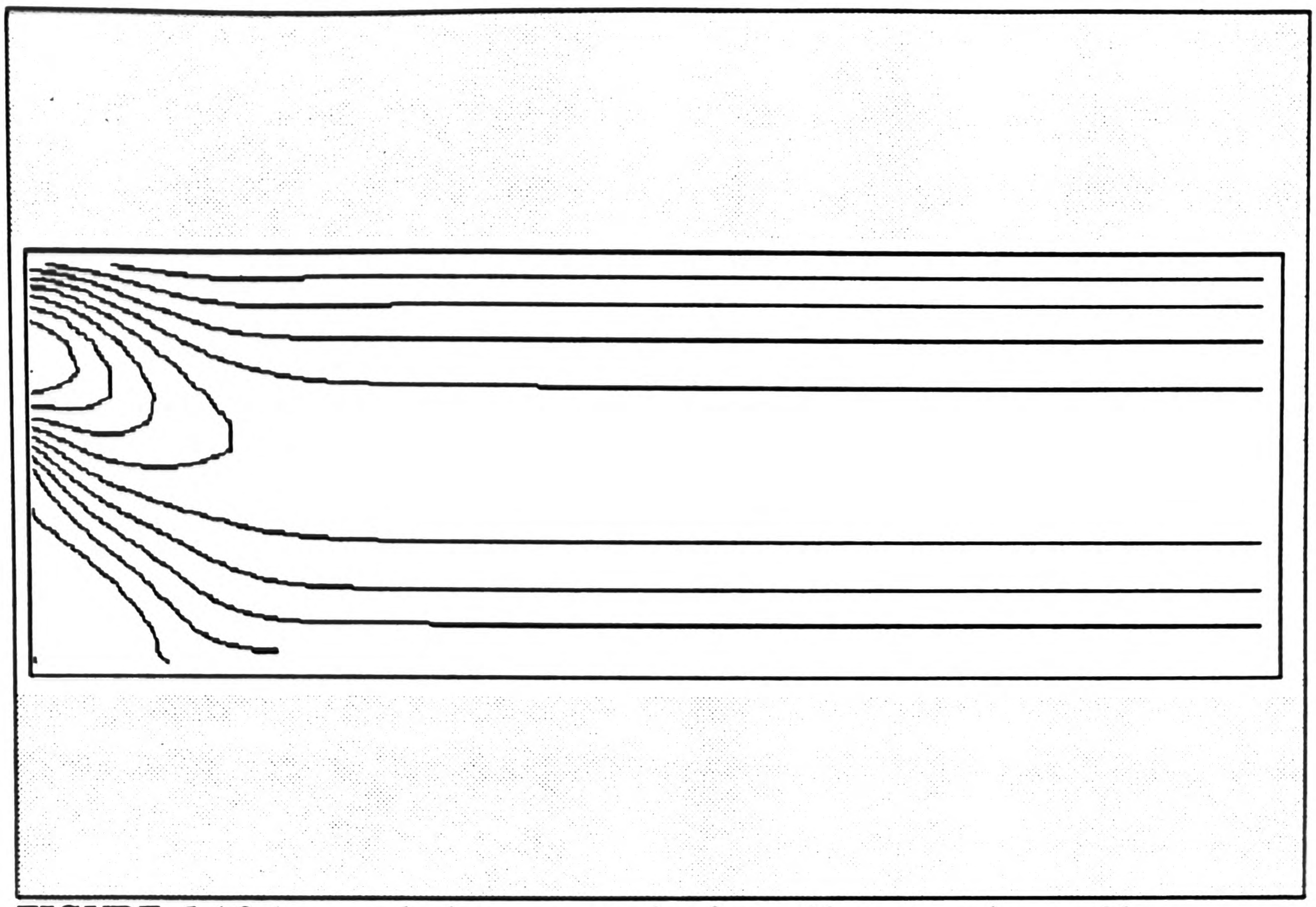


FIGURE 5.4.2-1c v-velocity contour plot for sudden expansion problem. Contours at -0.106 (0.0302) 0.196

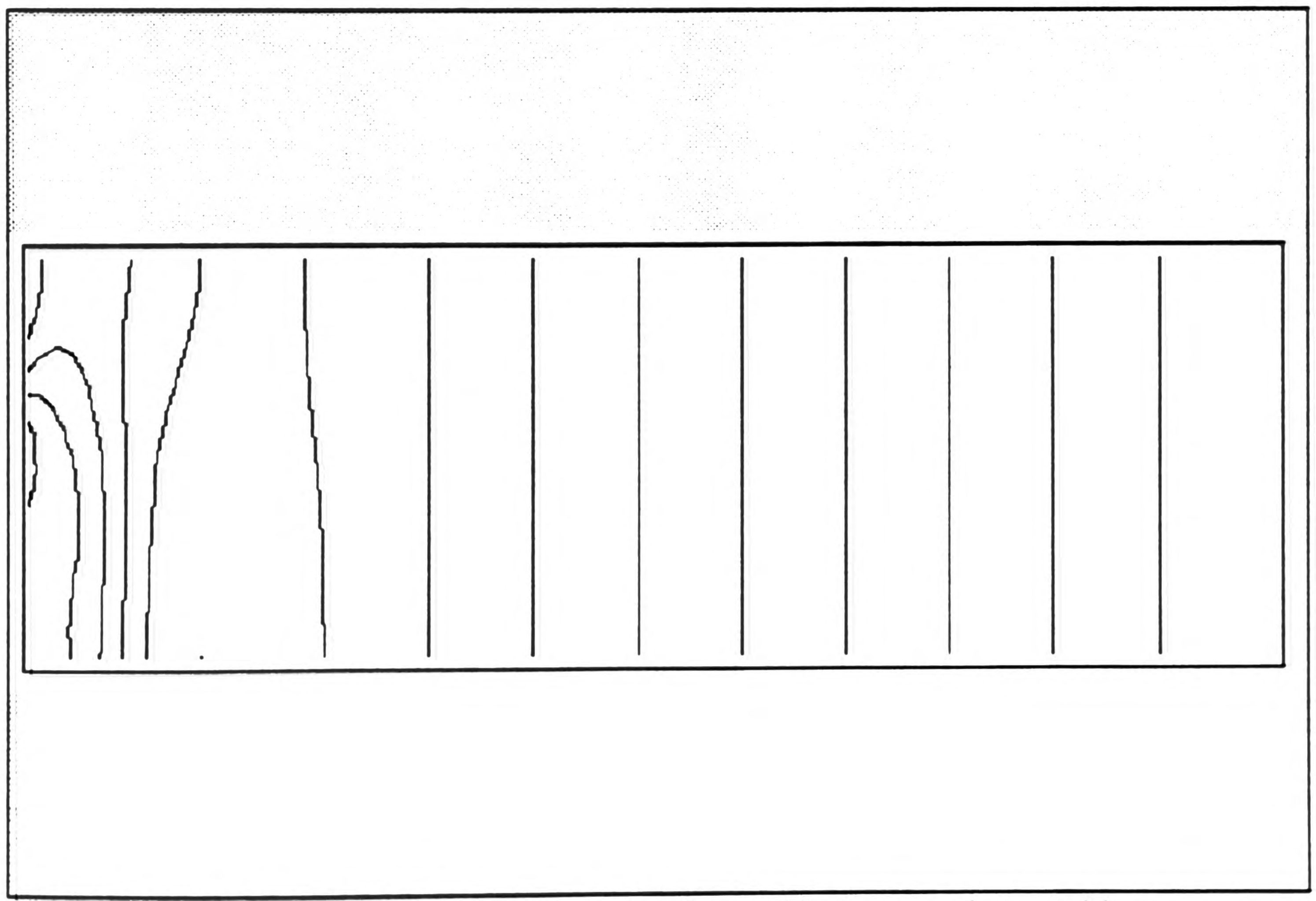


FIGURE 5.4.2-1d Pressure contour plot for sudden expansion problem. Contours at -0.121 (0.0717) 0.596

time provides a history of the convergence for this problem and is shown in figure 5.4.2-2. The plot shows that there is little to choose between the LSOR and JCG algorithms, the JCG algorithm being marginally faster. Both the LSOR and JCG algorithms are over 3 times faster than the JUR algorithm. Unlike the cavity problem where the coupling is weak between the governing equations, here the coupling is more pronounced and as a result there is a non-monotonic decrease in the residuals. Difficulties in convergence were experienced with the JCG algorithm, and as a result it was necessary to relax the solution. This is in marked contrast to the results obtained in Chapter 4. Although it is not clear why the algorithm behaved in this way, a possible explanation can be due to the representation of the matrix by the preconditioner. In the Laplace problem the Jacobi preconditioner is an adequate approximation, however for the pressure-correction matrix it is not so good and this causes an increase in the condition number of the matrix.

5.5 Distribution of computation effort in the SIMPLE procedure

To determine which portion of the SIMPLE procedure should be vectorised, it is first necessary to determine the percentage of CPU time spent in each step. This evaluation is naturally dependent on the problem being solved, the size of the grid used and the choice of algorithm used to solve the resulting discretised equations. For these reasons details of the percentage CPU times are presented for the cavity problem with Re=100 (Table 5.5-1a), Re=400 (Table 5.5.1b) and for the sudden expansion problem (Table 5.5-1c). The tables include the effect of using either a LSOR, JCG or JUR algorithm for the solution of the pressure-correction equation. The most striking feature, regardless of the problem being solved or the algorithm used, is that the solution of the pressure-correction equation is by far the most

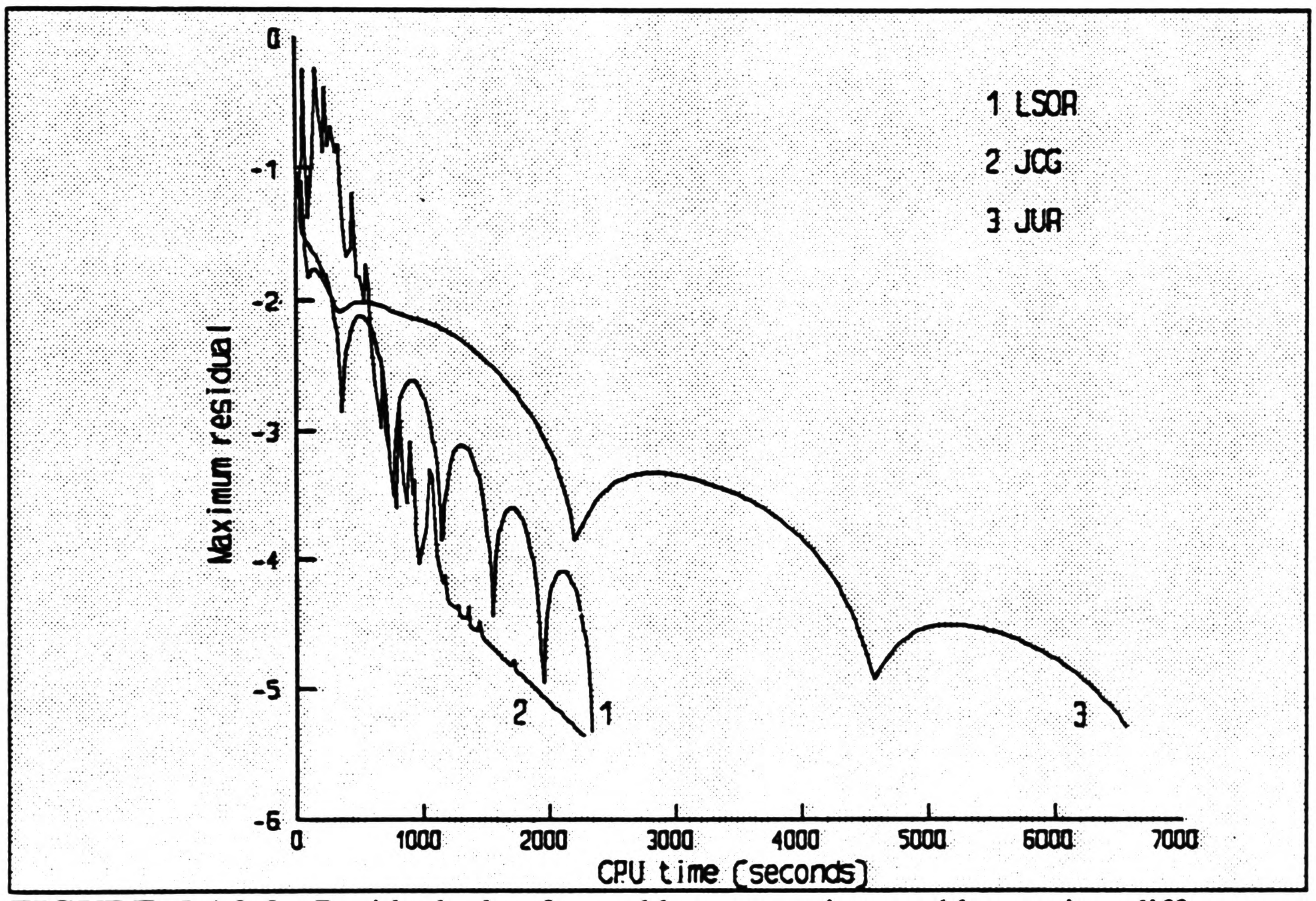


FIGURE 5.4.2-2 Residual plot for sudden expansion problem using different scalar algorithms to solve the pressure-correction equation

(a)

	LSOR	JUR	JCG
Set up source terms for u-momentum equation	2.3	.7	1.7
Set up u-momentum equation coefficients	6.1	1.9	4.6
Solve for u-momentum values u^*	1.9	.6	1.5
Set up source terms for v-momentum equation	2.3	.7	1.7
Set up v-momentum equation coefficients	6.1	1.9	4.6
Solve for v-momentum values v^*	1.9	.6	1.5
Set up pressure-correction coefficients	12.9	4.1	9.7
Solve for pressure-correction values p'	65.4	89.2	73.9
Correct u^*, v^* and p^* to produce u, v and p	1.1	.3	.8

(b)

	LSOR	JUR	JCG
Set up source terms for u-momentum equation	2.3	.7	1.7
Set up u-momentum equation coefficients	5.9	1.9	4.5
Solve for u-momentum values u^*	2.0	.6	1.5
Set up source terms for v-momentum equation	2.3	.7	1.7
Set up v-momentum equation coefficients	5.9	1.9	4.5
Solve for v-momentum values v^*	2.0	.6	1.5
Set up pressure-correction coefficients	12.9	4.1	9.9
Solve for pressure-correction values p'	65.6	89.2	73.9
Correct u^*, v^* and p^* to produce u, v and p	1.1	.3	.8

(c)

	LSOR	JUR	JCG
Set up source terms for u-momentum equation	1.6	.7	1.1
Set up u-momentum equation coefficients	4.1	1.7	2.9
Solve for u-momentum values u^*	1.4	.6	1.0
Set up source terms for v-momentum equation	1.6	.7	1.2
Set up v-momentum equation coefficients	4.3	1.8	3.0
Solve for v-momentum values v^*	1.4	.6	1.0
Set up pressure-correction coefficients	9.0	3.8	6.4
Solve for pressure-correction values p'	75.9	89.8	82.9
Correct u^*, v^* and p^* to produce u, v and p	.7	.3	.5

TABLE 5.5-1 Percentage breakdown of the SIMPLE procedure for (a) cavity problem Re=100 (b) cavity problem Re=400 (c) sudden expansion problem

computationally expensive step in the SIMPLE procedure. The percentage time varies according to the algorithm used. If the LSOR algorithm is used up to 76% of the total time is spent in solving the pressure-correction, this increases to 83% if the JCG algorithm is used and increases further still to 90% if the JUR algorithm is used. Therefore, the vectorisation of the pressure-correction ought to produce a reasonable improvement in speed.

To predict the speed-up factors which might be obtained, an analysis using Amdahl's law is carried out for the JUR and JCG algorithms. In solving the cavity problem the fraction of code that can be vectorised using the JCG algorithm is f_v=0.739, and the expected range of speed-up factors (S) is given by

$$\frac{1}{0.261 + 0.739/4.35} < S < \frac{1}{0.261 + 0.739/173.2}$$

$$2.32 < S < 3.77 \qquad (5.5\text{-}1)$$

When applied to the sudden expansion problem f_v=0.829 and the expected range of speed-up factors are given by

$$\frac{1}{0.171 + 0.829/4.35} < S < \frac{1}{0.171 + 0.829/173.2}$$

$$2.77 < S < 5.69 \qquad (5.5\text{-}2)$$

Therefore, when the vectorised JCG algorithm is used, a maximum speed-up of 3.77 can be expected for the solution of the cavity problem and a factor up to 5.69 is expected for the sudden expansion problem, when compared with the equivalent scalar algorithm.

When the JUR algorithm is used, the fraction of code that can be vectorised for the cavity problem is f_v=0.892, which leads to expected speed-up factors in the range

$$\frac{1}{0.108 + 0.892/4.35} < S < \frac{1}{0.108 + 0.892/173.2}$$

$$3.19 < S < 8.84 \qquad (5.5\text{-}3)$$

When applied to the sudden expansion problem, the fraction of code which can be vectorised is increased to f_v=0.898 and the expected speed-up factors are given by

$$\frac{1}{0.102 + 0.898/4.35} < S < \frac{1}{0.102 + 0.898/173.2}$$

$$3.24 < S < 9.33 \qquad (5.5\text{-}4)$$

Therefore, when the vectorised JUR algorithm is used, a maximum speed-up of 8.84 can be expected for the solution of the cavity problem and a factor up to 9.33 is expected for the sudden expansion problem, when compared with the equivalent scalar algorithm.

5.6 Vectorisation of the pressure-correction equation

From the general control-volume equation (3.4.7-1) the pressure-correction equation can be written as

$$a^P_{ij}p'_{ij} - a^E_{ij}p'_{i+1j} - a^W_{ij}p'_{i-1j} - a^N_{ij}p'_{ij+1} - a^S_{ij}p'_{ij-1} = b^P_{ij} \qquad \begin{array}{l} i=1(1)n \\ j=1(1)m \end{array} \qquad (5.6\text{-}1)$$

where

$$a^P_{ij} = a^E_{ij} + a^W_{ij} + a^N_{ij} + a^S_{ij}$$

The coefficients a_{ij} are related to the coefficients defined in the momentum equations and the right-hand-side term b^P_{ij} relates to the continuity residual for a given control-volume. The resulting matrix is symmetric and diagonally dominant and providing the structure is not destroyed, only half the matrix needs to be stored. The matrix is stored in a diagonal format so that the vectors are typically of length *nm* and suitable for manipulation by the pipeline processor (Ierotheou, Richards and Cross [1989a]). The only vectors which need to be stored in the pipeline memory are the pressure-correction vector $\mathbf{p}'$, the central component coefficient vector $\mathbf{a}^P$, the east component coefficient vector $\mathbf{a}^E$, the north component coefficient vector $\mathbf{a}^N$ and the right-hand-side vector $\mathbf{b}^P$. The vectors $\mathbf{a}^W$ and $\mathbf{a}^S$ are not needed since they are contained within the vectors $\mathbf{a}^E$ and $\mathbf{a}^N$, respectively.

Figures 5.6-1 to 5.6-3 show the variation of residual with CPU time for the scalar and vector implementations of the JCG algorithms (denoted by JCGS and JCGV), when applied to the two test problems. The results indicate that for the cavity problem a factor of 3 improvement is achieved, and for the sudden expansion problem a factor of 5.4 is achieved. These results compare well with the predictions stated using Amdahl's law (5.5-1) and (5.5-2). Similar residual plots are presented for the scalar and vector implementations of the JUR algorithm, denoted by JURS and JURV, respectively (figures 5.6-4 to 5.6-6). For the solution of the cavity problem a speed-up factor of up to 7 is achieved and for the solution of the sudden expansion a factor of up to 8 is achieved. Again, these speed-up factors are in good agreement with those stated using Amdahl's law (5.5-3) and (5.5-4).

Finally, figures 5.6-7 to 5.6-9 show a comparison between the scalar LSOR algorithm and the vector JUR and JCG algorithms. For the cavity problem the results indicate that there is little to choose between the JURV and JCGV algorithms, the JCGV being marginally faster. Both vector algorithms are a factor of 2.3 faster than the scalar LSOR algorithm. In the case of the sudden expansion problem, the JCGV algorithm is over 2 times more efficient than the JURV algorithm and nearly 6 times more efficient than the scalar LSOR algorithm.

These results compare favourably with other works found in the literature. These include the efforts of Spradley et al [1981], Hemker et al [1984], Vanka and Misengades [1987] and Schonauer and Schnepf [1988]. In each case the emphasis was on the solution of an algebraic system of linear equations and comparisons were based on the scalar and vector equivalent algorithms.

The distribution of computational effort for the SIMPLE procedure with a vectorised algorithm used to solve the pressure-correction equation is shown in tables 5.6-1a to 5.6-1c for both test problems. Two factors are significant, firstly, the percentage CPU time taken to solve the pressure-correction equation is dramatically reduced. For the cavity problem this is reduced from 74% to 5% if the JCGV algorithm is used, and from 89% to 9% if the JURV algorithm is used. The reductions are not as substantial for the sudden expansion problem, for the JCGV algorithm the reduction is from 83% to 18% and for the JURV algorithm the reduction is from 90% to 21%. The second significant factor is that the generation of the coefficients (from the momentum and pressure-correction equations) are now the major contributors to the total CPU time, taking up to 68% of the total time.

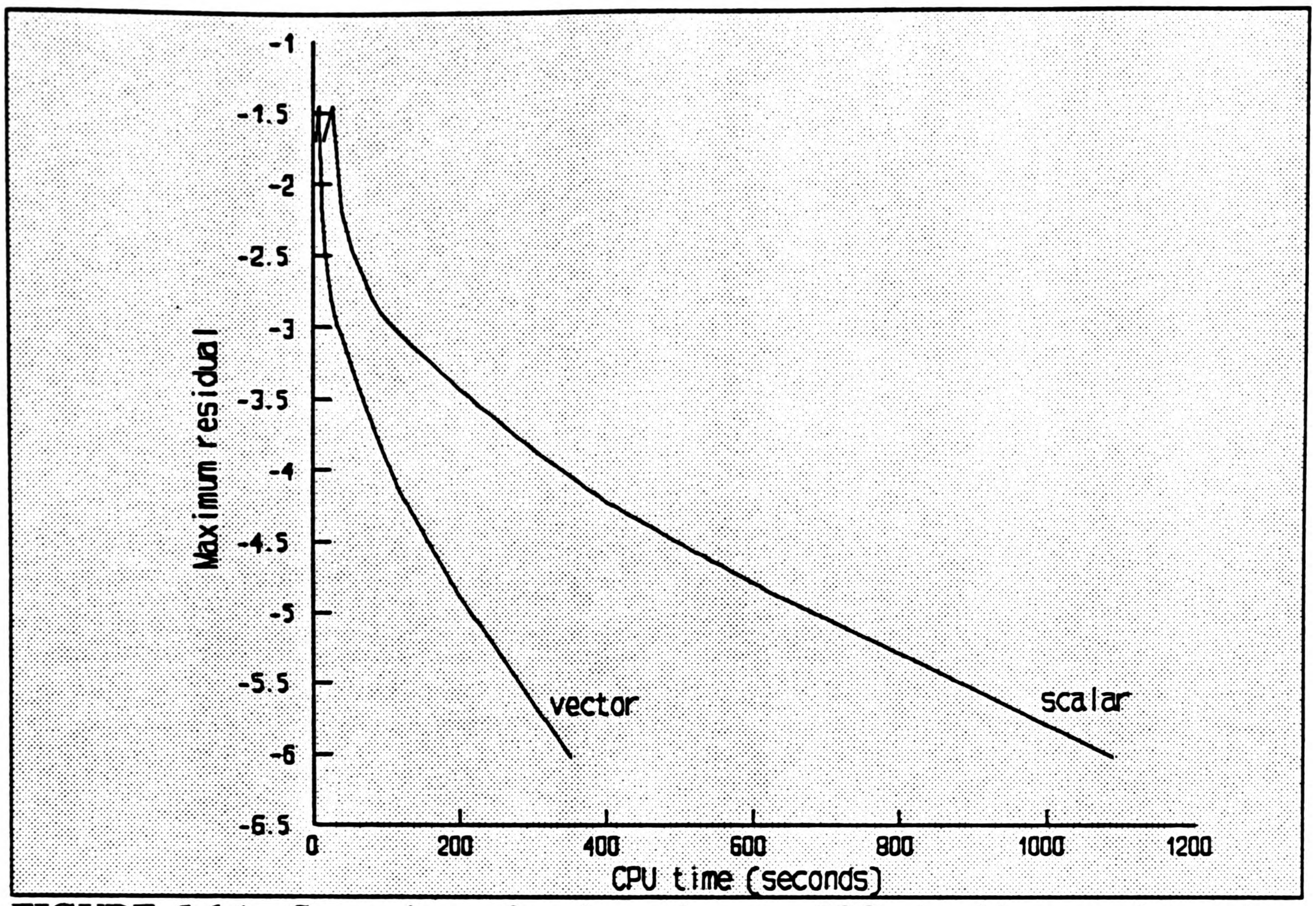


FIGURE 5.6-1 Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the cavity problem (Re=100)

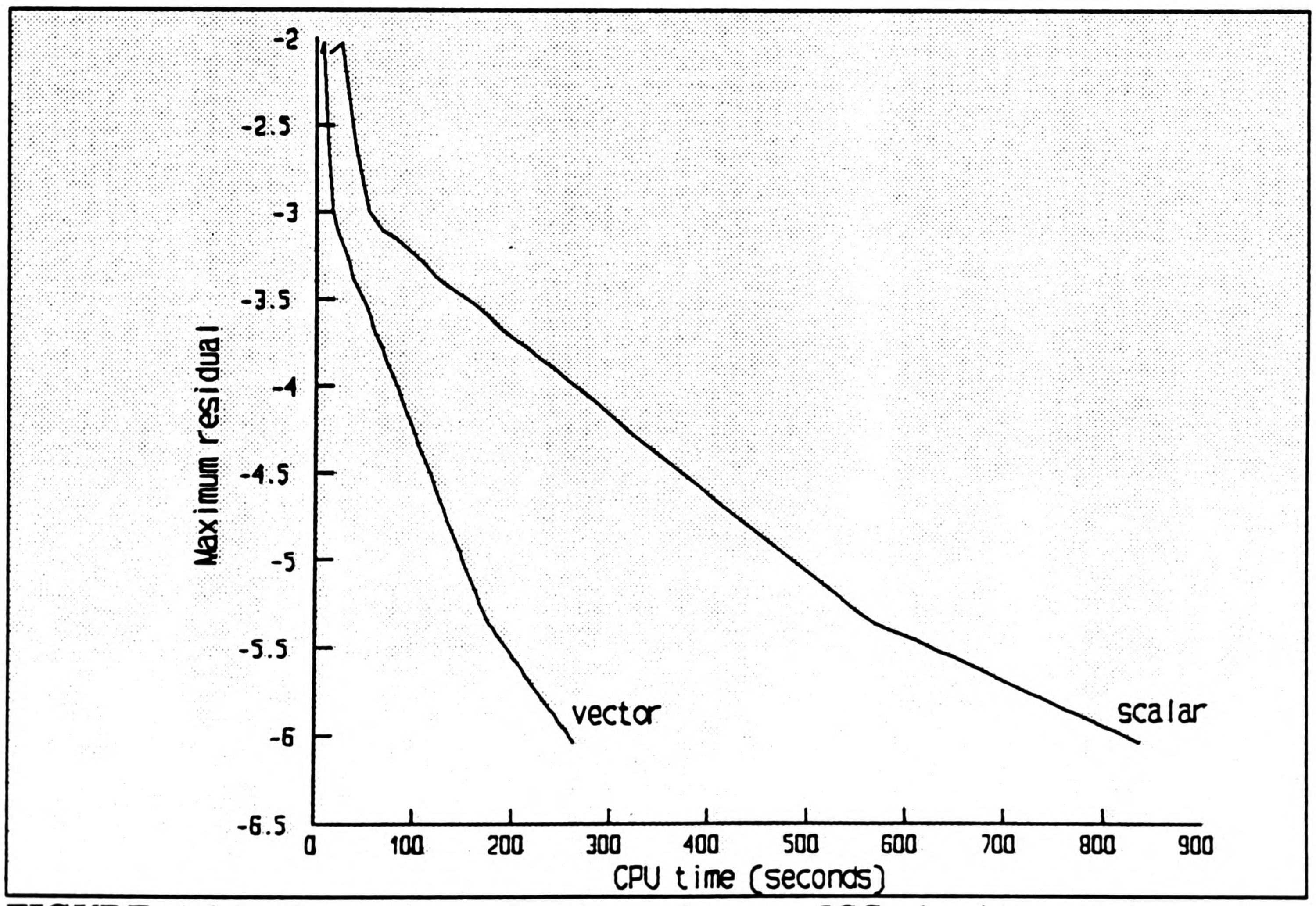


FIGURE 5.6-2 Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the cavity problem (Re=400)

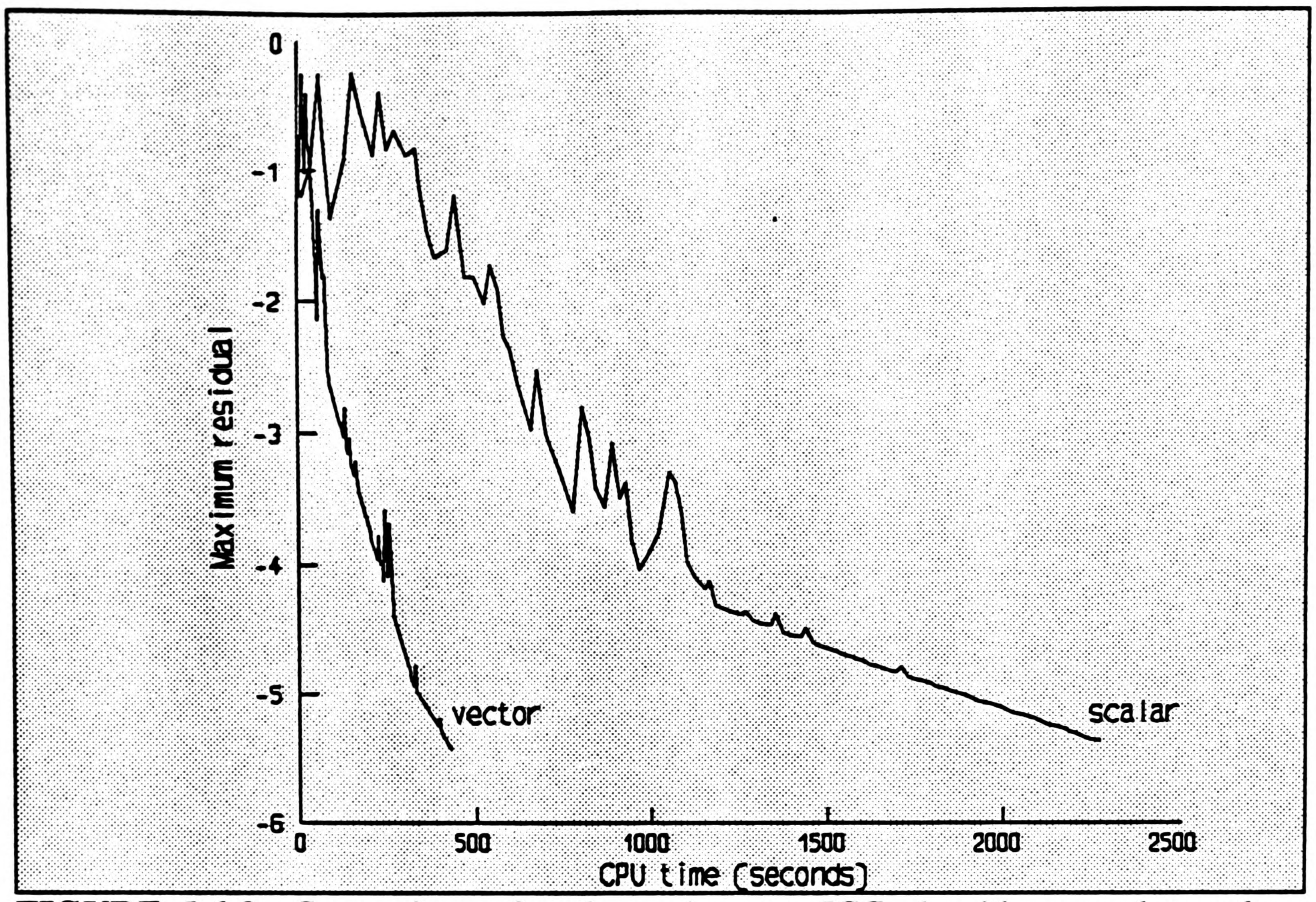


FIGURE 5.6-3 Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the sudden expansion problem

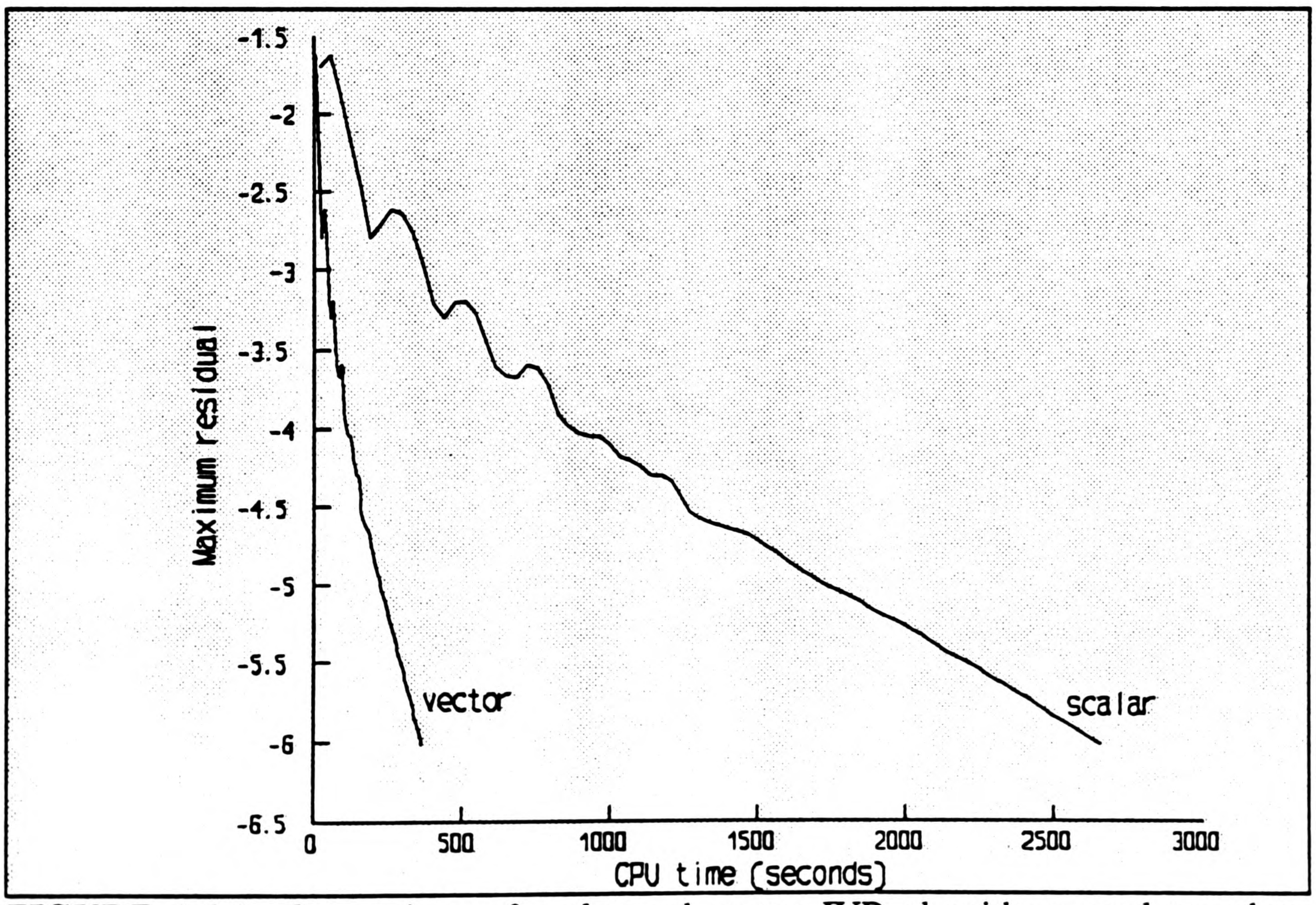


FIGURE 5.6-4 Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the cavity problem (Re=100)

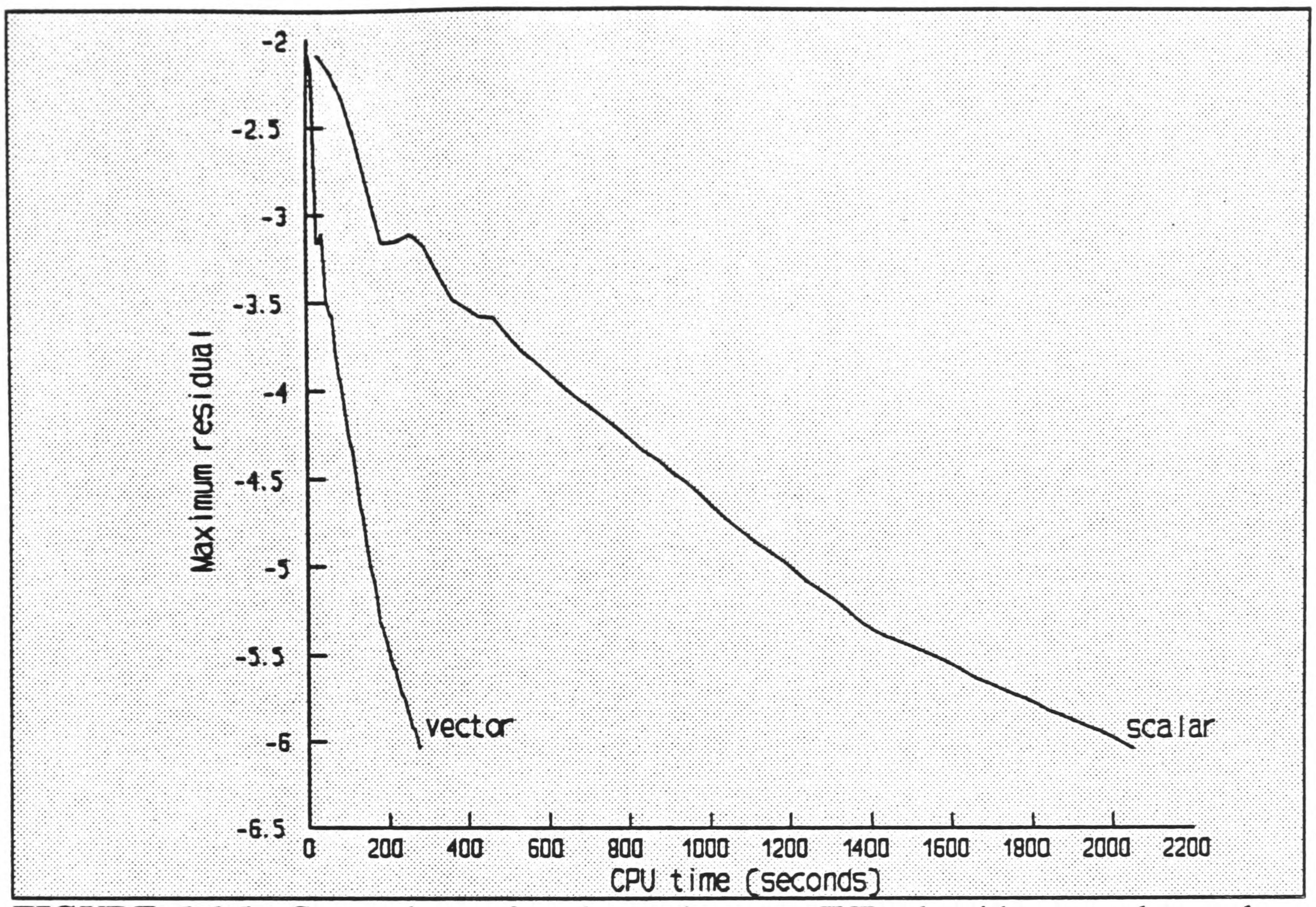


FIGURE 5.6-5 Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the cavity problem (Re=400)

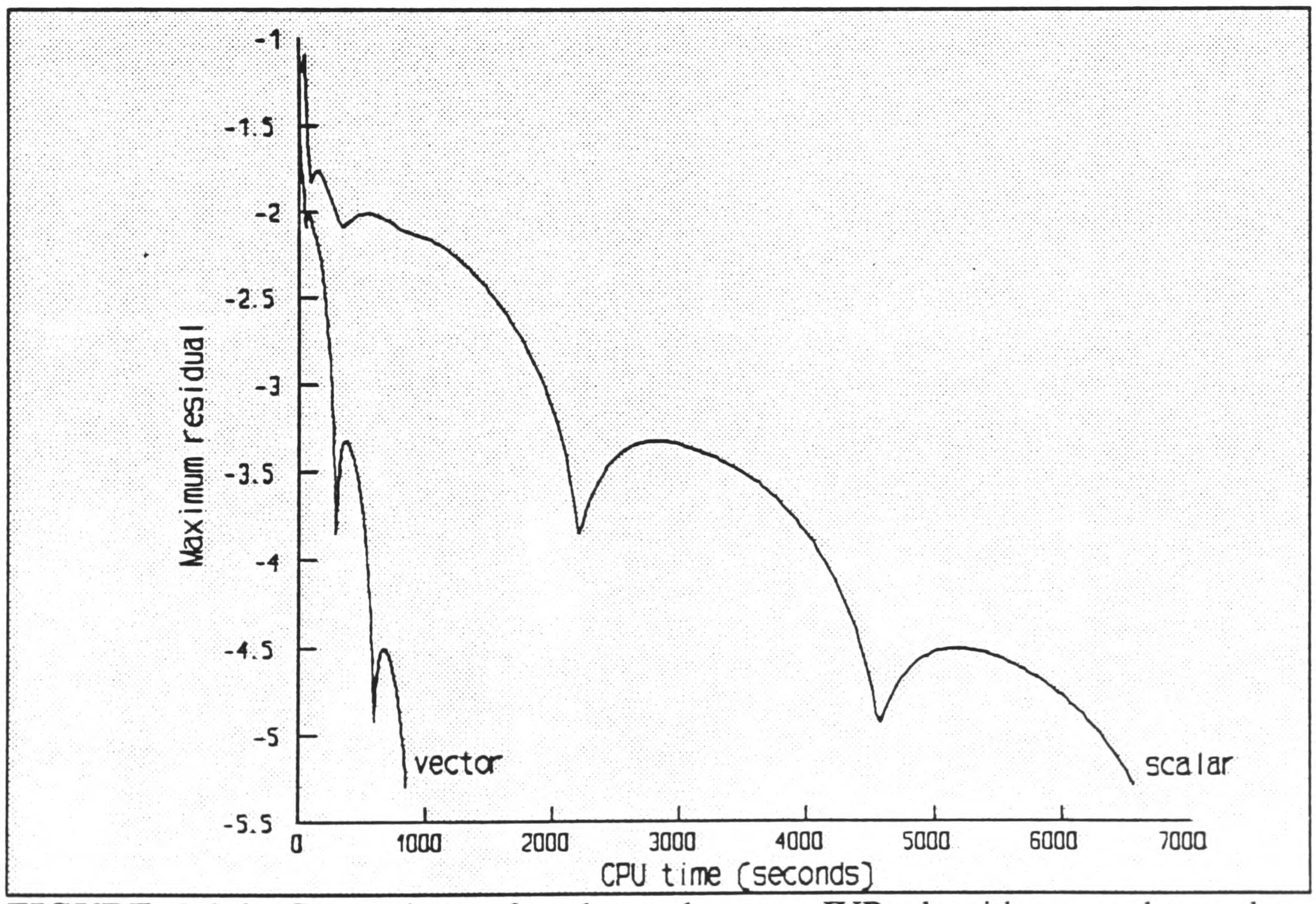


FIGURE 5.6-6 Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the sudden expansion problem

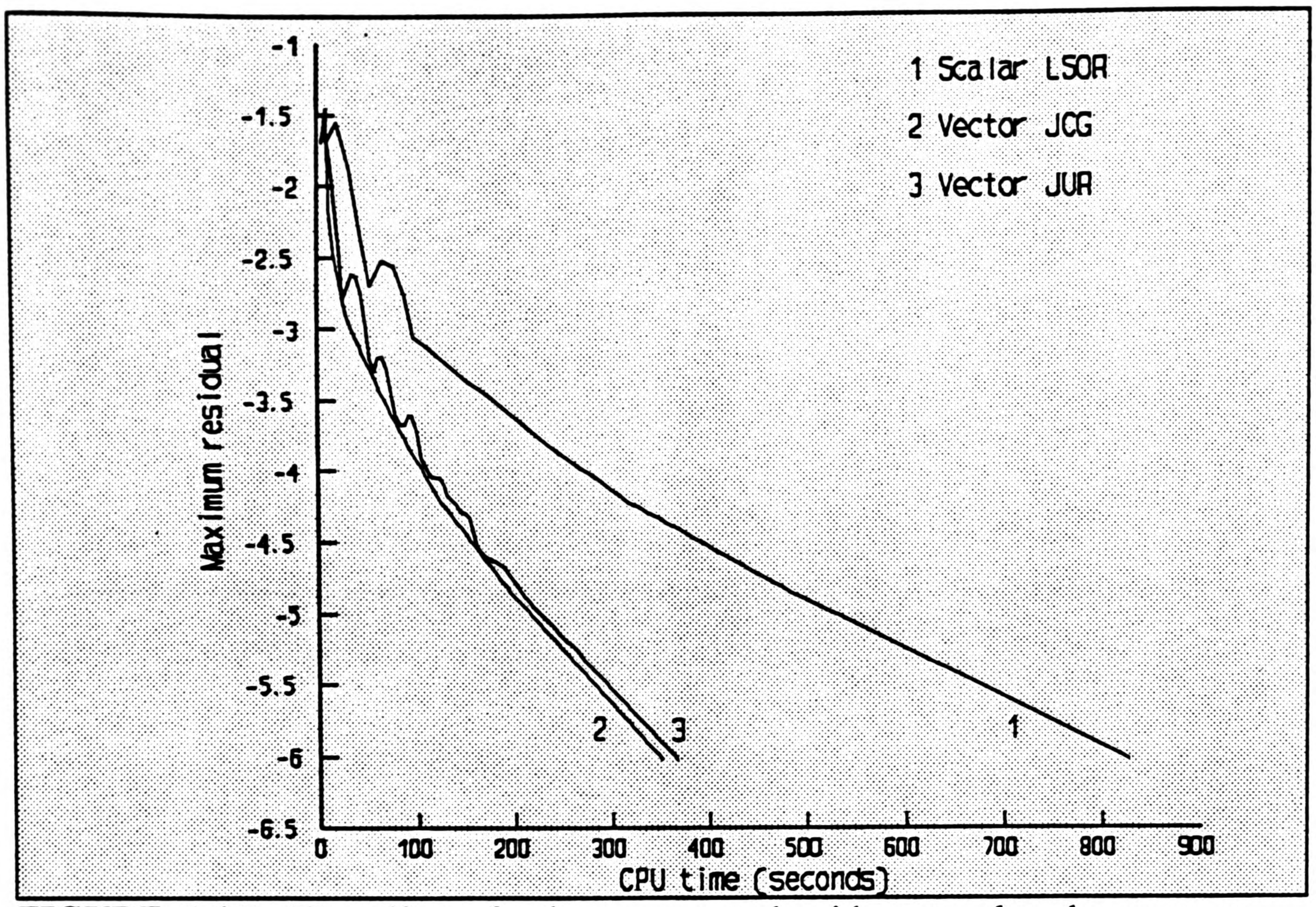


FIGURE 5.6-7 The effect of using a vector algorithm to solve the pressure-correction equation in the cavity problem (Re=100)

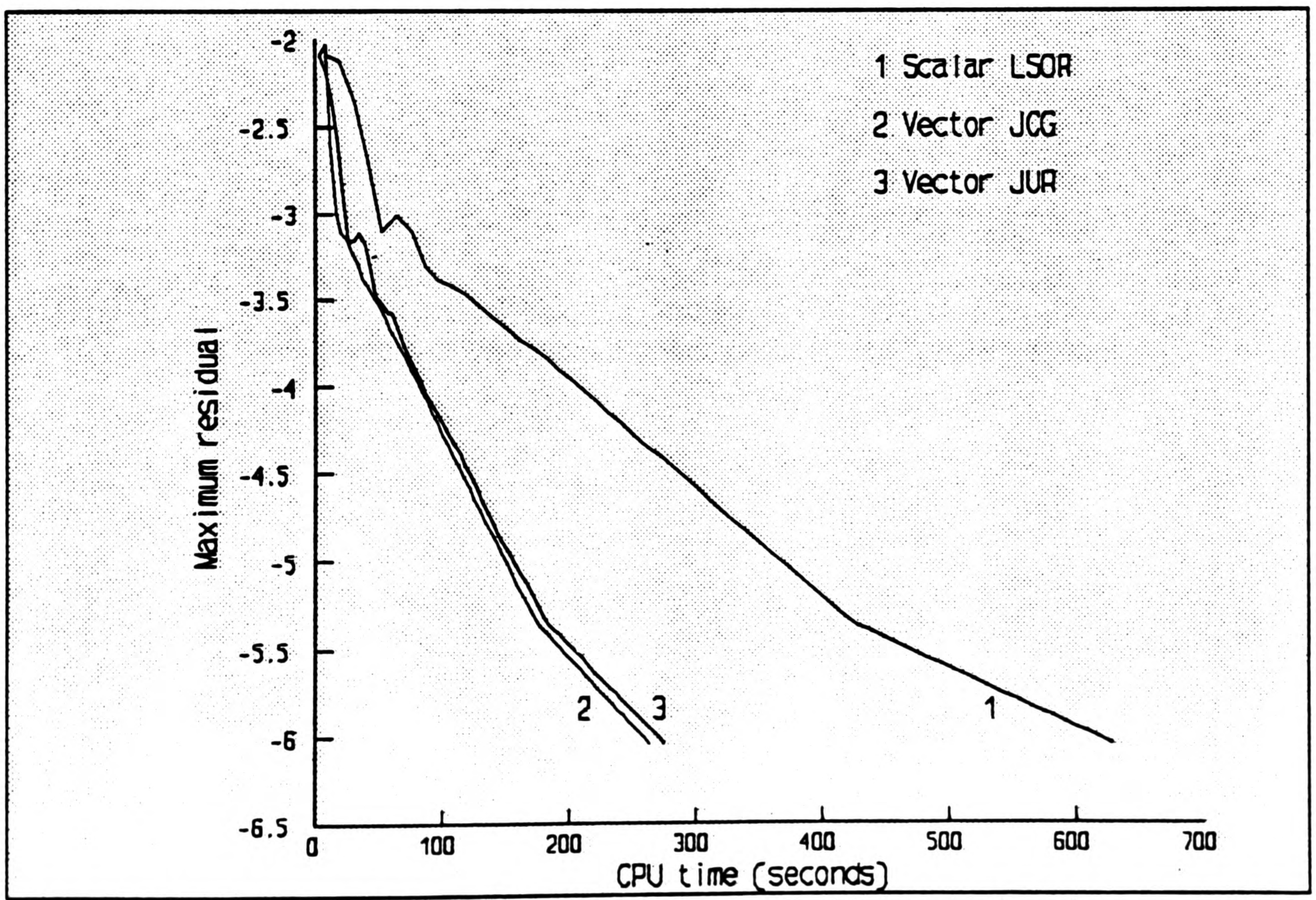


FIGURE 5.6-8 The effect of using a vector algorithm to solve the pressure-correction equation in the cavity problem (Re=400)

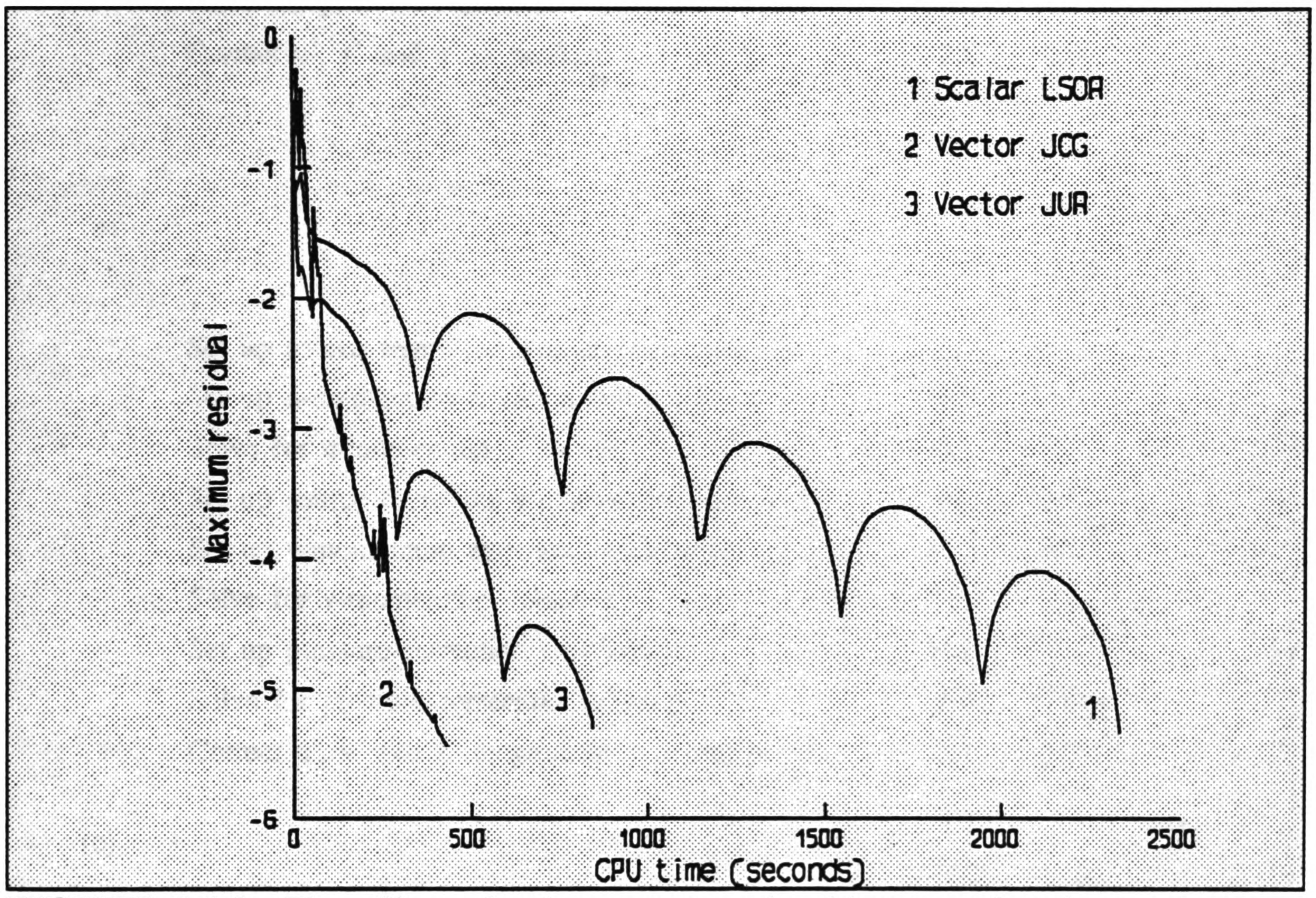


FIGURE 5.6-9 The effect of using a vector algorithm to solve the pressure-correction equation in the sudden expansion problem

(a)

	JUR	JCG
Set up source terms for u-momentum equation	6.0	6.3
Set up u-momentum equation coefficients	15.7	16.4
Solve for u-momentum values u^*	5.3	5.5
Set up source terms for v-momentum equation	6.0	6.3
Set up v-momentum equation coefficients	15.7	16.4
Solve for v-momentum values v^*	5.3	5.5
Set up pressure-correction coefficients	34.2	35.7
Solve for pressure-correction values p'	9.0	4.9
Correct u^*, v^* and p^* to produce u, v and p	2.8	3.0

(b)

	JUR	JCG
Set up source terms for u-momentum equation	6.0	6.3
Set up u-momentum equation coefficients	15.7	16.4
Solve for u-momentum values u^*	5.3	5.5
Set up source terms for v-momentum equation	6.0	6.3
Set up v-momentum equation coefficients	15.7	16.4
Solve for v-momentum values v^*	5.3	5.5
Set up pressure-correction coefficients	34.2	35.7
Solve for pressure-correction values p'	9.0	4.9
Correct u^*, v^* and p^* to produce u, v and p	2.8	3.0

(c)

	JUR	JCG
Set up source terms for u-momentum equation	5.1	5.3
Set up u-momentum equation coefficients	13.4	13.9
Solve for u-momentum values u^*	4.5	4.7
Set up source terms for v-momentum equation	5.3	5.5
Set up v-momentum equation coefficients	14.0	14.5
Solve for v-momentum values v^*	4.7	4.9
Set up pressure-correction coefficients	29.6	30.7
Solve for pressure-correction values p'	20.9	18.0
Correct u^*, v^* and p^* to produce u, v and p	2.5	2.5

TABLE 5.6-1 Percentage breakdown of the SIMPLE procedure for (a) cavity problem Re=100 (b) cavity problem Re=400 (c) sudden expansion problem. Vectorised pressure-correction solver.

It is clear that although there is a significant reduction in the total time taken to solve the pressure-correction equation the overall improvement is limited by the time taken to generate the necessary coefficients. Hence, further vectorisation of the solution procedure is necessary.

5.7 Further vectorisation of the SIMPLE solution procedure

This is made more difficult because the following features do not fit readily into the pipeline processor environment:

(i) The boundary conditions. These are highly problem dependent and are implemented as part of the source term contribution. In addition they are only applied to a subset of the nodes being solved and this would lead to an inefficient use of the pipeline processor.

(ii) The variable fluid properties. These are also problem dependent and may involve complex formulae.

(iii) Complicated source terms. These can also involve complex formulae typified by those used in turbulence modelling. Like the boundary conditions, they are problem dependent and are only applied to a subset of the nodes.

The implementation of these features is further hindered by the presence of condition statements and because of these difficulties all source terms and variable fluid properties are computed using the scalar processor. All other calculations are carried out using the pipeline processor. The evaluation of the convection terms using an upwind differencing scheme is also computed using the pipeline processor despite the use of comparison statements. This problem is overcome by re-structuring the scheme in a suitable form for vectorisation and involves no condition statements, instead the coefficient for a typical east node is evaluated as

$$C_e = D_e + 1/2(|F_e| - F_e).$$

where D_e is the diffusion coefficient and F_e is the convective flux.

The solution of the momentum equations also need to be reconsidered. In order to use the pipeline processor to solve the momentum equations a different numerical algorithm is needed instead of the LJUR algorithm. The reason for this is the inefficient usage of the pipeline processor when using a line-by-line algorithm because of the relatively small vector lengths involved. It has already been shown that this makes tridiagonal solvers unattractive for use on the VA-1 pipeline processor (Chapter 4). Instead a basic JUR algorithm is to be used, mainly because it is straightforward to implement but also because it has demonstrated impressive speed-up rates (Chapter 4). Since the JUR and LJUR algorithms do not have the same rate of convergence the number of JUR iterations are adjusted so that approximately the same rate of convergence is achieved. The result is that the JUR iterations are increased to 2 for the cavity problem and to 12 for the sudden expansion problem to give the same residual reduction rate per SIMPLE iteration.

To avoid unnecessary transferring of data between the host processor and the pipeline processor, all geometrical data and initial fields are stored in the pipeline memory before the start of the SIMPLE procedure. The main steps of the procedure are now described:

(i) Compute the fluid properties (scalar processor).

(ii) Compute the linearised source terms for the u-momentum equations (scalar processor).

(iii) Transfer the data obtained from (i) to the pipeline memory.

(iv) Complete the coefficients and right-hand-side vectors, then solve the u-momentum equation (pipeline processor).

(v) Transfer the u-solution field back to host memory.

(vi) Repeat steps (ii)-(v) for the v-momentum equation.

(vii) Compute the linearised source term for the pressure-correction equation (scalar processor).

(viii) Transfer the data from step (vii) to the pipeline memory.

(ix) Assemble the pressure-correction coefficients and continuity residuals (pipeline processor).

(x) Solve the pressure-correction field (pipeline processor).

(xi) Use the pressure-correction field to correct the velocity and pressure fields (pipeline processor).

(xii) Transfer the velocity and pressure fields back to the host memory.

5.8 Results

To determine the speed-up factors when vectorising the SIMPLE procedure computations are also carried out using only the scalar processor. Tables 5.8-1a to 5.8-1c show that the solution of the pressure-correction equation is still the most computationally demanding step. Although the percentages are lower than those obtained when a LJUR algorithm is used to solve the momentum equations, the advantages of this approach are demonstrated by a much higher percentage of the SIMPLE procedure being vectorised. When the JCG algorithm is used to solve the pressure-correction equation up to 96% of the procedure can be vectorised and when the JUR algorithm is used this percentage is increased to 98%.

Using Amdahl's law, a prediction can be made for the improvements in speed which can be expected when the SIMPLE procedure is vectorised using the modified solution approach. For the case where the JUR algorithm is used to solve the pressure-correction equation the fraction of code vectorised is $f_v=0.98$ and the expected speed-up factors are

$$\frac{1}{0.02 + 0.98/4.35} < S < \frac{1}{0.02 + 0.98/173.2}$$

$$4.08 < S < 38.97 \tag{5.8-1}$$

For the sudden expansion problem $f_v=0.982$ and hence the expected speed-up factors are given by

$$\frac{1}{0.018 + 0.982/4.35} < S < \frac{1}{0.018 + 0.982/173.2}$$

$$3.19 < S < 42.25 \tag{5.8-2}$$

When the JCG algorithm is used, the fraction of code vectorised in the cavity problem is fv=0.957 and this gives a predicted speed-up range of

$$\frac{1}{0.043 + 0.957/4.35} < S < \frac{1}{0.043 + 0.957/173.2}$$

$$3.8 < S < 20.61 \tag{5.8-3}$$

and for the sudden expansion problem f_v=0.965, so the range of speed-up factors is given by

$$\frac{1}{0.035 + 0.965/4.35} < S < \frac{1}{0.035 + 0.965/173.2}$$

$$3.8 < S < 24.65 \tag{5.8-4}$$

Residual plots are shown in figures 5.8-1 to 5.8-3 for both test problems when the JUR algorithm is used to solve the pressure-correction equation, the results compare the JURS and JURV algorithms. In each case significant reductions in the CPU times are achieved, these range from a factor of 22 for the cavity problem to over 28 for the sudden expansion problem. These speed-up factors are in good agreement with those predicted in (5.8-1) and (5.8-2). Residual plots are also shown for the JCG algorithm (figures 5.8-4 to 5.8-6). Again, there are significant reductions in the CPU time with factors of 11 for the cavity problem and up to 20 for the sudden expansion problem. These factors are also in good agreement with those predicted in (5.8-3) and (5.8-4).

As a result of vectorising as many of the steps in the SIMPLE procedure, a final study of how the computation effort was distributed is presented in tables 5.8-2a to 5.8-2c. For the cavity problem the changes to fluid properties such as the viscosity has little effect on the overall performance of vectorisation, although this may not be the case for higher Reynolds numbers.

In both problems the single major contribution to the CPU time is now the generation of the source term for the momentum and pressure-correction equations. The solution of the pressure-correction equation using the JUR algorithm still constitutes up to 31% of the total CPU time and this performance may be enhanced with the introduction of a multigrid method, for example.

If the coupling between the momentum and pressure equations is strong (as demonstrated by the sudden expansion problem), this will result in a greater computation effort required by the algorithm to solve the pressure-correction equation. This would be necessary to keep the continuity residuals under control, failure to do so could result in exceptionally high simulation times and even divergence of the procedure.

The speed-up factors appear to be very flattering to the JUR and JCG algorithms, however, what really matters is a 'practical' speed-up and this may not be as impressive. A comparison is therefore made between the best scalar algorithm and the two vector algorithms (figures 5.8-7 to 5.8-9). For the solution of the cavity problem there is little to choose between the LSOR and JCGS algorithms, the LSOR algorithm being marginally faster. Thus comparing the LSOR with the vector algorithms, the JURV algorithm is 5 times faster and the JCGV algorithm up to 6 times faster than the LSOR algorithm. In the solution of the sudden expansion problem the JCGS algorithm is found to be marginally faster than the LSOR algorithm. Comparing the JCGS algorithm with the JURV and JCGV algorithms, factors of over 7 and 29 are achieved in favour of the vector algorithms.

The results show that the best vector algorithm can solve the cavity problem 6 times faster than the best scalar algorithm and for the sudden expansion problem this factor is increased to 29. In the solution of the cavity problem the coupling of

(a)

	JUR	JCG
Set up source terms for u-momentum equation	.7	1.6
Set up u-momentum equation coefficients	1.8	4.3
Solve for u-momentum values u^*	2.0	4.6
Set up source terms for v-momentum equation	.7	1.6
Set up v-momentum equation coefficients	1.8	4.3
Solve for v-momentum values v^*	2.0	4.6
Set up pressure-correction coefficients	4.2	9.7
Solve for pressure-correction values p'	86.5	68.5
Correct u^*, v^* and p^* to produce u, v and p	.3	.8

(b)

	JUR	JCG
Set up source terms for u-momentum equation	.7	1.6
Set up u-momentum equation coefficients	1.9	4.1
Solve for u-momentum values u^*	2.0	4.5
Set up source terms for v-momentum equation	.7	1.6
Set up v-momentum equation coefficients	1.9	4.1
Solve for v-momentum values v^*	2.0	4.5
Set up pressure-correction coefficients	4.2	9.4
Solve for pressure-correction values p'	86.3	69.4
Correct u^*, v^* and p^* to produce u, v and p	.3	.8

(c)

	JUR	JCG
Set up source terms for u-momentum equation	.7	1.3
Set up u-momentum equation coefficients	1.3	2.7
Solve for u-momentum values u^*	9.4	19.2
Set up source terms for v-momentum equation	.7	1.3
Set up v-momentum equation coefficients	1.6	3.2
Solve for v-momentum values v^*	9.5	19.4
Set up pressure-correction coefficients	3.3	6.8
Solve for pressure-correction values p'	73.2	45.6
Correct u^*, v^* and p^* to produce u, v and p	.3	.5

TABLE 5.8-1 Percentage breakdown of the SIMPLE procedure for (a) cavity problem Re=100 (b) cavity problem Re=400 (c) sudden expansion problem. JUR used to solve momentum equations.

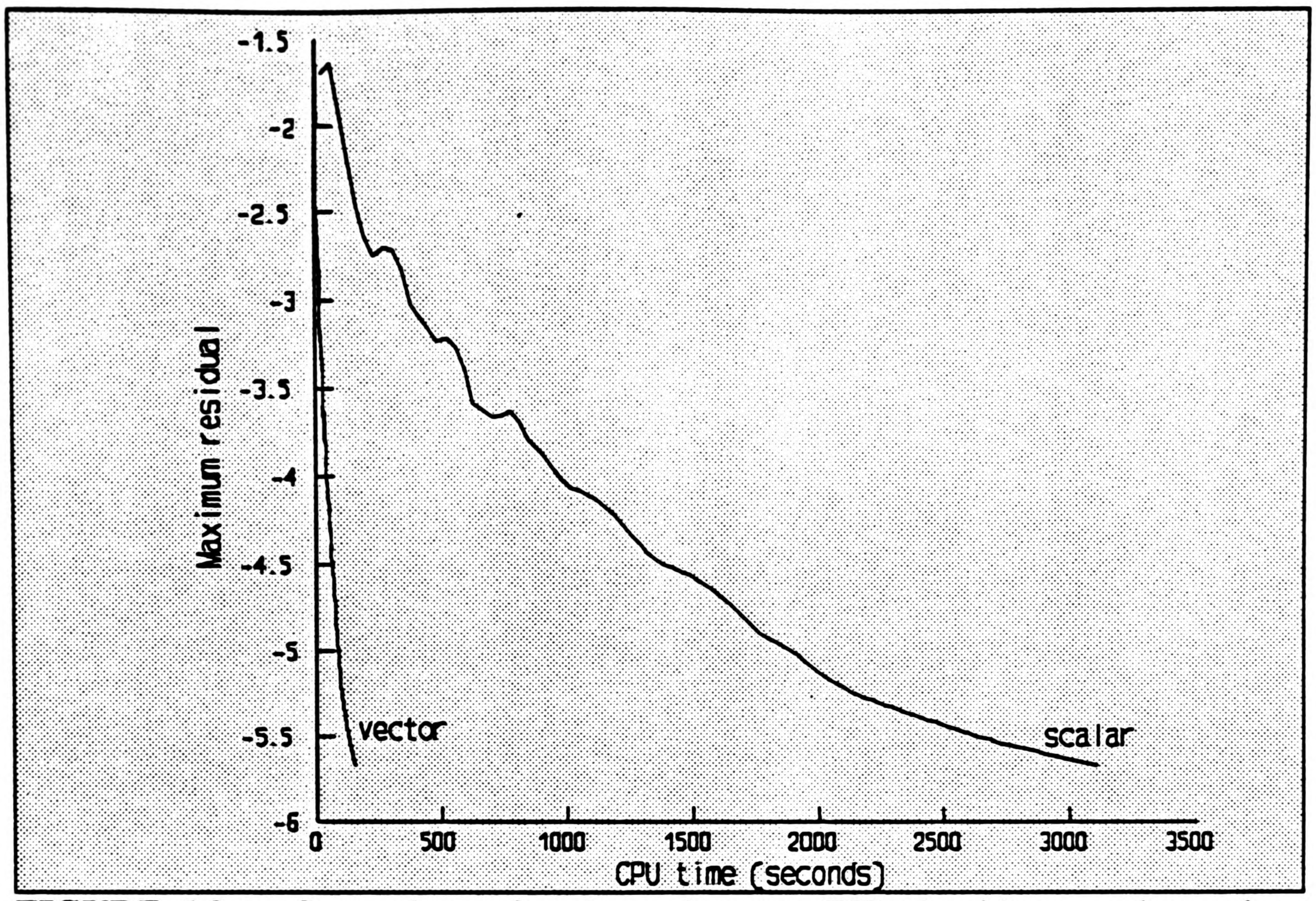


FIGURE 5.8-1 Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the cavity problem (Re=100). JUR used to solve the momentum equations.

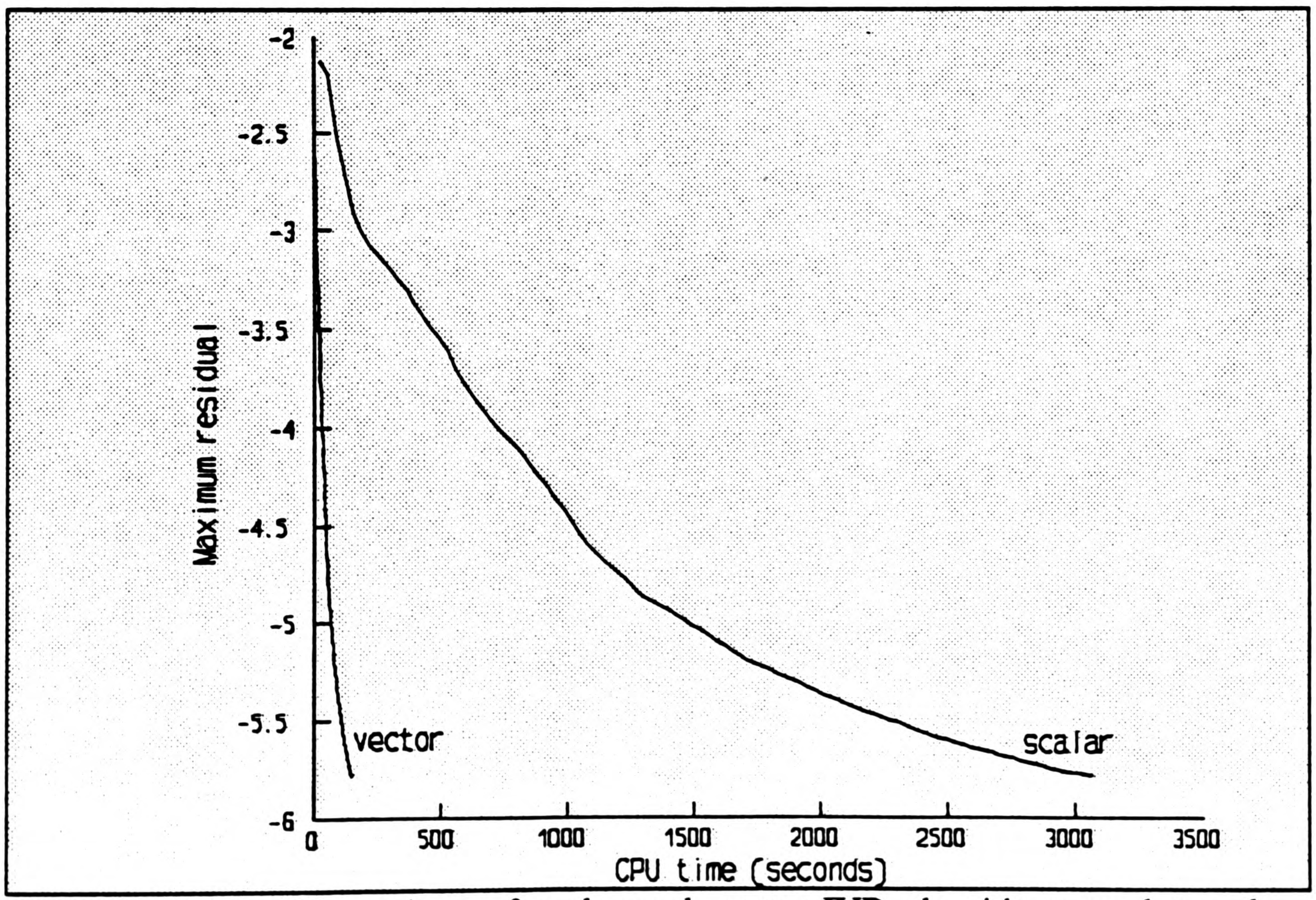


FIGURE 5.8-2 Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the cavity problem (Re=400). JUR used to solve the momentum equations.

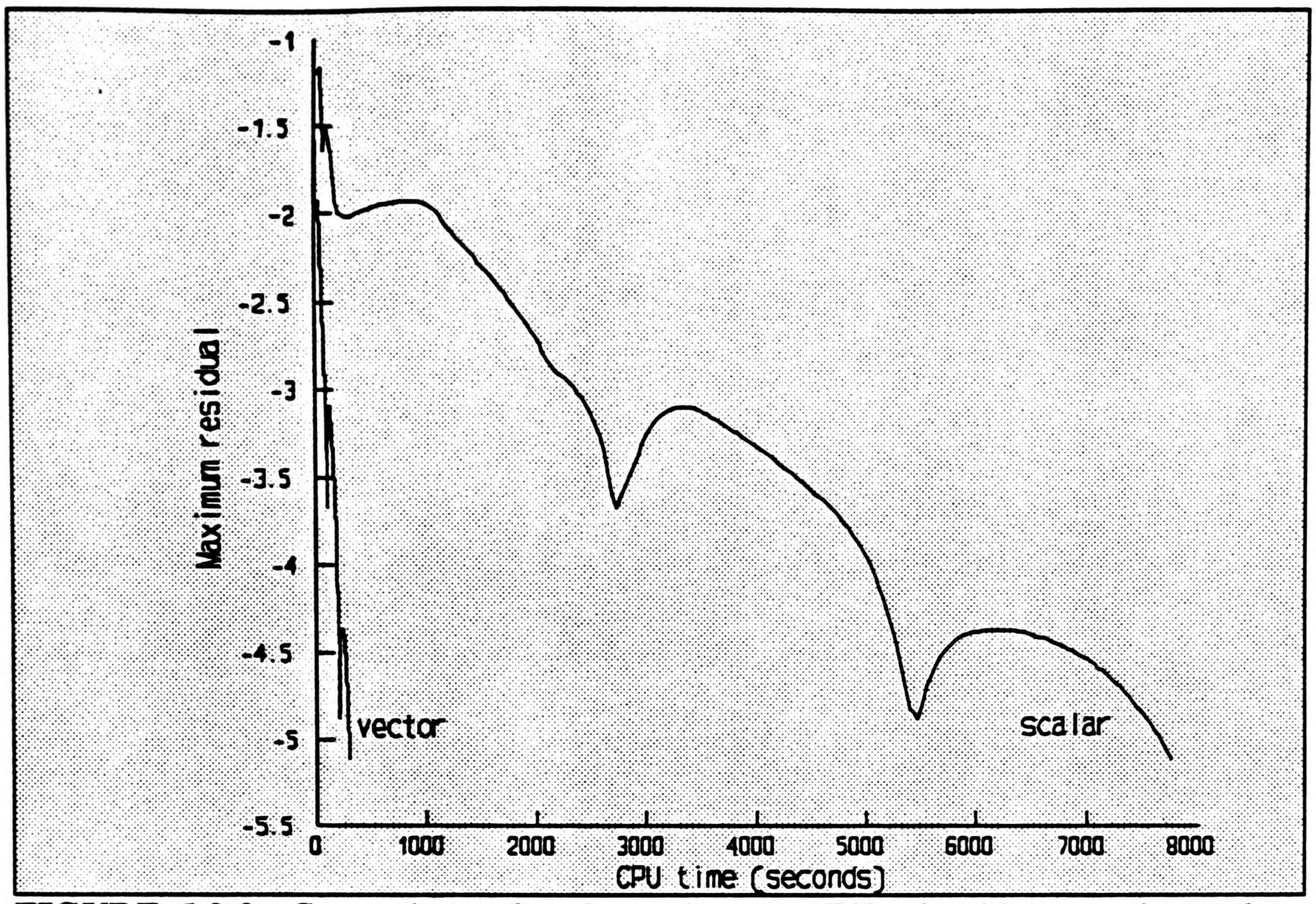


FIGURE 5.8-3 Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the sudden expansion problem. JUR used to solve the momentum equations.

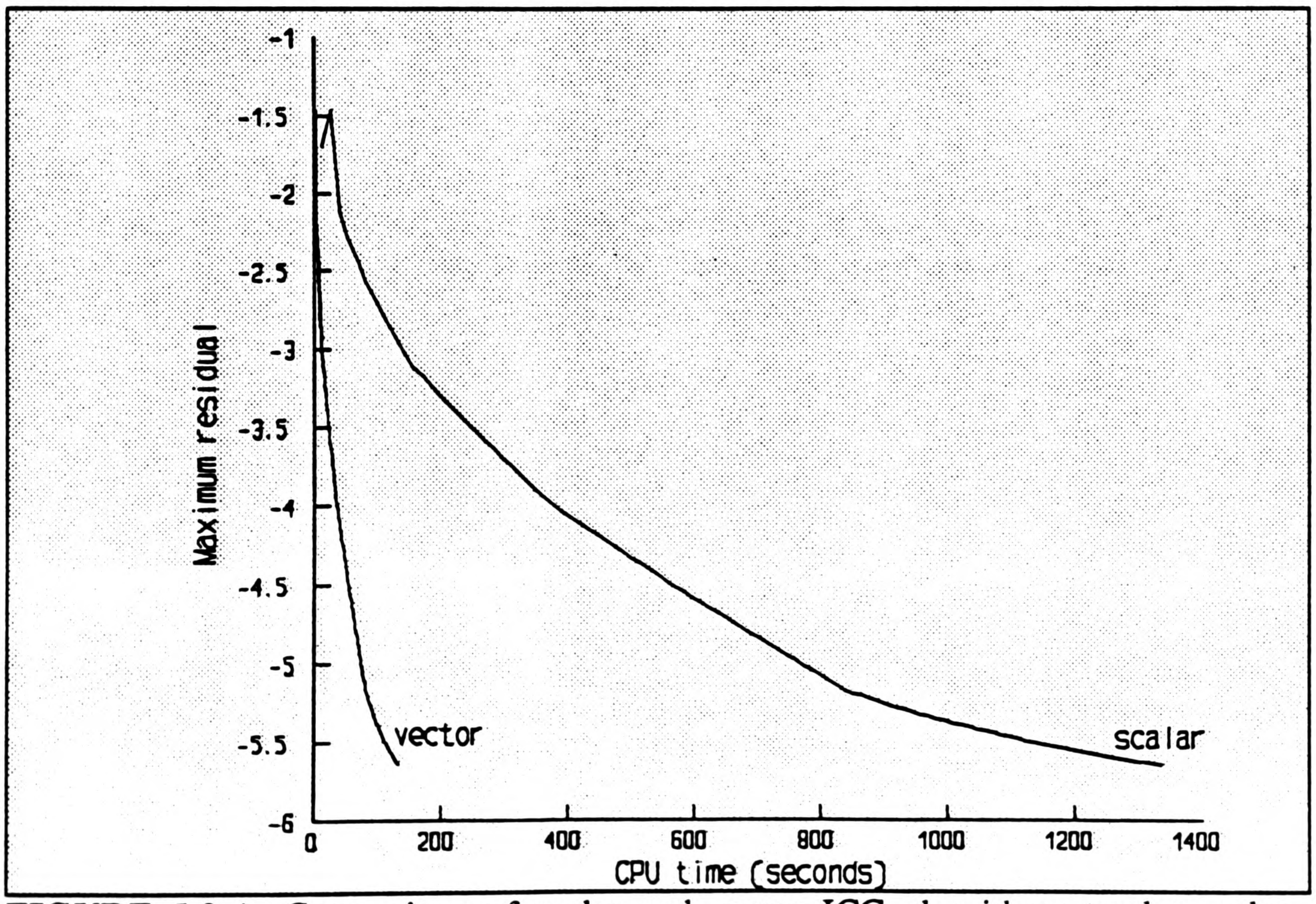


FIGURE 5.8-4 Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the cavity problem (Re=100). JUR used to solve the momentum equations.

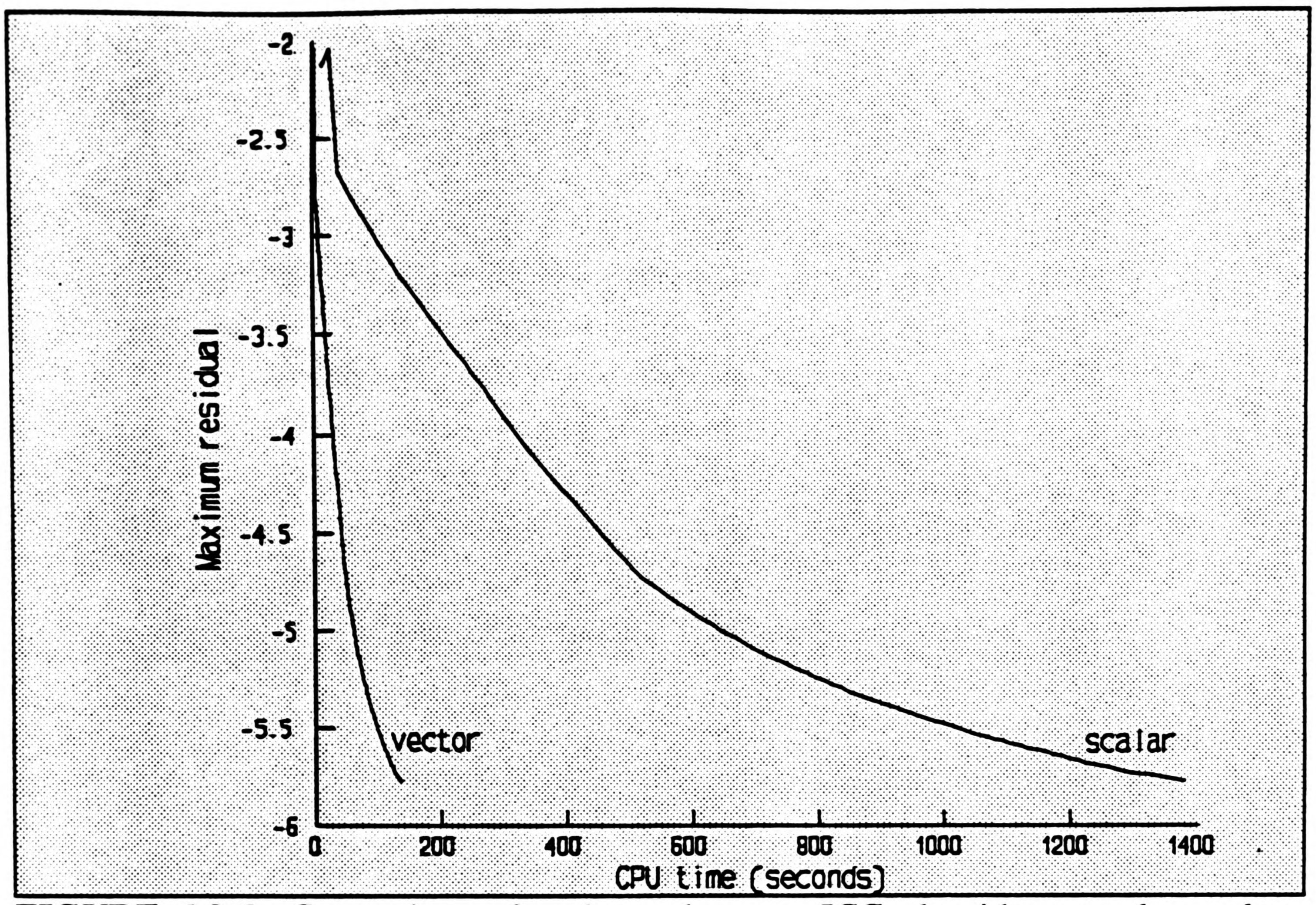


FIGURE 5.8-5 Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the cavity problem (Re=400). JUR used to solve the momentum equations.

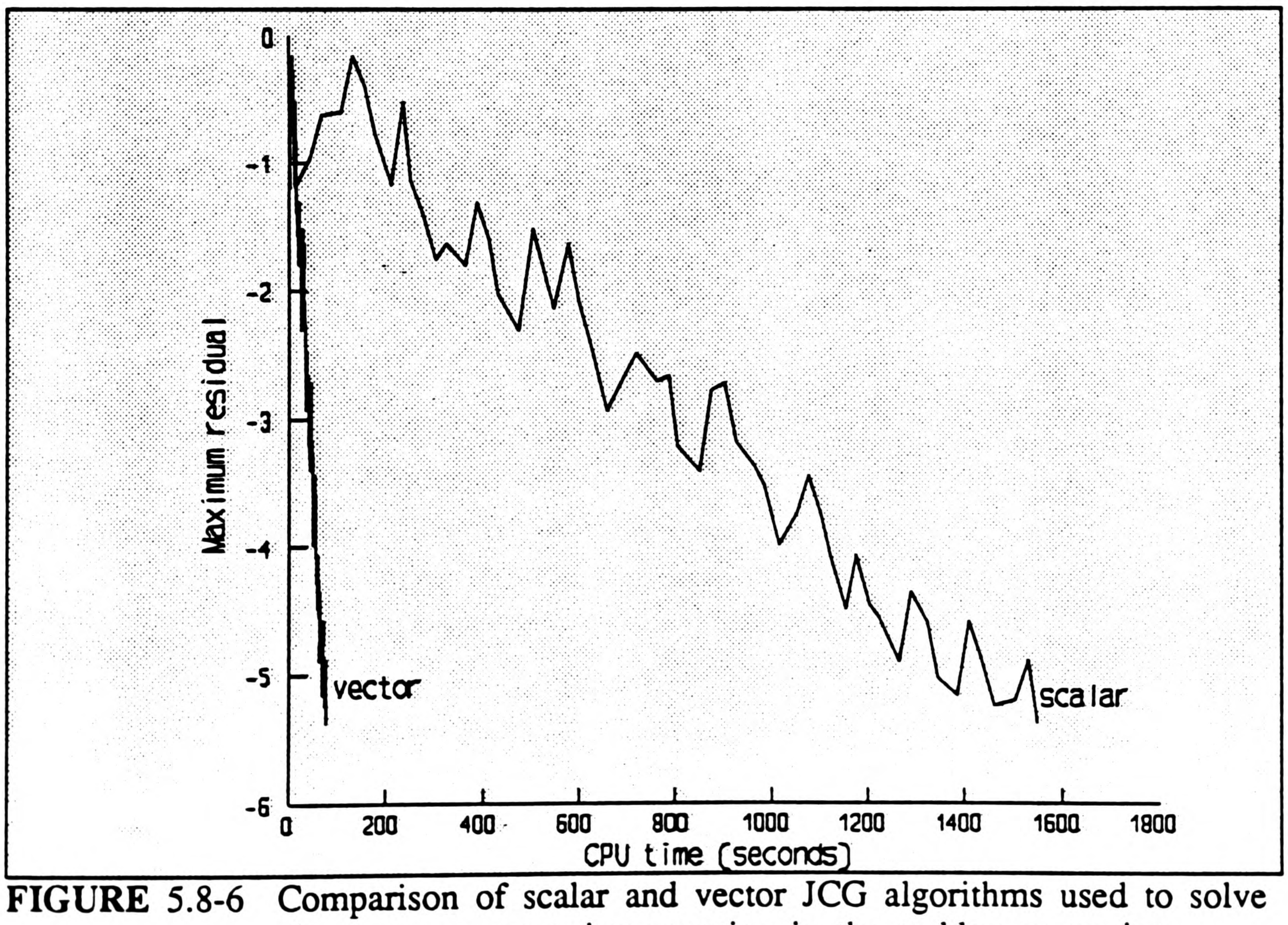


FIGURE 5.8-6 Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the sudden expansion problem. JUR used to solve the momentum equations.

(a)

	JUR	JCG
Set up source terms for u-momentum equation	20.1	22.7
Set up u-momentum equation coefficients	1.0	1.1
Solve for u-momentum values u^*	1.1	1.2
Set up source terms for v-momentum equation	20.1	22.7
Set up v-momentum equation coefficients	1.0	1.1
Solve for v-momentum values v^*	1.1	1.2
Set up pressure-correction coefficients	34.7	39.1
Solve for pressure-correction values p'	20.4	10.9
Correct u^*, v^* and p^* to produce u, v and p	.2	.2

(b)

	JUR	JCG
Set up source terms for u-momentum equation	20.2	22.6
Set up u-momentum equation coefficients	1.0	1.1
Solve for u-momentum values u^*	1.1	1.2
Set up source terms for v-momentum equation	20.2	22.6
Set up v-momentum equation coefficients	1.0	1.1
Solve for v-momentum values v^*	1.1	1.2
Set up pressure-correction coefficients	34.8	39.1
Solve for pressure-correction values p'	20.4	10.9
Correct u^*, v^* and p^* to produce u, v and p	.2	.2

(c)

	JUR	JCG
Set up source terms for u-momentum equation	17.2	19.6
Set up u-momentum equation coefficients	1.0	1.1
Solve for u-momentum values u^*	1.0	1.1
Set up source terms for v-momentum equation	17.3	19.7
Set up v-momentum equation coefficients	1.0	1.1
Solve for v-momentum values v^*	1.0	1.1
Set up pressure-correction coefficients	30.4	34.7
Solve for pressure-correction values p'	30.8	21.3
Correct u^*, v^* and p^* to produce u, v and p	.3	.3

TABLE 5.8-2 Percentage breakdown of the SIMPLE procedure for (a) cavity problem Re=100 (b) cavity problem Re=400 (c) sudden expansion problem. Full vectorisation.

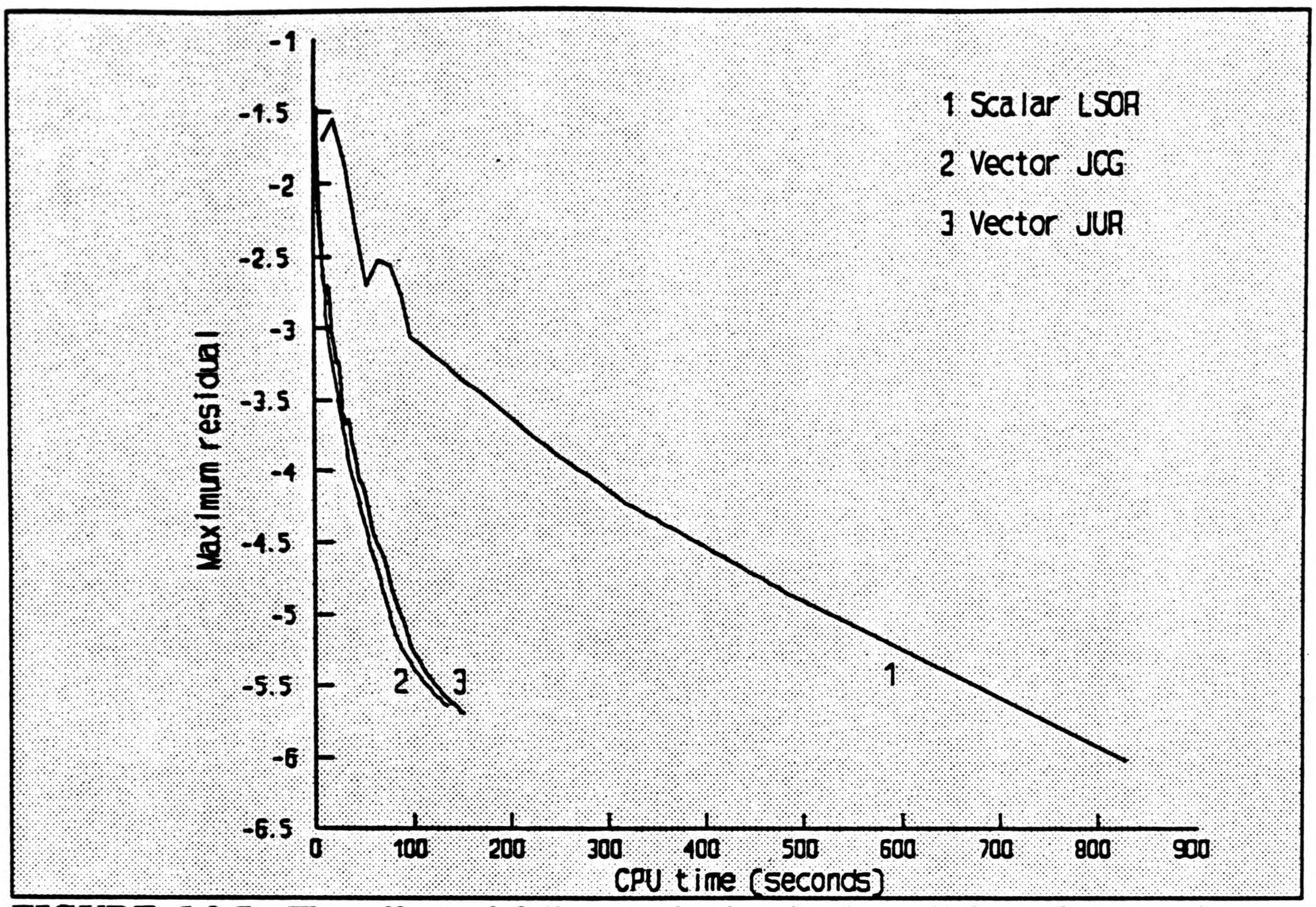


FIGURE 5.8-7 The effect of full vectorisation in the solution of the cavity problem (Re=100). JUR used to solve the momentum equations.

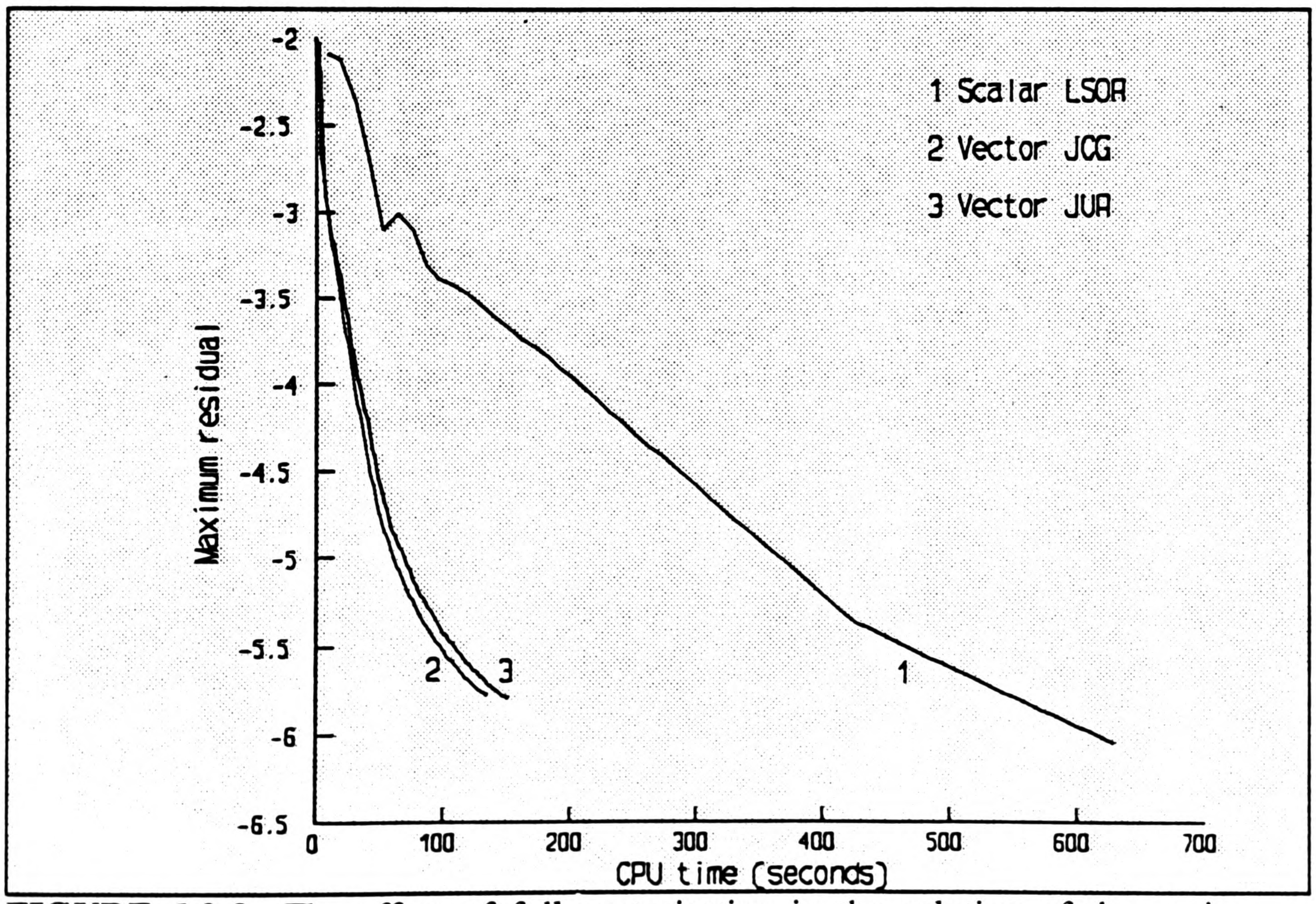


FIGURE 5.8-8 The effect of full vectorisation in the solution of the cavity problem (Re=400). JUR used to solve the momentum equations.

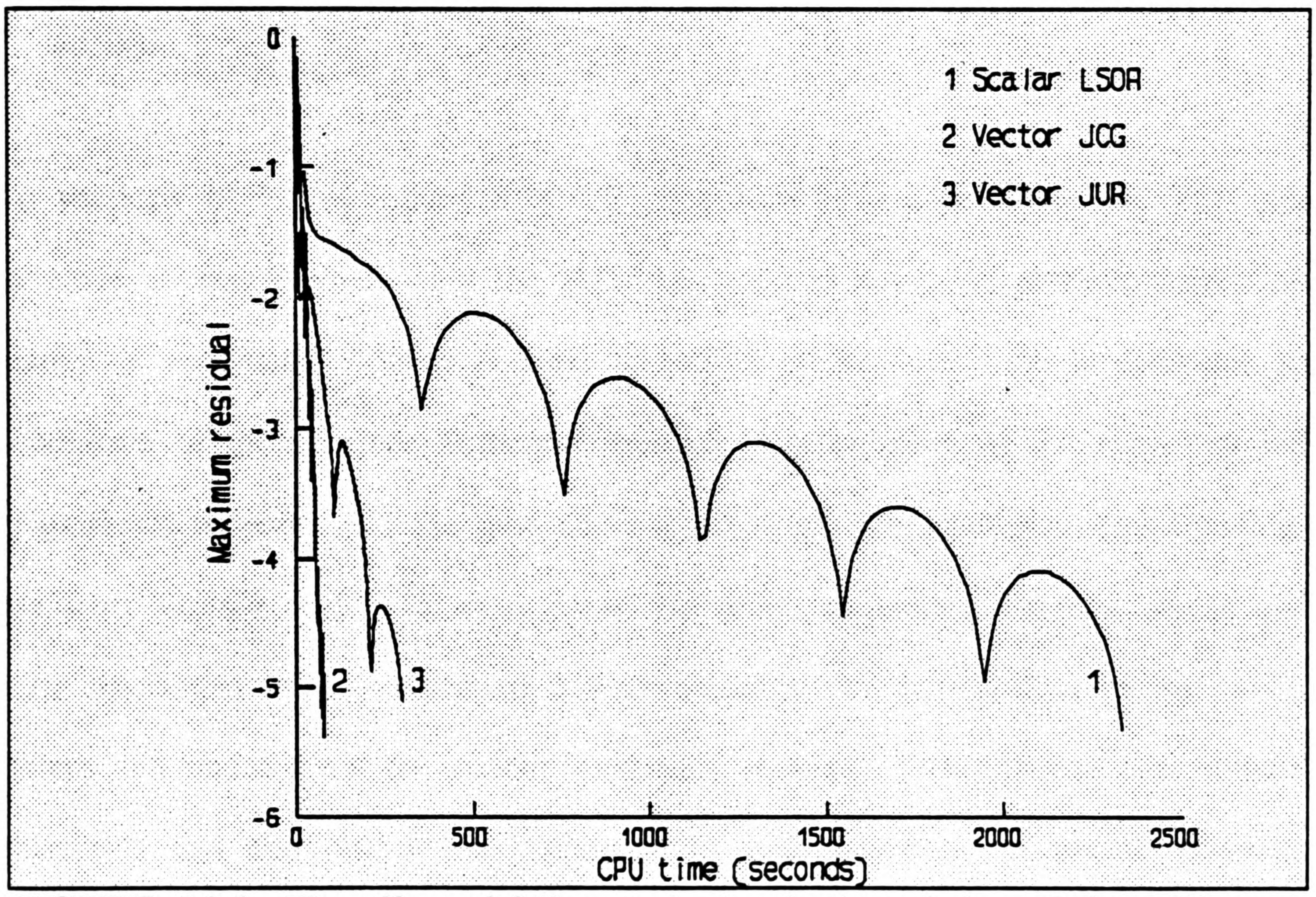


FIGURE 5.8-9 The effect of full vectorisation in the solution of the sudden expansion problem. JUR used to solve the momentum equations.

the governing equations is weak and therefore the solution of the pressure-correction equation is not required to a high degree of convergence, thus the significant improvement in speed results from the vectorisation of the remainder of the solution procedure. In the case of the sudden expansion problem the coupling is much stronger and here the improvements have been further enhanced through the solution of the pressure-correction equation.

5.9 Closure

Two versions of the vectorised SIMPLE procedure have been presented. The first is a naive approach where only the algorithm used to solve the pressure-correction equation is vectorised. This is because 65%-90% of the total time is dedicated to the solution of the pressure-correction equation. Vectorisation of the algorithm led to significant reductions in the time taken to solve the pressure-correction equation, these were as low as 5% in some cases. However, comparing the best scalar and vector algorithms improvements of 2 were achieved for the solution of the cavity problem and under 6 for the sudden expansion problem. These factors may appear surprisingly low considering the proportion of code which is vectorised, however an analysis using Amdahl's law reveals that such speed-up factors are to be expected.

In the second approach as much of the SIMPLE procedure as possible is vectorised. Full vectorisation is prohibited because features such as boundary conditions, fluid properties and source terms do not allow for efficient vectorisation on the VA-1 processor. As a result they are computed on the scalar processor. Also, to allow the solution of the momentum equations to be carried out using the pipeline processor the JUR algorithm is used instead of the LJUR algorithm. Comparisons between the best scalar and vector algorithms show that factors of up to 6 are achieved for the solution of the cavity problem and up to 29 for the sudden expansion problem. The successful implementation of 95%-98% of the SIMPLE procedure is the reason for the high speed-up factors achieved.

Although the JCG is seen to be the most efficient algorithm, it is also found to be unreliable and lacks robustness. The tendency for the algorithm to become inefficient or to diverge means that relaxation is necessary, and this needs to be taken into consideration when choosing a best algorithm.

CHAPTER SIX

6.0 ADDITION OF SCALAR QUANTITIES

6.1 Introduction

The purpose of this chapter is to determine what effect the introduction of scalar quantities have on the vectorised SIMPLE procedure. So far only laminar flow problems involving the solution of the velocity and pressure fields have been tackled. Consideration is now given to the solution of scalar variables typified by enthalpy (or temperature) and turbulence. It is not surprising that the introduction of scalar equations will cause a significant increase in the computation time, although this will depend largely on the strength of coupling between the equations. Generally, strong coupling can make convergence difficult and cause a significant increase in the computation time.

In this chapter two test problems are studied, both of which involve the scalar equations for enthalpy and the time-averaged form of turbulence. The first problem is a study of L-shaped turbulent flow in a duct. The second involves the natural convection of air in an enclosed cavity, simulations are performed for Rayleigh numbers varying from 10^3-10^6 (laminar) and 10^7 (turbulent).

6.2 The scalar equations

The turbulence model used here has been described by many authors, see for example Launder and Spalding [1974] for details. The model is based on turbulent kinetic energy (k) and its rate of dissipation (ε) and is called the k-ε model. Assuming a two-dimensional cartesian framework, the equation for kinetic energy is given by

$$\frac{\partial}{\partial x}(\rho u k) + \frac{\partial}{\partial y}(\rho v k) = \frac{\partial}{\partial x}\left(\frac{\mu_{eff}}{\sigma_k}\frac{\partial k}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{\mu_{eff}}{\sigma_k}\frac{\partial k}{\partial y}\right) + S_k \qquad (6.2\text{-}1a)$$

$$S_k = G_k + G_B - \rho\varepsilon \qquad (6.2\text{-}1b)$$

and the equation for the dissipation rate is

$$\frac{\partial}{\partial x}(\rho u \varepsilon) + \frac{\partial}{\partial y}(\rho v \varepsilon) = \frac{\partial}{\partial x}\left(\frac{\mu_{eff}}{\sigma_\varepsilon}\frac{\partial \varepsilon}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{\mu_{eff}}{\sigma_\varepsilon}\frac{\partial \varepsilon}{\partial y}\right) + S_\varepsilon \qquad (6.2\text{-}2a)$$

$$S_\varepsilon = \frac{\varepsilon}{k}\left(c_1 G_k - c_2\rho\varepsilon + c_3 G_B\right) \qquad (6.2\text{-}2b)$$

G_k is the shear production term and is defined by

$$G_k = \mu_t\left[2\left\{\left[\frac{\partial u}{\partial x}\right]^2 + \left[\frac{\partial v}{\partial y}\right]^2\right\} + \left[\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right]^2\right] \qquad (6.2\text{-}3)$$

and G_B is the buoyancy production term

$$G_B = -\beta g \frac{\mu_t}{\sigma_t}\left[\frac{\partial T}{\partial y}\right] \qquad (6.2\text{-}4)$$

In this implementation of the k-ε model the empirical constants are set up as c_1=1.44, c_2=1.92 and c_3=1.0, if buoyancy is not included then c_3=0.0. The effective viscosity μ_{eff} is given as a function of the laminar and turbulent viscosities μ and μ_t respectively, thus

$$\mu_{eff} = \mu + \mu_t \qquad (6.2\text{-}5a)$$

$$\mu_t = \frac{C_D \rho k^2}{\varepsilon} \qquad (6.2\text{-}5b)$$

The constant C_D is set to 0.09.

The temperature equation is given by

$$\frac{\partial(\rho uT)}{\partial x} + \frac{\partial(\rho vT)}{\partial y} = \frac{\partial}{\partial x}\left(\left[\frac{\mu}{\sigma} + \frac{\mu_t}{\sigma_t}\right]\frac{\partial T}{\partial x}\right) + \frac{\partial}{\partial y}\left(\left[\frac{\mu}{\sigma} + \frac{\mu_t}{\sigma_t}\right]\frac{\partial T}{\partial y}\right) + S_T \qquad (6.2\text{-}6)$$

In laminar flows the no-slip condition is assumed at a wall and the specification of boundary conditions for scalar quantities is straightforward. In turbulent flows the changes to the unknown quantities can be dramatic near a wall. To describe this change, wall functions assuming a logarithmic profile are used in the near-wall region.

6.3 PROBLEM 3: L-shaped turbulent flow problem

The fluid enters through an opening at one end of a square duct and exits at the other end, the restricted outlet is perpendicular to the inlet (figure 6.3-1). The walls are heated and the solution for the steady state turbulent flow is simulated.

6.3.1 Physical and geometrical specification

The inlet u-velocity has a plug-flow profile with a velocity of $50ms^{-1}$ and an inlet Reynolds number of 10^5. At the outlet a fixed pressure of zero is defined and all walls are at a constant temperature of 500K. The initial velocity and pressure fields are $u=50ms^{-1}$, $v=0ms^{-1}$ and $p=0Nm^{-2}$, respectively. The initial internal temperature was 500K. The domain is discretised using a non-uniform 64x16 grid and has a maximum cell aspect ratio of 10.

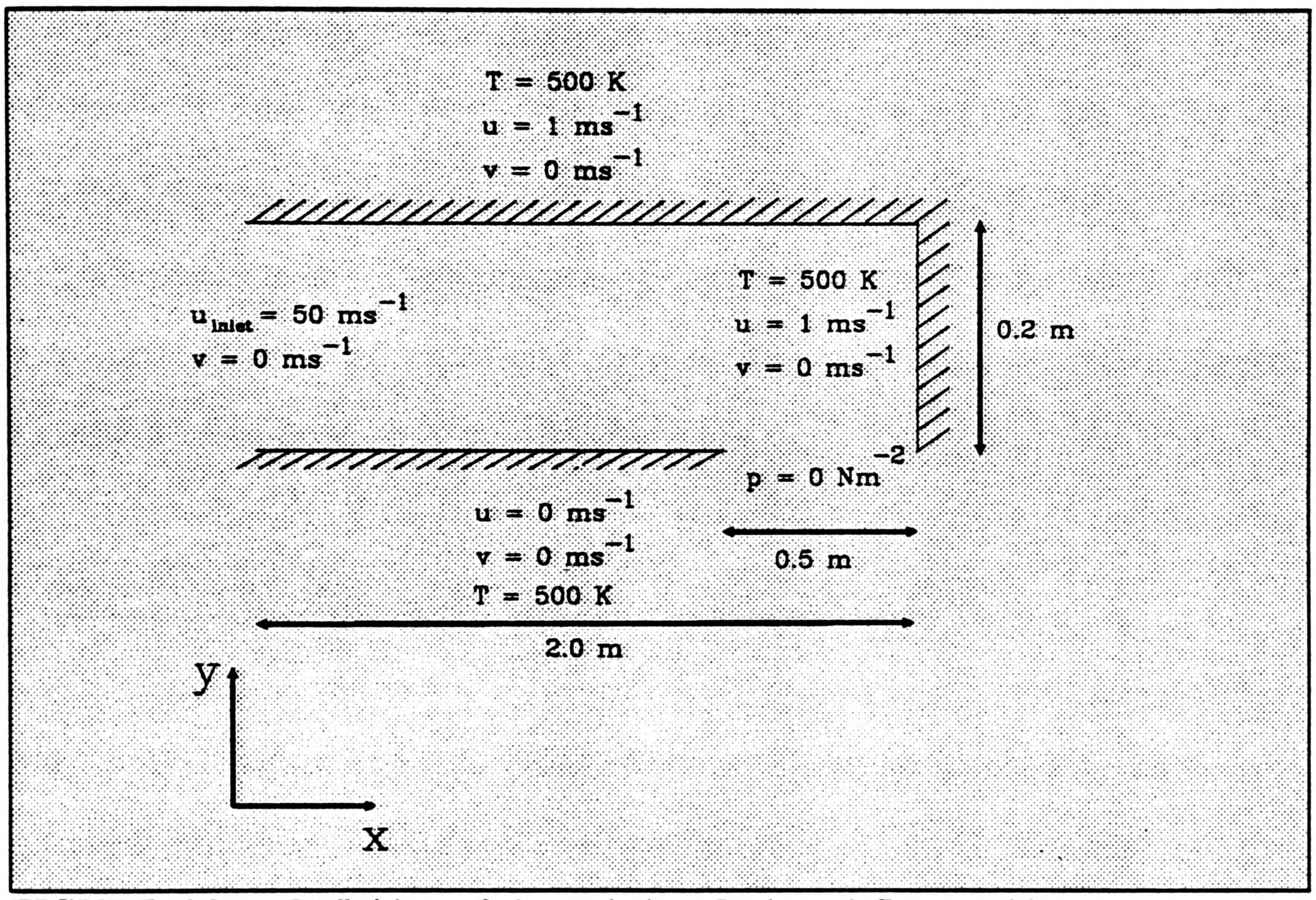


FIGURE 6.3-1 Definition of the turbulent L-shaped flow problem

6.3.2 Results using the scalar algorithms

The velocity vector plot is shown in figure 6.3.2-1a. The developed turbulent profile can be clearly seen together with the recirculation region in the top-right corner of the domain. Contour plots of velocity, pressure and temperature are shown in figures 6.3.2-1b to 6.3.2-1e.

A plot of the maximum residual with CPU time provides a useful insight into the performance of some algorithms. This is shown in figure 6.3.2-2 and shows that the LSOR algorithm is the most efficient of the scalar algorithms considered. Taking a suitable level of convergence, for example, when the maximum residual is less than 1.75×10^{-5}, the LSOR algorithm is 40% faster than the JCG algorithm and nearly 3 times faster than the JUR algorithm.

6.4 PROBLEM 4: Natural convection in a square cavity problem

In this final problem, the convective flow is buoyancy-driven and there is a temperature difference between the two vertical walls as one is cold and the other hot. This test case was proposed by Jones [1979] as a suitable test case for validating computer codes, as well as being of practical interest. This problem is often referred to as the 'double glazing' problem (Jones [1979], de Vahl Davis and Jones [1983] and de Vahl Davis [1983]) and is solved here as a two dimensional steady state problem.

The differential equations make use of the Boussinesq approximation for steady state flows, this assumes that the density variations are negligible except in the

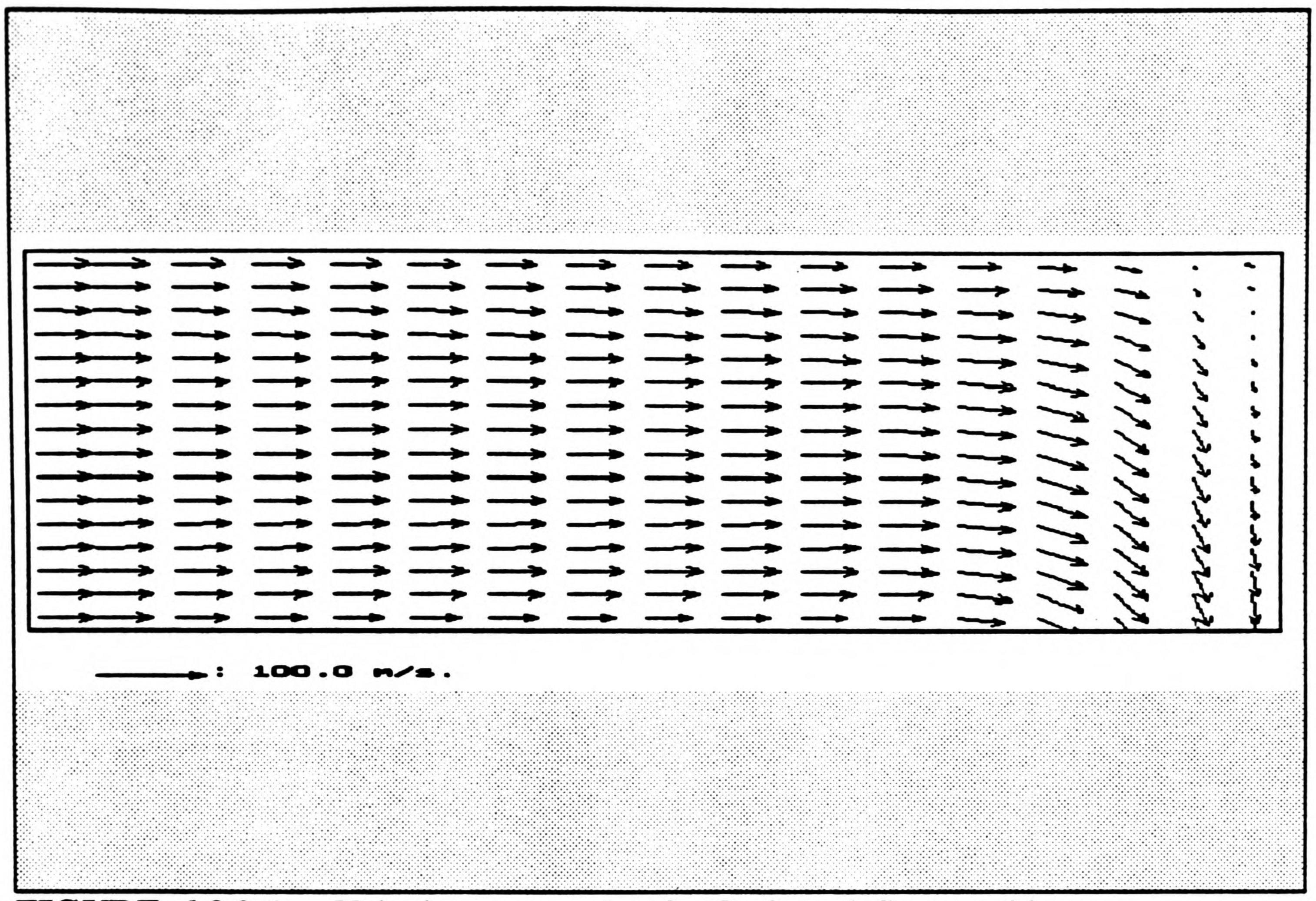


FIGURE 6.3.2-1a Velocity vector plot for L-shaped flow problem

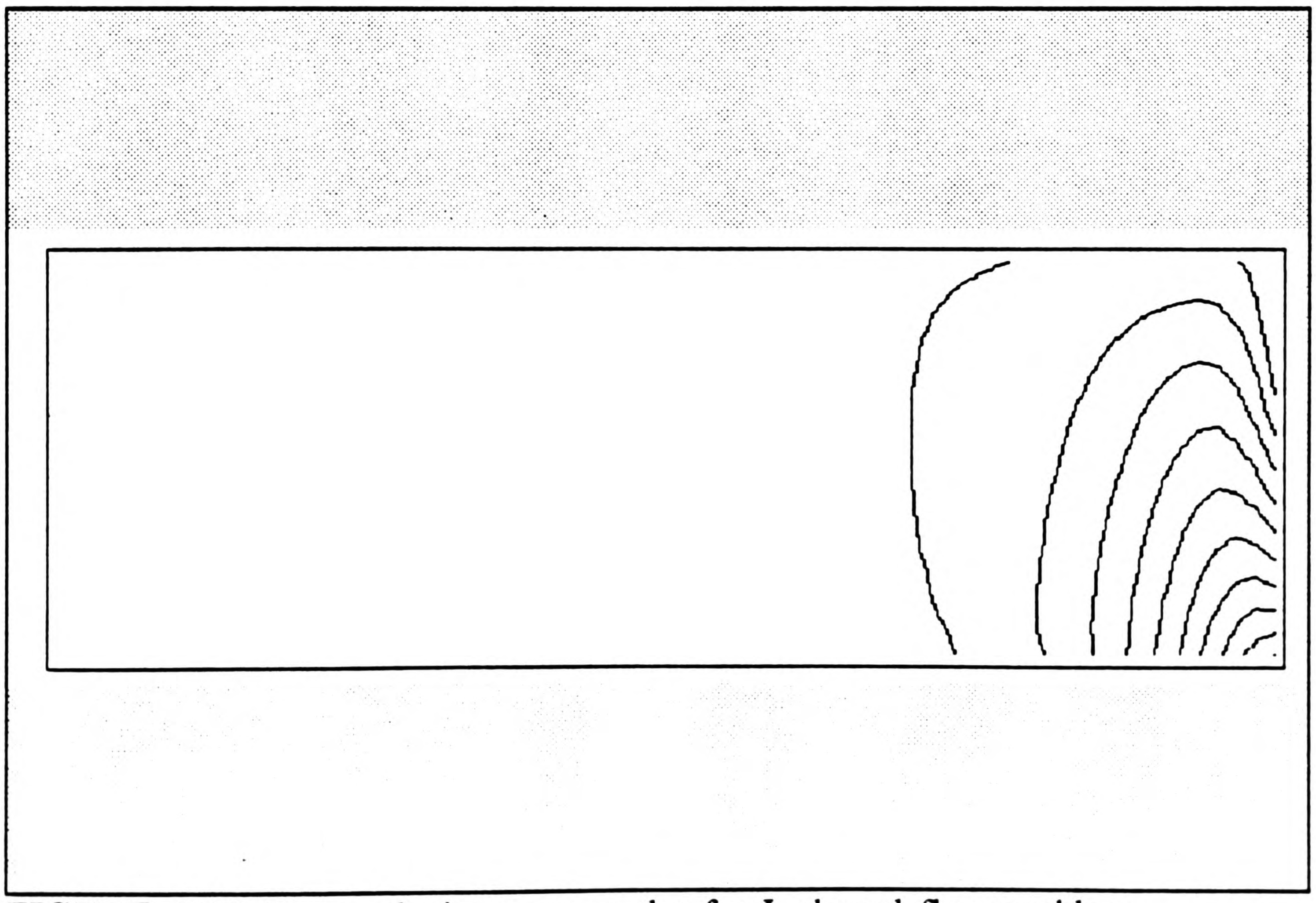


FIGURE 6.3.2-1b u-velocity contour plot for L-shaped flow problem. Contours at -6.38 (5.928) 52.9

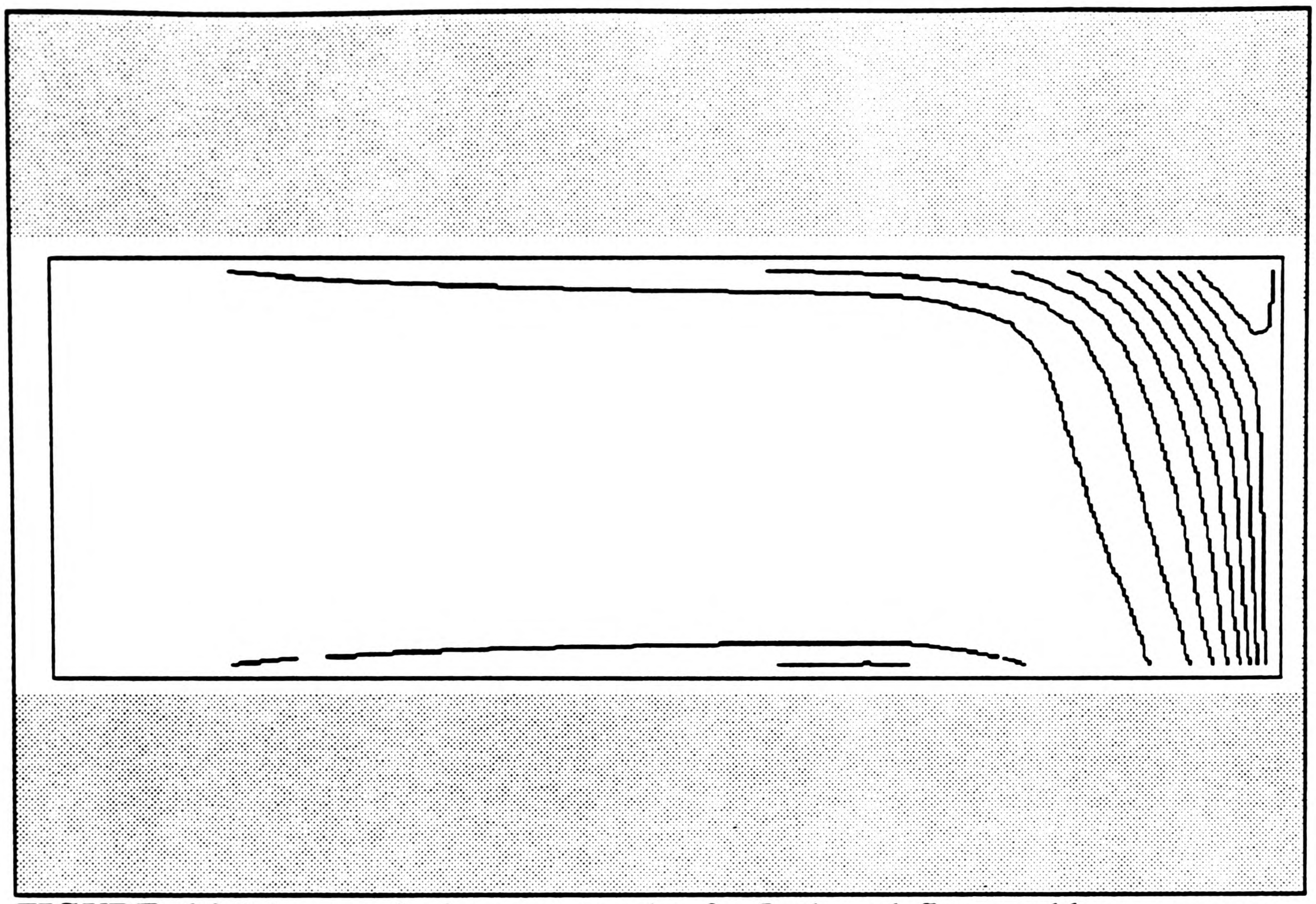


FIGURE 6.3.2-1c v-velocity contour plot for L-shaped flow problem.
Contours at -4.61 (5.101) 46.4

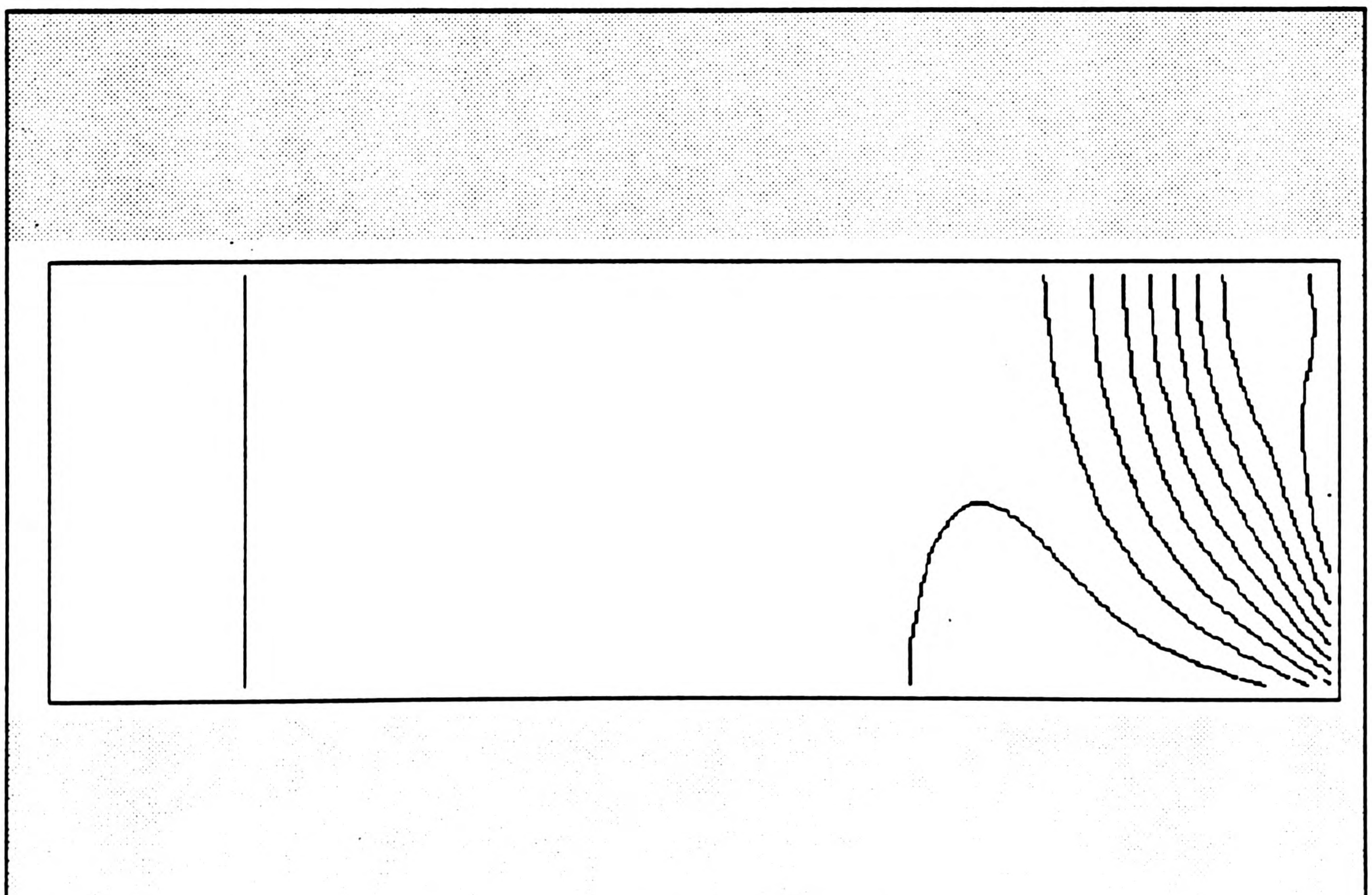


FIGURE 6.3.2-1d Pressure contour plot L-shaped flow problem.
Contours at 6.68 (98.532) 992

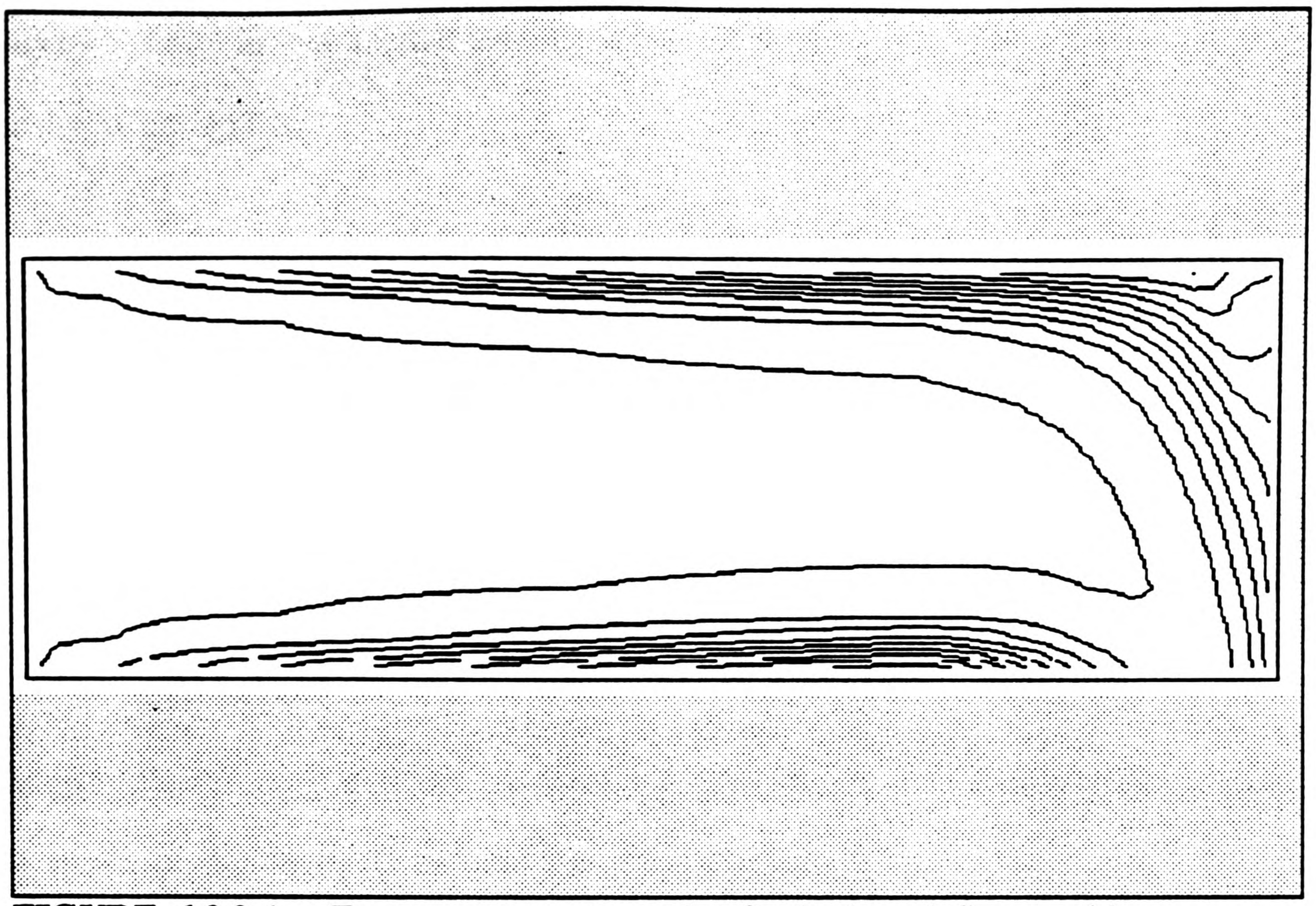


FIGURE 6.3.2-1e Temperature contour plot for L-shaped flow problem. Contours at 358 (14.2) 500

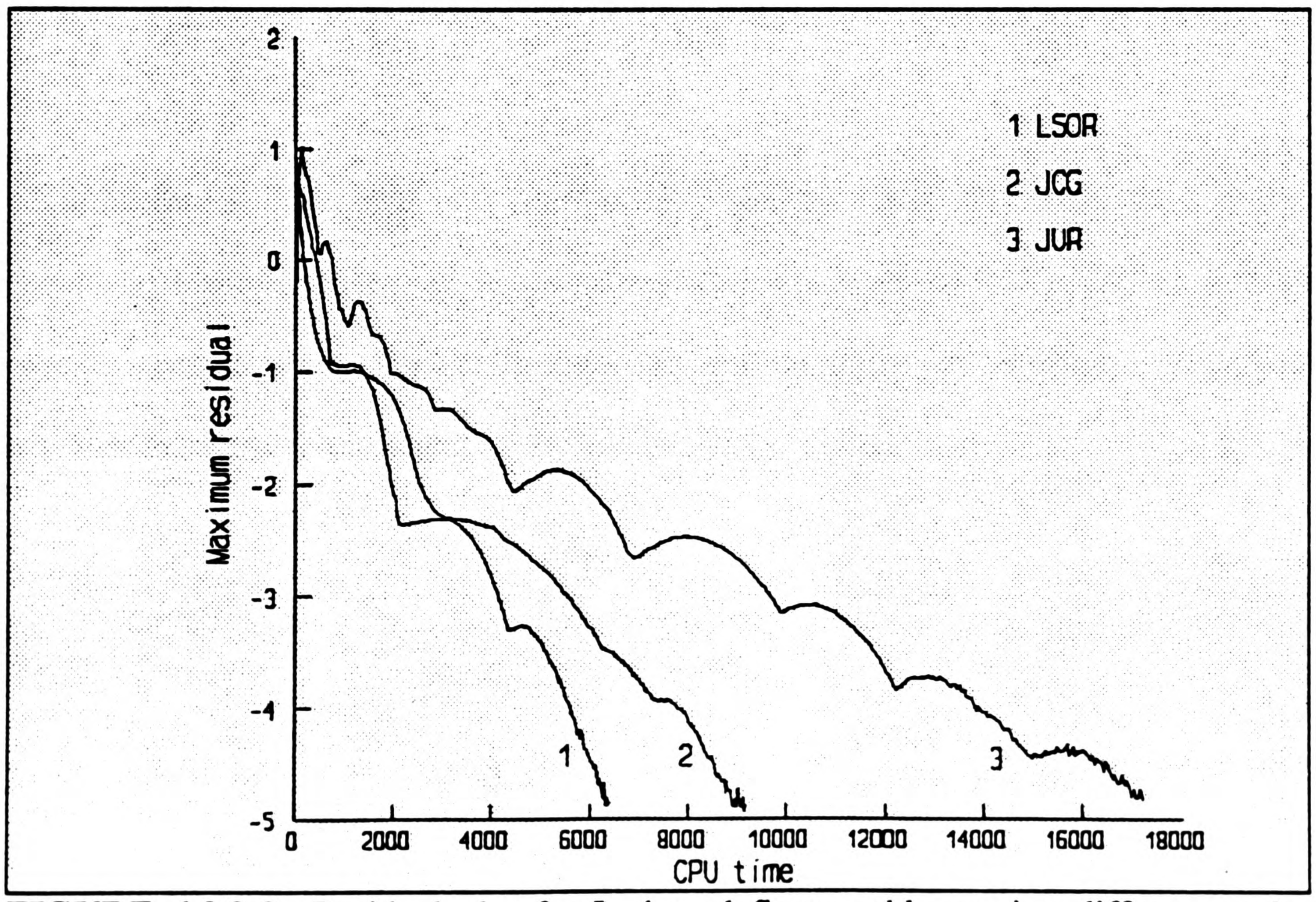


FIGURE 6.3.2-2 Residual plot for L-shaped flow problem using different scalar algorithms to solve the pressure-correction equation buoyancy term in the momentum equations. Therefore in the buoyancy term the density is defined by

$$\rho = \rho_0 \beta \Delta T \tag{6.4-1}$$

and elsewhere the density is constant, i.e

$$\rho = \rho_0 \tag{6.4-2}$$

6.4.1 Physical and geometrical specification

The coordinates x, y and the velocity components are non-dimensionalised using the length of the cavity D and the thermal diffusivity *k*, thus

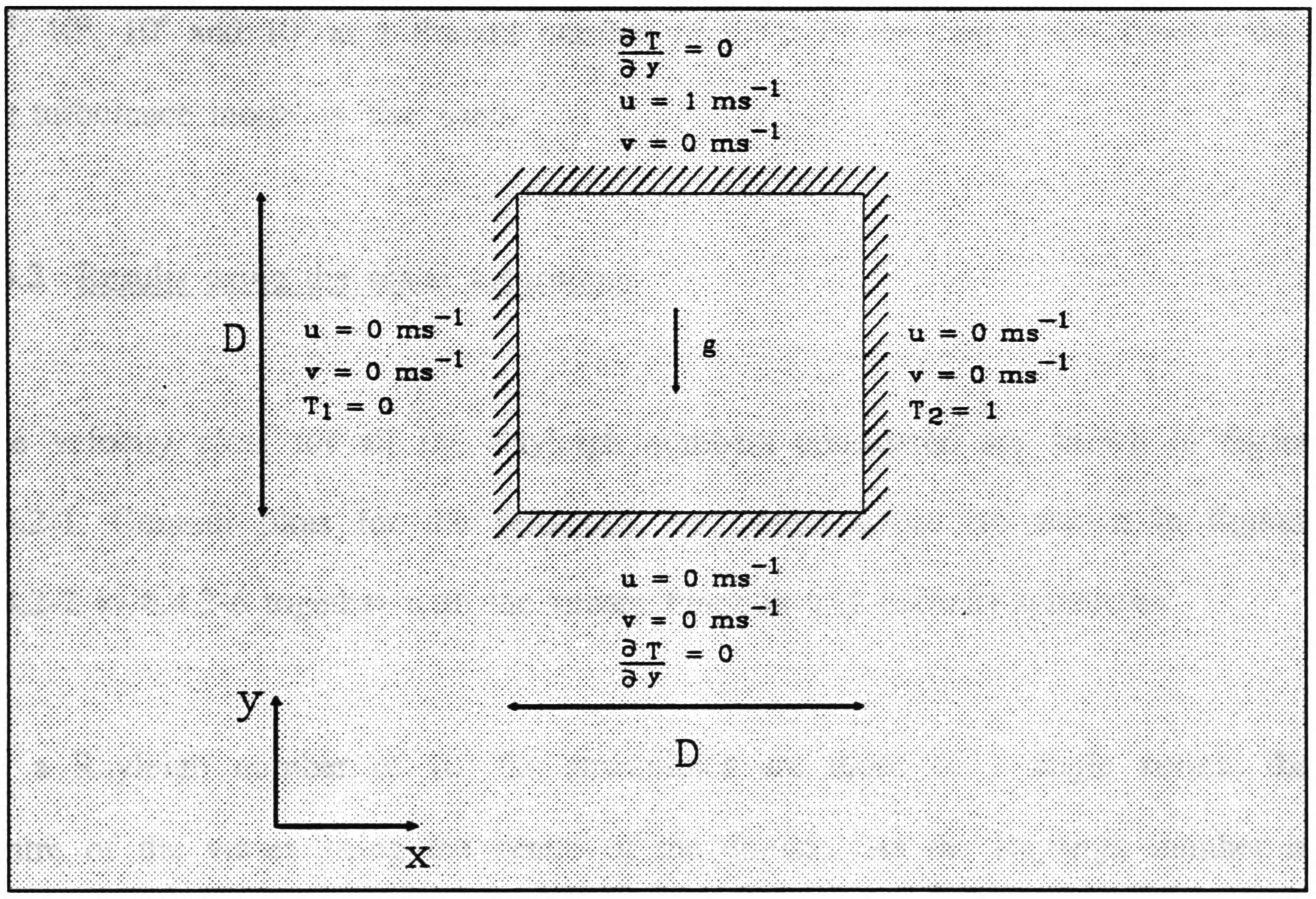
$$\begin{aligned} x &= x/D \\ y &= y/D \\ u &= uD/k \\ v &= vD/k \end{aligned} \tag{6.4.1-1}$$

The temperature difference between the two vertical walls ΔT is set to 1 (figure 6.4.1-1), at x=0 the cold wall is set to a scaled temperature T_1=0.0 and at x=1 the hot wall temperature is T_2=1.0. The top and bottom walls are insulated and defined by a zero temperature gradient

$$\left.\frac{\partial T}{\partial y}\right|_{y=0,1} = 0 \tag{6.4.1-2}$$

The dimension (D) of the cavity is also used to modify the Rayleigh number (Ra), this is defined by

$$Ra = \beta g \Delta T D^3 / k\nu \tag{6.4.1-3a}$$


$\frac{\partial T}{\partial y} = 0$
$u = 1\ ms^{-1}$
$v = 0\ ms^{-1}$
D
$u = 0\ ms^{-1}$
$v = 0\ ms^{-1}$
$T_1 = 0$
g
$u = 0\ ms^{-1}$
$v = 0\ ms^{-1}$
$T_2 = 1$
$u = 0\ ms^{-1}$
$v = 0\ ms^{-1}$
$\frac{\partial T}{\partial y} = 0$
D
y
x

where ν is the kinematic viscosity

$$\nu = \mu/\rho \qquad (6.4.1\text{-}3b)$$

The initial velocity and pressure fields are $u=0.0ms^{-1}$, $v=0.0ms^{-1}$ and $p=0Nm^{-2}$, respectively and the initial temperature field was set to T=0.5. A non-uniform 32x32 grid was used, and solutions were determined for Rayleigh numbers of 10^3, 10^4, 10^5, 10^6 and 10^7 at a Prandtl number of 0.71. In the case when $Ra=10^7$ the k-ε turbulence model is also used.

6.4.2 Results using the scalar algorithms

The velocity plots for all the Rayleigh numbers considered are shown in figure 6.4.2-1. Contour plots for the velocities, and temperatures are shown in figures 6.4.2-2 to 6.4.2-6 together with the stream function and vorticity quantities.

At a Rayleigh number of 10^3 the streamlines are those of a single vortex, the centre of the vortex being the centre of the domain. As the Rayleigh number is increased the centre streamline becomes elliptic ($Ra=10^4$), extending to produce two secondary vortices inside it ($Ra=10^5$). These vortices tend towards the direction of the flow, moving towards the walls. The isotherms indicate that for low Rayleigh numbers, most of the heat transfer is through heat conduction, the effect of convection is seen as the deviation of isotherms from the vertical. For high Rayleigh numbers the heat transfer is mainly through convection in what have now become the thin boundary layers.

The results for Rayleigh numbers $10^3 \leq Ra \leq 10^6$ are compared with the benchmark solution (de Vahl Davis [1983]). The qualitative agreement between the present results and the benchmark solutions are good for low Rayleigh numbers, despite the fact that finer grids were used in the benchmark solution. For high Rayleigh numbers there are differences, this is because the benchmark used a finer grid (81x81) and was more accurate in defining the thin boundary layers. Table 6.4.2-1 shows the maximum and minimum local Nusselt numbers on the cold wall and their location; The maximum u-velocity and its location on the vertical mid-plane and the maximum v-velocity and its location on the horizontal mid-plane.

An investigation into the performance of various algorithms reveals that the most efficient is the LSOR algorithm (figures 6.4.2-7 to 6.4.2-11). This is up to 75% faster than the JCG and a factor of about 3.5 faster than the JUR algorithm. The results were obtained for a convergence level where the maximum residual was less than 2.5×10^{-5}.

6.5 Distribution of computation effort in the SIMPLE procedure

A breakdown of the SIMPLE procedure is carried out to determine the expected reductions in computation time when the pipeline processor is used. A detailed breakdown of the procedure for both the JUR and JCG algorithms reveals two main trends (Table 6.5-1 and Table 6.5-2). Firstly, the time taken to solve the pressure-correction equation, although still a major contribution to the total time, is now lower than in the laminar problems studied in Chapter 5. Secondly, the proportion of computation essentially scalar (such as the generation of source terms) has increased because more unknowns are being solved. For example, in the

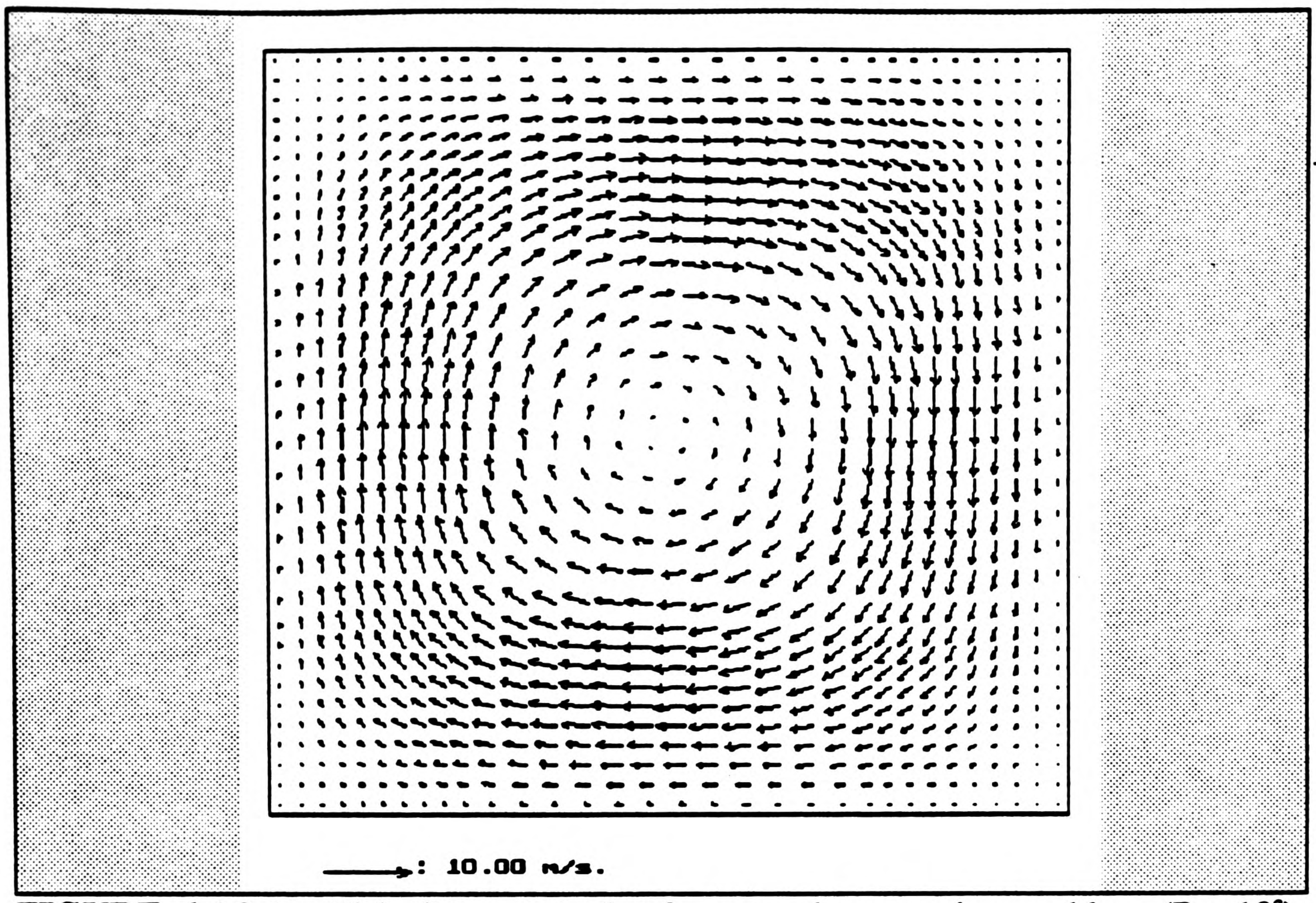


FIGURE 6.4.2-1a Velocity vector plot for natural convection problem (Ra=10^3)

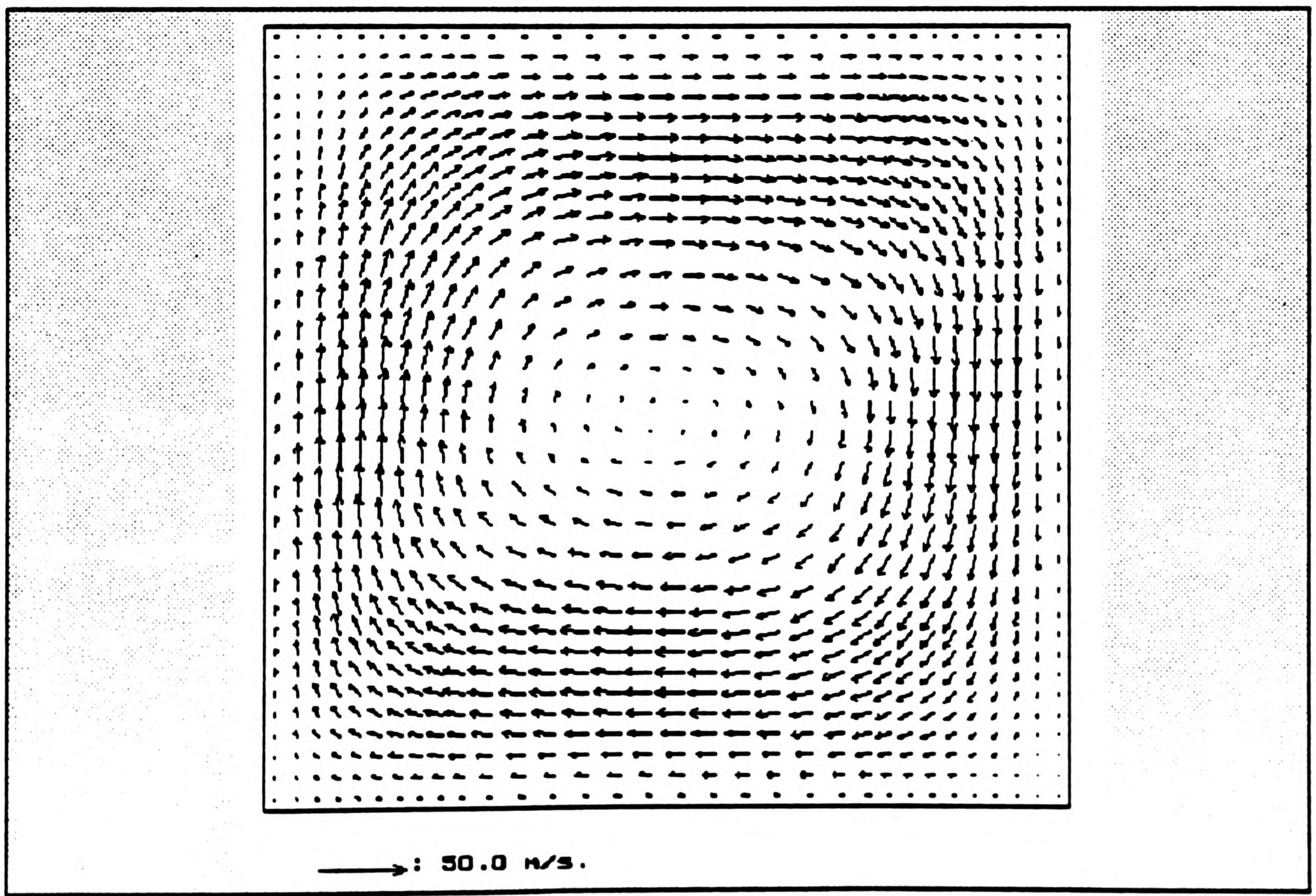


FIGURE 6.4.2-1b Velocity vector plot for natural convection problem (Ra=10^4)

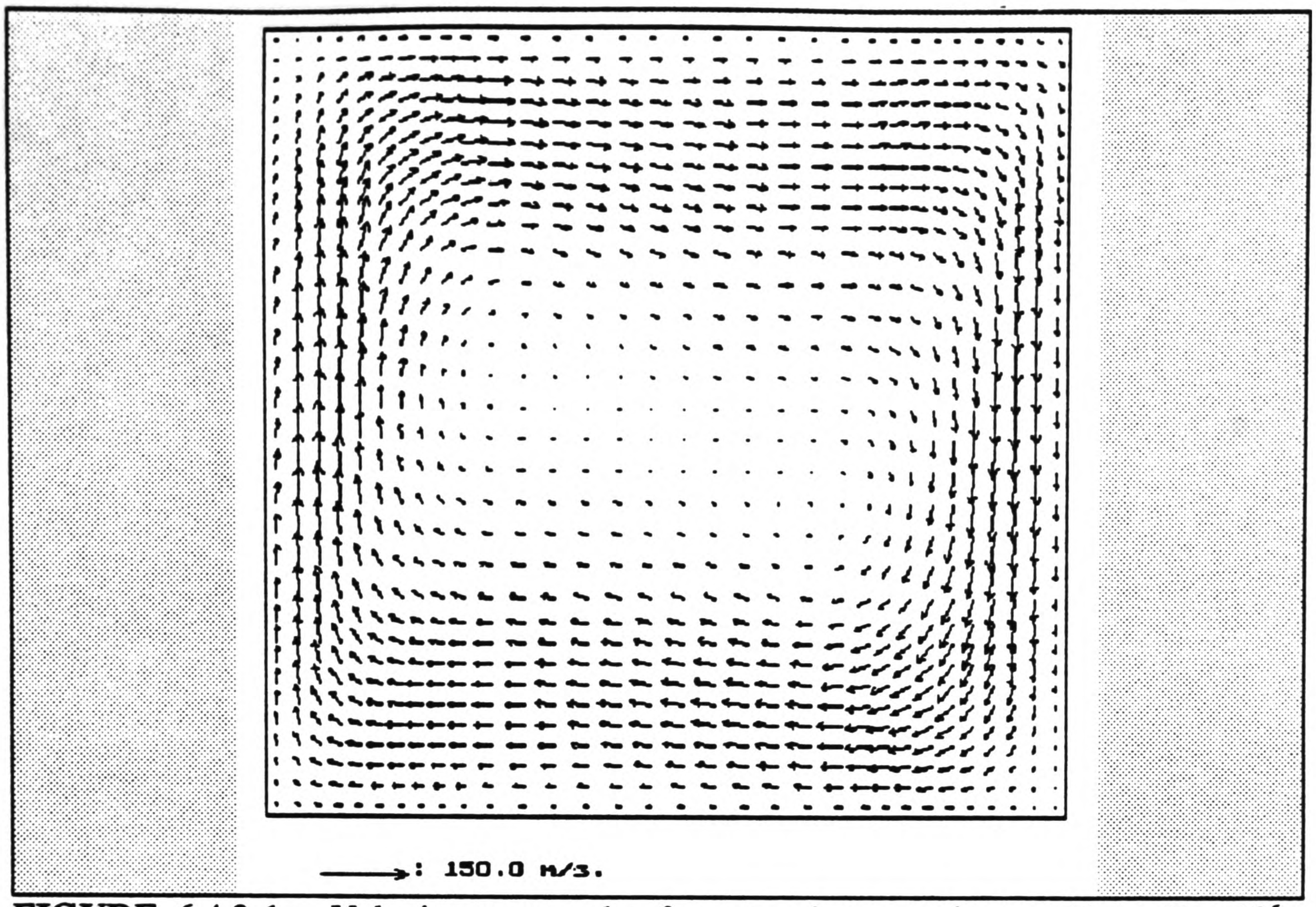


FIGURE 6.4.2-1c Velocity vector plot for natural convection problem (Ra=10^5)

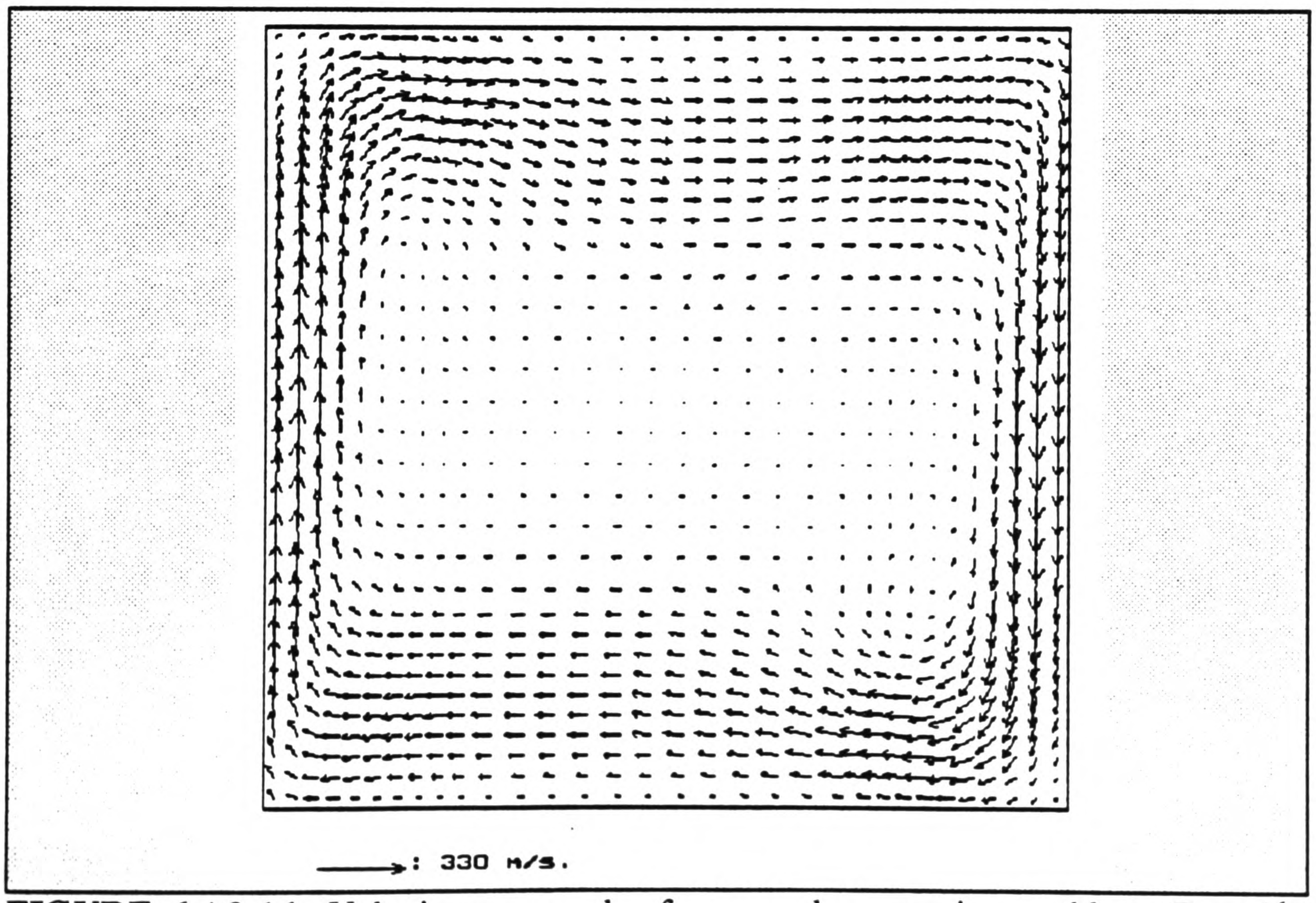


FIGURE 6.4.2-1d Velocity vector plot for natural convection problem (Ra=10^6)

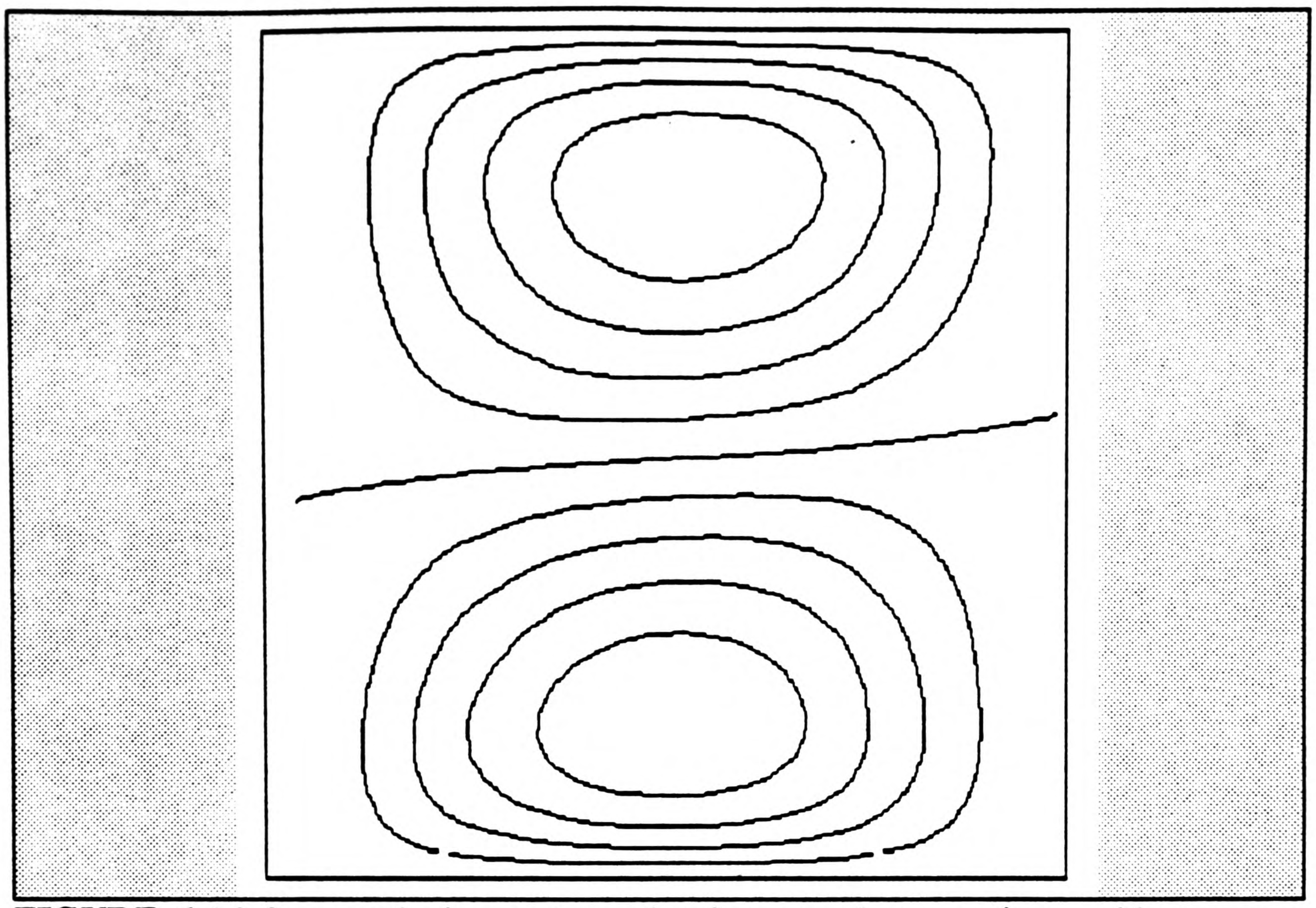


FIGURE 6.4.2-2a u-velocity contour plot for natural convection problem ($Ra=10^3$). Contours at -3.64 (0.728) 3.64

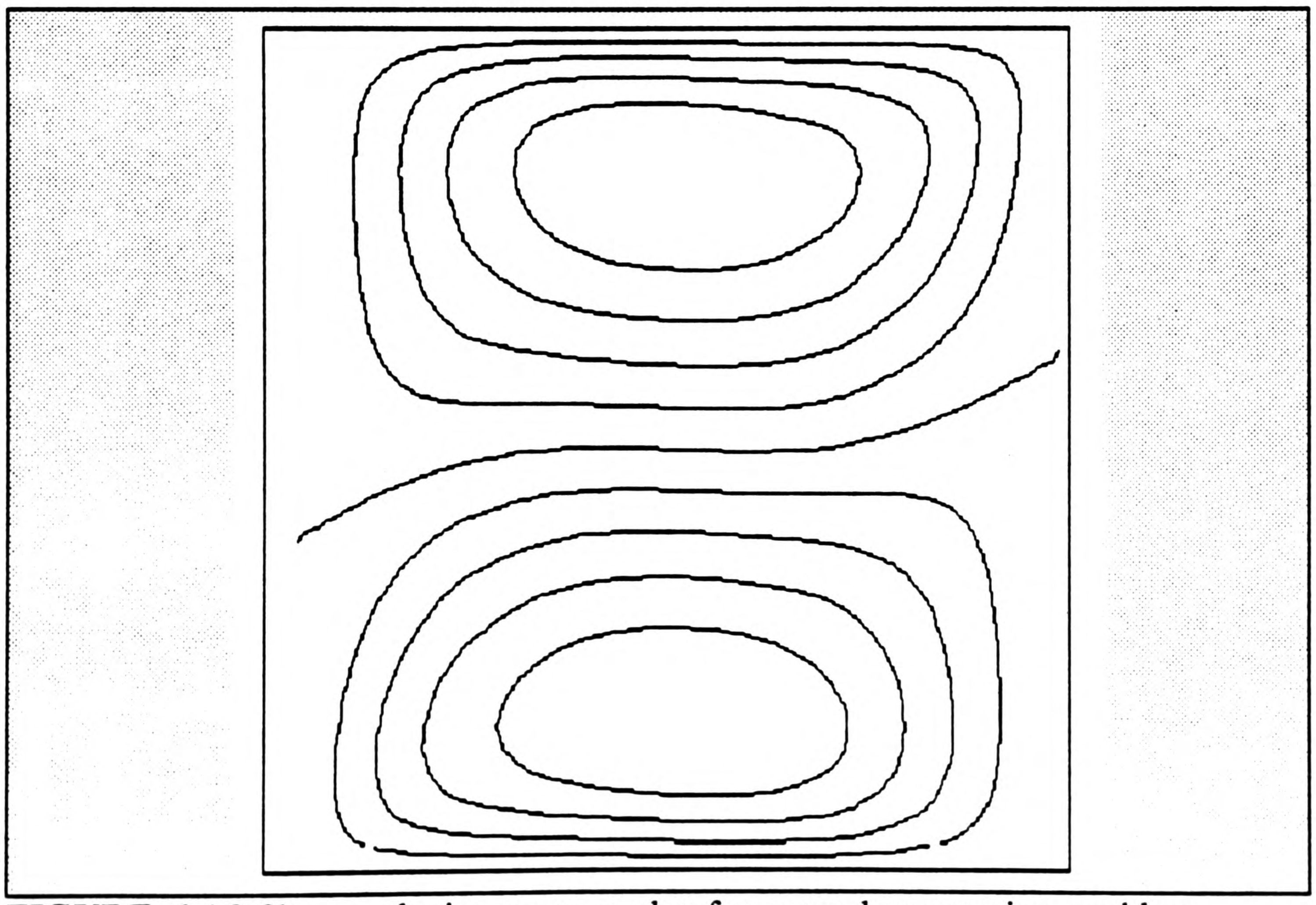


FIGURE 6.4.2-2b u-velocity contour plot for natural convection problem ($Ra=10^4$). Contours at -16.05 (3.21) 16.05

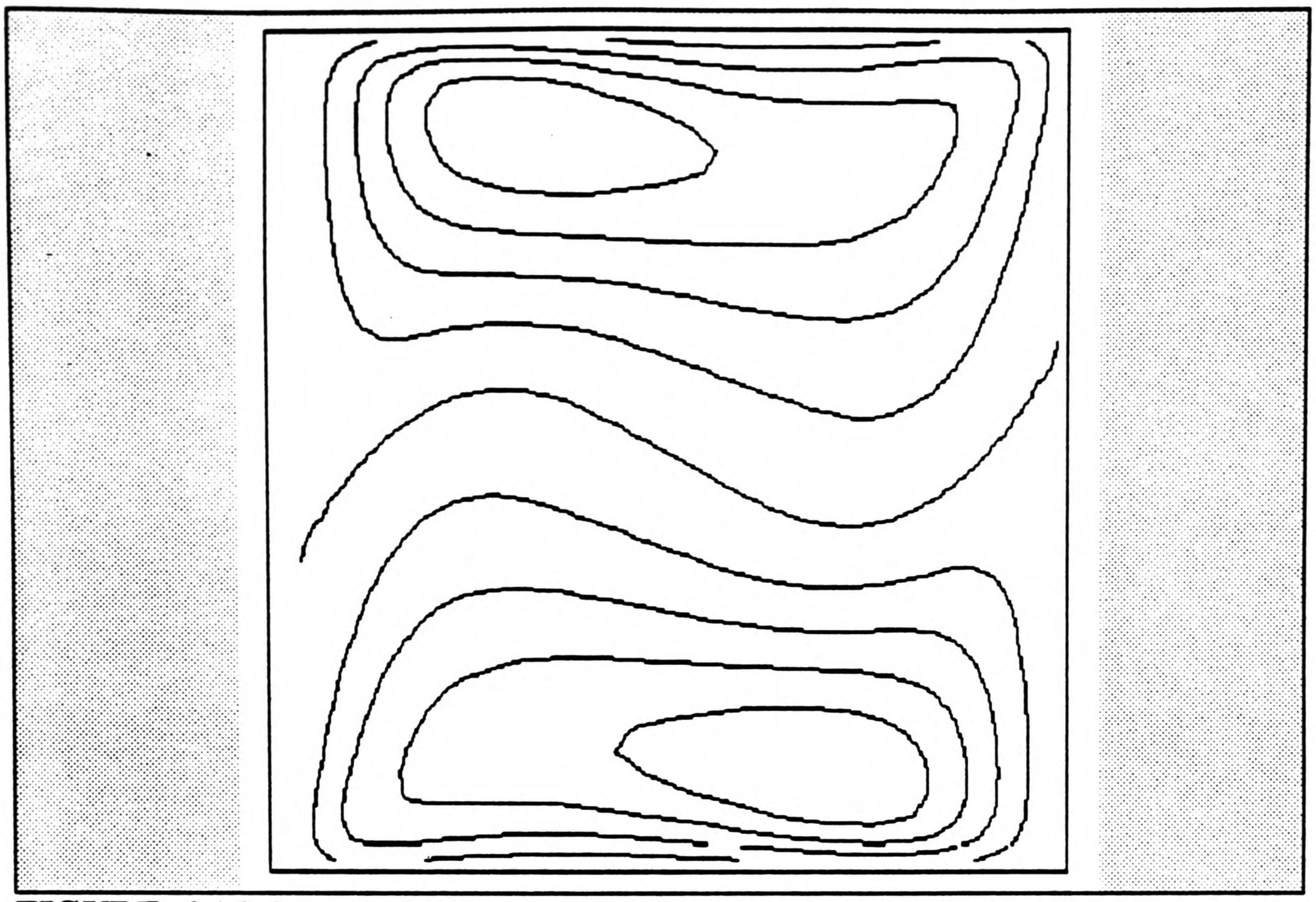


FIGURE 6.4.2-2c u-velocity contour plot for natural convection problem ($Ra=10^5$). Contours at -43.5 (8.7) 43.5

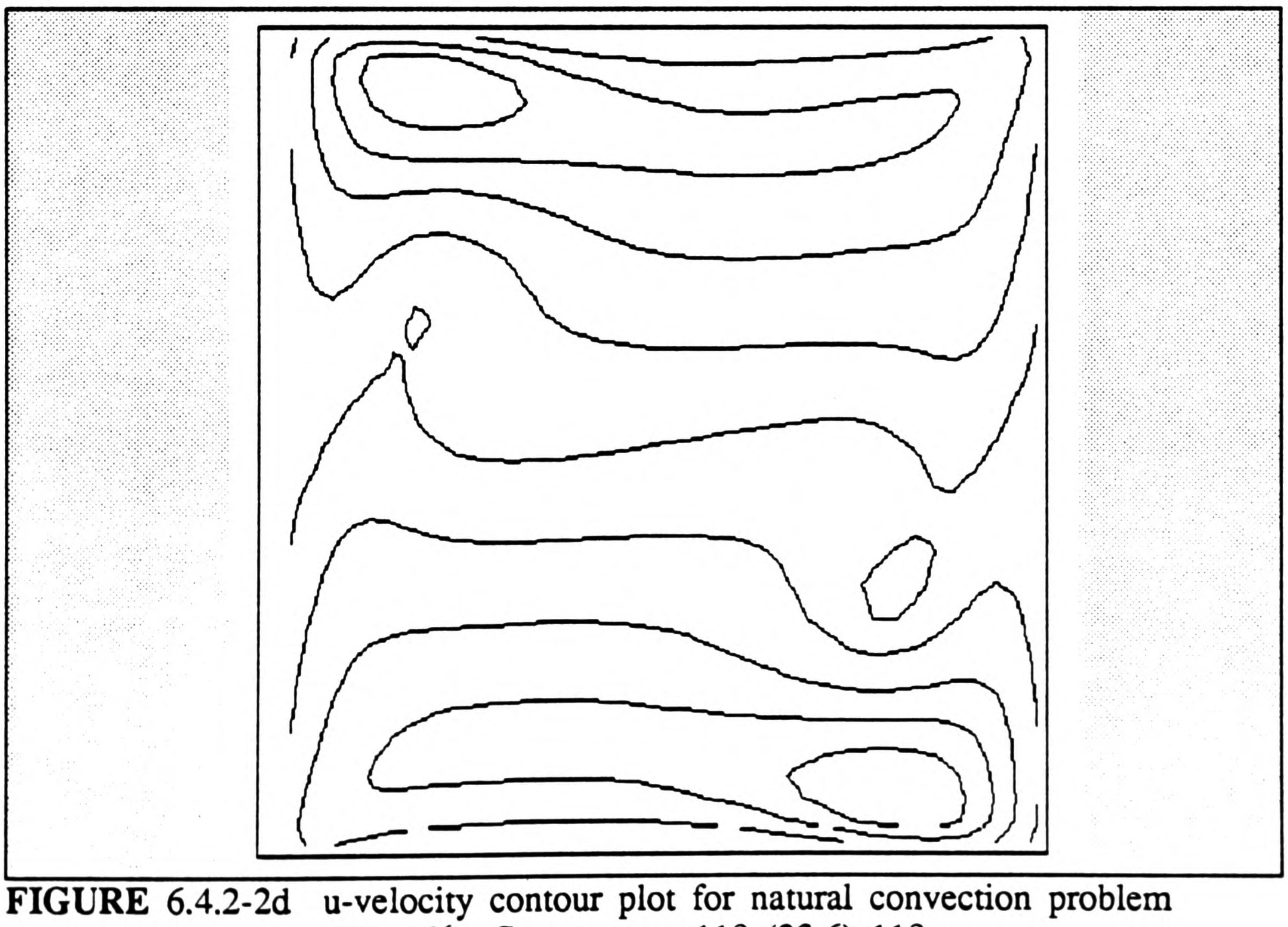


FIGURE 6.4.2-2d u-velocity contour plot for natural convection problem ($Ra=10^6$). Contours at -118 (23.6) 118

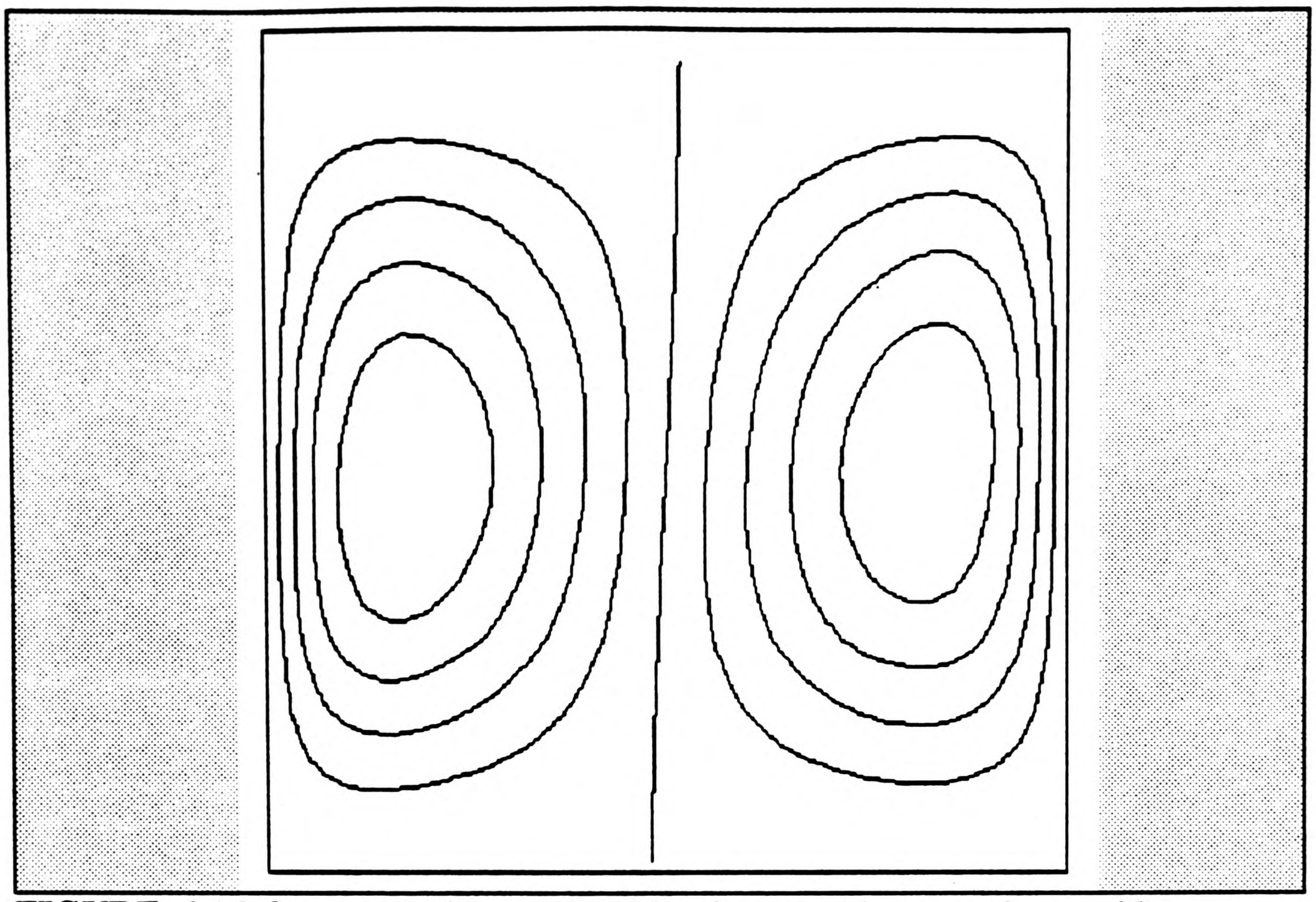


FIGURE 6.4.2-3a v-velocity contour plot for natural convection problem ($Ra=10^3$). Contours at -3.67 (0.734) 3.67

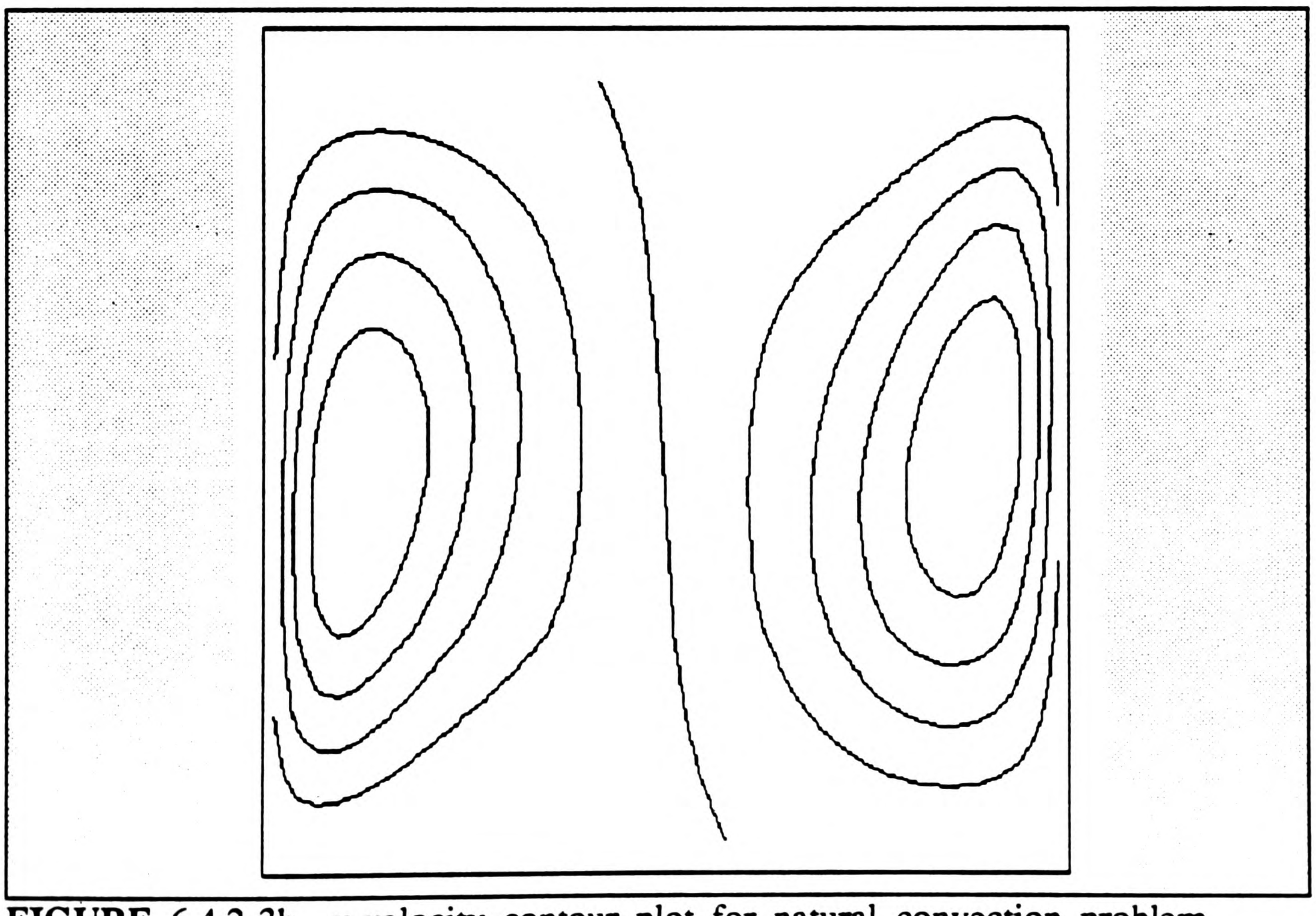


FIGURE 6.4.2-3b v-velocity contour plot for natural convection problem ($Ra=10^4$). Contours at -19.44 (3.88) 19.4

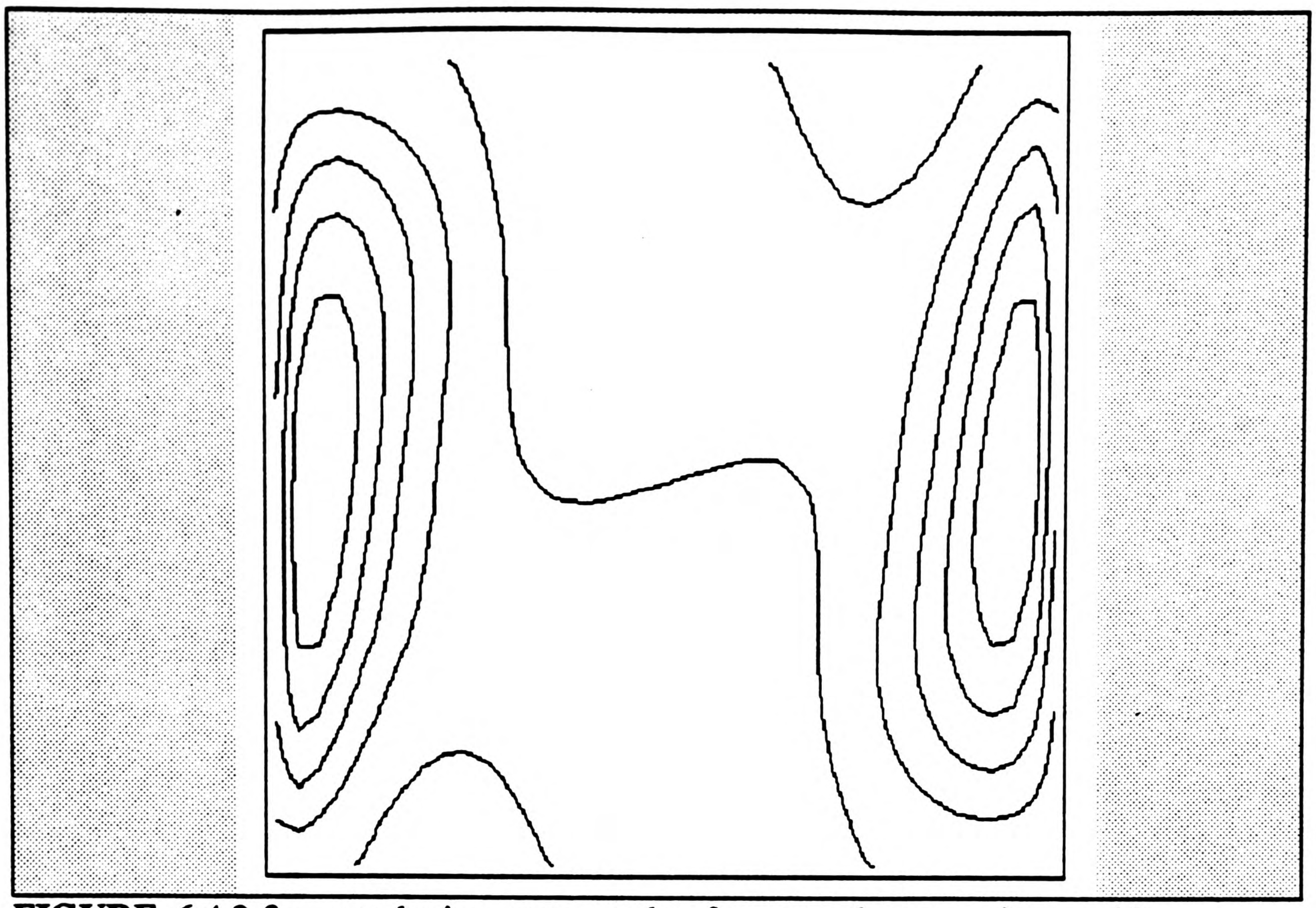


FIGURE 6.4.2-3c v-velocity contour plot for natural convection problem ($Ra=10^5$). Contours at -69.3 (13.86) 69.3

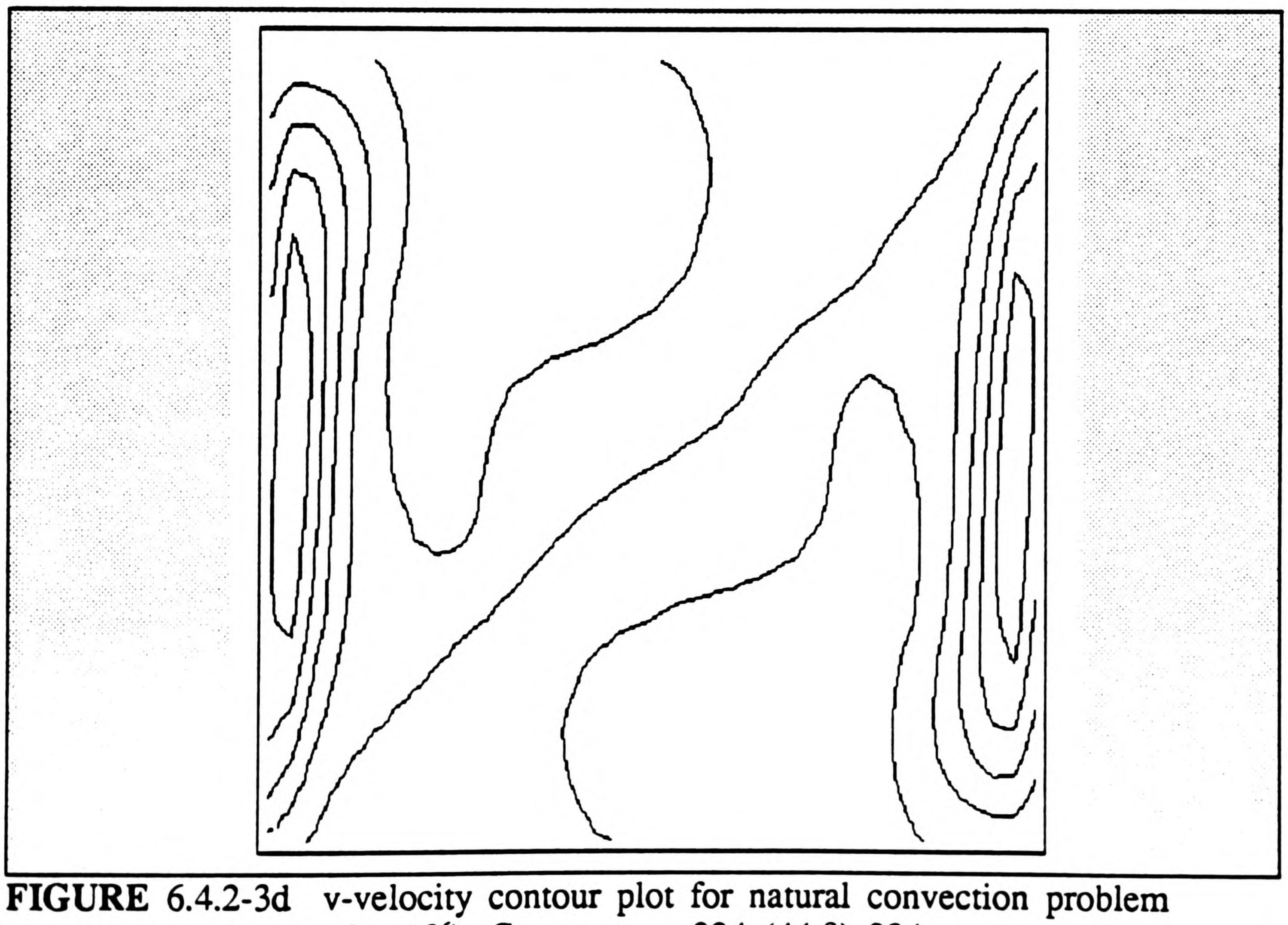


FIGURE 6.4.2-3d v-velocity contour plot for natural convection problem ($Ra=10^6$). Contours at -224 (44.8) 224

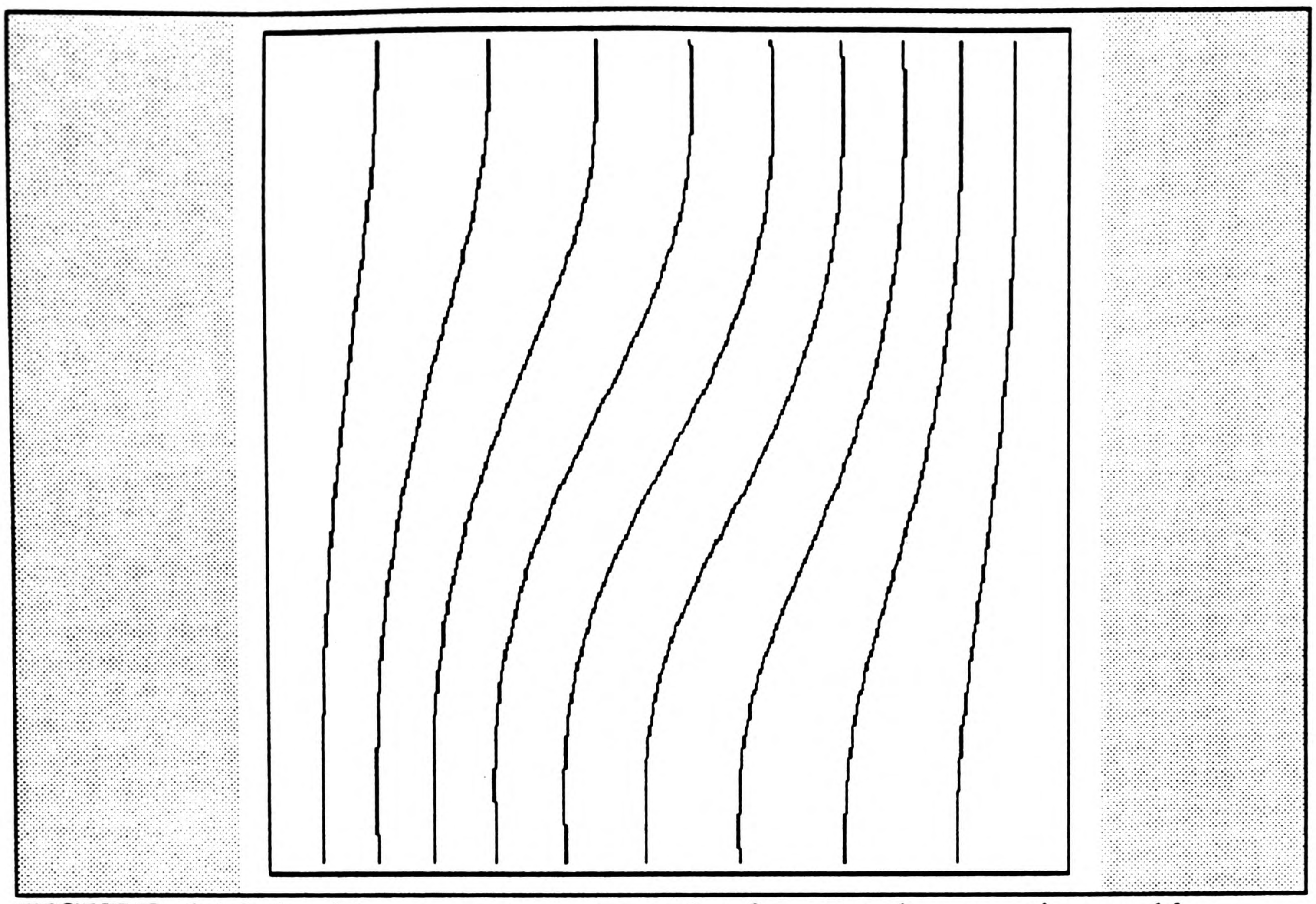


FIGURE 6.4.2-4a Temperature contour plot for natural convection problem ($Ra=10^3$). Contours at 0 (0.1) 1

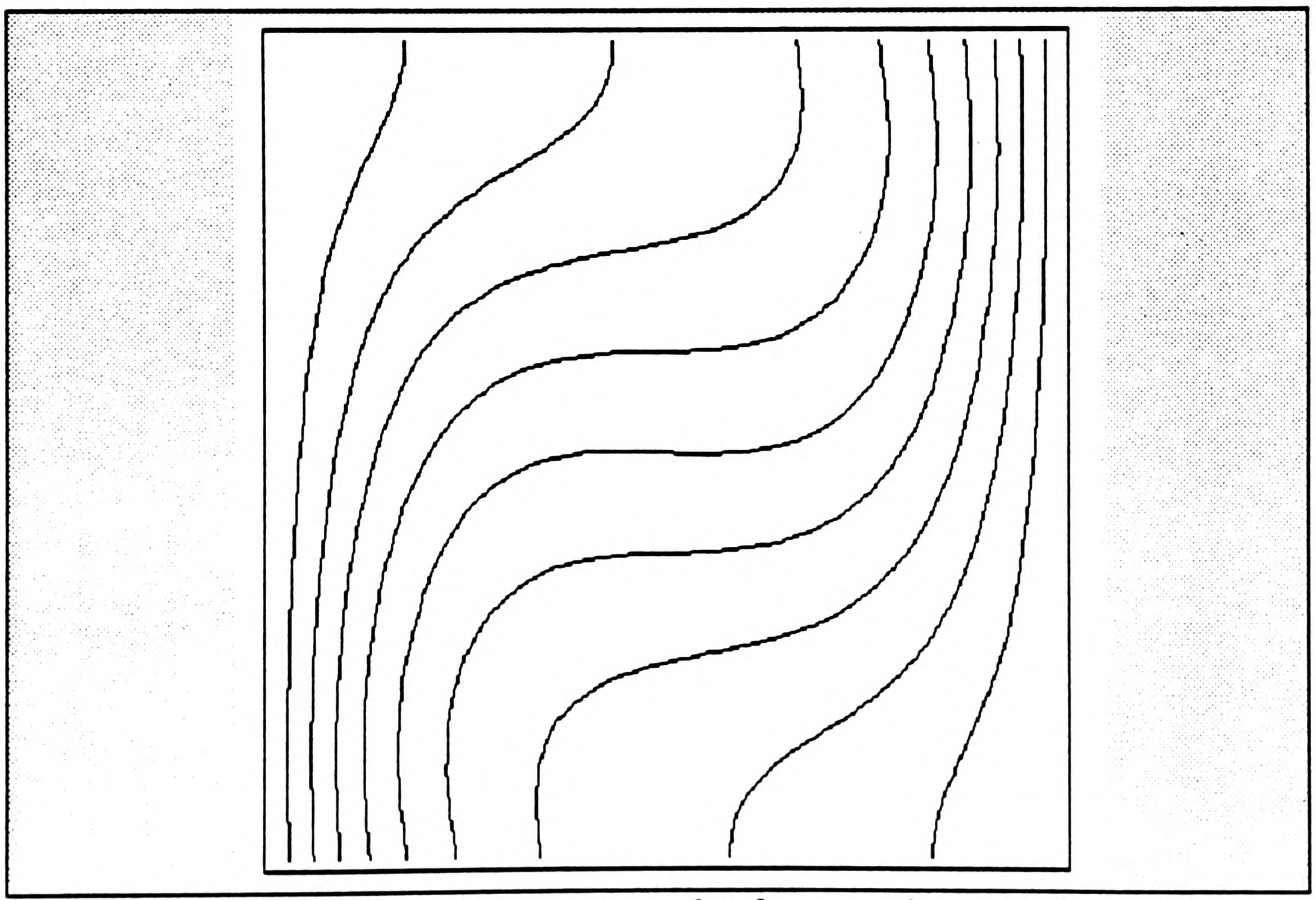


FIGURE 6.4.2-4b Temperature contour plot for natural convection problem ($Ra=10^4$). Contours at 0 (0.1) 1

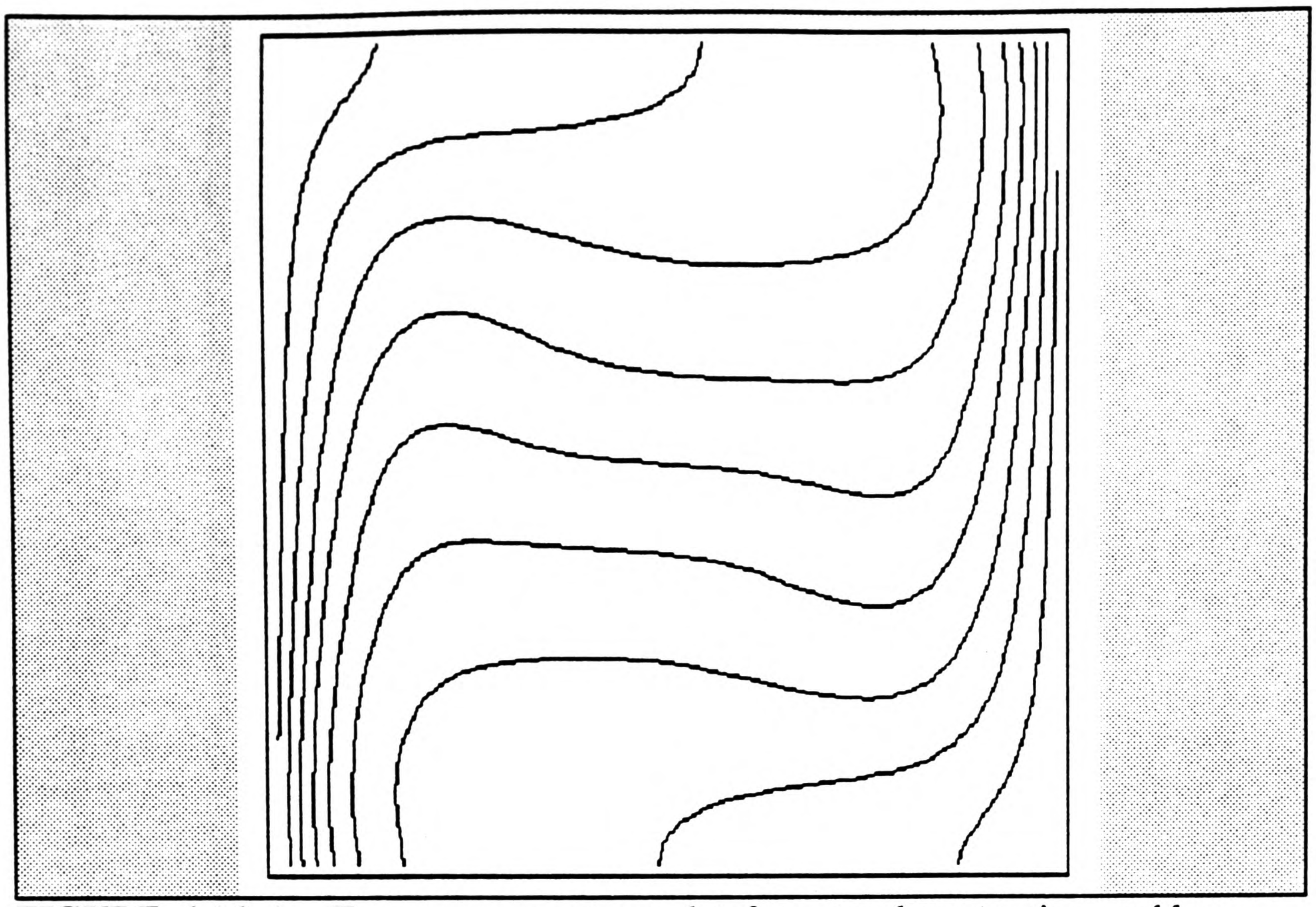


FIGURE 6.4.2-4c Temperature contour plot for natural convection problem ($Ra=10^5$). Contours at 0 (0.1) 1

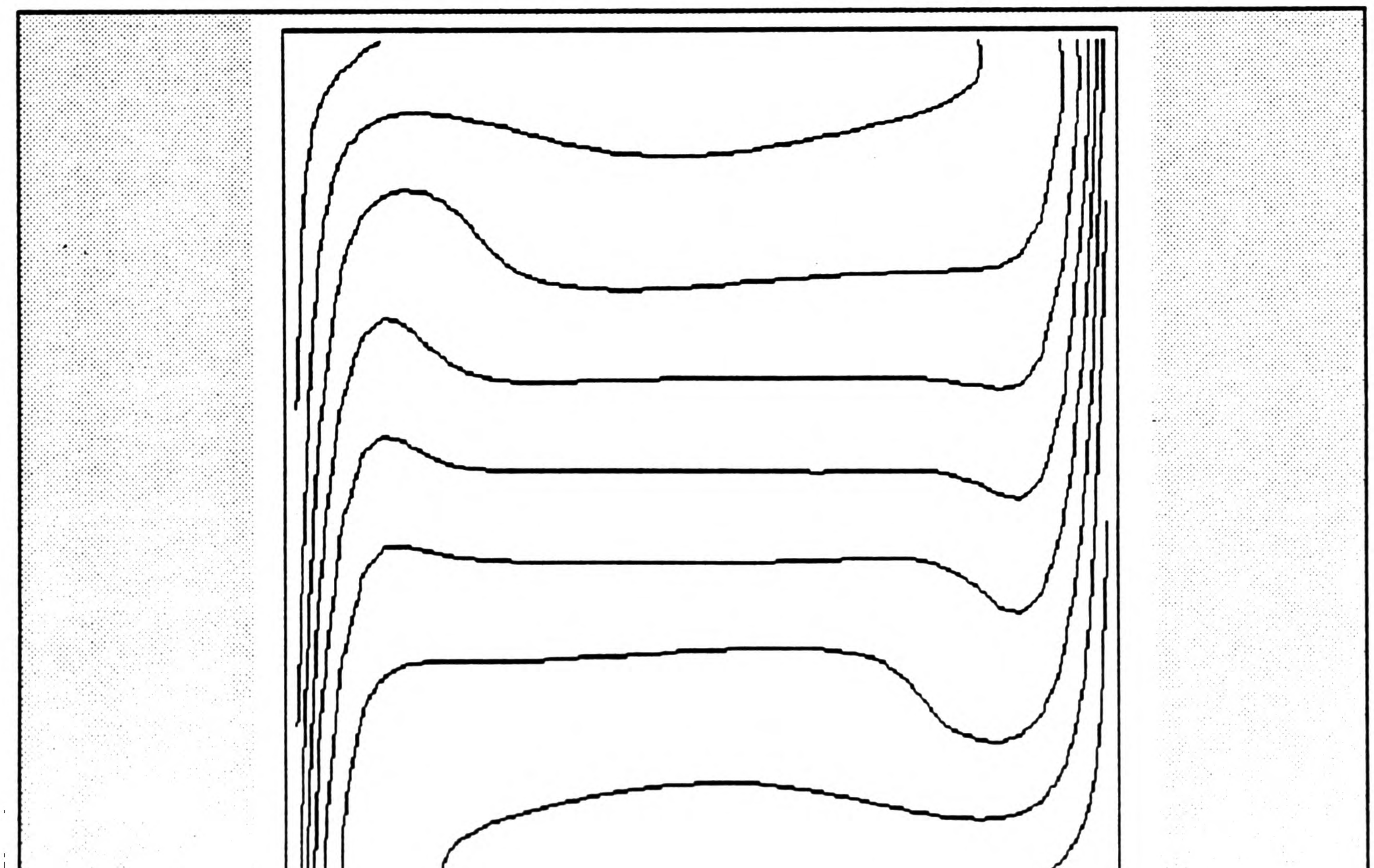


FIGURE 6.4.2-4d Temperature contour plot for natural convection problem ($Ra=10^6$). Contours at 0 (0.1) 1

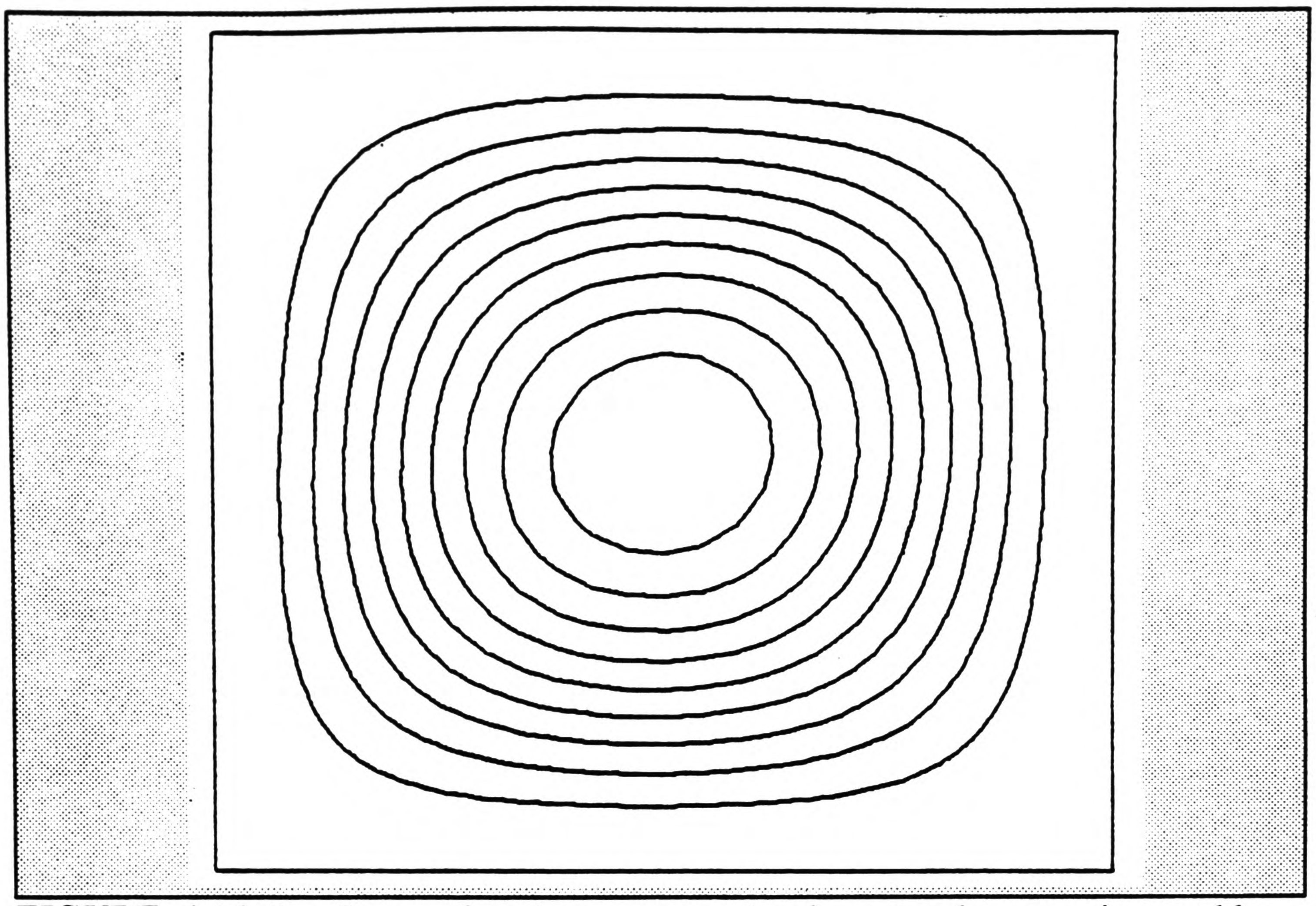


FIGURE 6.4.2-5a Stream function contour plot for natural convection problem ($Ra=10^3$). Contours at -1.172 (0.1172) 0

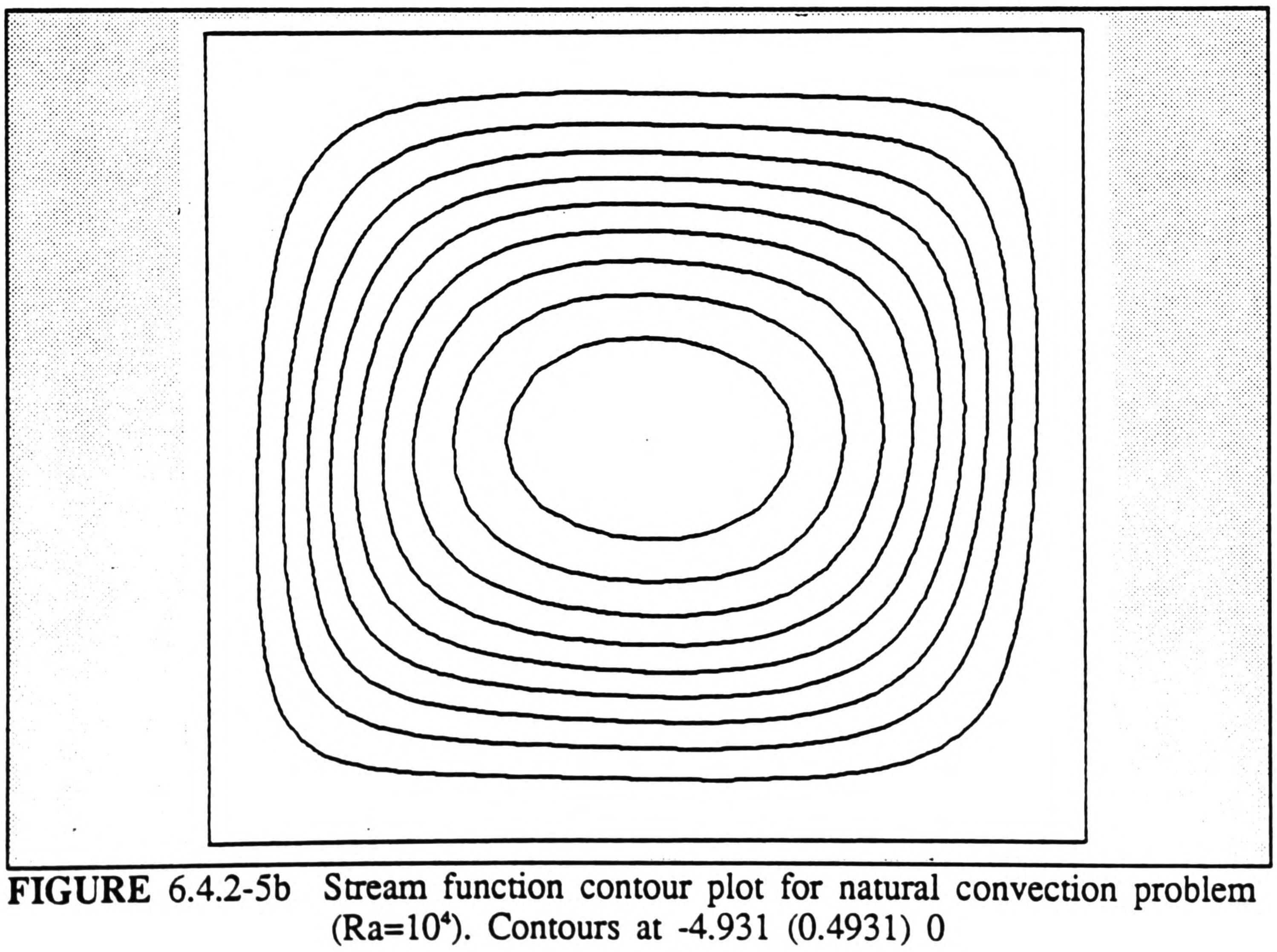


FIGURE 6.4.2-5b Stream function contour plot for natural convection problem ($Ra=10^4$). Contours at -4.931 (0.4931) 0

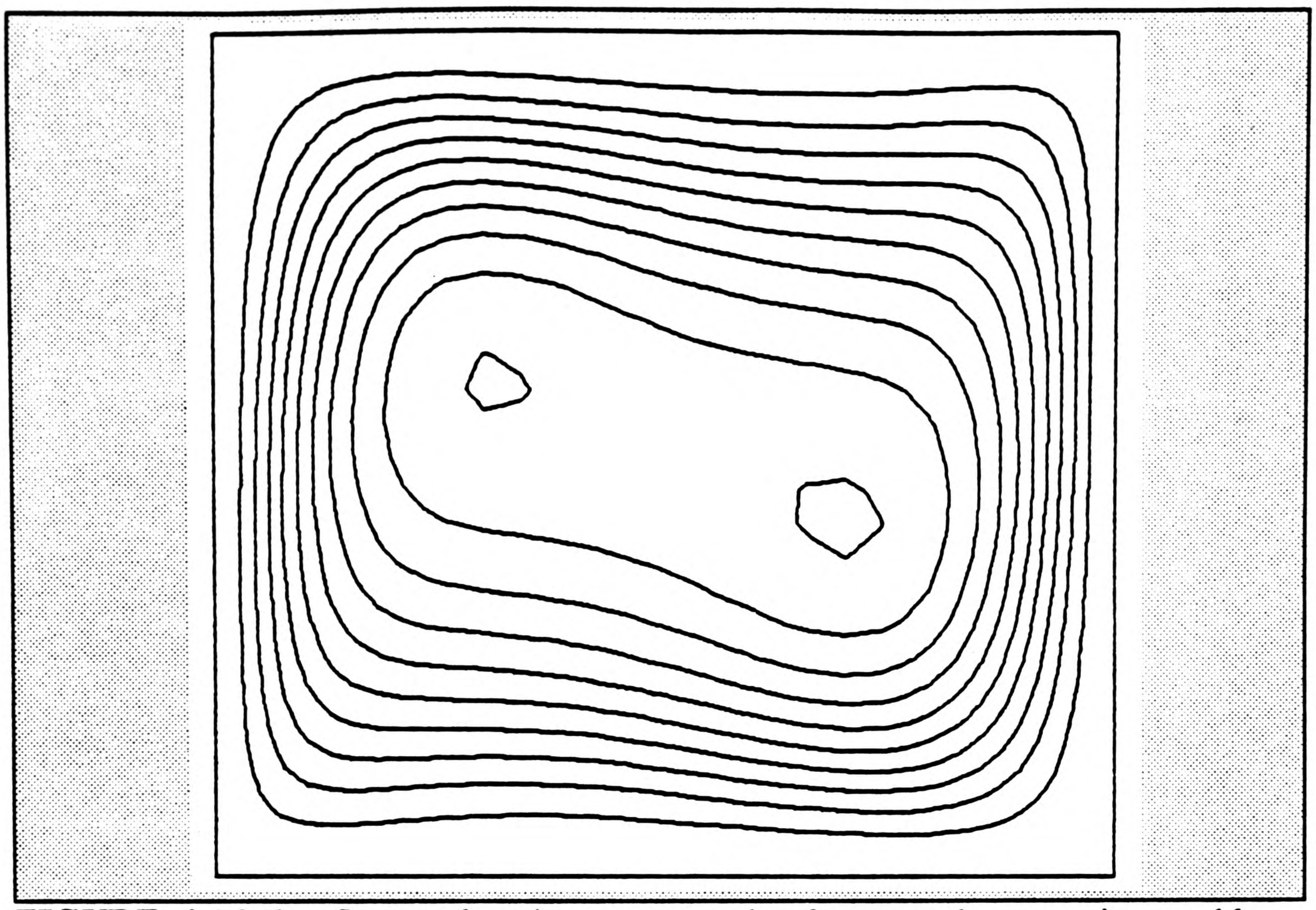


FIGURE 6.4.2-5c Stream function contour plot for natural convection problem ($Ra=10^5$). Contours at -9.469 (0.9469) 0

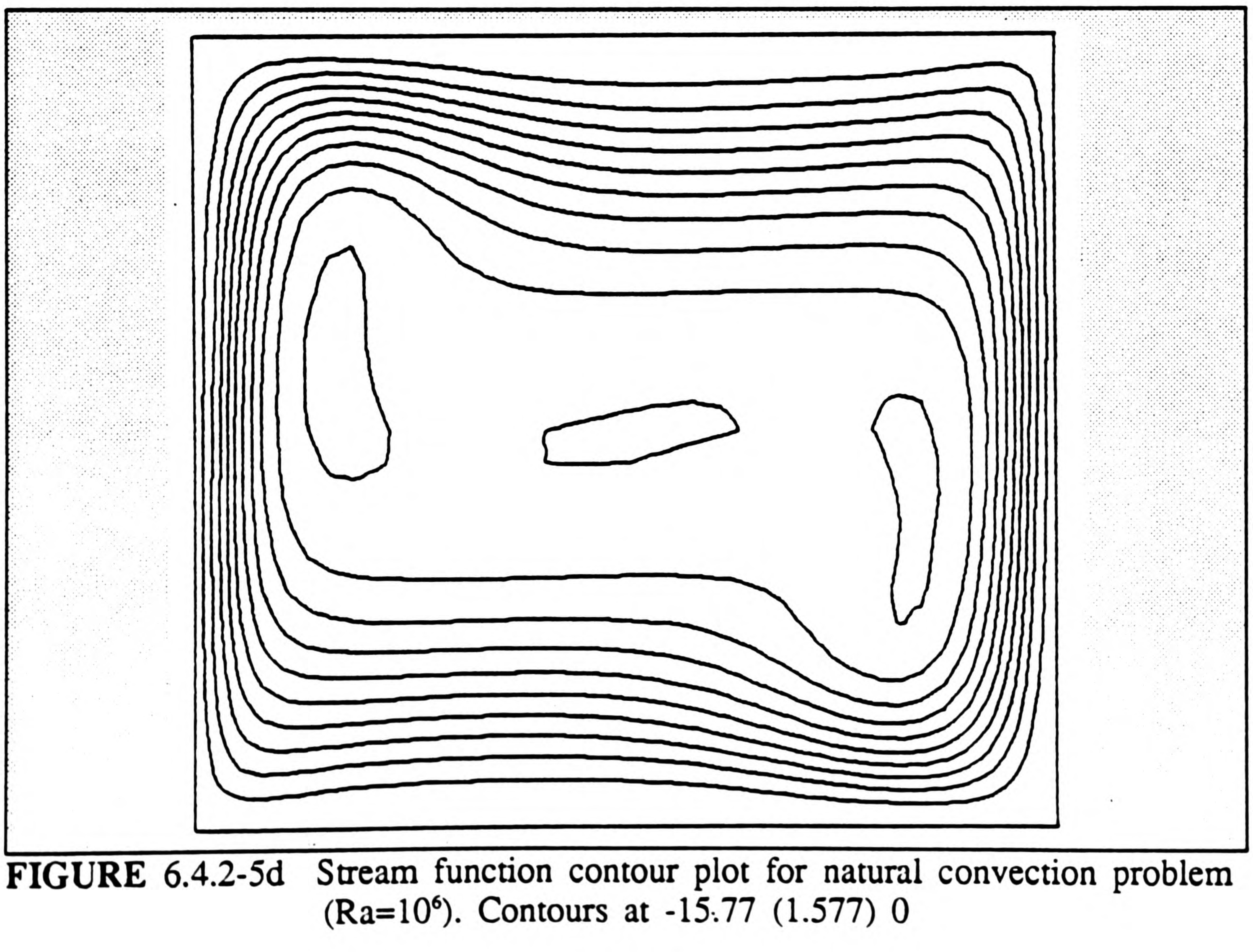


FIGURE 6.4.2-5d Stream function contour plot for natural convection problem ($Ra=10^6$). Contours at -15.77 (1.577) 0

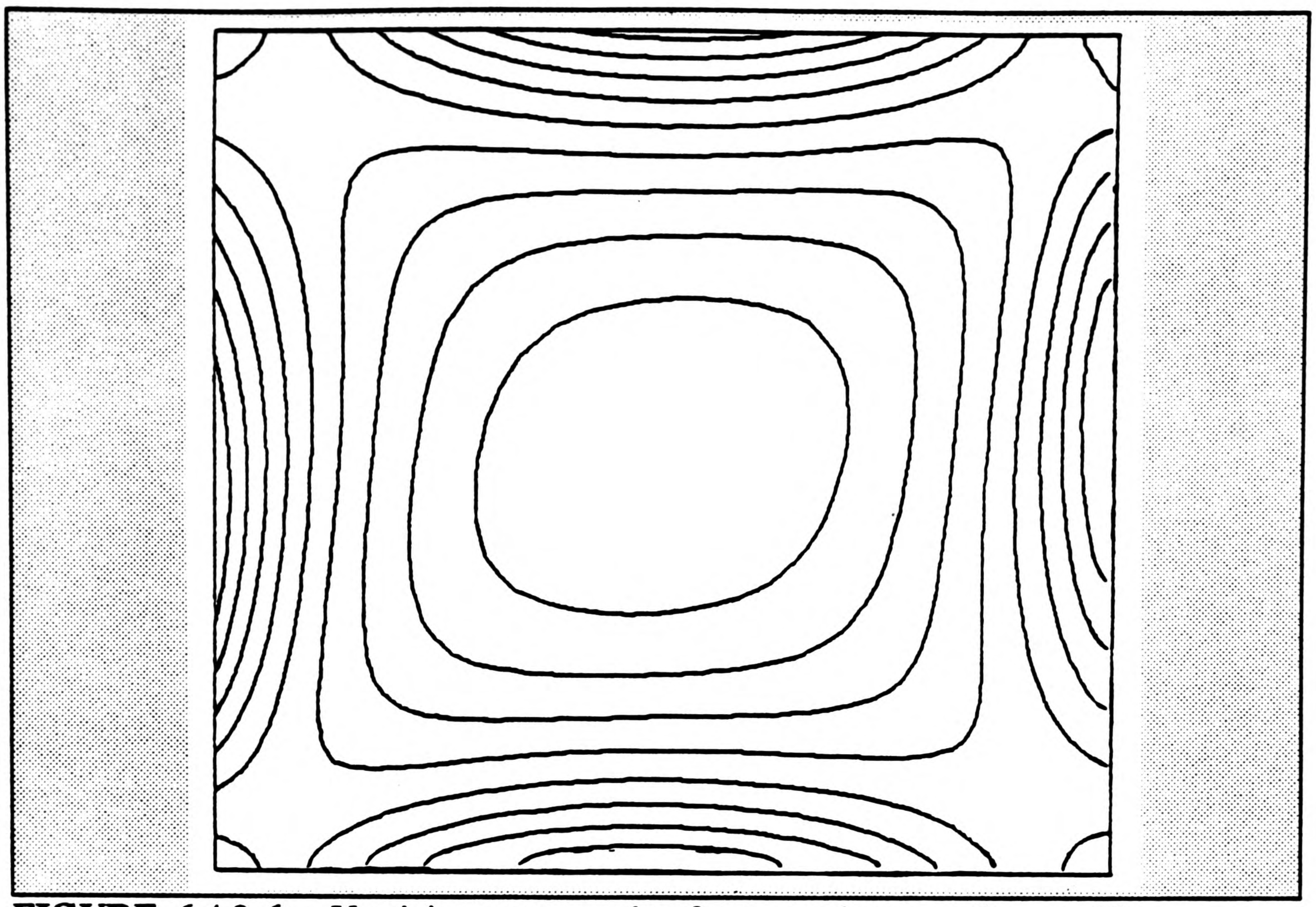


FIGURE 6.4.2-6a Vorticity contour plot for natural convection problem ($Ra=10^3$). Contours at -31.39 (8.300) 51.61

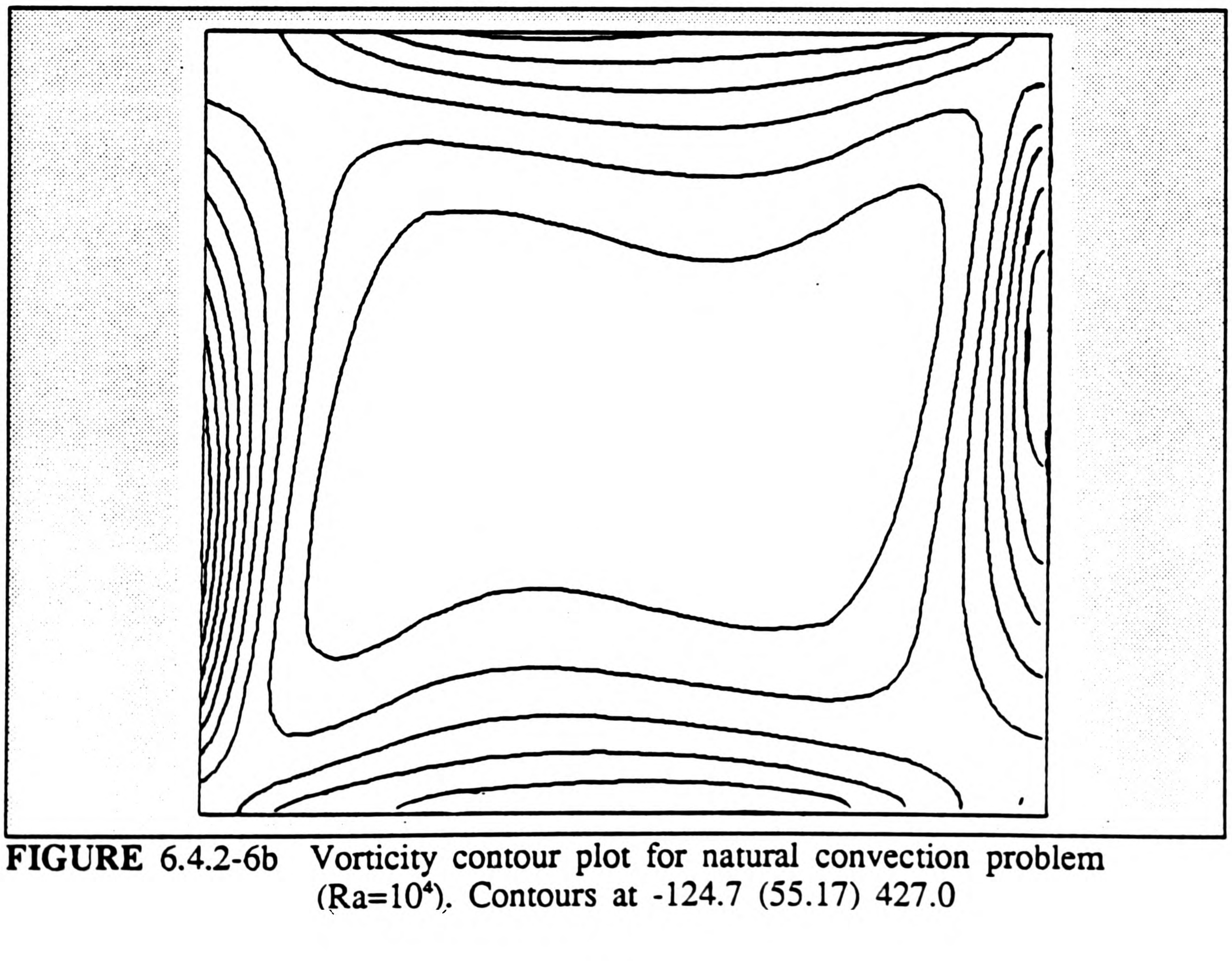


FIGURE 6.4.2-6b Vorticity contour plot for natural convection problem ($Ra=10^4$). Contours at -124.7 (55.17) 427.0

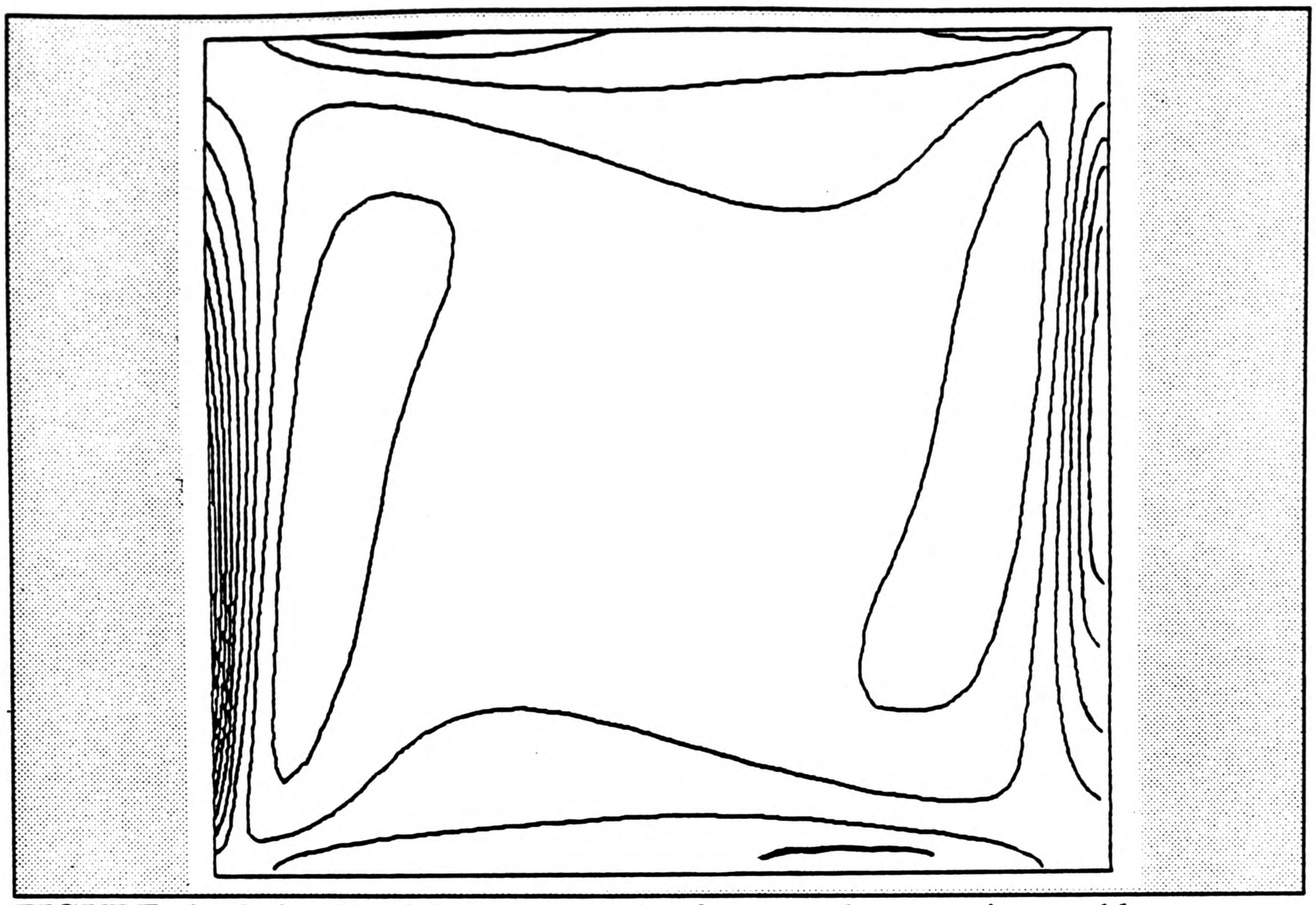


FIGURE 6.4.2-6c Vorticity contour plot for natural convection problem ($Ra=10^5$). Contours at -598.0 (301.5) 2417

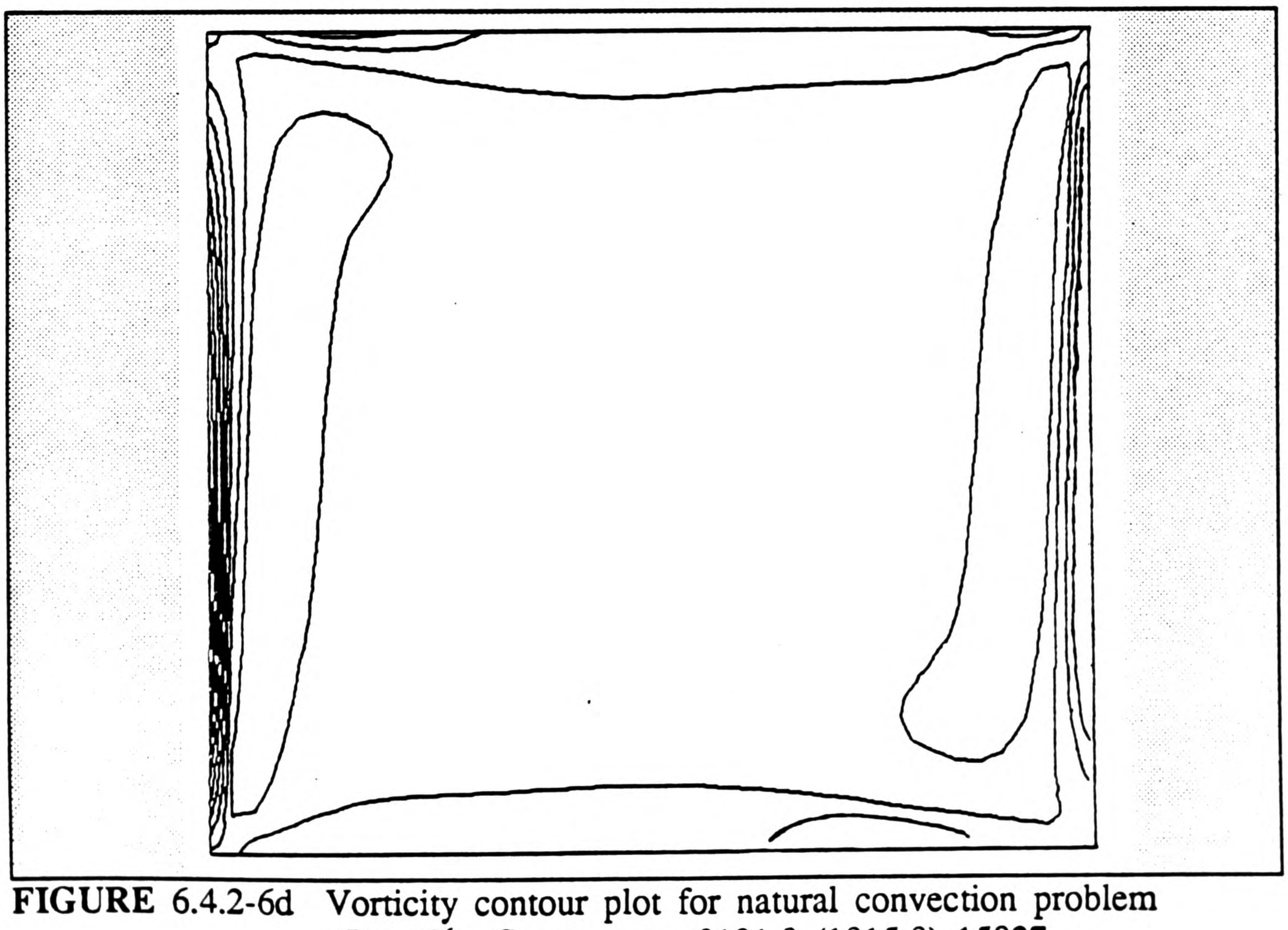


FIGURE 6.4.2-6d Vorticity contour plot for natural convection problem ($Ra=10^6$). Contours at -3131.0 (1815.8) 15027

	Rayleigh number, Ra							
	10^3		10^4		10^5		10^6	
Nu_{min}	0.719	(0.692)	0.640	(0.585)	0.890	(0.729)	1.428	(0.989)
y	1.000	(1.000)	1.000	(1.000)	1.000	(1.000)	1.000	(1.000)
Nu_{max}	1.509	(1.505)	3.617	(3.528)	8.300	(7.717)	20.624	(17.925)
y	0.104	(0.092)	0.129	(0.143)	0.083	(0.081)	0.051	(0.038)
u_{max}	3.532	(3.694)	16.045	(16.178)	37.708	(34.73)	73.93	(64.63)
y	0.805	(0.813)	0.831	(0.823)	0.857	(0.855)	0.883	(0.850)
v_{max}	3.604	(3.697)	19.356	(19.617)	69.26	(68.59)	223.99	(219.36)
x	0.169	(0.178)	0.117	(0.119)	0.065	(0.066)	0.039	(0.038)

( ) benchmark solution

TABLE 6.4.2-1 A comparison between the present study and the benchmark solution (de Vahl Davis [1983b]).

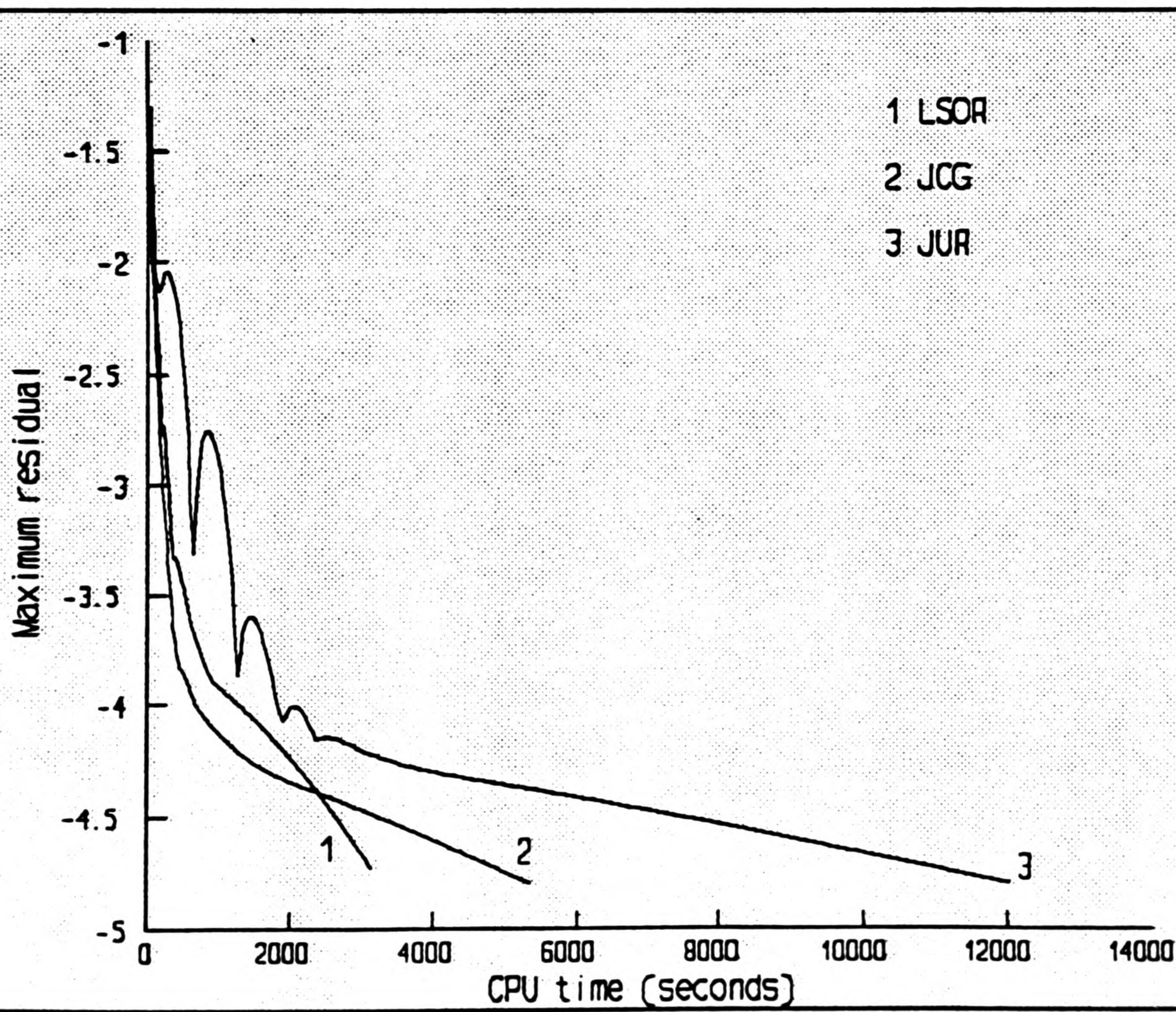


FIGURE 6.4.2-7 Residual plot for natural convection problem (Ra=10^3) using different scalar algorithms to solve the pressure-correction equation

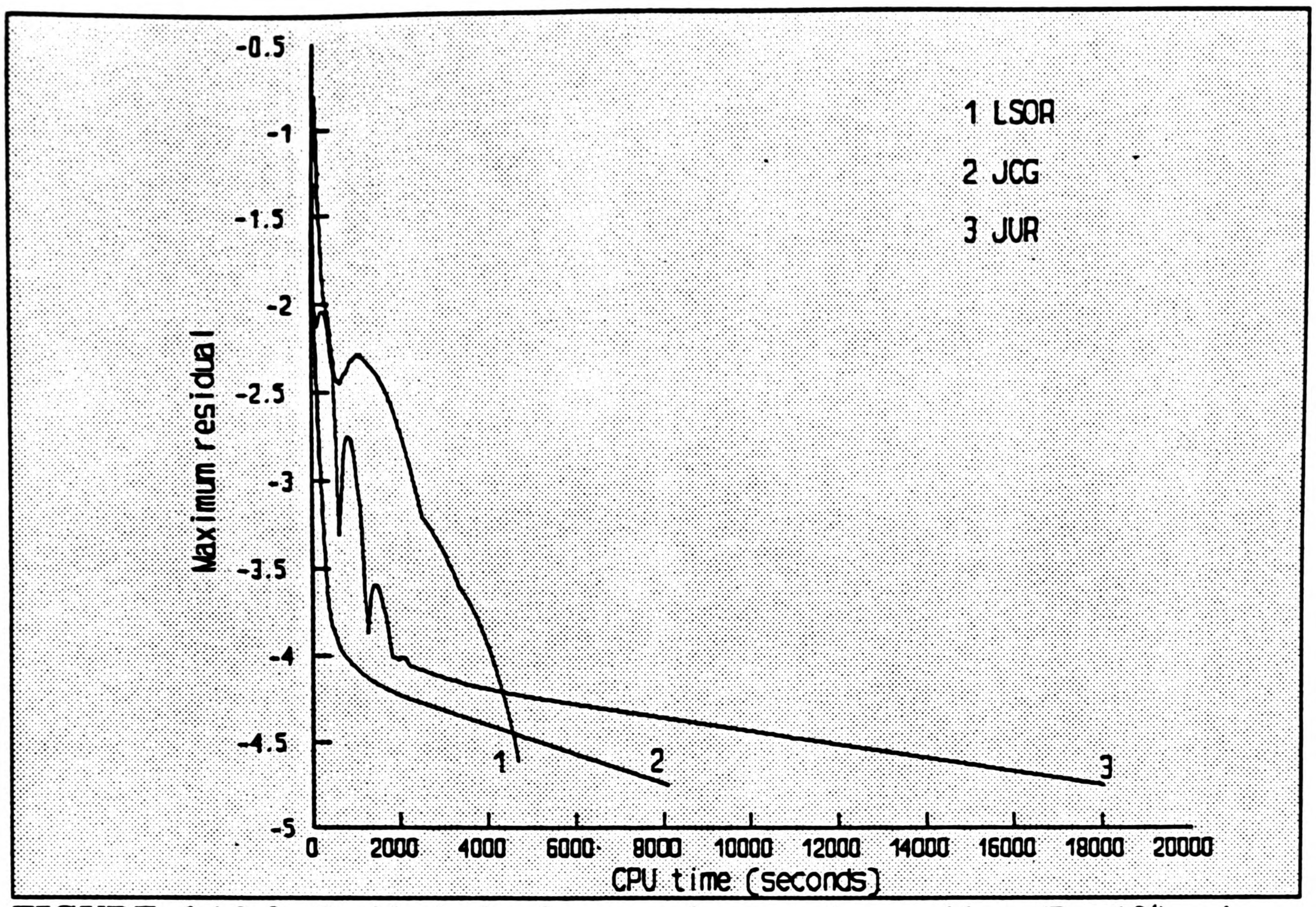


FIGURE 6.4.2-8 Residual plot for natural convection problem (Ra=10^4) using different scalar algorithms to solve the pressure-correction equation

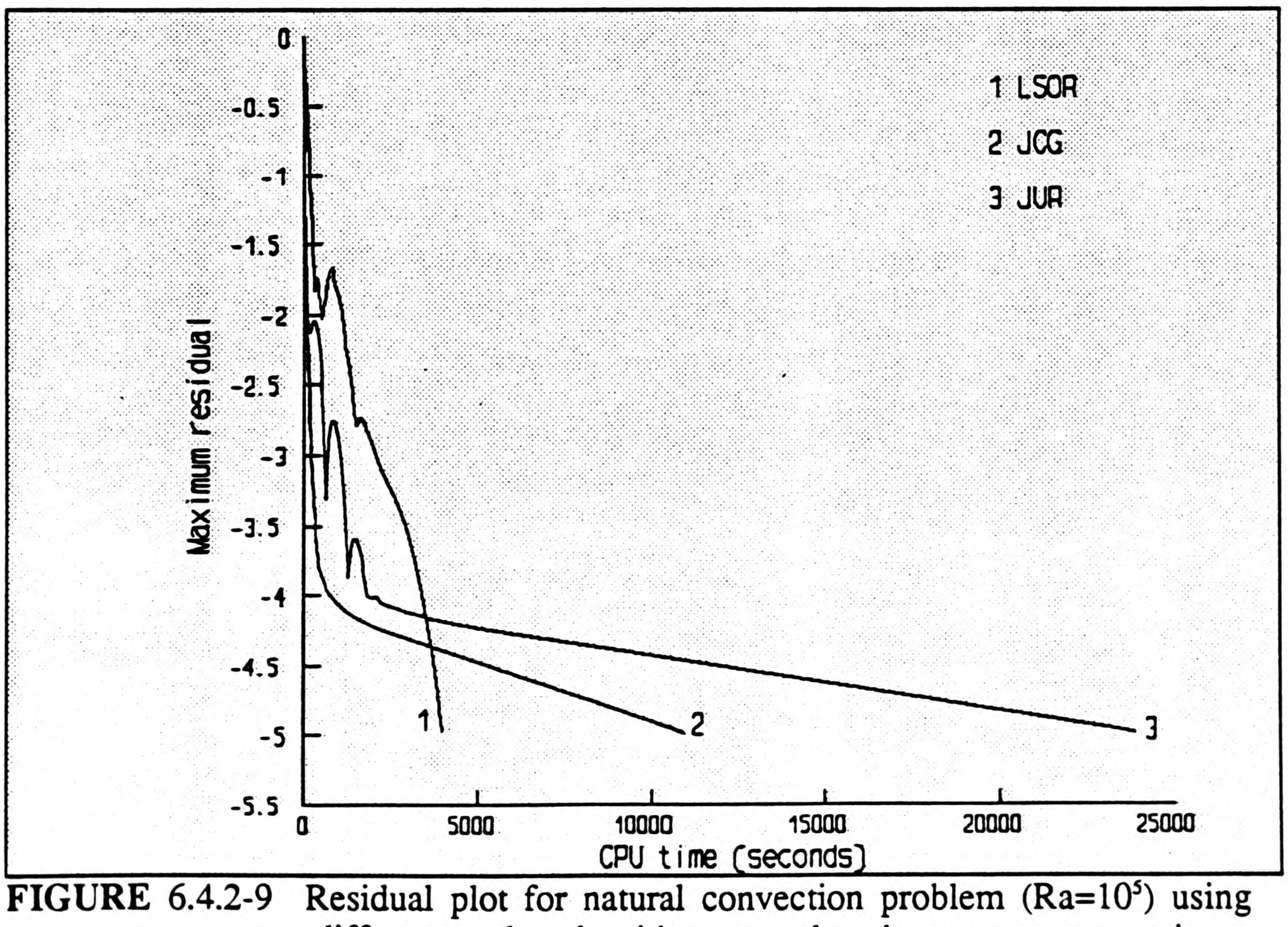


FIGURE 6.4.2-9 Residual plot for natural convection problem (Ra=10^5) using different scalar algorithms to solve the pressure-correction equation

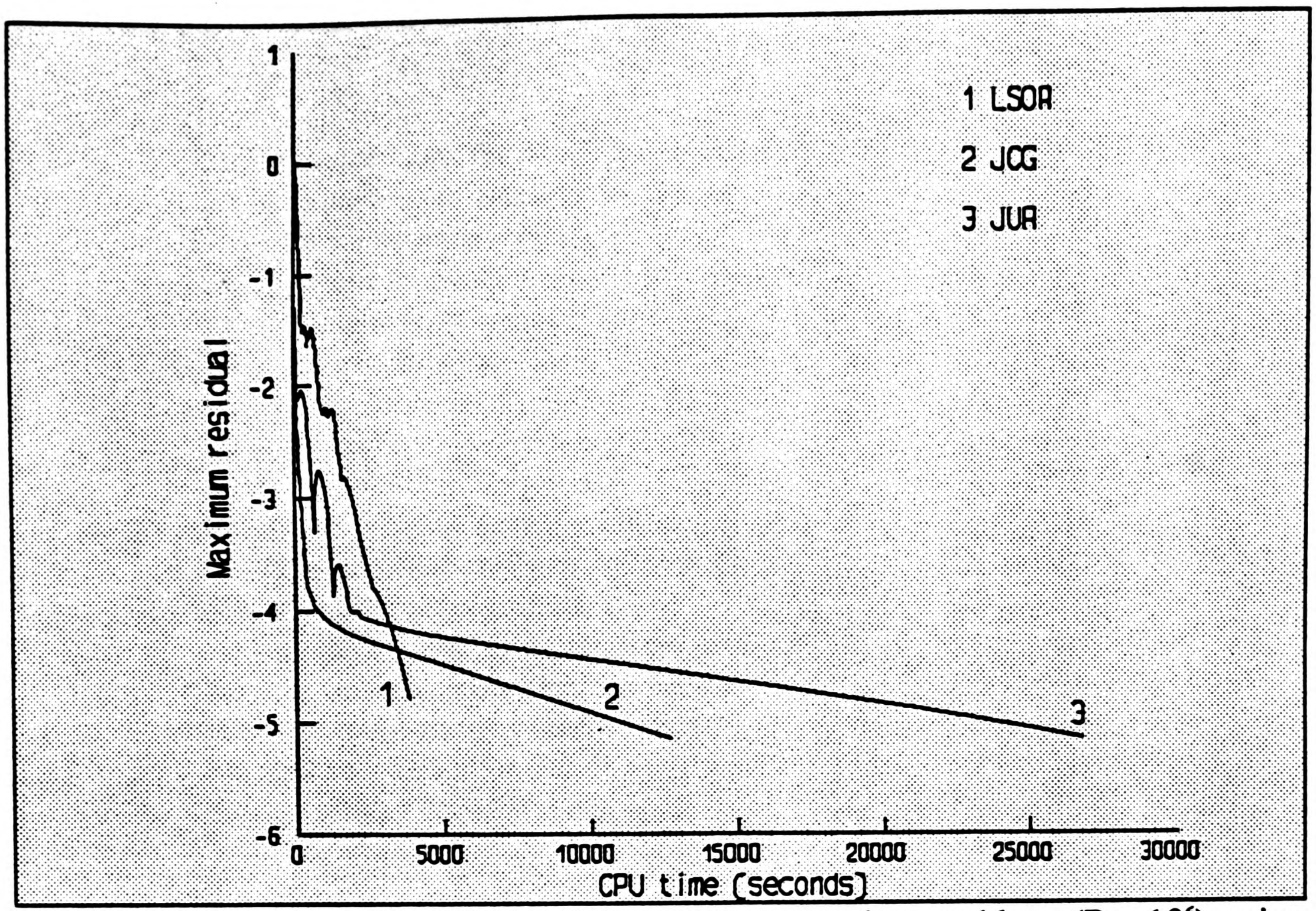


FIGURE 6.4.2-10 Residual plot for natural convection problem (Ra=10^6) using different scalar algorithms to solve the pressure-correction equation

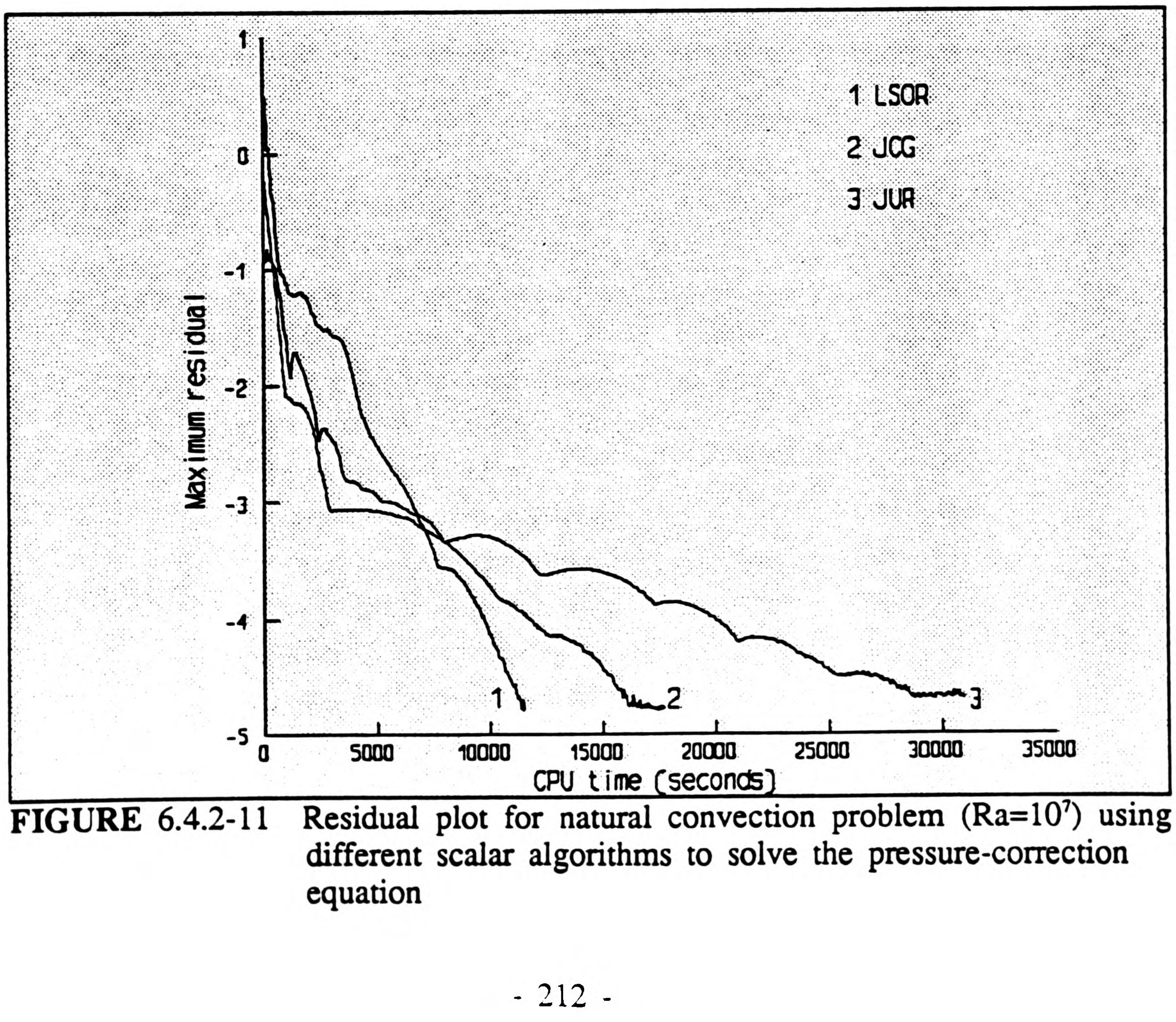


FIGURE 6.4.2-11 Residual plot for natural convection problem (Ra=10^7) using different scalar algorithms to solve the pressure-correction equation

	JUR	JCG
Set up source terms for u-momentum equation	.6	1.0
Set up u-momentum equation coefficients	2.7	4.6
Solve for u-momentum values u^*	1.9	3.2
Set up source terms for v-momentum equation	.6	1.0
Set up v-momentum equation coefficients	2.6	4.5
Solve for v-momentum values v^*	2.0	3.4
Set up pressure-correction coefficients	6.0	10.4
Solve for pressure-correction values p'	70.5	49.3
Correct u^*, v^* and p^* to produce u, v and p	.4	.7
Set up source terms for k turbulence equation	1.8	3.1
Set up k turbulence equation coefficients	1.5	2.6
Solve for k turbulence values	1.3	2.2
Set up source terms for ε turbulence equation	1.8	3.1
Set up ε turbulence equation coefficients	1.5	2.6
Solve for ε turbulence values	1.3	2.2
Set up source terms for enthalpy equation	.7	1.3
Set up enthalpy equation coefficients	1.5	2.6
Solve for enthalpy values h	1.3	2.2

TABLE 6.5-1 Percentage breakdown of the SIMPLE procedure for L-shaped flow problem.

	JUR	JCG
Set up source terms for u-momentum equation	.8	1.8
Set up u-momentum equation coefficients	3.2	7.2
Solve for u-momentum values u^*	2.2	4.9
Set up source terms for v-momentum equation	.9	2.1
Set up v-momentum equation coefficients	3.3	7.4
Solve for v-momentum values v^*	2.2	4.9
Set up pressure-correction coefficients	7.0	15.7
Solve for pressure-correction values p'	76.2	45.2
Correct u^*, v^* and p^* to produce u, v and p	.5	3.1
Set up source terms for enthalpy equation	.4	.2
Set up enthalpy equation coefficients	1.9	4.2
Solve for enthalpy values h	1.4	3.3

TABLE 6.5-2a Percentage breakdown of the SIMPLE procedure for natural convection problem ($Ra=10^3$).

	JUR	JCG
Set up source terms for u-momentum equation	.7	1.6
Set up u-momentum equation coefficients	2.8	6.3
Solve for u-momentum values u^*	1.9	4.3
Set up source terms for v-momentum equation	.8	1.8
Set up v-momentum equation coefficients	2.9	6.5
Solve for v-momentum values v^*	1.9	4.3
Set up pressure-correction coefficients	6.2	13.8
Solve for pressure-correction values p'	79.0	52.9
Correct u^*, v^* and p^* to produce u, v and p	.4	1.0
Set up source terms for enthalpy equation	.4	.9
Set up enthalpy equation coefficients	1.7	3.7
Solve for enthalpy values h	1.3	2.9

TABLE 6.5-2b Percentage breakdown of the SIMPLE procedure for natural convection problem ($Ra=10^4$).

	JUR	JCG
Set up source terms for u-momentum equation	.7	1.5
Set up u-momentum equation coefficients	2.7	5.8
Solve for u-momentum values u^*	1.8	4.0
Set up source terms for v-momentum equation	.8	1.7
Set up v-momentum equation coefficients	2.7	6.0
Solve for v-momentum values v^*	1.8	4.0
Set up pressure-correction coefficients	5.9	12.8
Solve for pressure-correction values p'	80.0	56.5
Correct u^*, v^* and p^* to produce u, v and p	.4	.9
Set up source terms for enthalpy equation	.4	.8
Set up enthalpy equation coefficients	1.6	3.4
Solve for enthalpy values h	1.2	2.6

TABLE 6.5-2c Percentage breakdown of the SIMPLE procedure for natural convection problem ($Ra=10^5$).

	JUR	JCG
Set up source terms for u-momentum equation	.7	1.5
Set up u-momentum equation coefficients	2.9	6.0
Solve for u-momentum values u^*	2.0	4.1
Set up source terms for v-momentum equation	.8	1.7
Set up v-momentum equation coefficients	2.9	6.2
Solve for v-momentum values v^*	2.0	4.1
Set up pressure-correction coefficients	6.3	13.2
Solve for pressure-correction values p'	78.6	55.3
Correct u^*, v^* and p^* to produce u, v and p	.4	.9
Set up source terms for enthalpy equation	.4	.8
Set up enthalpy equation coefficients	1.7	3.5
Solve for enthalpy values h	1.3	2.7

TABLE 6.5-2d Percentage breakdown of the SIMPLE procedure for natural convection problem ($Ra=10^6$).

	JUR	JCG
Set up source terms for u-momentum equation	.7	1.1
Set up u-momentum equation coefficients	2.9	4.6
Solve for u-momentum values u^*	1.0	3.1
Set up source terms for v-momentum equation	.8	1.3
Set up v-momentum equation coefficients	3.0	4.7
Solve for v-momentum values v^*	2.0	3.1
Set up pressure-correction coefficients	6.3	10.0
Solve for pressure-correction values p'	67.4	48.4
Correct u^*, v^* and p^* to produce u, v and p	.5	.7
Set up source terms for k turbulence equation	2.4	3.9
Set up k turbulence equation coefficients	1.7	2.7
Solve for k turbulence values	1.3	2.1
Set up source terms for ε turbulence equation	2.4	3.8
Set up ε turbulence equation coefficients	1.7	2.7
Solve for ε turbulence values	1.3	2.1
Set up source terms for enthalpy equation	.6	.9
Set up enthalpy equation coefficients	1.7	2.7
Solve for enthalpy values h	1.3	2.1

TABLE 6.5-2e Percentage breakdown of the SIMPLE procedure for natural convection problem ($Ra=10^7$).

natural convection problem, the introduction of the k and ε scalars increases the essentially scalar computation from 8% to 13%.

An analysis using Amdahl's law is used to determine the theoretical benefits in using the JCG and JUR algorithms. For the L-shaped flow problem the fraction of code which can be vectorised using the JUR algorithm is f_v=0.933 and the expected speed-up factors are given by

$$\frac{1}{0.067 + 0.933/4.35} < S < \frac{1}{0.067 + 0.933/173.2}$$

$$3.55 < S < 13.81 \tag{6.5-1}$$

For the natural convection problem ($10^3 \leq Ra \leq 10^6$) f_v=0.966 and the speed-up factors are given by

$$\frac{1}{0.034 + 0.966/4.35} < S < \frac{1}{0.034 + 0.966/173.2}$$

$$3.91 < S < 25.27 \tag{6.5-2}$$

and for $Ra=10^7$ f_v=0.916

$$\frac{1}{0.084 + 0.916/4.35} < S < \frac{1}{0.084 + 0.916/173.2}$$

$$3.40 < S < 11.20 \tag{6.5-3}$$

When the JCG algorithm is used, the fraction of code vectorised in the L-shaped flow problem is f_v=0.885 and the expected speed-up is given by

$$\frac{1}{0.115 + 0.885/4.35} < S < \frac{1}{0.115 + 0.885/173.2}$$

$$3.14 < S < 8.33 \tag{6.5-4}$$

For the natural convection problem ($10^3 \leq Ra \leq 10^6$) $f_v=0.928$ and the speed-up factors are given by

$$\frac{1}{0.072 + 0.928/4.35} < S < \frac{1}{0.072 + 0.928/173.2}$$

$$3.50 < S < 12.93 \qquad (6.5\text{-}5)$$

and for $Ra=10^7$ $f_v=0.866$

$$\frac{1}{0.134 + 0.866/4.35} < S < \frac{1}{0.134 + 0.866/173.2}$$

$$3.00 < S < 7.19 \qquad (6.5\text{-}6)$$

6.6 Results

The residual plots show the effect of using the JURS and JURV algorithms on the two test problems (figures 6.6-1 and 6.6-2). Significant reductions are achieved with the vectorised version, these range from a factor of 13 for the L-shaped flow problem to over 23 for the natural convection problem. These factors agree well with the predictions in (6.5-1) and (6.5-3). Residual plots are also shown for the JCGS and JCGV algorithms (figures 6.6-3 and 6.6-4). The reductions in CPU time range from 8 for the L-shaped flow problem to 12 for the natural convection problem, again, there is good agreement with the predictions (6.5-4) and (6.5-6).

Although the speed-up factors appear flattering to the JUR and JCG algorithms there is a need to compare the best scalar and vector results. Generally, comparisons show that the LSOR algorithm is the best scalar algorithm (figures 6.6-5 and 6.6-6). The best vector algorithm can solve the L-shaped flow problem over 5 times faster than the best scalar algorithm. For the natural convection problem ($Ra \leq 10^6$) this increases to a factor of 11. The introduction of turbulence in

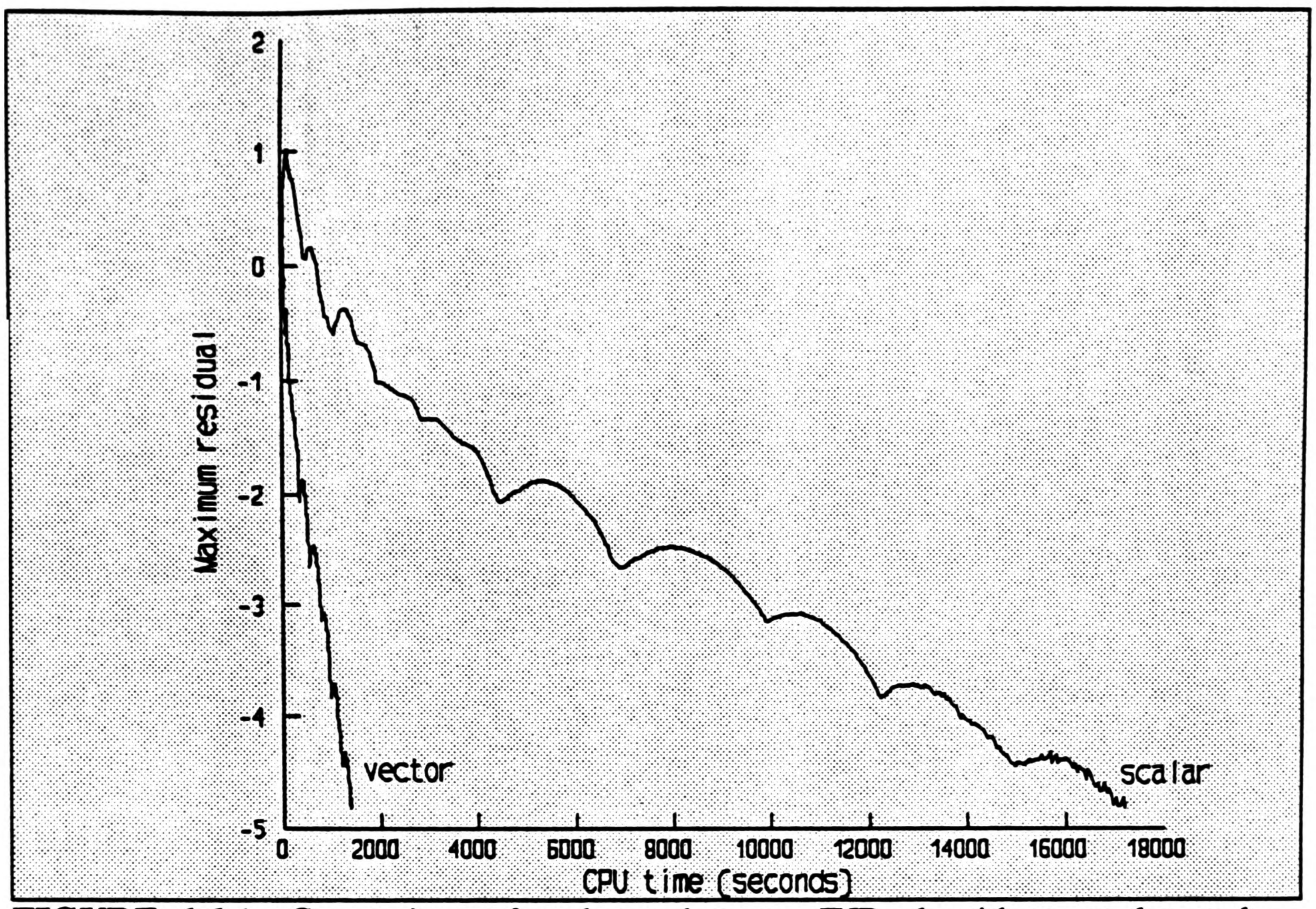


FIGURE 6.6-1 Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the L-shaped flow problem

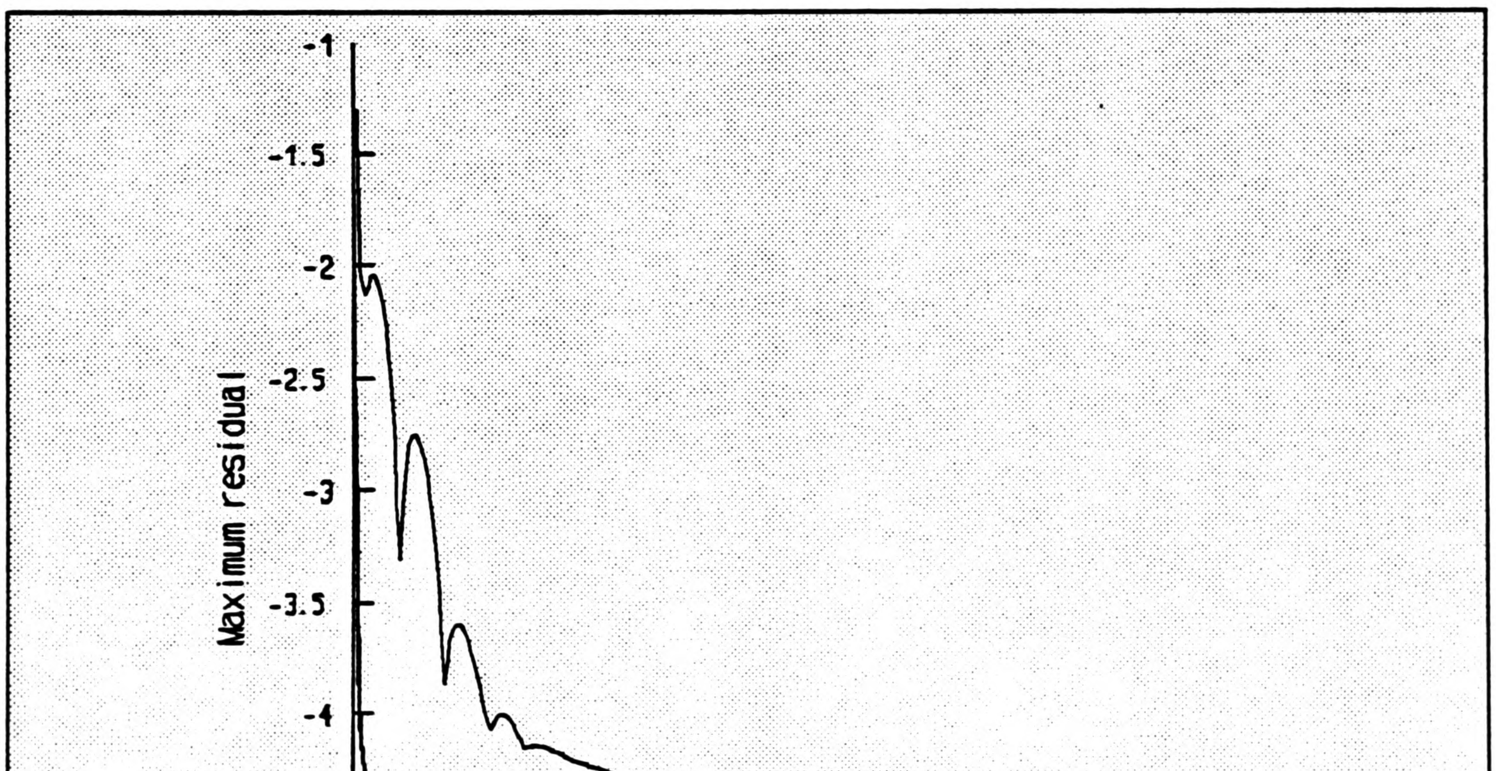


FIGURE 6.6-2a Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the natural convection problem (Ra=10^3)

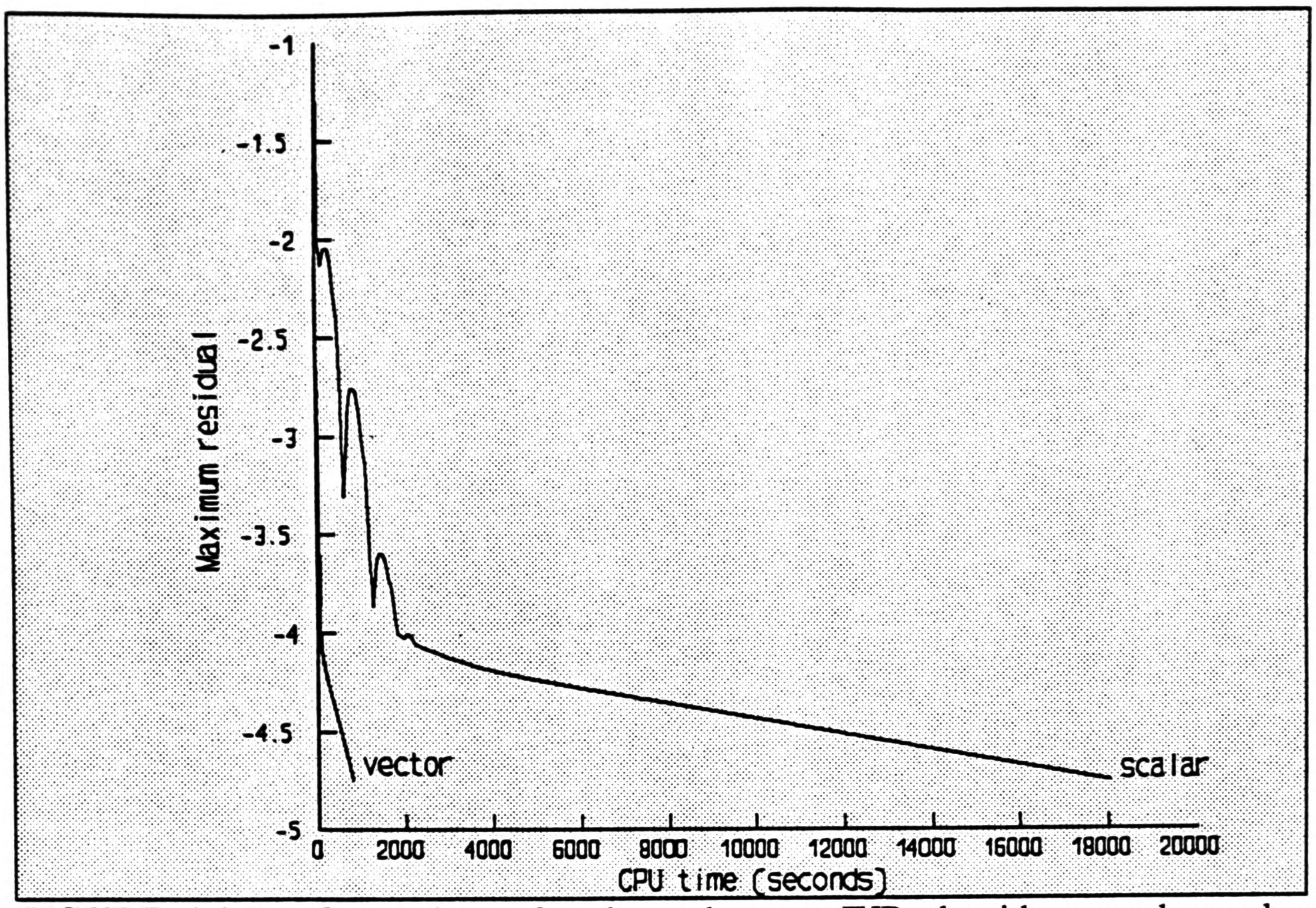


FIGURE 6.6-2b Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the natural convection problem (Ra=10^4)

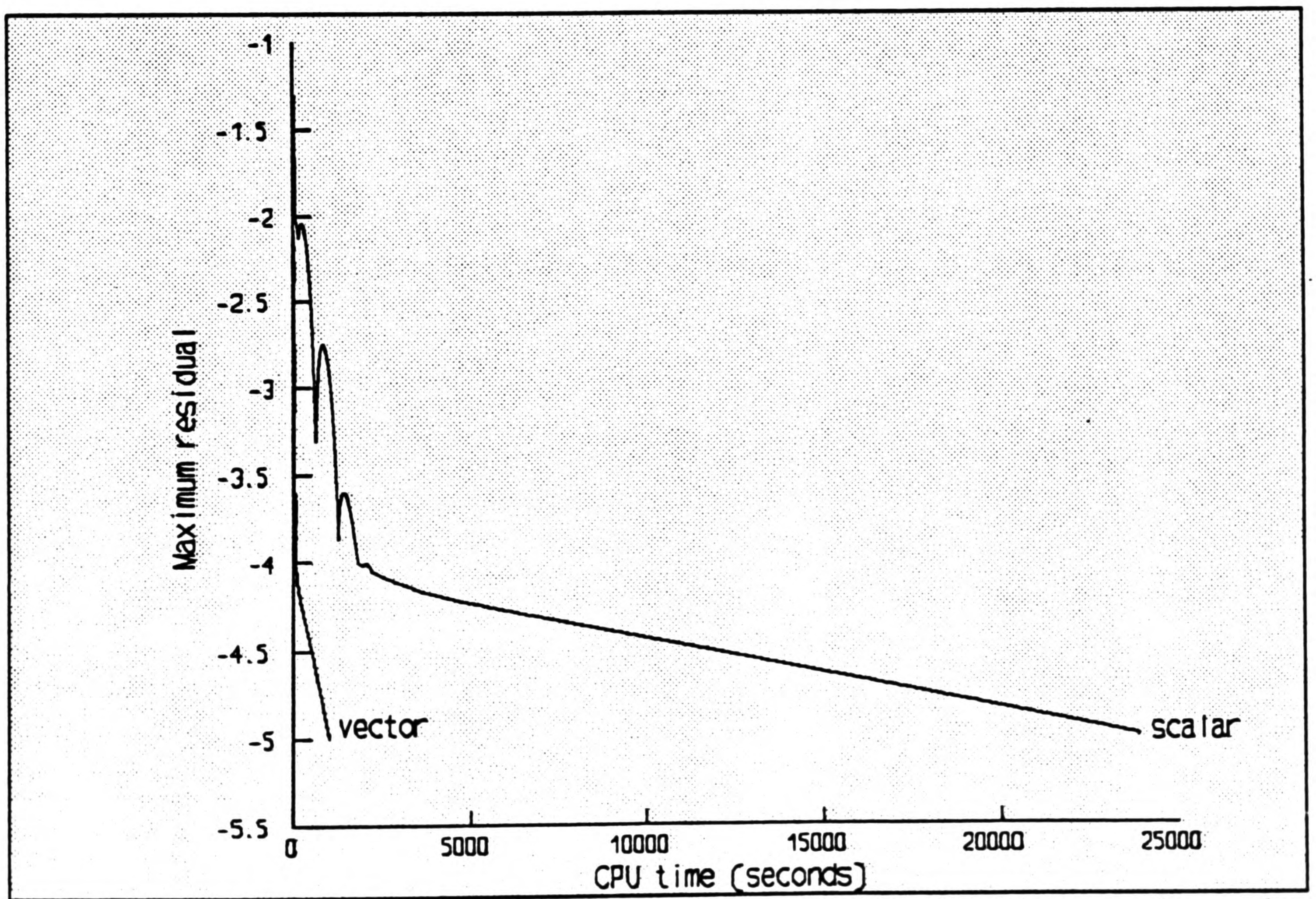


FIGURE 6.6-2c Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the natural convection problem (Ra=10^5)

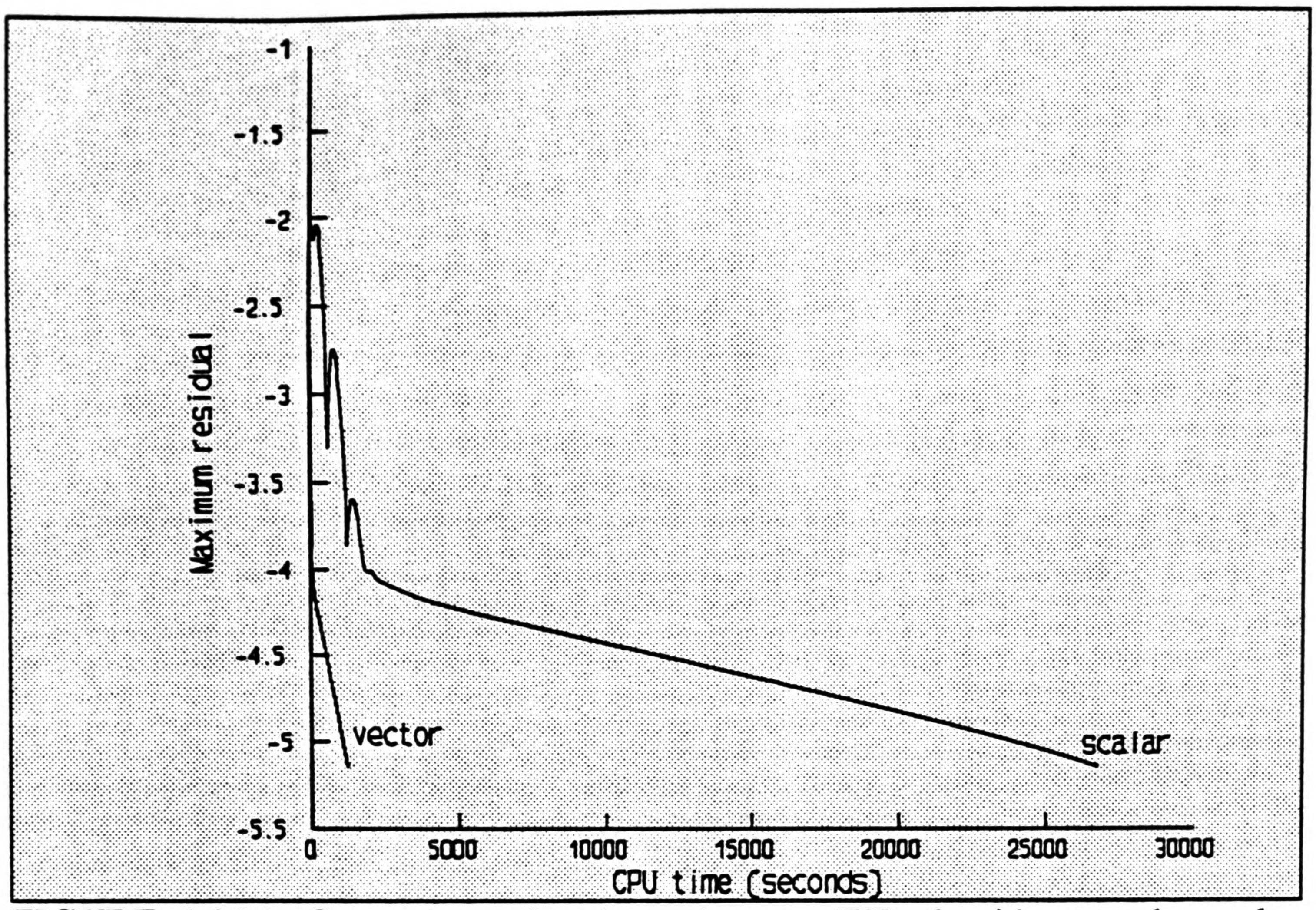


FIGURE 6.6-2d Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the natural convection problem (Ra=10^6)

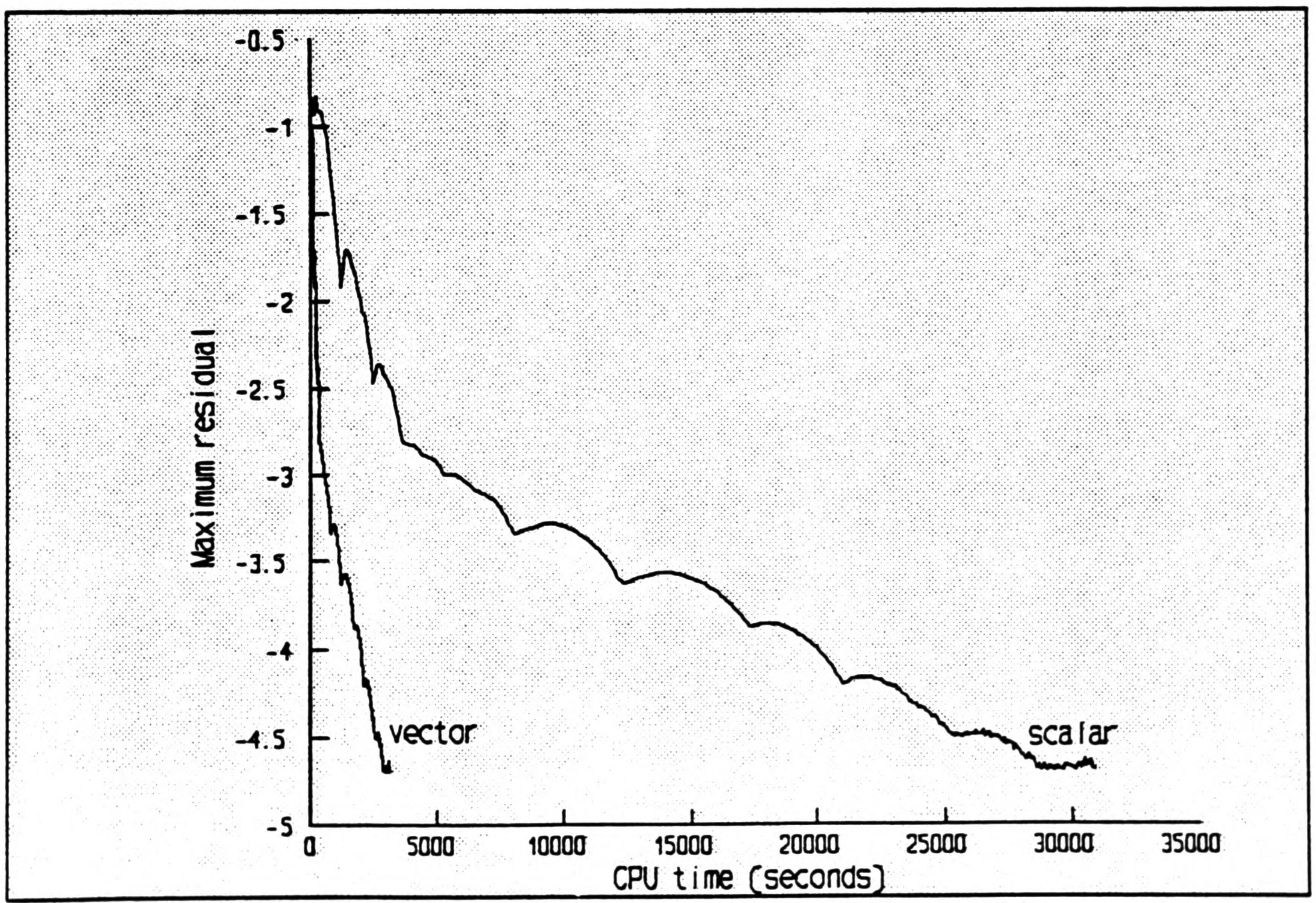


FIGURE 6.6-2e Comparison of scalar and vector JUR algorithms used to solve the pressure-correction equation in the natural convection problem (Ra=10^7)

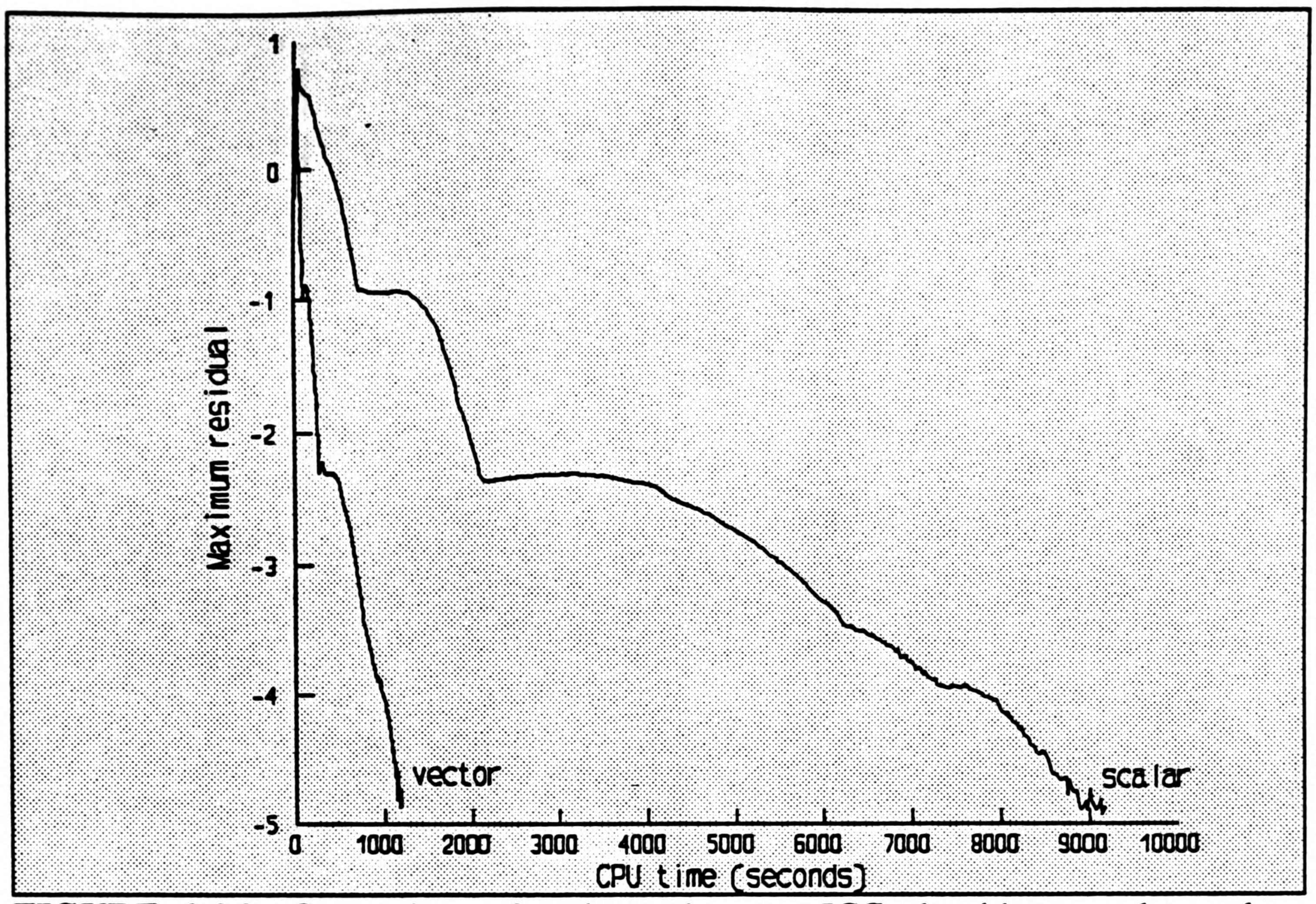


FIGURE 6.6-3 Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the L-shaped flow problem

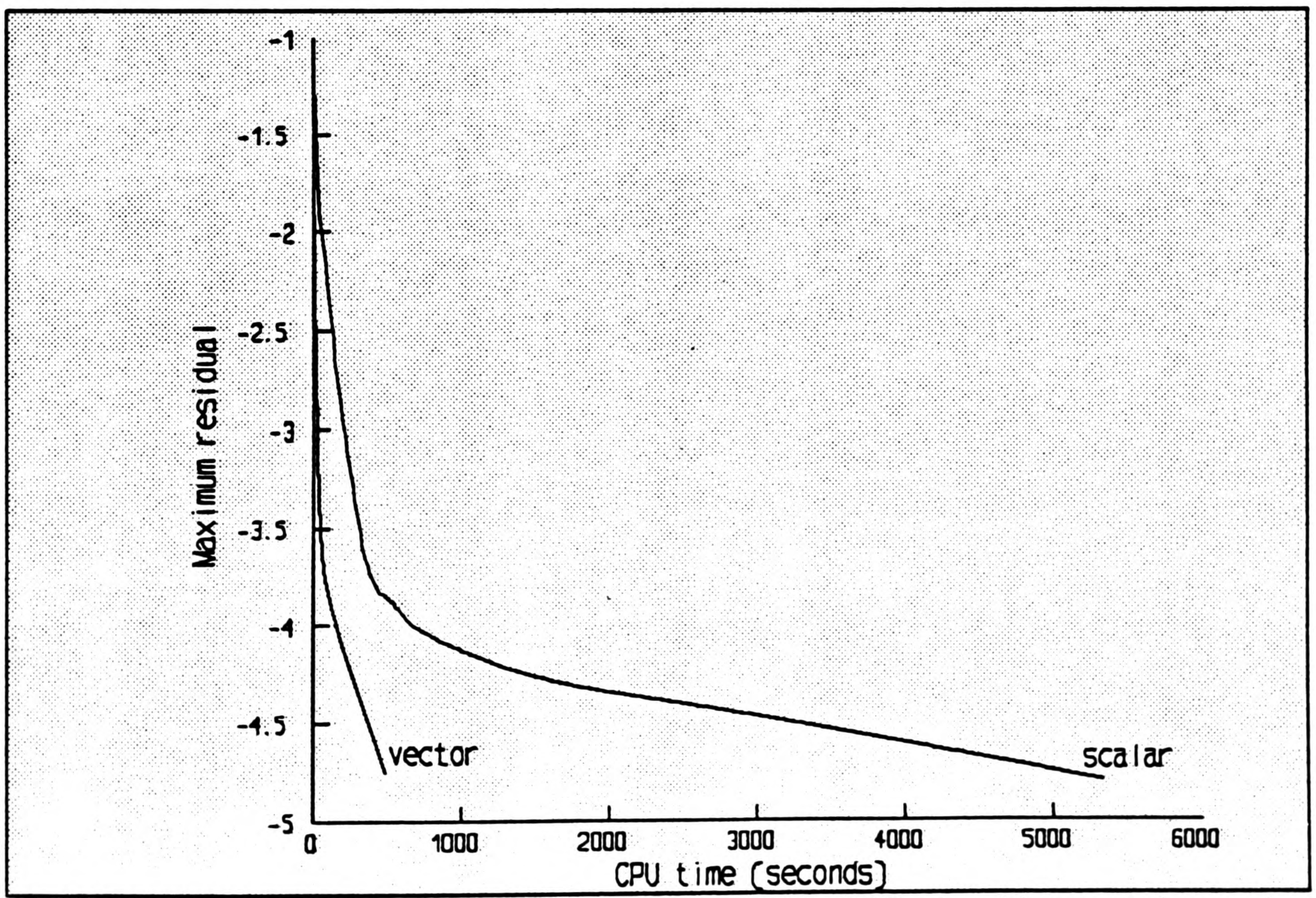


FIGURE 6.6-4a Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the natural convection problem (Ra=10^3)

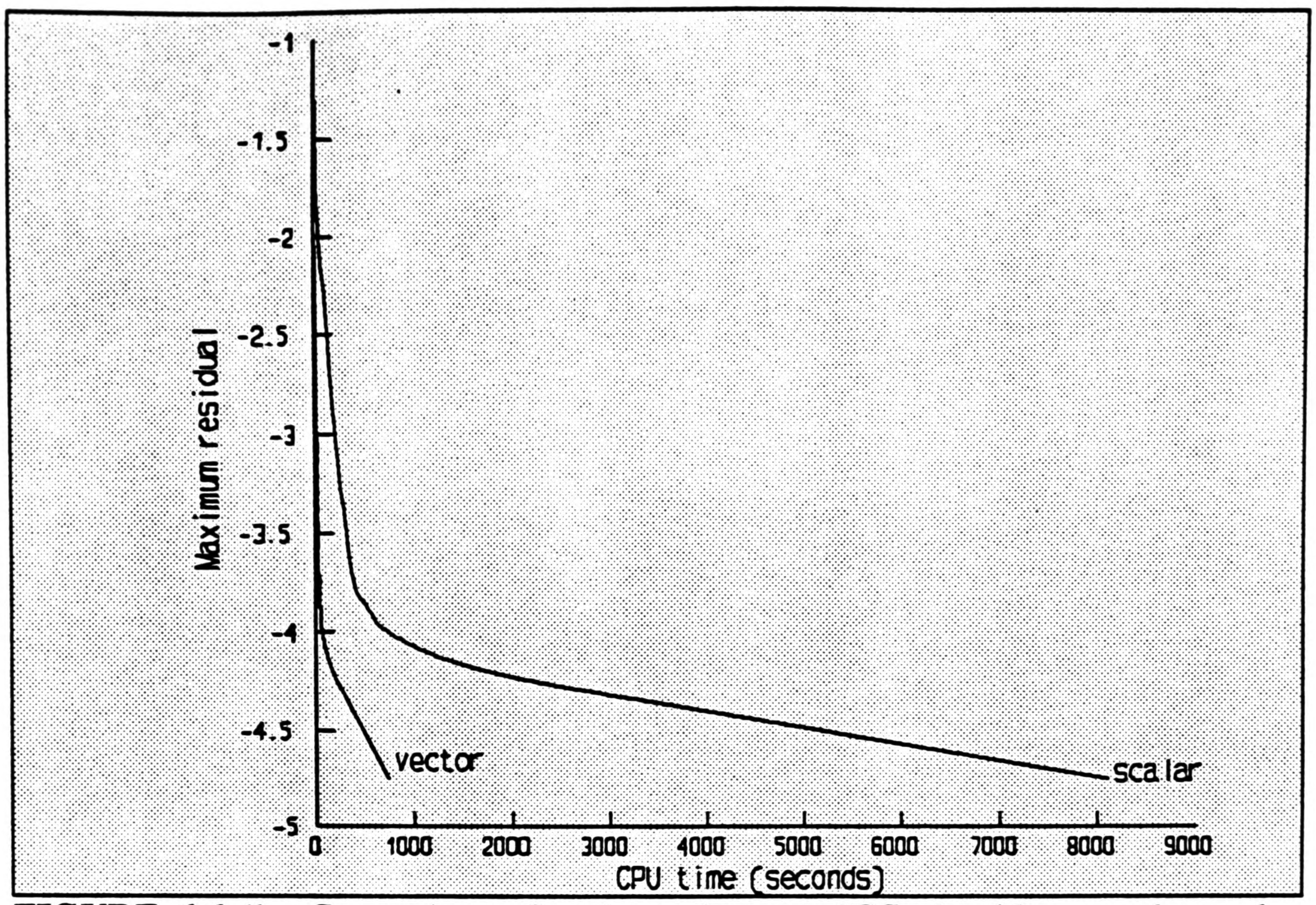


FIGURE 6.6-4b Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the natural convection problem (Ra=10^4)

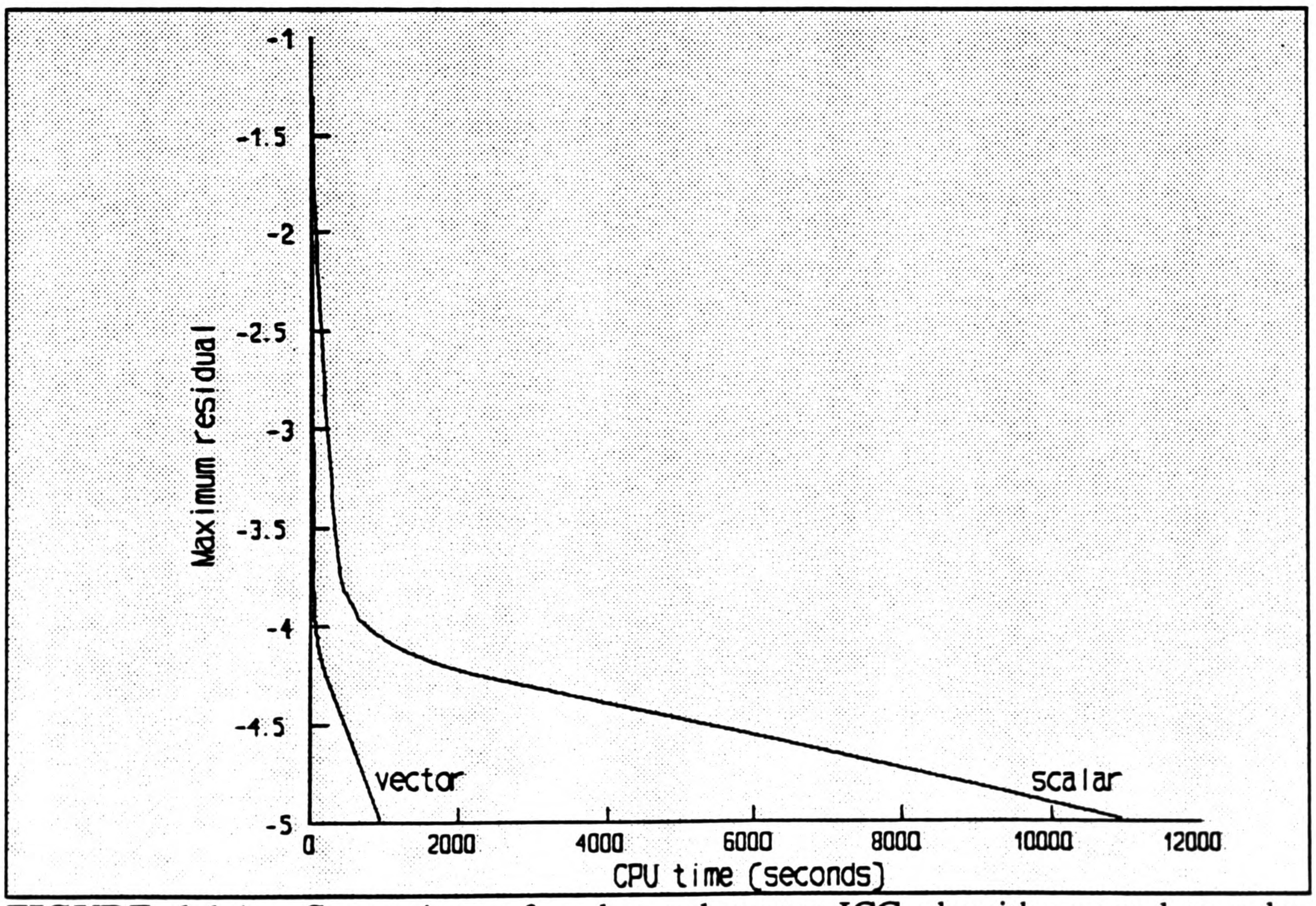


FIGURE 6.6-4c Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the natural convection problem (Ra=10^5)

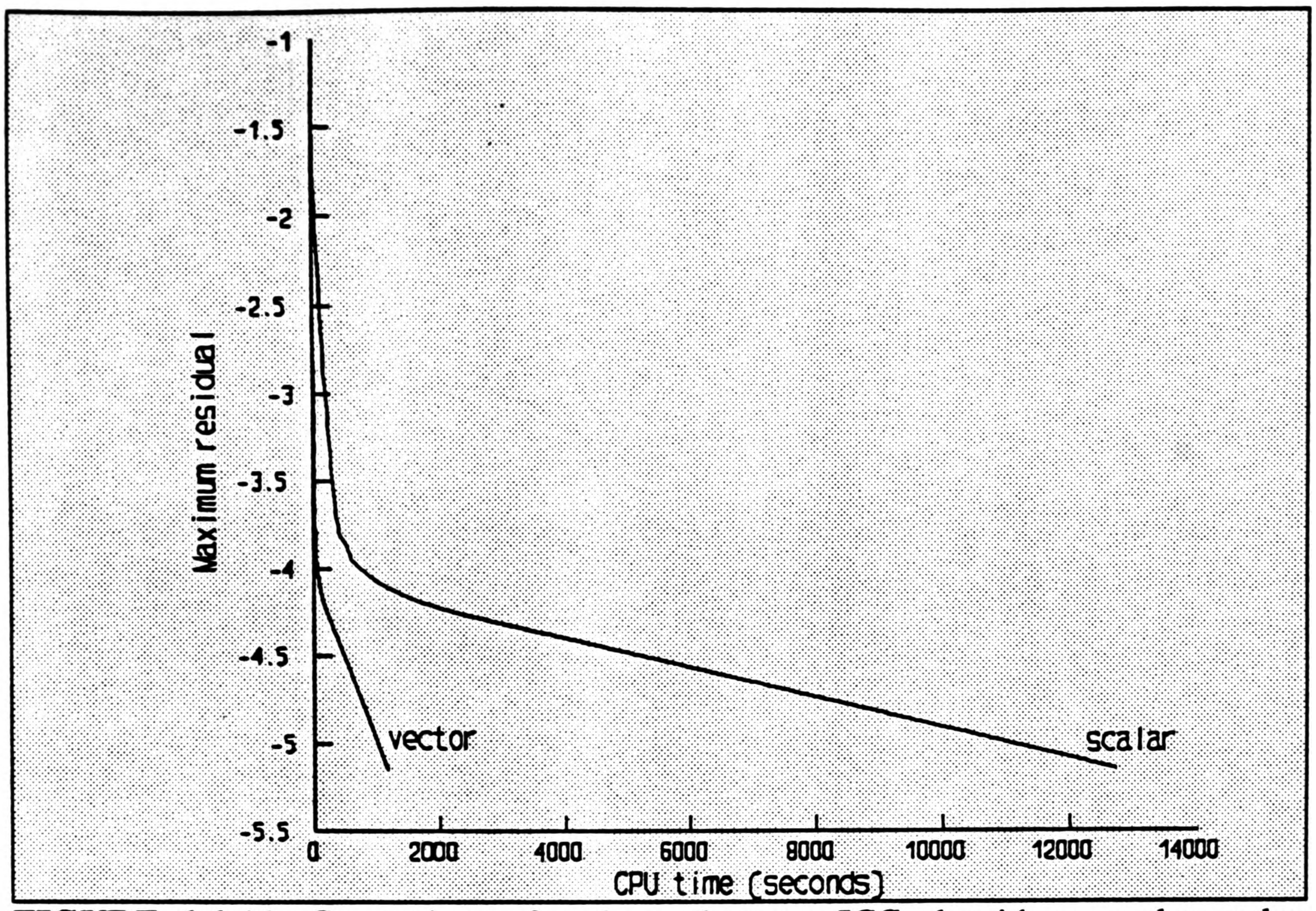


FIGURE 6.6-4d Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the natural convection problem (Ra=10^6)

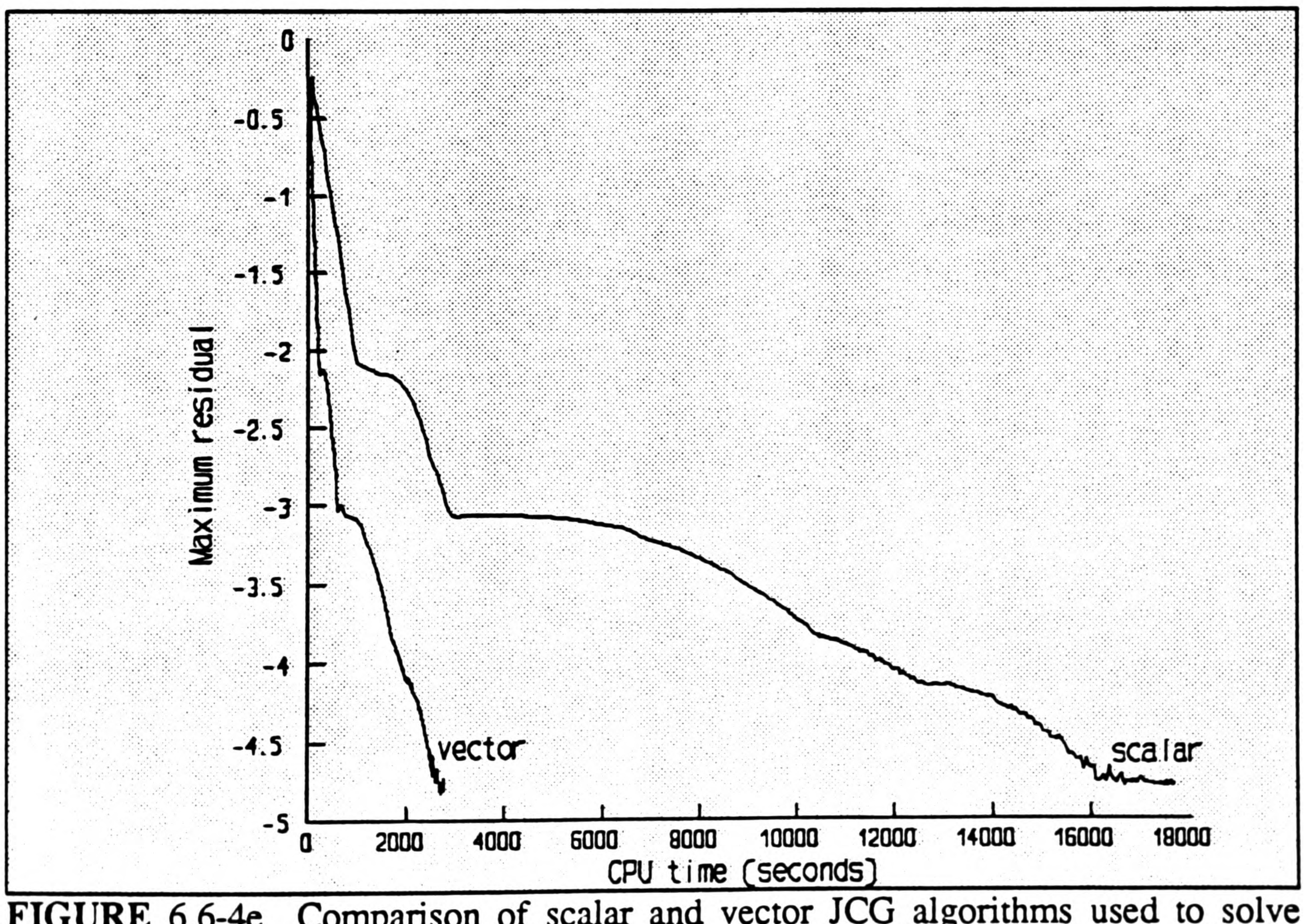


FIGURE 6.6-4e Comparison of scalar and vector JCG algorithms used to solve the pressure-correction equation in the natural convection problem (Ra=10^7)

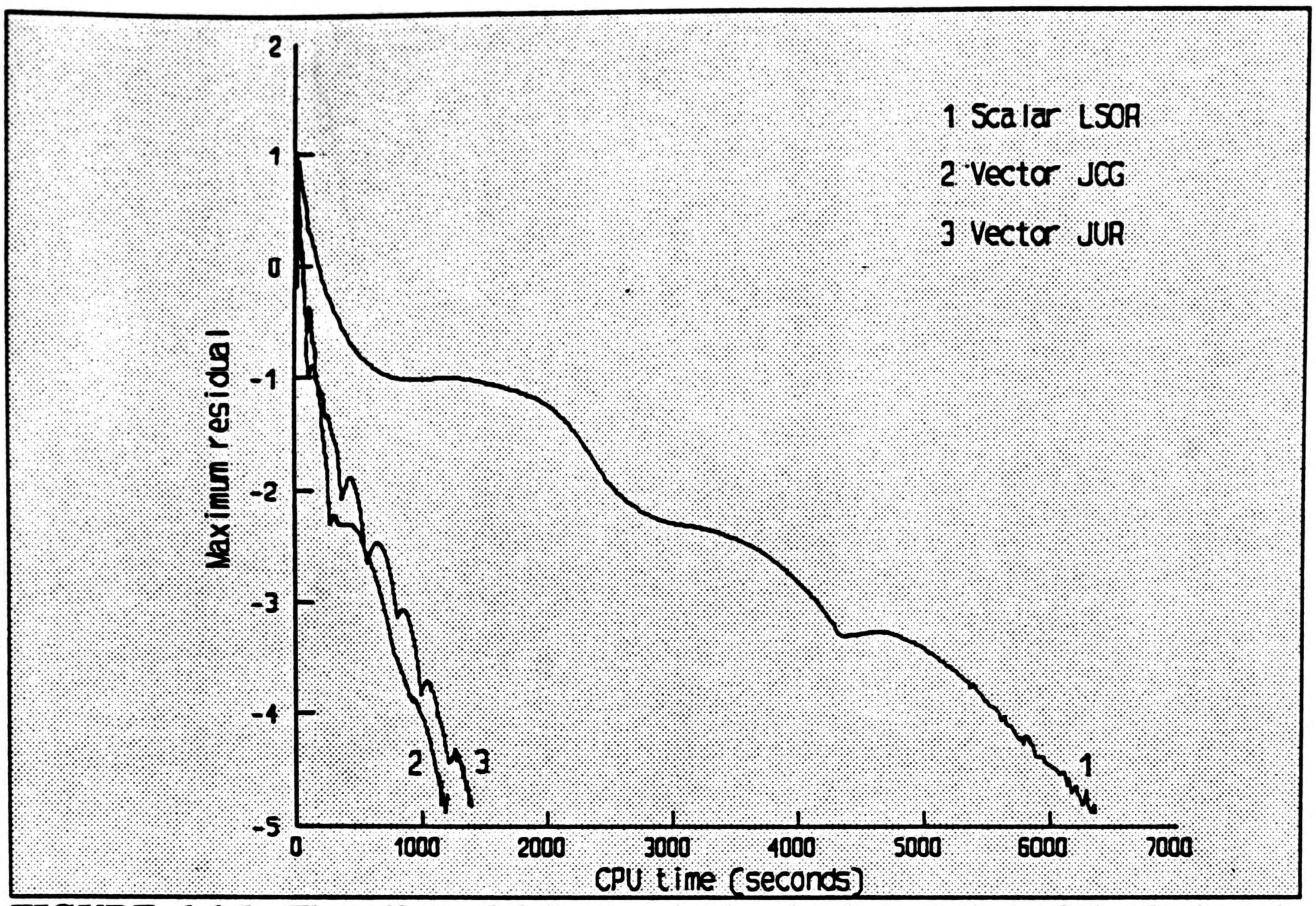


FIGURE 6.6-5 The effect of full vectorisation in the solution of the L-shaped flow problem. JUR is used to solve the momentum equations.

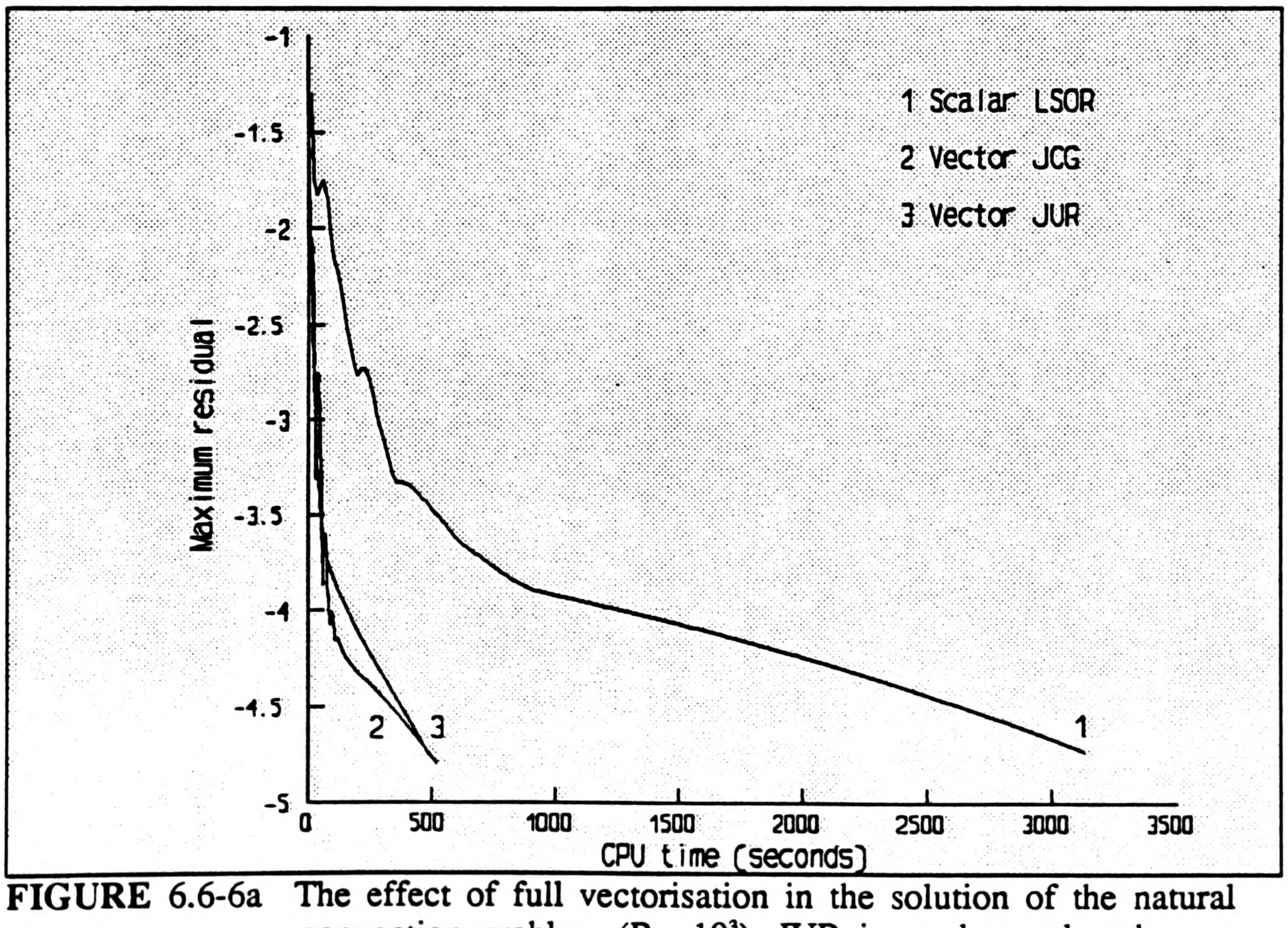


FIGURE 6.6-6a The effect of full vectorisation in the solution of the natural convection problem (Ra=10^3). JUR is used to solve the momentum equations.

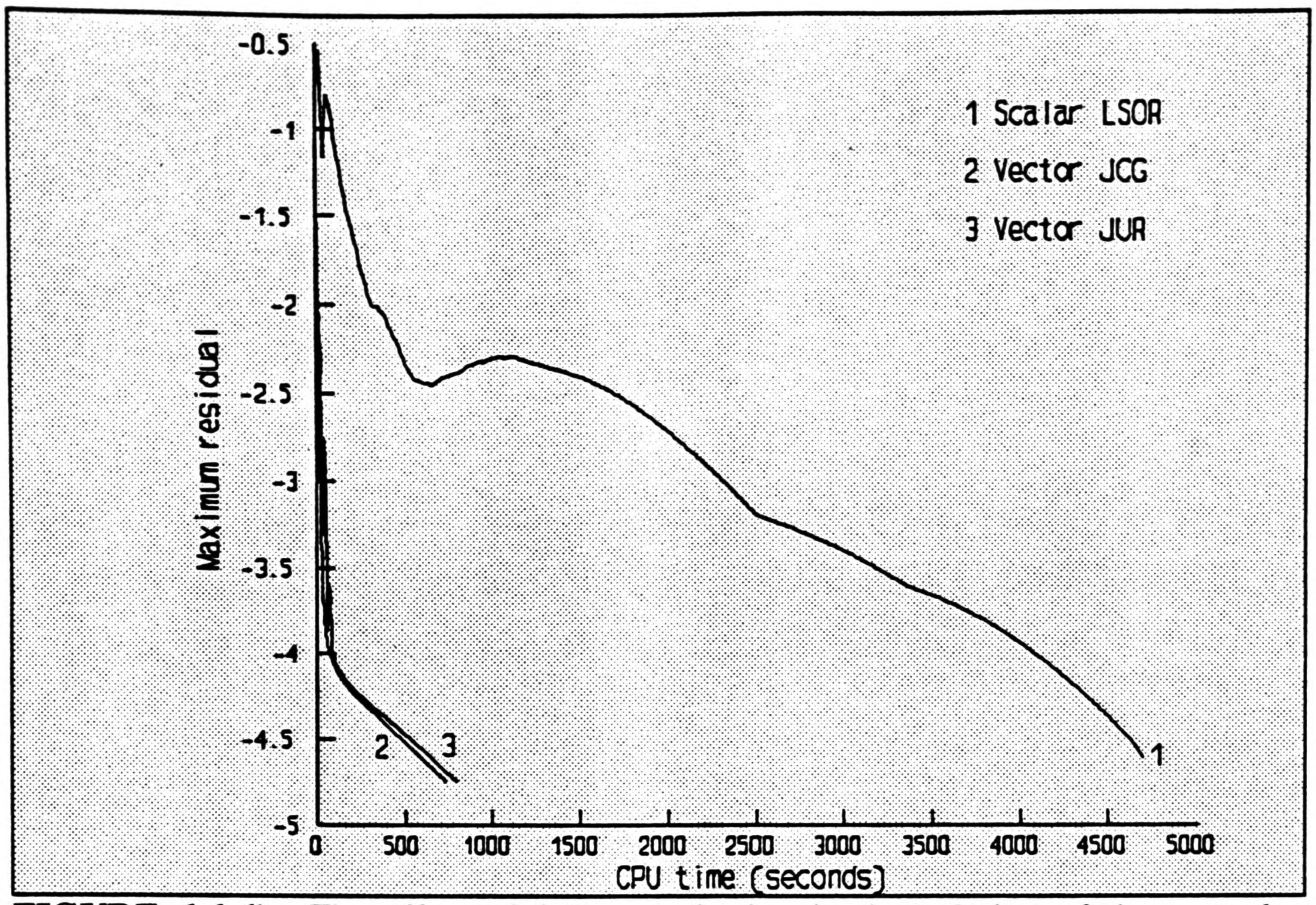


FIGURE 6.6-6b The effect of full vectorisation in the solution of the natural convection problem (Ra=10^4). JUR is used to solve the momentum equations.

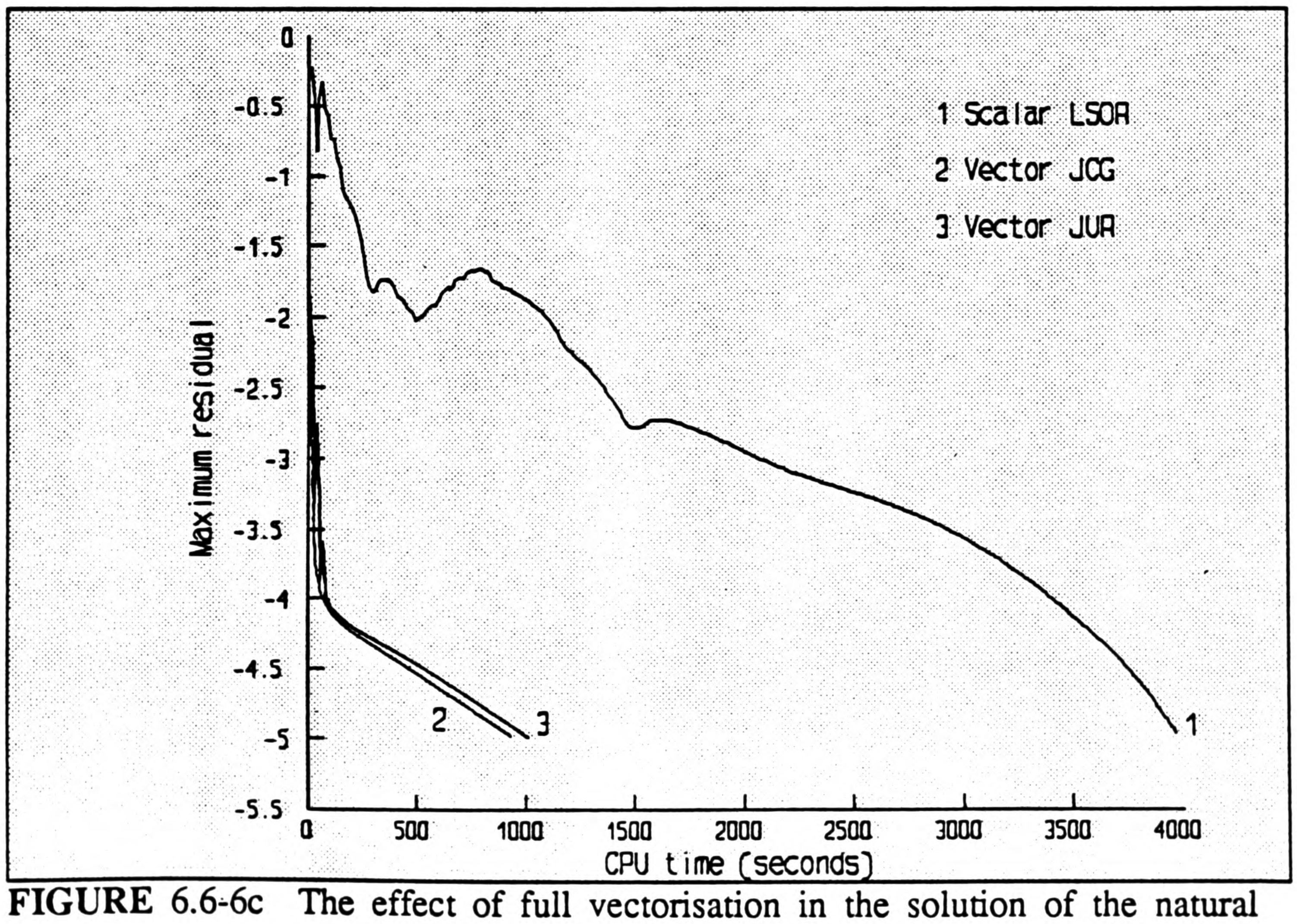


FIGURE 6.6-6c The effect of full vectorisation in the solution of the natural convection problem (Ra=10^5). JUR is used to solve the momentum equations.

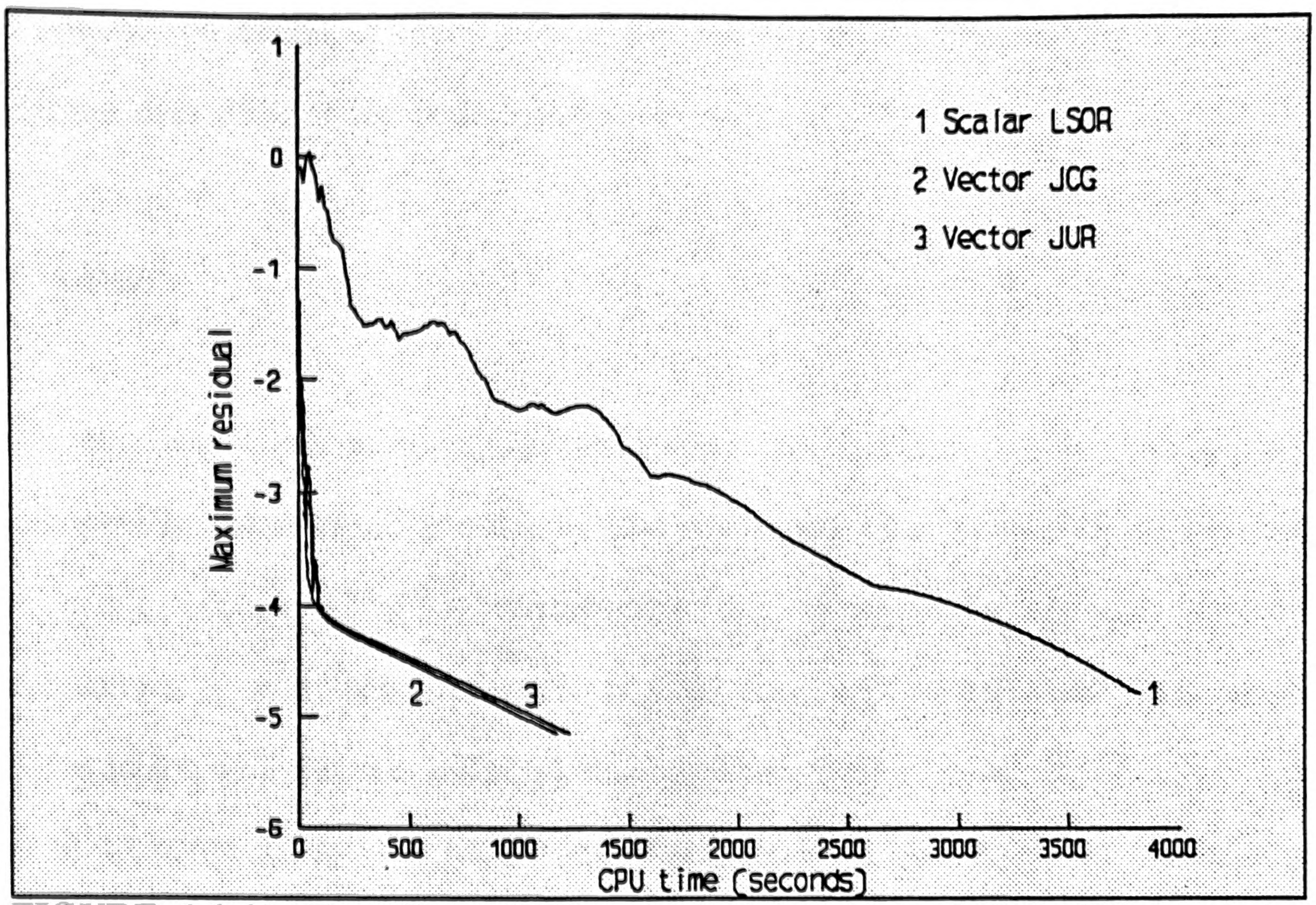


FIGURE 6.6-6d The effect of full vectorisation in the solution of the natural convection problem (Ra=10^6). JUR is used to solve the momentum equations.

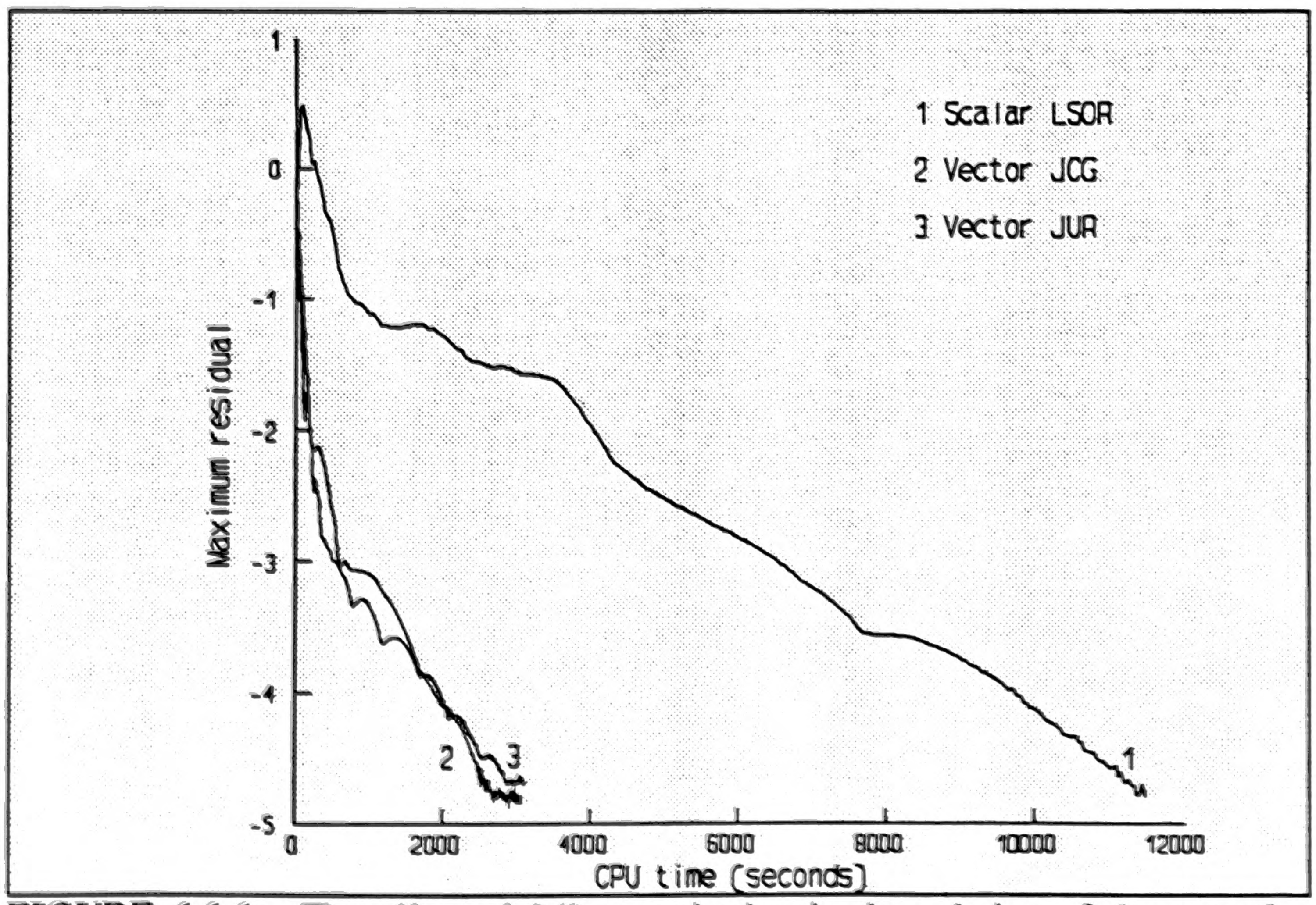


FIGURE 6.6-6e The effect of full vectorisation in the solution of the natural convection problem (Ra=10^7). JUR is used to solve the momentum equations.

the natural convection problem reduces the benefits of vectorisation by a factor of 2 despite over 85% of the code being vectorised.

6.7 Closure

The introduction of scalar quantities causes an increase in the total scalar component in the SIMPLE procedure. This can account for about 10% of the code not being vectorised, as a consequence the speed-up factors are lowered. Nevertheless, a worthwhile reduction in CPU time can be achieved. This ranges from a factor of 5 for the L-shaped flow problem (which solves for u, v, p, T, k and ε) to about 11 for the natural convection problem (which solves for u, v, p and T). The addition of turbulence in the latter problem, for $Ra=10^7$, gives a speed-up factor of 5.

CHAPTER SEVEN

7.0 THE IMPACT OF USING A MULTIGRID METHOD

7.1 Introduction

Although the classical iterative algorithms have served well, there is a general tendency for these algorithms to become less efficient as the number of nodes are increased. There are two main reasons for this trend, the first is due to an increase in the number of floating point operations which need to be performed per iteration. These are directly proportional to the total number of nodes $1/h^\alpha$ where h is the grid spacing and α is the number of dimensions. Secondly, there is an increase in the number of iterations needed to reduce the error in the approximation to a suitable level. The relationship between the number of iterations and the computation time is given by

$$\mathrm{CPU} \propto n^\beta$$

where n is the number of nodes and β is greater than 1. Ideally, the relationship between the time and the number of nodes should be linear. Whilst pipeline processing has been used to overcome the former problem (Chapters 4 and 5), it does not address the latter problem. To this extent, the concept of using a number of different grids called *multigrids* is considered.

The general feature of a classical iterative algorithm is such that although the initial convergence is rapid, it soon slows down and can become inefficient. The performance of these algorithms can be explained within the context of the errors present in the approximate solution. As the number of nodes are increased the convergence difficulties become more apparent. In the initial iterations the algebraic equations are solved locally and this causes a significant reduction in the local errors. This is because the errors eliminated have a wavelength λ which is of the same order as the grid spacing h. The deteriorating performance of the algorithm in the latter stages is due to the poor way in which the errors of wavelength $\lambda >> h$ are eliminated.

Since the iterative algorithms are most efficient for errors with magnitude equal to the grid spacing, it is worth considering a technique which will take advantage of this. Errors which cannot be effectively reduced on a given grid, may be handled more effectively on a coarser grid. Taking a series of such grids can lead to optimal use of the iterative algorithm - this is the essence of the multigrid method. The major advantage of a multigrid method is the reduction in the computation time which is a direct result of the arithmetic being performed on the coarser grids.

Amongst the first authors to discuss such an approach were Fedorenko [1962] and Bakhvalov [1966], but it was not until the late seventies when two independent schools of thought emerged as to the importance of multigrid methods. One concentrates on the convergence rate properties and characteristics of multigrid methods (Hackbusch [1978]), the other considers the practical application of multigrid methods in the solution of systems of algebraic equations (Brandt et al [1977, 1979, 1980, 1982]). The latter has become more popular and has done much to revive the popularity of existing classical iterative algorithms. Much of the work is largely based on the solution of linear problems, but more recently they have been used to solve non-linear problems. To place the present work into context some of the popular approaches are outlined.

Brandt and Dinar [1977] describe a distributed Gauss-Seidel method (DGS) coupled with a full approximation storage scheme (FAS) to solve a highly elliptic fluid flow system. After the relaxation of the velocity components a 'distributive' relaxation is used to update the pressure, and continuity is satisfied by adding corrections to the velocity and pressure fields. The DGS method has also been used by Fuchs [1983] employing a primitive variable formulation.

Vanka [1986] suggested a block-implicit method called Symmetrical Coupled Gauss-Seidel (SCGS). This solved the velocity and pressure fields simultaneously at each node using a staggered grid. Unlike the DGS approach which solves the unknowns by decoupling them, the SCGS maintains the coupling between the unknowns. The SCGS method has been used by Gaskell and Wright [1988] together with the FAS scheme to solve recirculating flow problems. It has also been considered for vectorisation by Vanka and Misengades [1987].

7.2 The SIMPLE-based procedure as a multigrid smoother

Sivaloganathan and Shaw [1988a] used a local mode Fourier analysis to assess the performance of pressure-linked procedures as multigrid smoothers. This work was later supplemented with a fluid flow example involving the solution of a shear-driven cavity problem (Sivaloganathan and Shaw [1988b]). In the formulation a staggered grid was used with the FAS scheme. A single coarse grid control-volume was made up of four fine grid control-volumes, for a bi-grid structure the outline of the procedure is given by

(a) Apply a given number of sweeps to the SIMPLE procedure (pre-smoothing)

(b) Set up and solve the coarse grid problem using the SIMPLE procedure. Then transfer the corrections to the fine grid solution (coarse grid correction)

(c) Re-apply a given number of sweeps of the SIMPLE procedure (post-smoothing).

For a series of coarser grids the process can be used to solve the equations at stage (b) and repeated until the coarsest grid is reached. On the coarsest grid the equations are solved to convergence. Lonsdale [1988] applied a similar technique (using SIMPLEC) to the steady state solution of fluid flow between two corotating discs.

7.3 SIMPLE-based procedures using multigrids as a linear solver

The SIMPLE-based procedures can also use multigrid methods as linear equation solvers. They can be used to determine the solution of algebraic equations which result from discretisation, such as the momentum, continuity and scalar equations.

Phillips and Schmidt [1984] considered the solution of the diffusion equation using a multigrid method. The motivation for using a multigrid method was that an accurate solution was needed, however, the accuracy of the solution could be affected by the presence of regions containing large gradients. The multigrid method was used in two ways. Firstly, the domain was covered with coarse grids, and secondly, selected regions were covered with fine grids where large gradients were suspected in the dependent variable. The work was extended to solve the advection-diffusion equation (Phillips and Schmidt [1985a]) and also to recirculating flow problems (Phillips and Schmidt [1985b]). However, the emphasis was on the accuracy of the solutions rather than the efficiency of the multigrid methods.

Miller and Schmidt [1988] used the SIMPLEC procedure to solve two-dimensional fluid flow problems considering the Gauss-Seidel, LSOR and Stone's strongly implicit algorithms. The multigrid method of Phillips and Schmidt [1985b] was implemented for the solution of a two-dimensional 'lid-driven cavity' problem and the 'sudden contraction in a pipe' problem. In all cases a 32x32 grid was used and reductions in CPU time were reported for the Gauss-Seidel and Stone's algorithms. In the cavity problem the best reductions (in work units) were about 37% and for the sudden contraction problem a factor of five was achieved.

7.4 The additive correction multigrid method (ACM)

Hutchinson and Raithby [1986] describe an additive correction technique which is a generalisation of the block correction method proposed by Settari and Aziz [1973]. The ACM method is essentially a multigrid method and has much in common with the classical methods of Brandt [1977]. For example, both are used to accelerate the convergence rates of iterative algorithms by using a series of coarser meshes. The ACM method forms the coarse grid equations by ensuring integral conservation over the coarse block of control-volumes, Brandt requires the discretisation of the governing equation on the coarse grids. In the ACM method there is a physical significance to the formulation of the coarse grid equations and the corrections from the coarse grid are added to the fine grid solution, Brandt carries out an interpolation of the coarse grid corrections up to the fine grid. Finally, the ACM method is not restricted to coarse blocks made up of 2x2 fine grid control-volumes and requires no special treatment of boundary conditions on the coarse grids.

In the ACM method the iterative process is accelerated by adding corrections to blocks of control-volumes. These corrections do not ensure that the residual in the discretised equation is satisfied at the fine level control-volume, but rather conservation is preserved over the coarse block.

7.5 The ACM method applied to the pressure-correction equation

Hutchinson and Raithby [1986] only considered the solution of a single conserved variable and showed that significant reductions can be made when the ACM method is used. However, it has been shown by Ierotheou, Richards and Cross [1988, 1989b] that the method can be equally applied to the solution of the pressure-correction equation as part of the whole-field SIMPLE solution procedure.

7.5.1 The one-dimensional ACM method

The ACM method is first described for a one-dimensional flow situation because of its simplicity and with reference to a bi-grid (one fine mesh and one coarse mesh). Consider *n* control-volumes in the x cartesian direction, then the discretised pressure-correction equation can be expressed as

$$a_i^W p'_{i-1} - a_i^P p'_i + a_i^E p'_{i+1} + b_i = 0 \qquad i=1(1)n \qquad (7.5.1\text{-}1)$$

where

$$a_i^P = a_i^E + a_i^W$$

The coefficients a_i^P, a_i^E and a_i^W are related to the central coefficient in the momentum equation and b_i is related to the residual in the discretised continuity equation. At some stage during the iterative process the pressure-correction equation is represented by

$$r_i = a_i^W p_{i-1}^{\prime *} - a_i^P p_i^{\prime *} + a_i^E p_{i+1}^{\prime *} + b_i \qquad i=1(1)n \qquad (7.5.1\text{-}2)$$

where $p_i^{\prime *}$ represents the latest approximation to $p_i^{\prime}$ and r_i is the residual.

A coarse grid is used only if the iterative process becomes inefficient. For convenience the number of control-volumes are assumed to be even, then the coarse grid has $n/2$ blocks. Each block is made up of two adjacent control-volumes, thus block k comprises control-volumes i and i+1 and is related by index as 2k=i+1 (figure 7.5.1-1). Adding the two residual equations in block k gives

$$a_i^W p_{i-1}^{\prime *} + (a_{i+1}^W - a_i^P) p_i^{\prime *} + (a_i^E - a_{i+1}^P) p_{i+1}^{\prime *} + a_{i+1}^E p_{i+2}^{\prime *} + b_i + b_{i+1} = r_i + r_{i+1}$$
$$i=1(1)n \qquad (7.5.1\text{-}3)$$

This now represents the residual within the block. To ensure that the residual in the block is zero (even though the fine grid residual may not yet be zero), a correction δ_k is added to each pressure-correction approximation in block k. This gives

$$a_i^W (p_{i-1}^{\prime *} + \delta_{k-1}) + (a_{i+1}^W - a_i^P)(p_i^{\prime *} + \delta_k) + (a_i^E - a_{i+1}^P)(p_{i+1}^{\prime *} + \delta_k) + a_{i+1}^E (p_{i+2}^{\prime *} + \delta_{k+1}) + b_i + b_{i+1} = 0$$
$$i=1(1)n$$
$$k=1(1)n/2 \qquad (7.5.1\text{-}4)$$

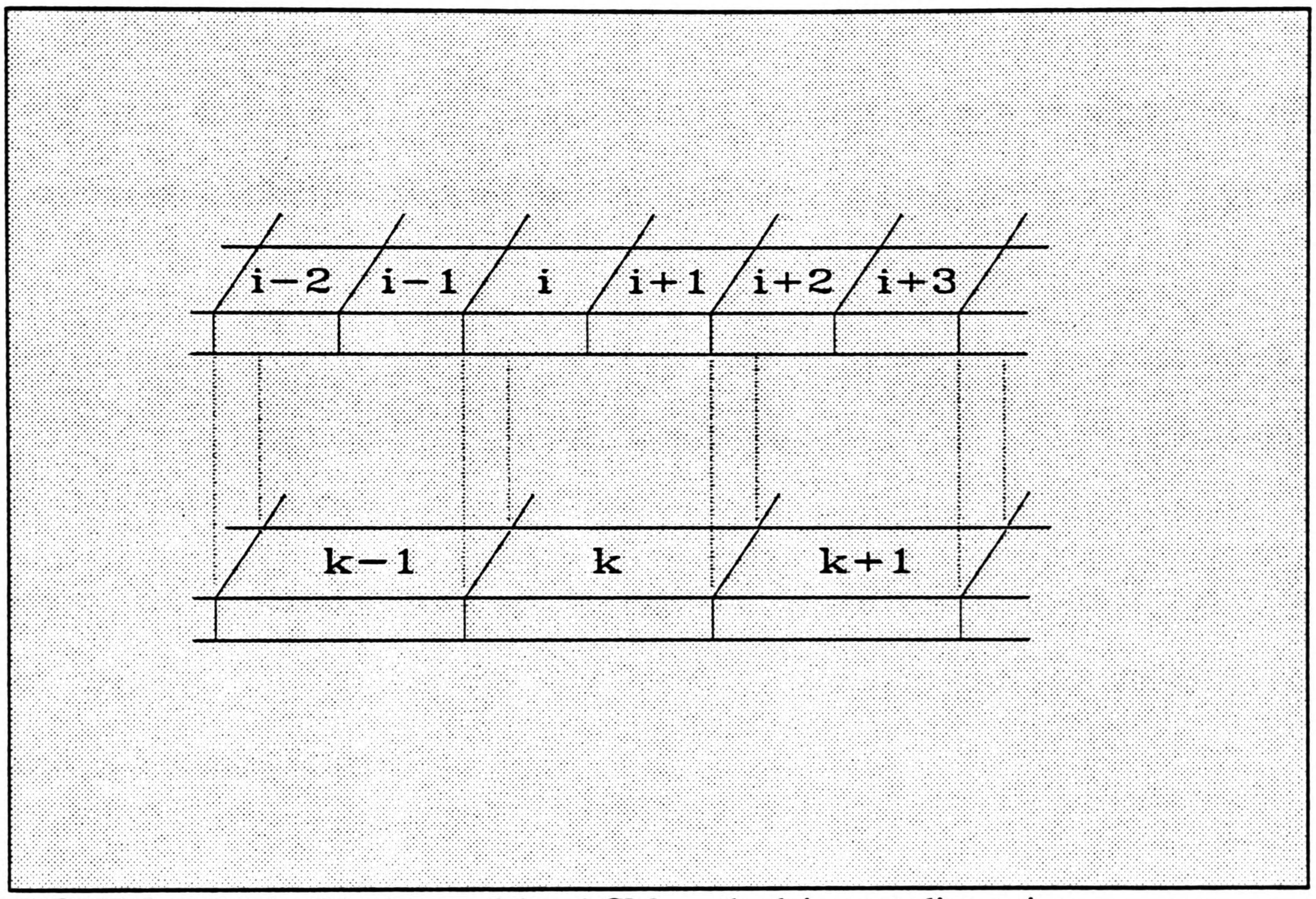


FIGURE 7.5.1-1 Blocks used by ACM method in one dimension

This can now be written in a form representative of the coarse grid only,

$$A_k^W\delta_{k-1} - A_k^P\delta_k + A_k^E\delta_{k+1} + B_k = 0 \qquad k=1(1)n/2 \qquad (7.5.1\text{-}5)$$

where

$$A_k^E = a_{i+1}^E$$

$$A_k^W = a_i^W$$

$$A_k^P = A_k^E + A_k^W$$

$$B_k = r_i + r_{i+1}$$

At this stage equation (7.5.1-5) is solved using a direct or iterative algorithm and the correction transferred back to the fine grid. The updated corrections are given by

$$p_i' = p_i'^{*} + \delta_k \qquad (7.5.1\text{-}6a)$$

$$p_{i+1}' = p_{i+1}'^{*} + \delta_k \qquad (7.5.1\text{-}6b)$$

However, the multigrid process can be repeated for the coarser grids if necessary since equation (7.5.1-5) has the same form as (7.5.1-1) and can therefore be treated in a similar manner.

7.5.2 The two-dimensional ACM method

There are now an even number of control-volumes in the y-direction given by *m*. The two-dimensional discretised pressure-correction equation is expressed as

$$a_{ij}^W p_{i-1j}' + a_{ij}^S p_{ij-1}' - a_{ij}^P p_{ij}' + a_{ij}^E p_{i+1}' + a_{ij}^N p_{ij+1}' + b_{ij} = 0 \qquad \begin{matrix} i=1(1)n \\ j=1(1)m \end{matrix} \qquad (7.5.2\text{-}1)$$

where

$$a_{ij}^P = a_{ij}^W + a_{ij}^S + a_{ij}^E + a_{ij}^N$$

At some stage during the iterative process the pressure-correction equation is updated by

$$r_{ij} = a^{W}_{ij}p'^{*}_{i-1j} + a^{S}_{ij}p'^{*}_{ij-1} - a^{P}_{ij}p'^{*}_{ij} + a^{E}_{ij}p'^{*}_{i+1j} + a^{N}_{ij}p'^{*}_{ij+1} + b_{ij}$$

$$i=1(1)n$$
$$j=1(1)m \quad (7.5.2\text{-}2)$$

Again, use is made of a coarse grid which has blocks made up of four neighbouring control-volumes from the fine grid, the coarse grid then has a total of *n*/2x*m*/2 blocks. Block kl comprises control-volumes ij, i+1j, ij+1 and i+1j+1 and is related by index as 2l=j+1 (figure 7.5.2-1). The addition of the four relevant residual equations in block kl gives

$$A^{W}_{kl}\delta_{k-1l} + A^{S}_{kl}\delta_{kl-1} - A^{P}_{kl}\delta_{kl} + A^{E}_{kl}\delta_{k+1l} + A^{N}_{kl}\delta_{kl+1} + B_{kl} = 0$$

$$k=1(1)n/2$$
$$l=1(1)m/2 \quad (7.5.2\text{-}3)$$

where

$$A^{E}_{kl} = a^{E}_{i+1j} + a^{E}_{i+1j+1}$$

$$A^{W}_{kl} = a^{W}_{ij} + a^{W}_{ij+1}$$

$$A^{N}_{kl} = a^{N}_{ij+1} + a^{N}_{i+1j+1}$$

$$A^{S}_{kl} = a^{S}_{ij} + a^{S}_{i+1j}$$

$$A^{P}_{kl} = A^{E}_{kl} + A^{W}_{kl} + A^{N}_{kl} + A^{S}_{kl}$$

$$B_{kl} = r_{ij} + r_{i+1j} + r_{ij+1} + r_{i+1j+1}$$

Equation (7.5.2-3) is solved and the solution δ_{kl} is used to correct the four control-volumes on the fine grid

$$p'_{ij} = p'^{*}_{ij} + \delta_{kl} \quad (7.5.2\text{-}4a)$$

$$p'_{ij+1} = p'^{*}_{ij+1} + \delta_{kl} \quad (7.5.2\text{-}4b)$$

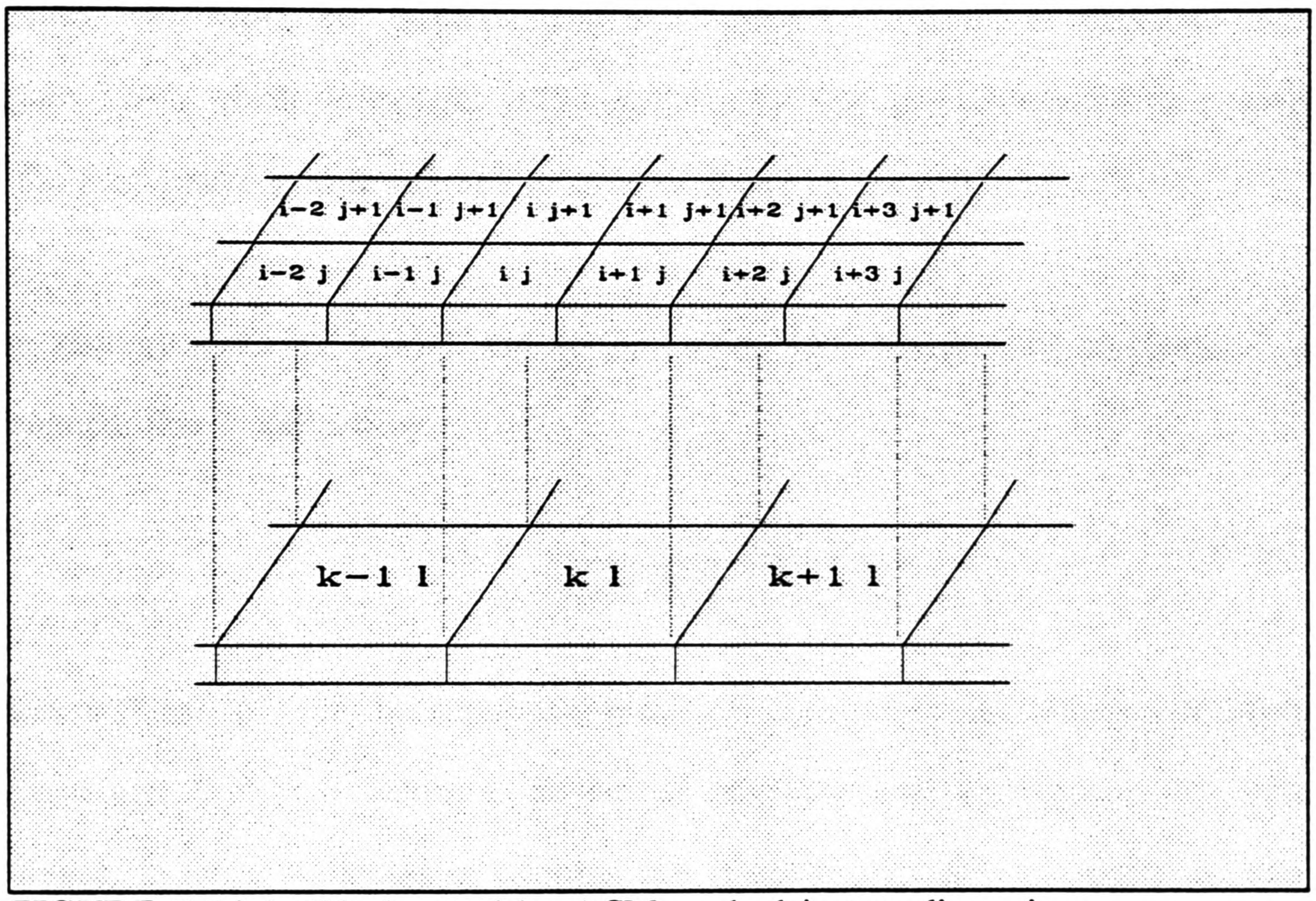


FIGURE 7.5.2-1 Blocks used by ACM method in two dimensions

$$p'_{i+1j} = p'^{*}_{i+1j} + \delta_{kl} \qquad (7.5.2\text{-}4c)$$

$$p'_{i+1j+1} = p'^{*}_{i+1j+1} + \delta_{kl} \qquad (7.5.2\text{-}4d)$$

7.6 The flexible cycle C strategy

There are several algorithms for carrying out the basic multigrid idea and each with their own variations. Brandt [1977] suggests three different algorithms, the most favourable is referred to as the *cycle C*. This strategy is adopted here together with some modifications. The basic cycle C is shown in flowchart form (figure 7.6-1) which utilises the ACM method, the fine grid is referred to as level 1, the first coarse grid is referred to as level 2 and so on.

The strategy starts on level 1 and at the end of each iteration the parameter ζ is evaluated, this is defined by

$$\zeta = \frac{\| \phi^{new} - \phi^{old} \|_2}{\mathrm{Max}\{10^{-10}, \| \phi^{new} \|_2\}} \qquad (7.6\text{-}1)$$

where ϕ^{new} and ϕ^{old} are the correction vectors at the current and previous iteration, respectively. If ζ falls below a pre-defined tolerance then convergence has been achieved. In the early stages of iteration this is not usually the case and a decision is made to determine if convergence of the algorithm is slow. The convergence is deemed to be slow if

$$\gamma \leq \frac{\zeta^{new}}{\zeta^{old}} \qquad (7.6\text{-}2)$$

where γ is determined experimentally for each algorithm. With the exception of the coarsest grid (level N) the computation switches from level i to level i+1. On the coarsest grid an accurate solution is required, this is done using an iterative

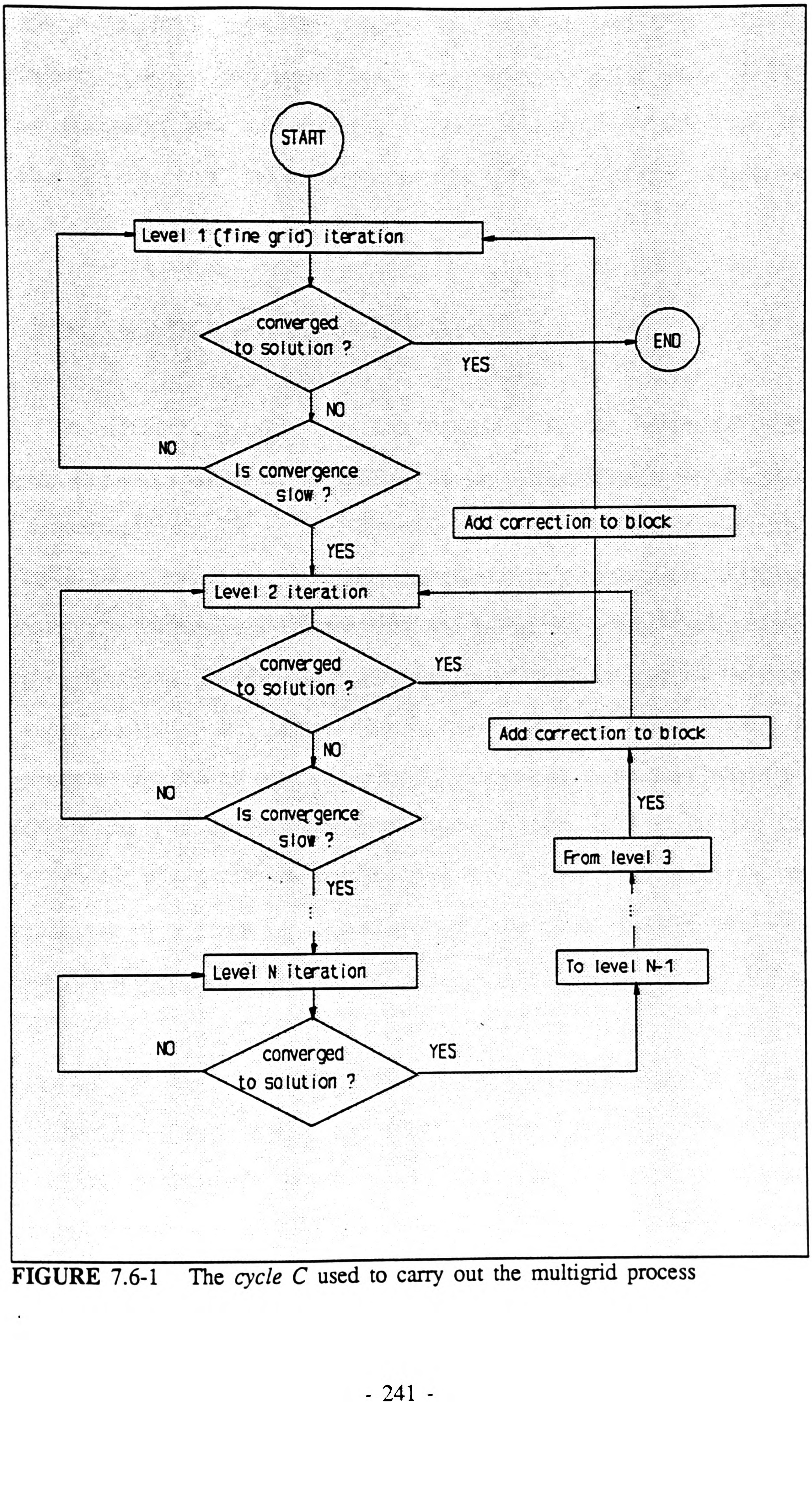


FIGURE 7.6-1 The *cycle C* used to carry out the multigrid process algorithm. The solution for the coarsest grid is then used to correct the approximations on the next highest level in a similar fashion to (7.5.2-3). The process is repeated until convergence is achieved on level 2. At this stage the correction δ_{k1} is used to correct the pressure-correction field. Again, if convergence is not adequate then the cycle is repeated.

7.7 Iterative algorithms used in the ACM method

The JUR and LSOR algorithms are both considered for use within the ACM strategy. In the case of the LSOR algorithm the Line Gauss-Seidel is used to solve the coarsest grid, and for the JUR algorithm the Jacobi algorithm is used. The choice of the parameter ζ (to determine convergence on a given level), is chosen to reflect the different convergence rates and quality of solution produced by different algorithms. For the LSOR algorithm $\zeta=10^{-3}$ is chosen and for the JUR algorithm $\zeta=10^{-4}$, this is consistent with the criteria employed in the solution of the pressure-correction equation in section 5.2. Robustness is the main criteria on which the choice of the parameter γ was made. By selecting a single value for each algorithm this allowed the multigrid method to perform satisfactorily, but not optimally, on all of the test cases considered. For the LSOR algorithm $\gamma=0.5$ and for the JUR algorithm $\gamma=0.9$.

An attempt was made to combine the JCG algorithm with the ACM method. Unfortunately, in the solution of the pressure-correction equation the residuals are not always monotonically decreasing and this has prohibited an efficient implementation. The general behaviour of the JCG algorithm is now summmarised; At some stage during the cycle the computation is switched to a coarser grid because $\gamma>1$ (7.6-2). After the solution of the correction field the result is added to the finer grid. However, the solution now gives a residual which is larger than that of the previous iteration, that is, before the ACM method was applied. This results in either a high inefficiency or sometimes divergence of the algorithm.

7.8 Implementation of the ACM method on a pipeline processor

The implementation of the ACM method on the MASSCOMP VA-1 pipeline processor was straightforward. Fortunately, much of the cycle C is vectorisable, the only essentially scalar operations were the switching and convergence criteria, these were carried out on the host processor. There was a marginal increase in the code size, but more significant was the increase of the data storage required. This was as high as 33% of the original storage required by the iterative algorithm. Use is also made of the gather and scatter vector operations. These operations are used to ensure that the vector computations are of maximum possible length, but at the expense of an overhead in preparation of the vectors. Here, the gather operation is used to generate the residual vector on the next higher level and the scatter operation is used correct the approximations on the next lower level.

7.9 Results using the ACM method

The four test problems introduced in Chapters 5 and 6 are used here to test the effect of using the ACM method. The comparisons include the effect of the ACM method on the scalar and vector algorithms as well as the most efficient algorithm in each problem.

7.9.1 PROBLEM 1: The cavity with moving lid problem

Up to four different grids are used in the ACM method 4x4, 8x8, 16x16 and 32x32 and the case when Re=100 is considered. Overall convergence is achieved when the maximum residual for all unknowns is less than 2.5×10^{-6}. The results show that the JURS algorithm improves by a factor of up to 2.5 when four levels are used (table 7.9.1-1). However, the JURV does not improve to the same extent as the scalar algorithm, showing only marginal improvements and working most efficiently with only 2 levels. Marginal improvements are also observed for the scalar LSOR algorithm. A complete history of the performance of the algorithms is shown in figures 7.9.1-1 to 7.9.1-3.

From these results and those obtained for the JCG algorithm in Chapter 5 the most efficient algorithm can be determined (figure 7.9.1-4). The most efficient scalar and vector algorithms are the LSOR with 4 levels and the JCGV algorithm, respectively. The JCGV is a factor of five faster than the LSOR with four levels.

7.9.2 PROBLEM 2: The sudden expansion problem

The four different grids used in the ACM method are 8x2, 16x4, 32x8 and 64x16. Table 7.9.2-1 shows the effect of using up to four levels with a given convergence criteria of residuals less than 1.0×10^{-5}. The scalar JUR algorithm shows an improvement of up to 4.5 when all four levels are used, the vectorised algorithm again shows most efficient use with just two levels. The scalar LSOR algorithm is the most efficient scalar algorithm with improvements of up to 3.5 over the single level LSOR algorithm. A complete history of the maximum residuals reveals that there is some oscillation for all three algorithms (figures 7.9.2-1 to 7.9.2-3). Therefore, care must be taken when making comparisons based on a single convergence value. The scalar LSOR is the best scalar algorithm, being up to 2.5 times faster than the JCG algorithm (figure 7.9.2-4), there is however little to choose between the JURV with two levels and the JCGV algorithm for residuals up to 10^{-4}. A final comparison between the best scalar and vector algorithms shows there is a factor of 8.5 in favour of the vector algorithm.

7.9.3 PROBLEM 3: Turbulent L-shaped flow problem

The same grid levels are used as those in PROBLEM 2 and the convergence level selected for this problem is 1.75×10^{-5}. Timings for the simulation are given in table 7.9.3-1 and show that the scalar JUR algorithm benefits by a factor of up to 2.7 when all four levels are used. The LSOR algorithm did not benefit as much as the JUR algorithm and there was little to choose between the results for two, three and four levels. Similarly, for the JURV algorithm the advantages of using the ACM method are less pronounced (figures 7.9.3-1 to 7.9.3-3). The LSOR algorithm with four levels is clearly the best scalar algorithm while there is little to choose between the vector algorithms (figure 7.9.3-4). The overall improvement for solving this problem is a factor of 4.5 in favour of the vector algorithms. Using the ACM method as a solver for the pressure-correction equation becomes less profitable as more scalar equations are solved. For this reason the speed-up factor for this problem between the best scalar and vector algorithms is lower than in the previous cases.

Number of ACM levels	Grid	LSOR	JURS	JURV
1	32x32	705.9	2980.4	137.3
2	16x16	616.7	1460.8	128.9
3	8x8	561.1	1274.5	133.8
4	4x4	575.0	1205.9	134.9

TABLE 7.9.1-1 The effect of using up to 4 levels of the ACM method for the solution of the cavity problem (Re=100).

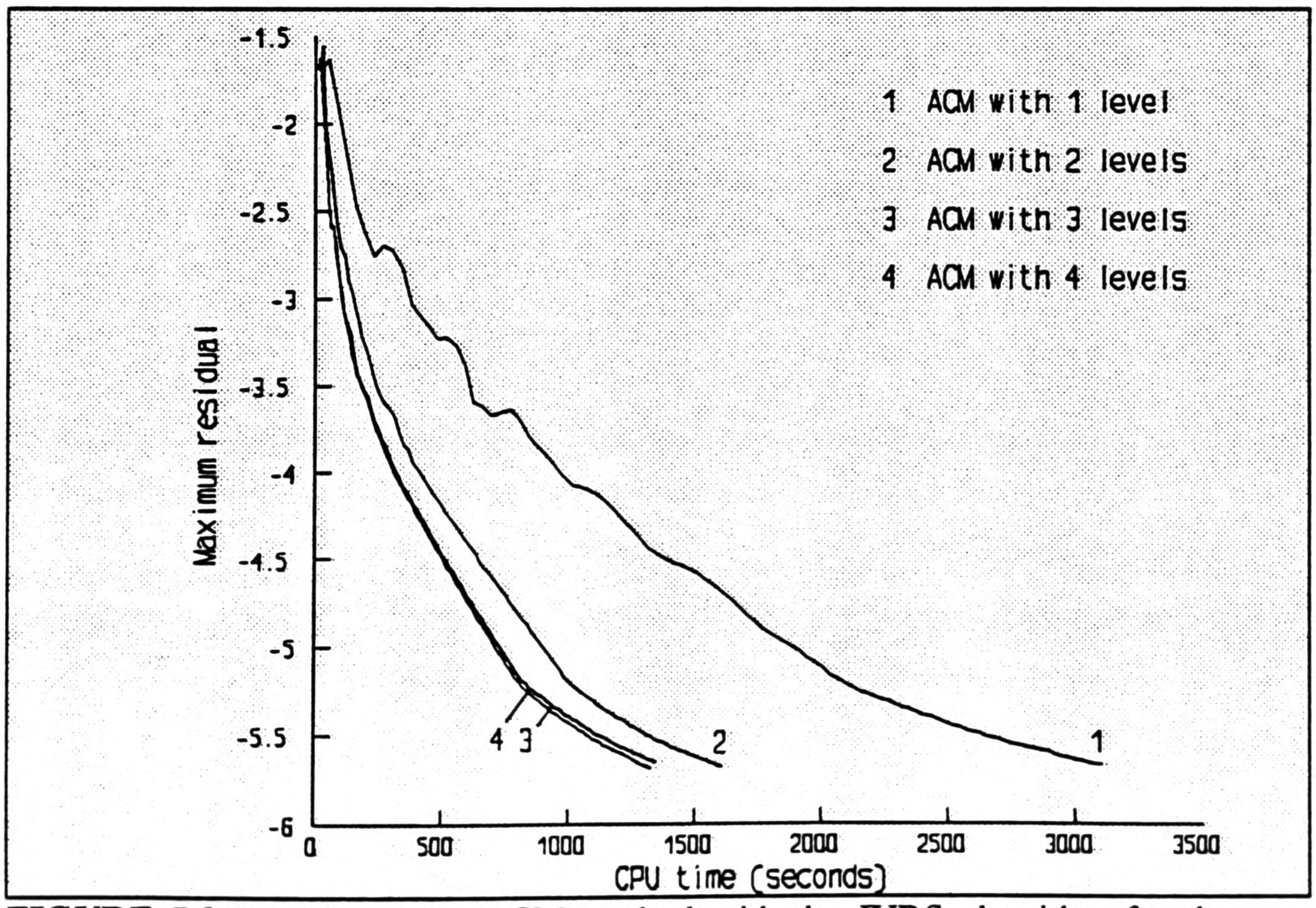


FIGURE 7.9.1-1 Using the ACM method with the JURS algorithm for the solution of the cavity problem (Re=100)

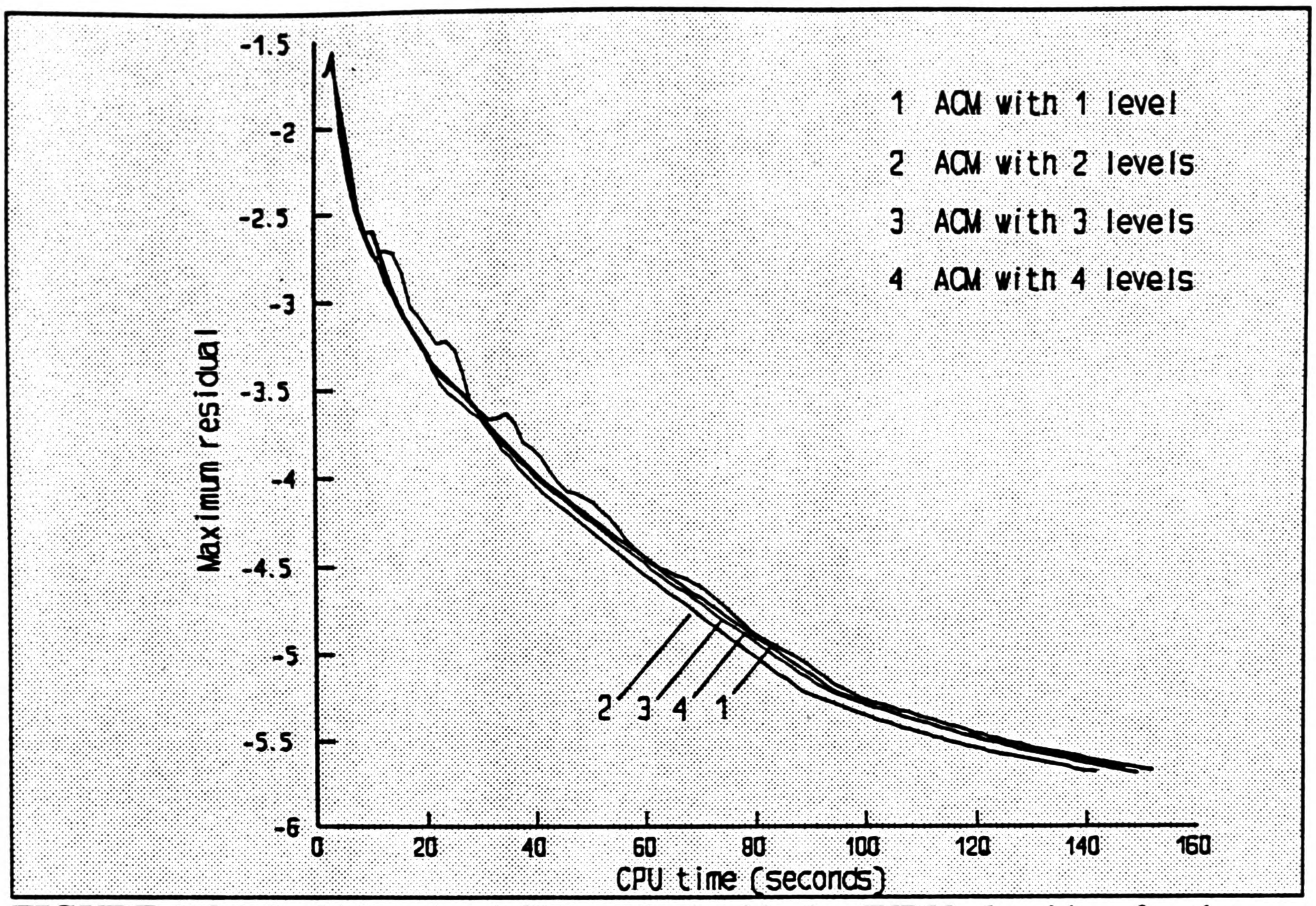


FIGURE 7.9.1-2 Using the ACM method with the JURV algorithm for the solution of the cavity problem (Re=100)

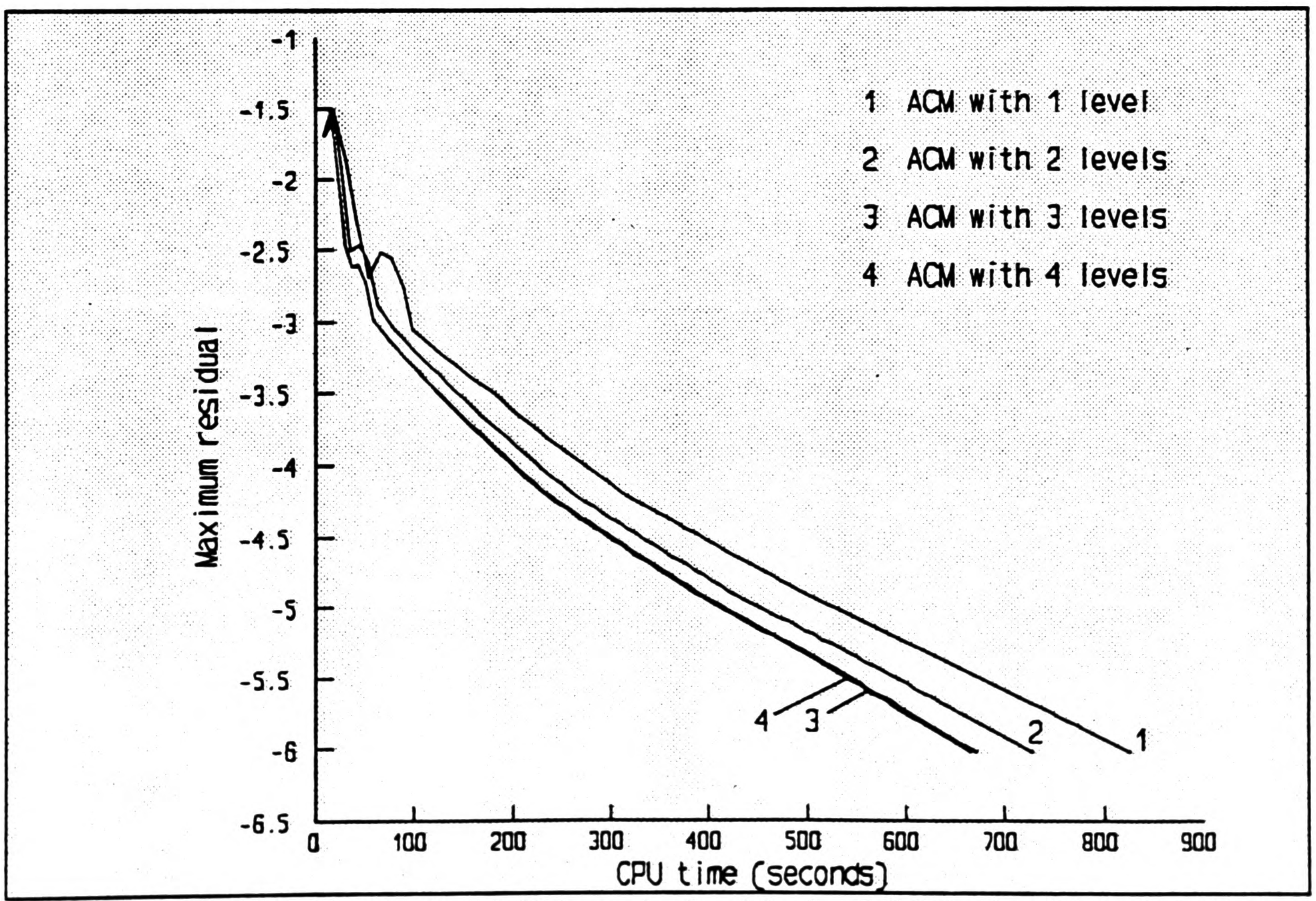


FIGURE 7.9.1-3 Using the ACM method with the LSOR algorithm for the solution of the cavity problem (Re=100)

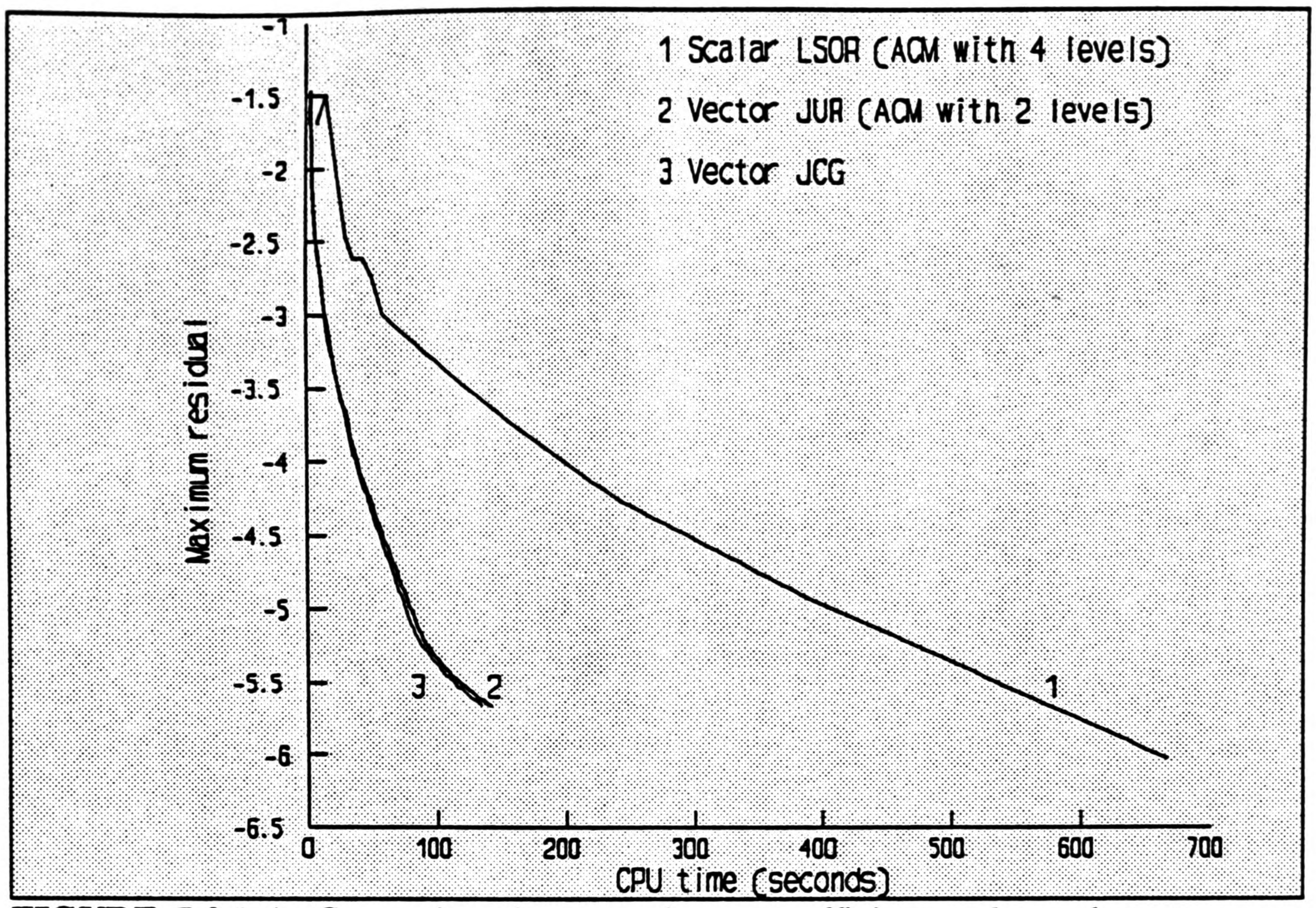


FIGURE 7.9.1-4 Comparison between the most efficient scalar and vector algorithm for the solution of the cavity problem (Re=100)

Number of ACM levels	Grid	LSOR	JURS	JURV
1	64x16	1955.6	7600.0	266.7
2	32x8	1050.0	3288.9	164.7
3	16x4	700.0	2244.4	203.3
4	8x2	566.7	1688.9	184.9

TABLE 7.9.2-1 The effect of using up to 4 levels of the ACM method for the solution of the sudden expansion problem.

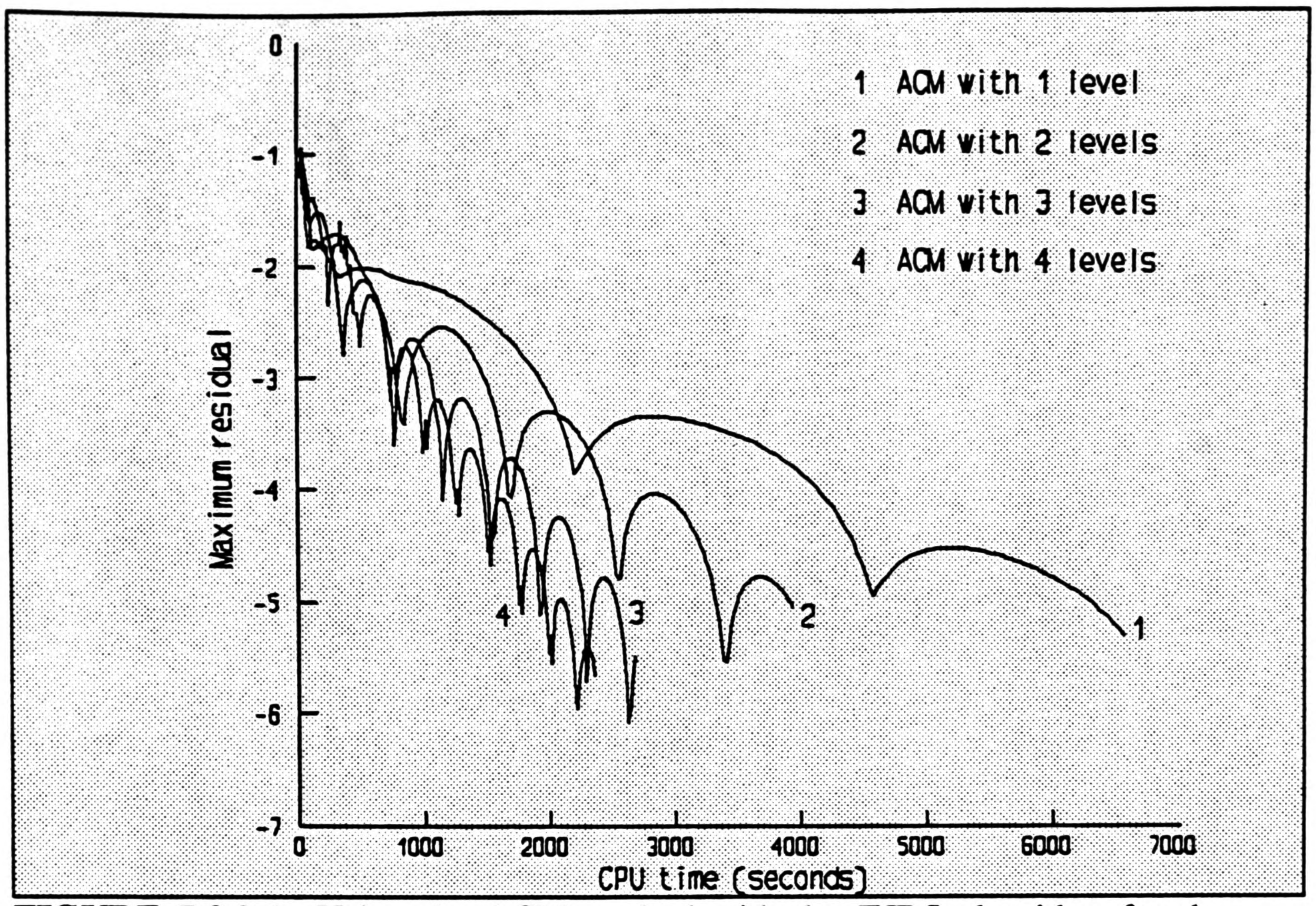


FIGURE 7.9.2-1 Using the ACM method with the JURS algorithm for the solution of the sudden expansion problem

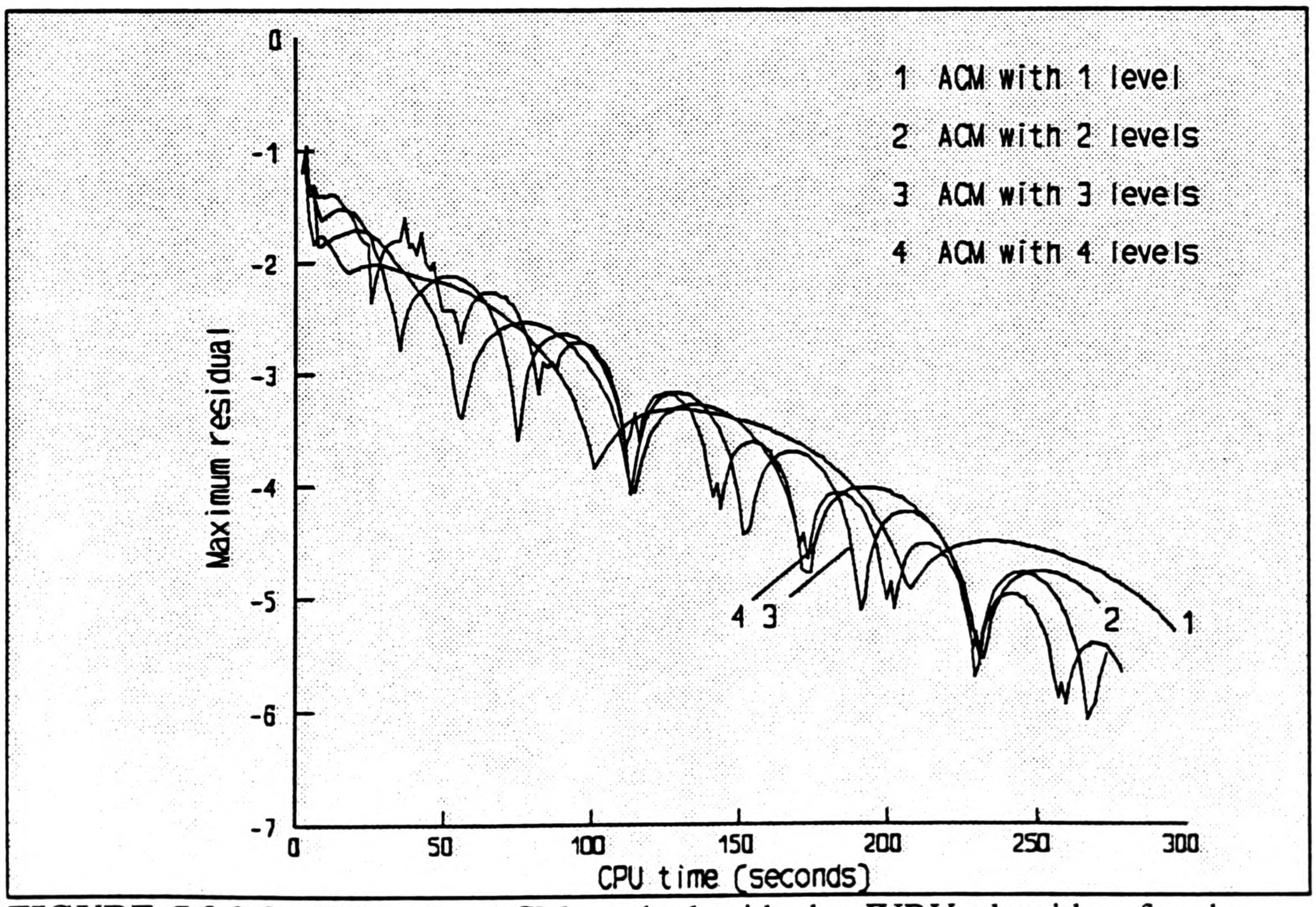


FIGURE 7.9.2-2 Using the ACM method with the JURV algorithm for the solution of the sudden expansion problem

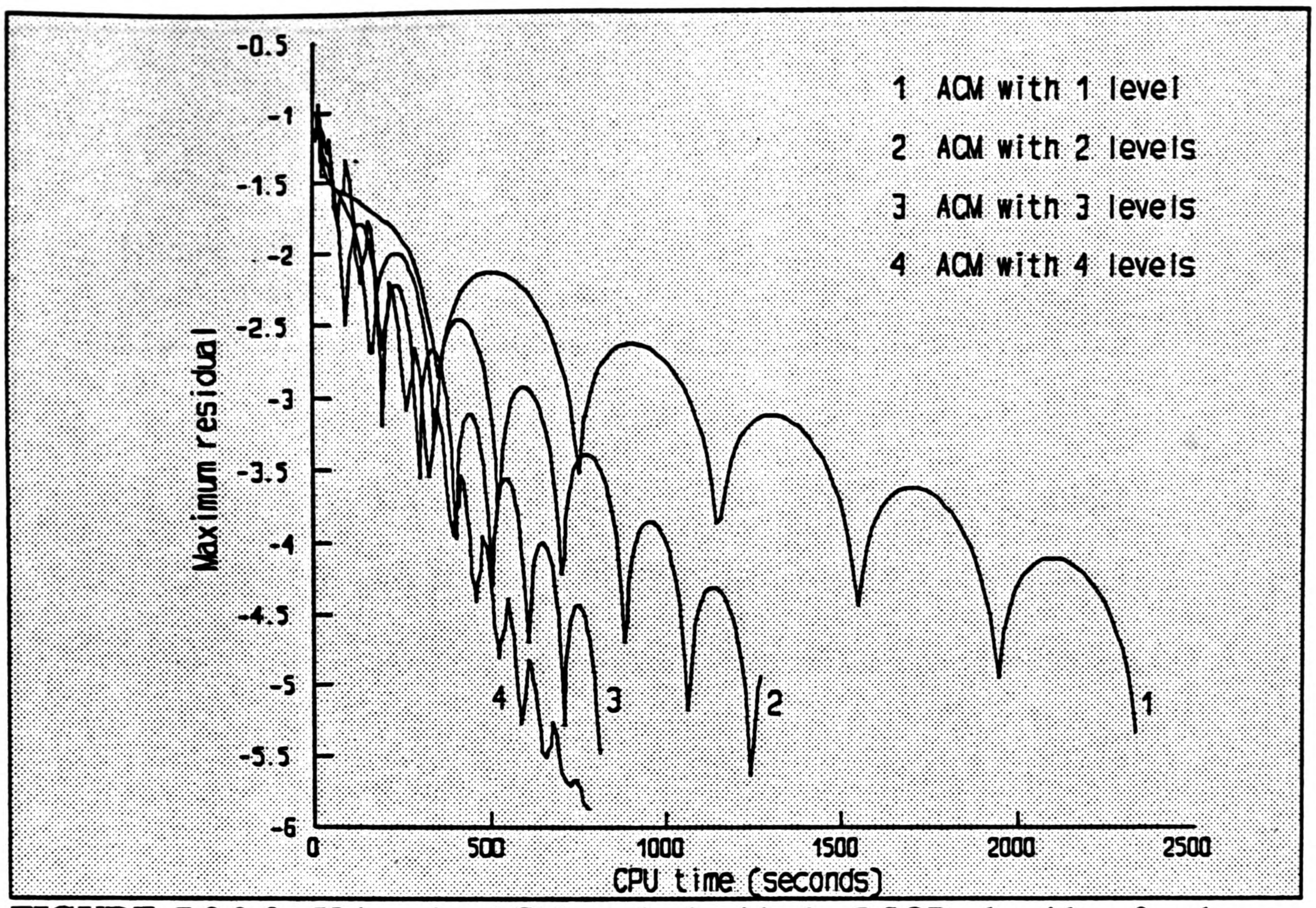


FIGURE 7.9.2-3 Using the ACM method with the LSOR algorithm for the solution of the sudden expansion problem

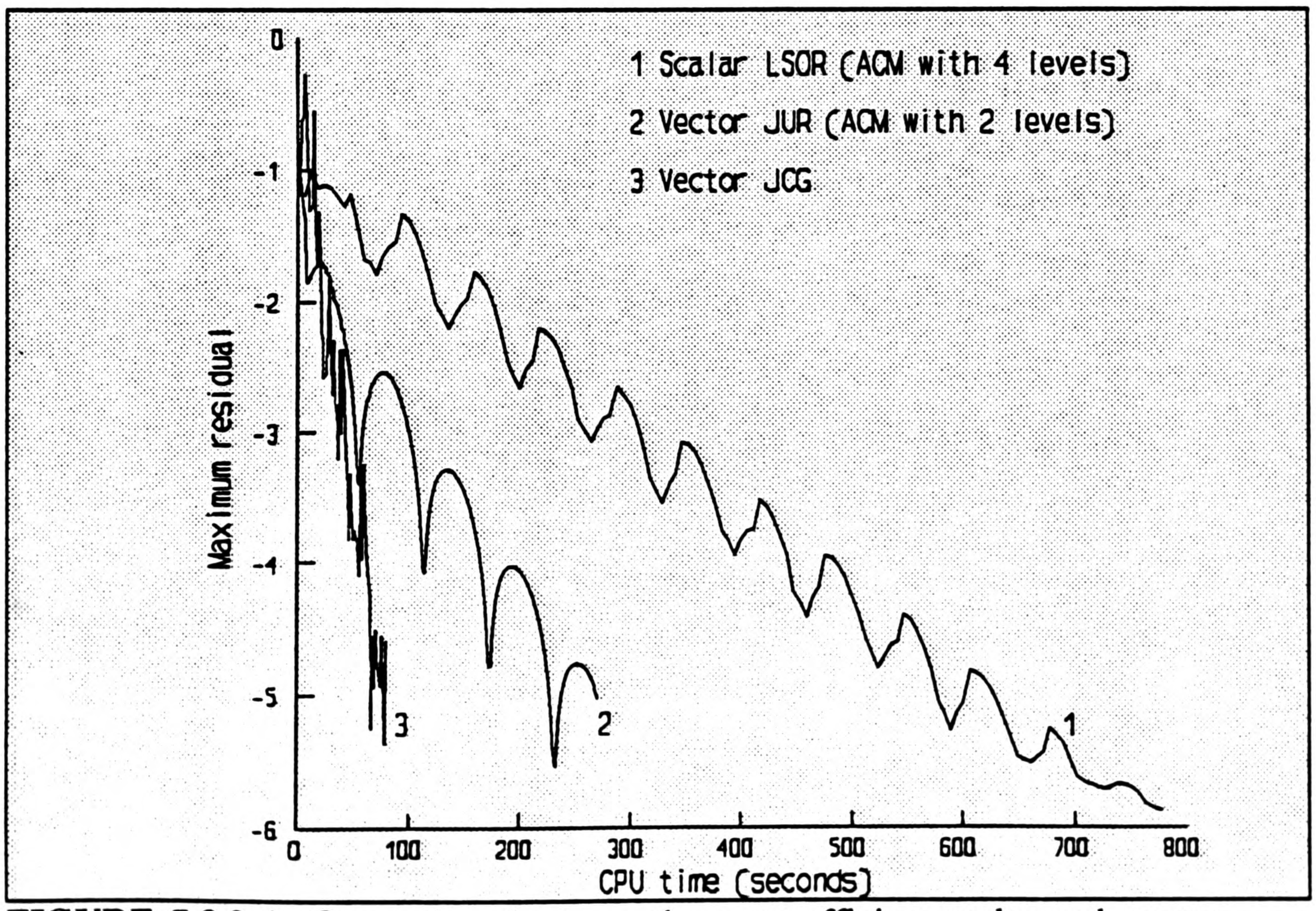


FIGURE 7.9.2-4 Comparison between the most efficient scalar and vector algorithm for the solution of the sudden expansion problem

Number of ACM levels	Grid	LSOR	JURS	JURV
1	64x16	6275.4	16777.8	1333.3
2	32x8	5327.3	7555.6	1136.4
3	16x4	5372.5	6222.2	1163.6
4	8x2	5417.6	6181.0	1181.8

TABLE 7.9.3-1 The effect of using up to 4 levels of the ACM method for the solution of the L-shaped flow problem.

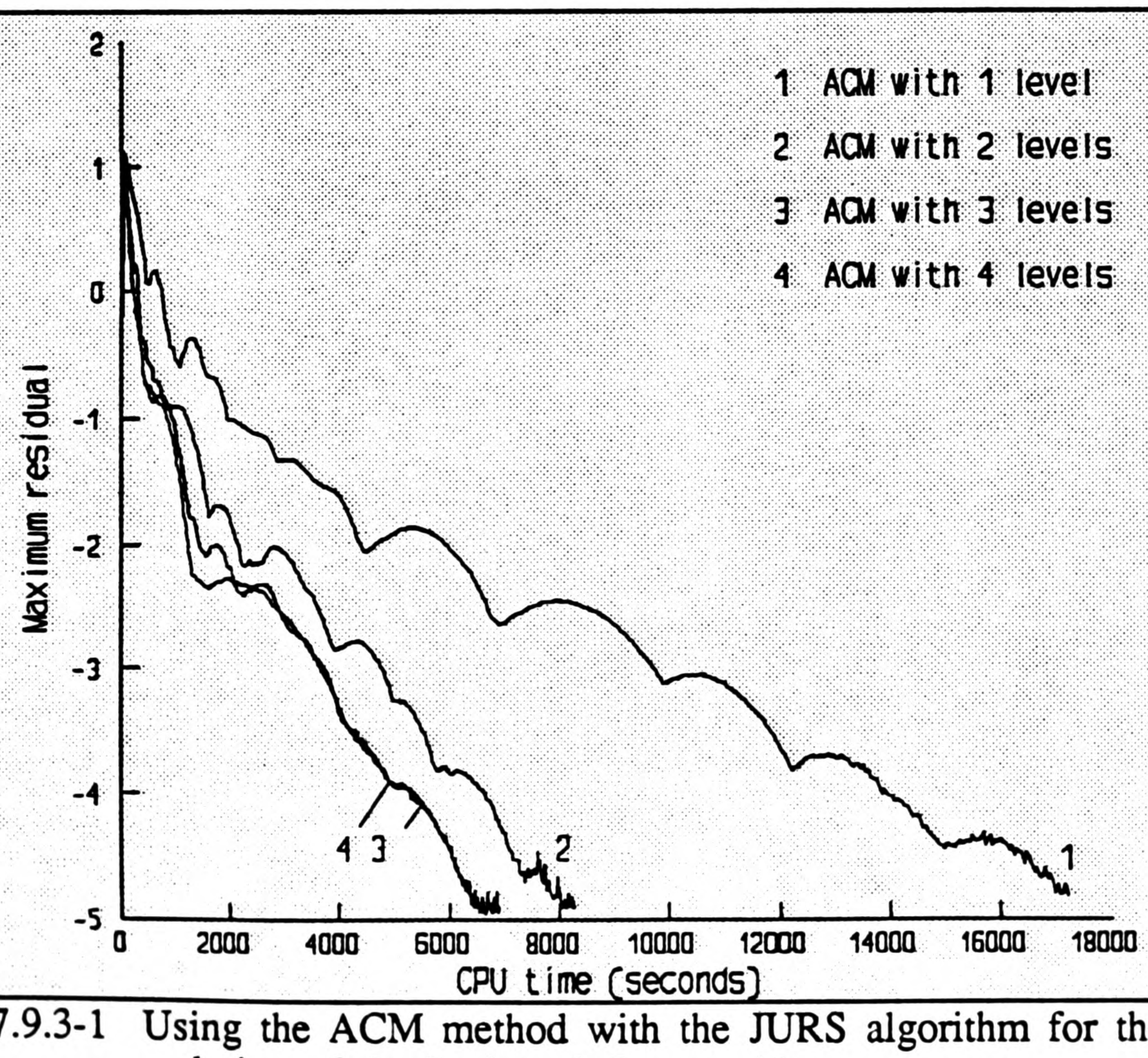


FIGURE 7.9.3-1 Using the ACM method with the JURS algorithm for the solution of the L-shaped flow problem

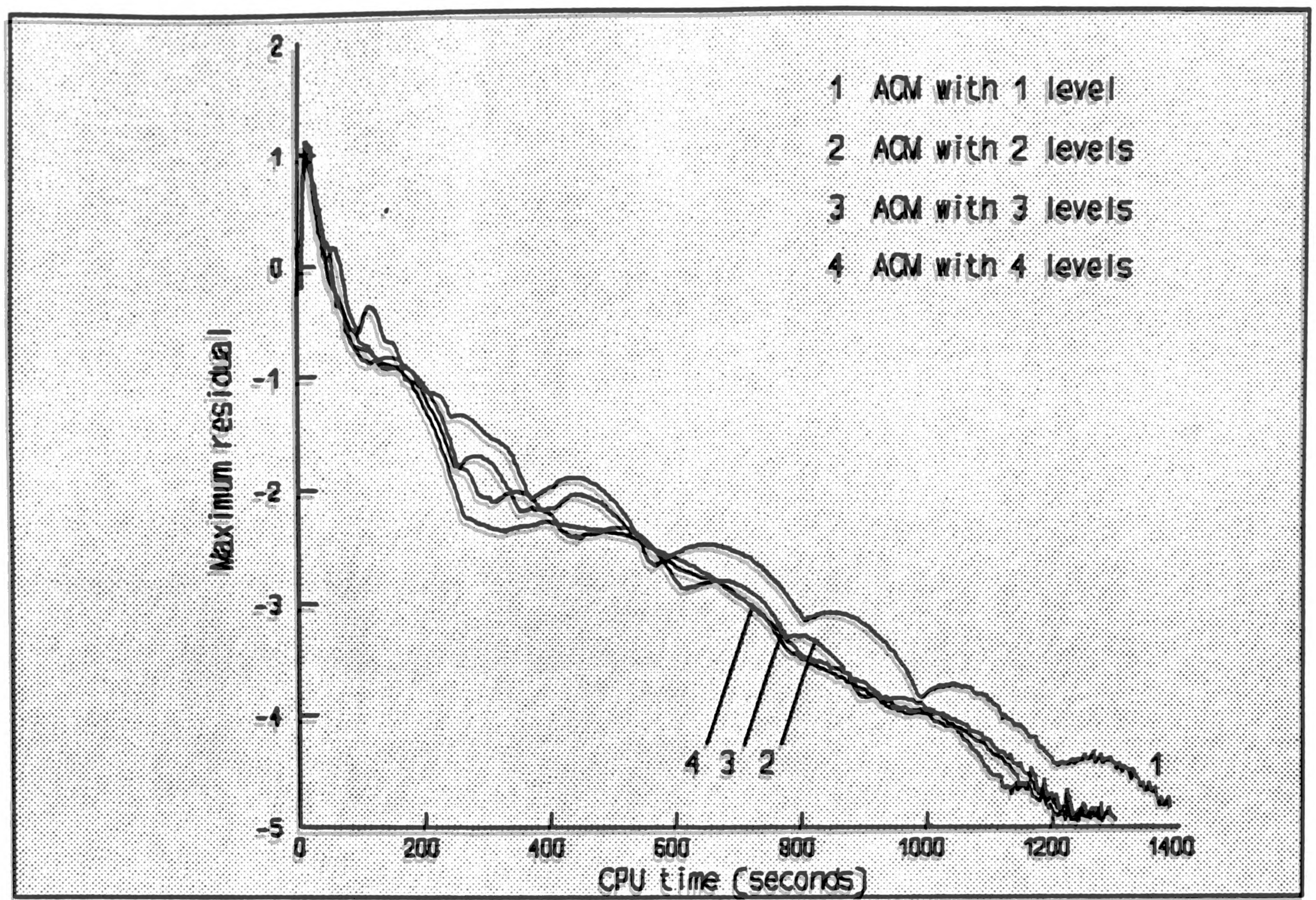


FIGURE 7.9.3-2 Using the ACM method with the JURV algorithm for the solution of the L-shaped flow problem

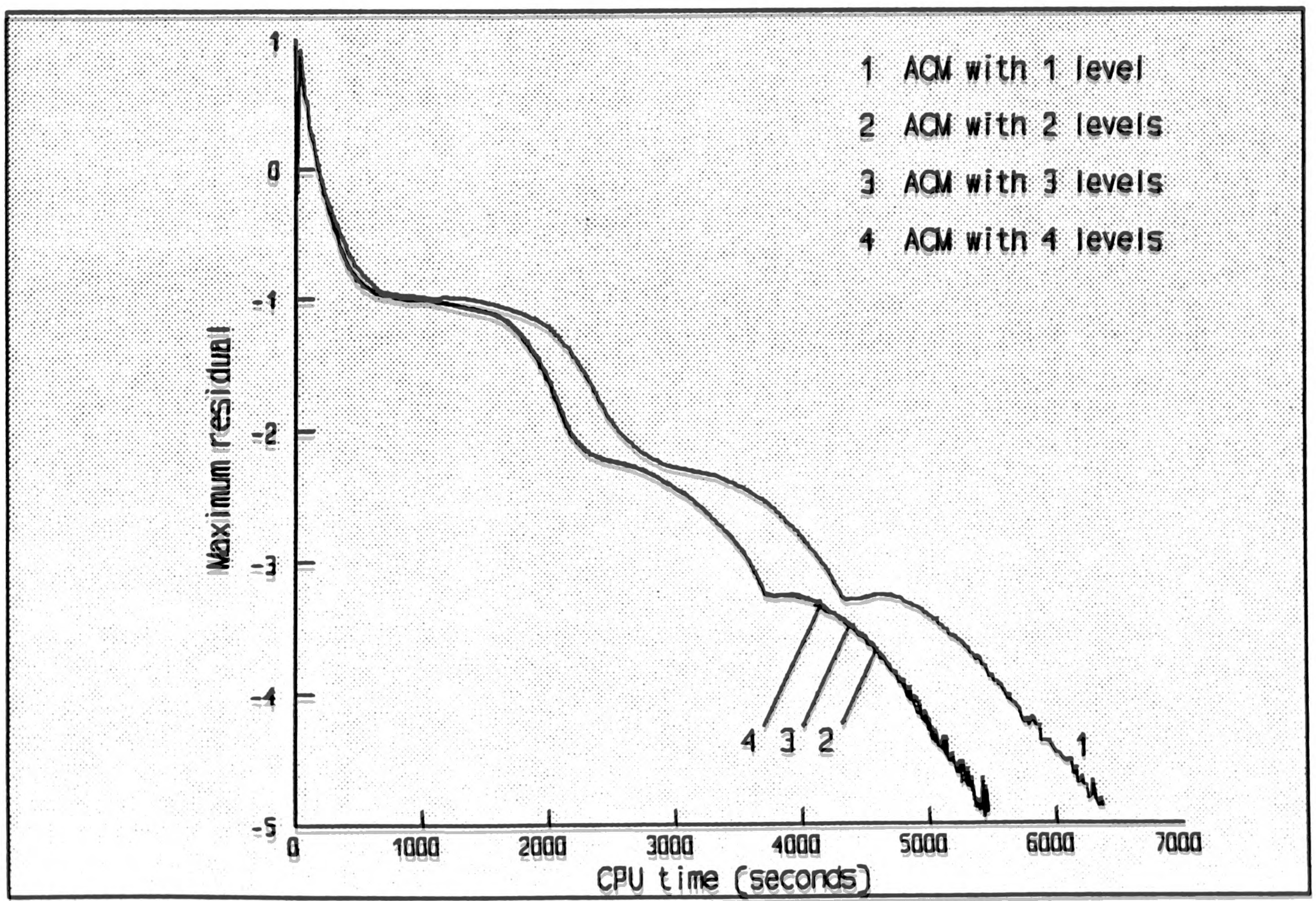


FIGURE 7.9.3-3 Using the ACM method with the LSOR algorithm for the solution of the L-shaped flow problem

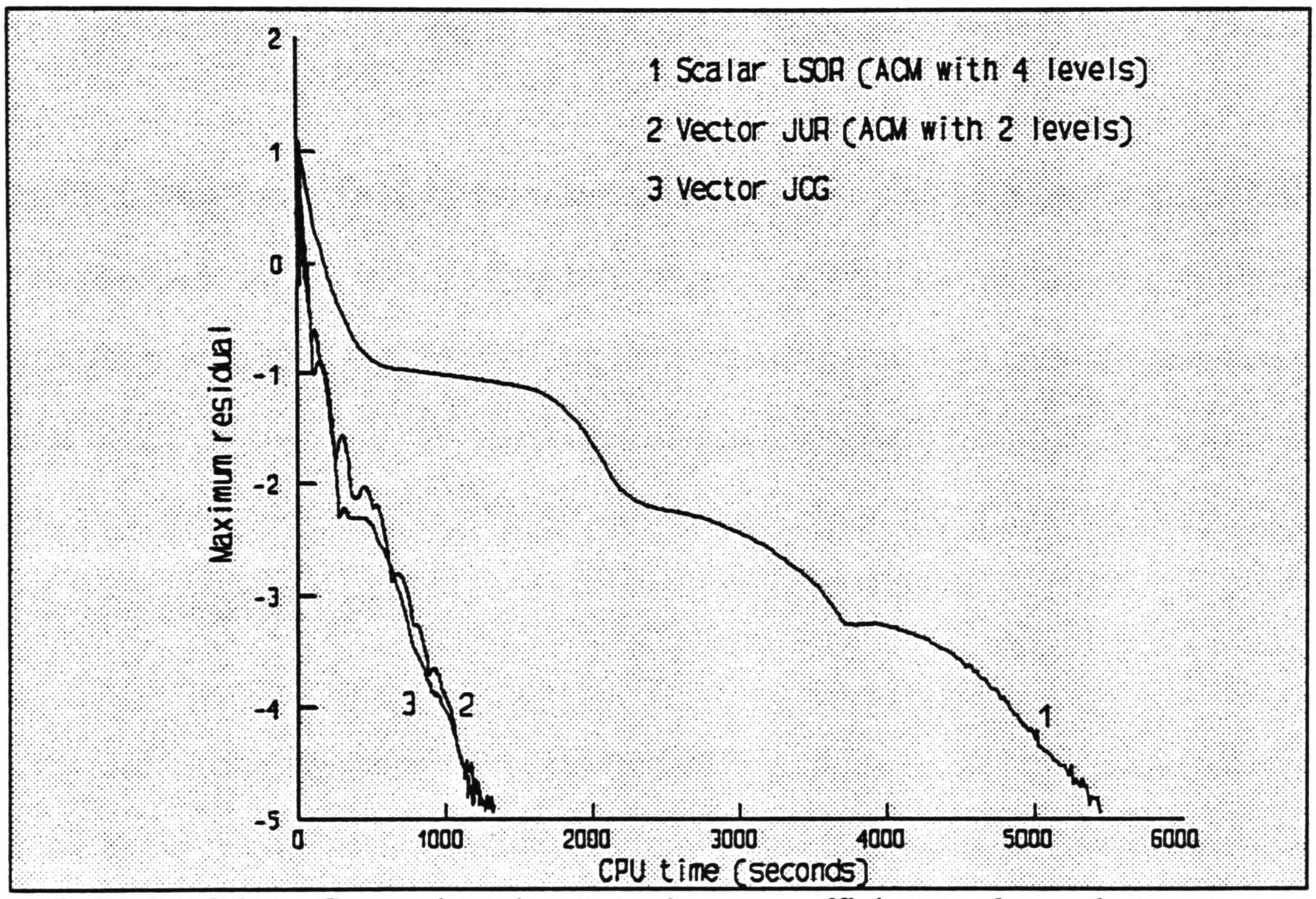


FIGURE 7.9.3-4 Comparison between the most efficient scalar and vector algorithm for the solution of the L-shaped flow problem

7.9.4 PROBLEM 4: Natural convection in a square cavity problem

Four levels of the ACM method are used and defined by 4x4, 8x8, 16x16, and 32x32. Table 7.9.4-1 gives timings for the simulation of various Rayleigh numbers at a convergence level of 1.75×10^{-5}. The scalar JUR algorithm benefits by up to a factor of 3 when all four levels are used. The LSOR algorithm did not benefit very much for low Rayleigh numbers but shows more improvement as the Rayleigh number is increased. The JURV algorithm shows only marginal improvements in all cases (figures 7.9.4-1 to 7.9.4-3). In general, the LSOR algorithm with four levels is the most efficient scalar algorithm and there is little to choose between the vector algorithms (figure 7.9.4-4). The overall improvements range from a factor of 8 ($Ra=10^3$) to 6.5 ($Ra=10^6$), and for the turbulent case a factor of 3.8 in favour of the vector algorithms is achieved.

7.10 Discussion of Results

In the two isothermal cases where the variables u, v and p are solved, the solution of the pressure-correction equation forms a major component of the computation time. Therefore, using the ACM method to assist in the solution of the pressure-correction equation is very worthwhile. This is particularly true for a slowly converging algorithm with typical reduction factors of up to 4.5. The use of the ACM method is less notable when it is vectorised, the best reductions are up to 60% and was achieved with two levels only.

From the first two cases it is apparent that the ACM method is most effective in situations where a predominant direction exists in the flow. In PROBLEM 2 a

(a)

Number of ACM levels	Grid	LSOR	JURS	JURV
1	32x32	2960.0	9200.0	400.0
2	16x16	2784.3	4156.9	373.3
3	8x8	2764.7	3372.5	376.6
4	4x4	2725.5	3215.7	378.3

(b)

Number of ACM levels	Grid	LSOR	JURS	JURV
1	32x32	4555.5	13666.7	611.1
2	16x16	4333.3	6222.2	554.5
3	8x8	4194.4	5121.1	563.6
4	4x4	4166.7	4777.5	600.0

(c)

Number of ACM levels	Grid	LSOR	JURS	JURV
1	32x32	3802.8	14788.7	633.8
2	16x16	3700.0	6338.0	586.7
3	8x8	3600.0	5352.1	613.3
4	4x4	3266.7	4929.6	600.0

(d)

Number of ACM levels	Grid	LSOR	JURS	JURV
1	32x32	3666.7	14000.0	666.7
2	16x16	3137.2	6166.7	619.6
3	8x8	2941.1	5000.0	635.3
4	4x4	2588.2	4666.7	650.9

(e)

Number of ACM levels	Grid	LSOR	JURS	JURV
1	32x32	11272.7	28039.2	2803.9
2	16x16	10303.1	13333.3	2430.1
3	8x8	9636.4	11176.5	2468.8
4	4x4	9333.3	10882.4	2517.6

TABLE 7.9.4-1 The effect of using up to 4 levels of the ACM method for the solution of the natural convection problem a) $Ra=10^3$ b) $Ra=10^4$ c) $Ra=10^5$ d) $Ra=10^6$ e) $Ra=10^7$.

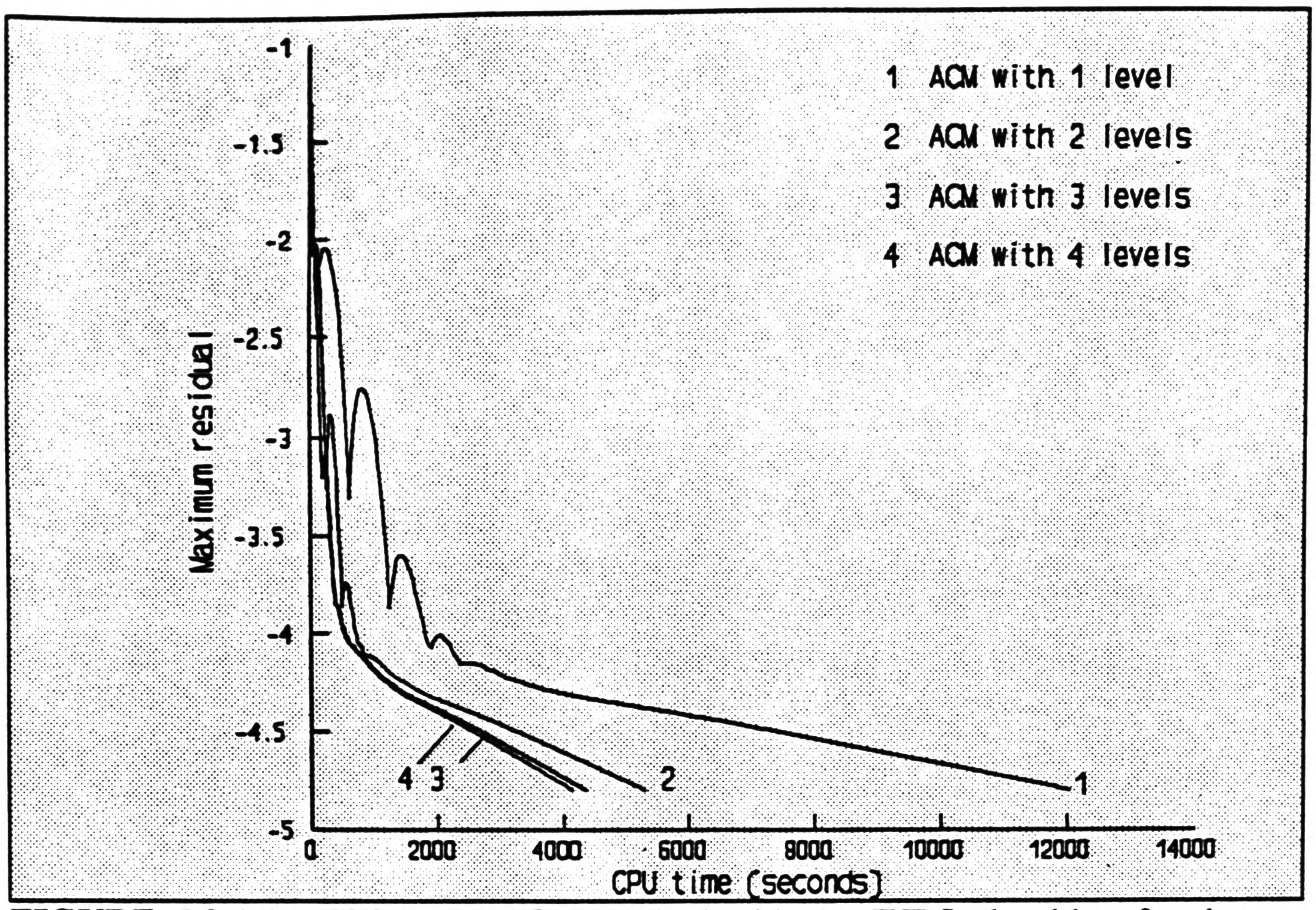


FIGURE 7.9.4-1a Using the ACM method with the JURS algorithm for the solution of the natural convection problem (Ra=10^3)

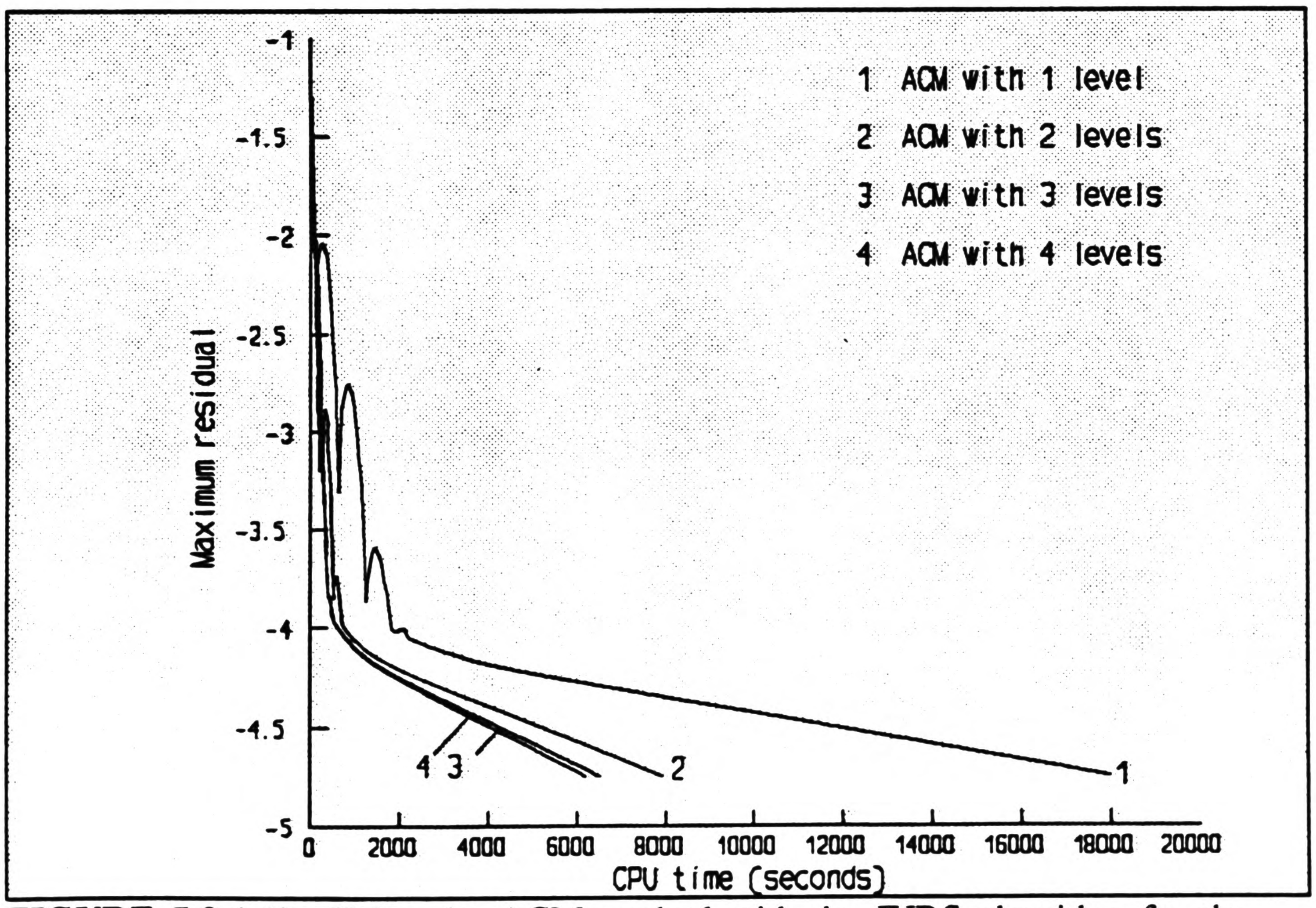


FIGURE 7.9.4-1b Using the ACM method with the JURS algorithm for the solution of the natural convection problem (Ra=10^4)

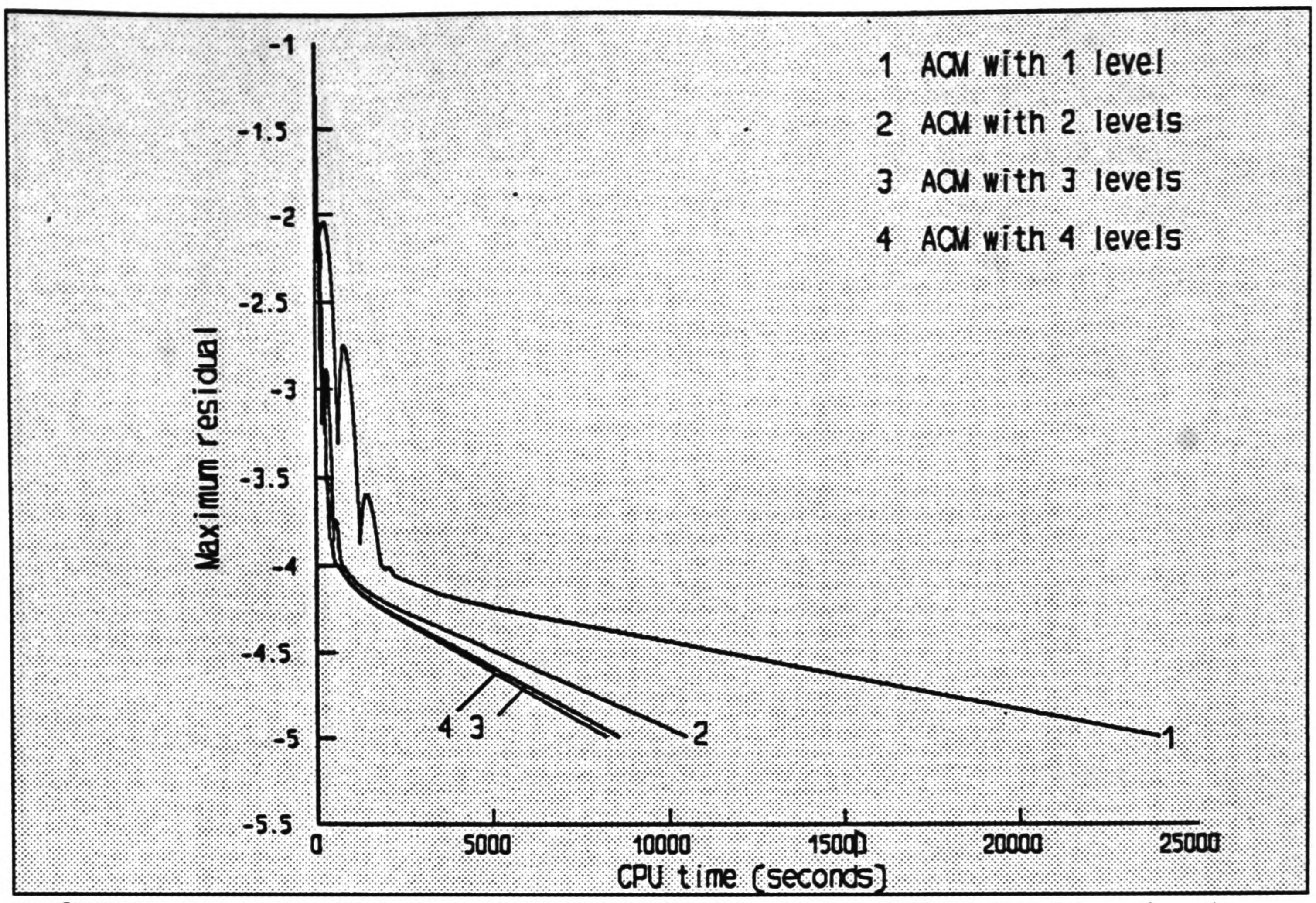


FIGURE 7.9.4-1c Using the ACM method with the JURS algorithm for the solution of the natural convection problem (Ra=10^5)

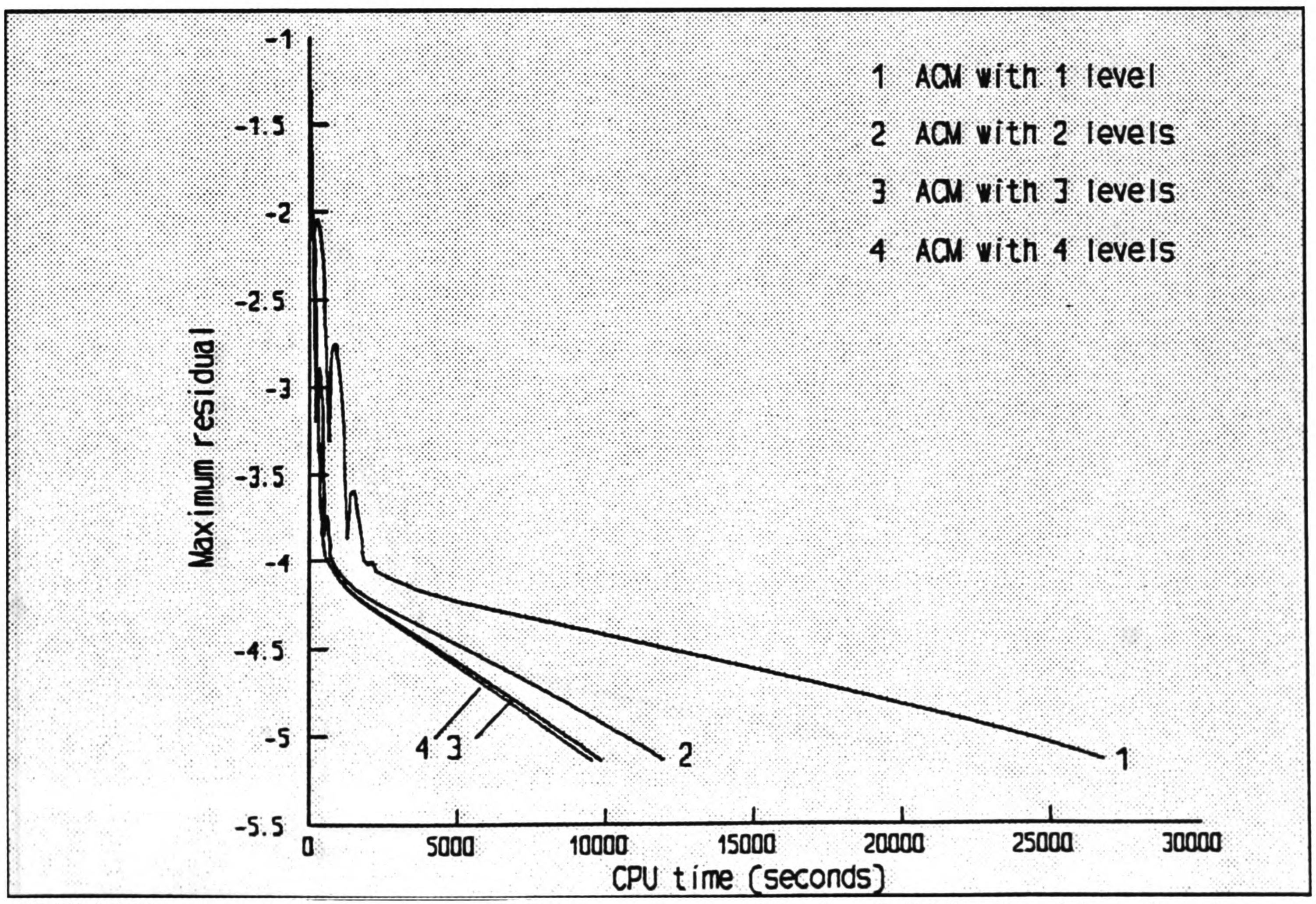


FIGURE 7.9.4-1d Using the ACM method with the JURS algorithm for the solution of the natural convection problem (Ra=10^6)

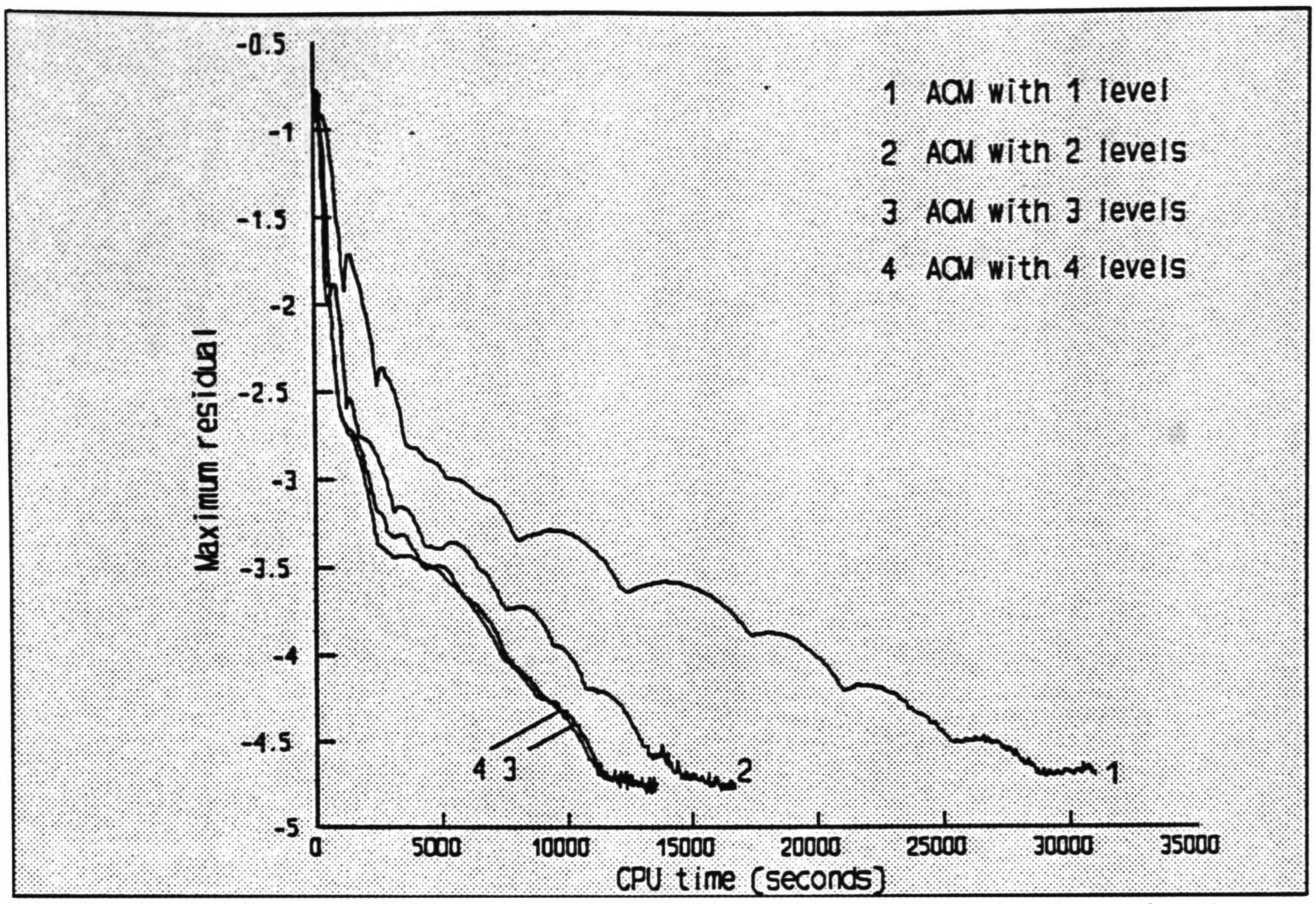


FIGURE 7.9.4-1e Using the ACM method with the JURS algorithm for the solution of the natural convection problem (Ra=10^7)

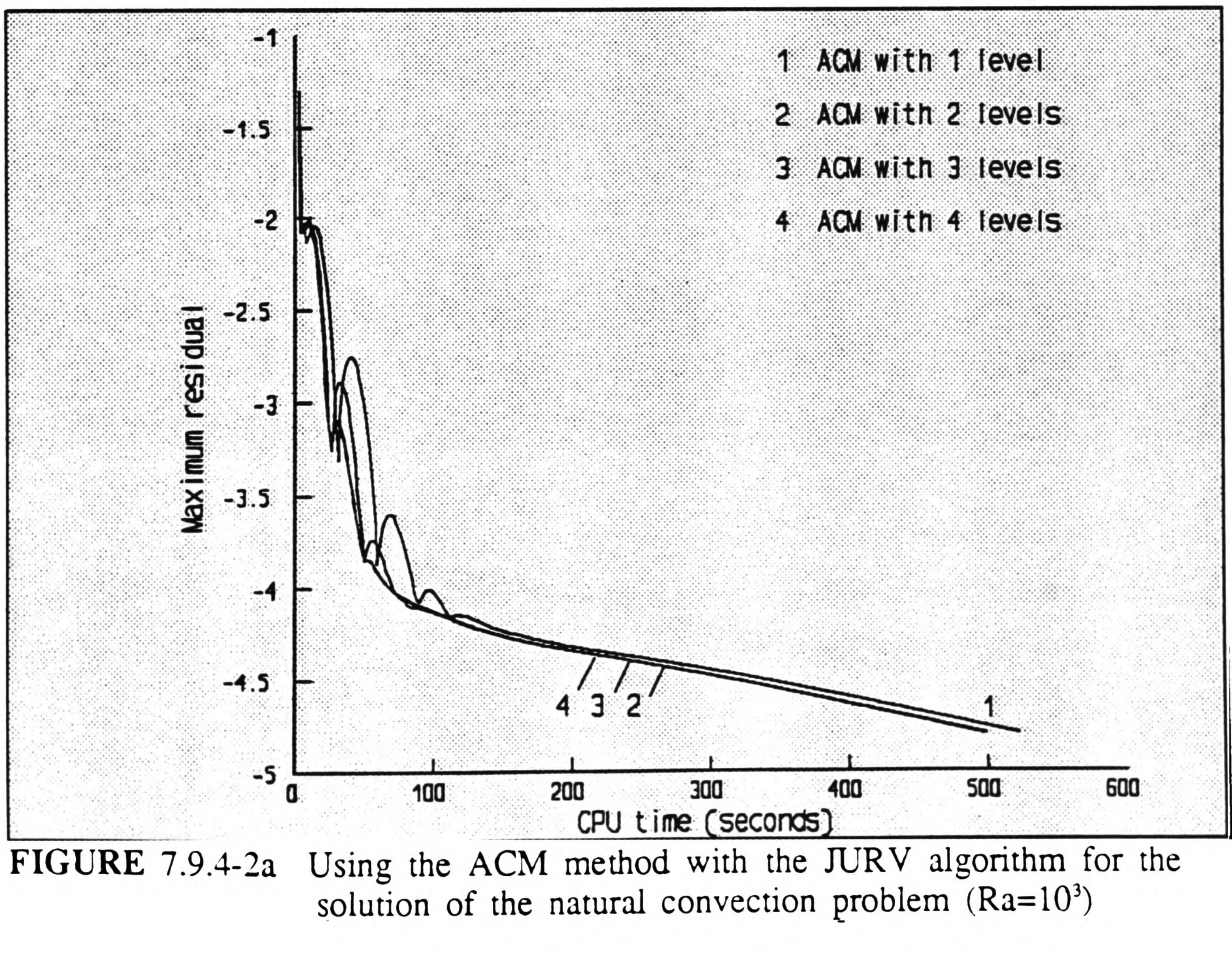


FIGURE 7.9.4-2a Using the ACM method with the JURV algorithm for the solution of the natural convection problem (Ra=10^3)

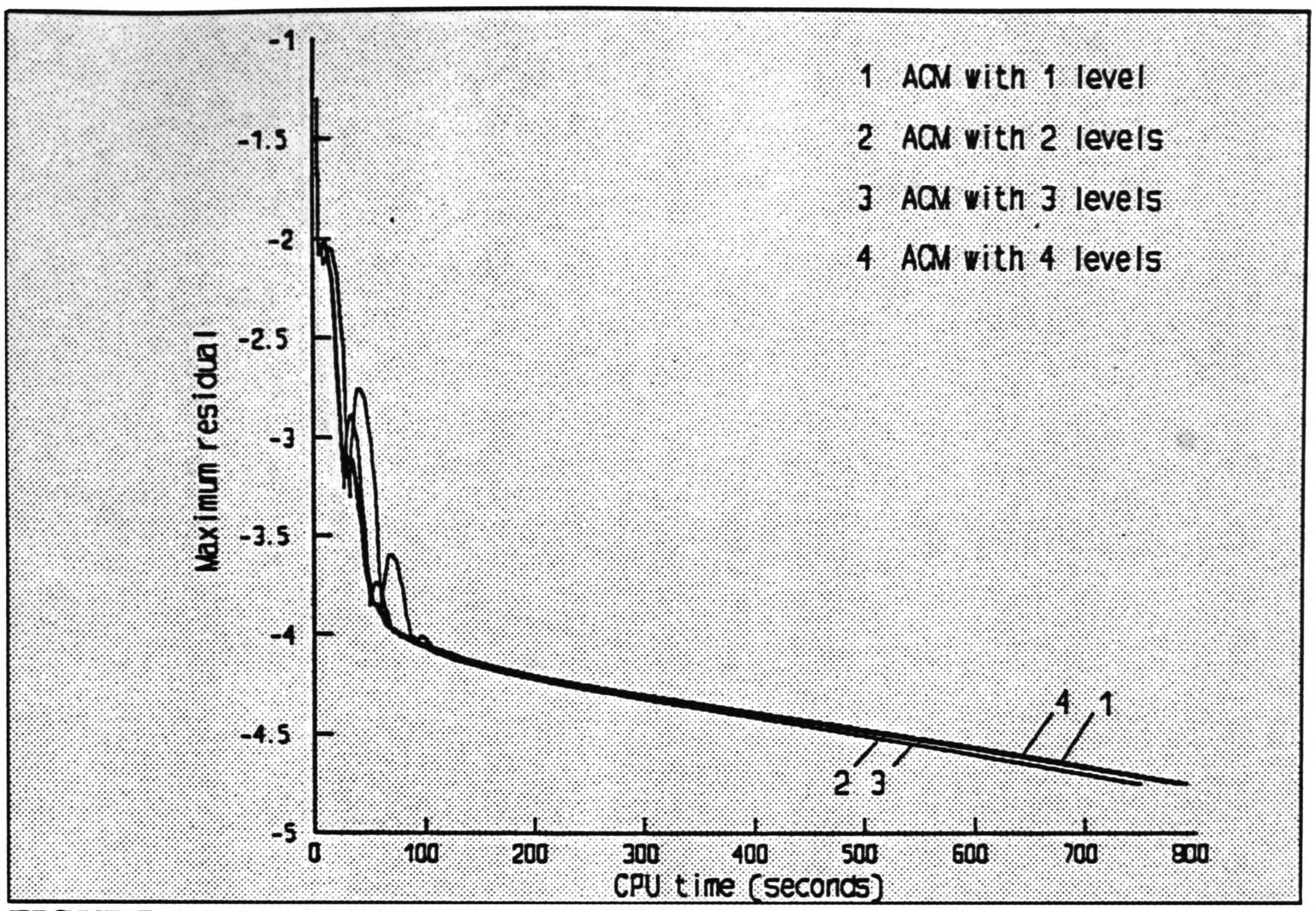


FIGURE 7.9.4-2b Using the ACM method with the JURV algorithm for the solution of the natural convection problem (Ra=10^4)

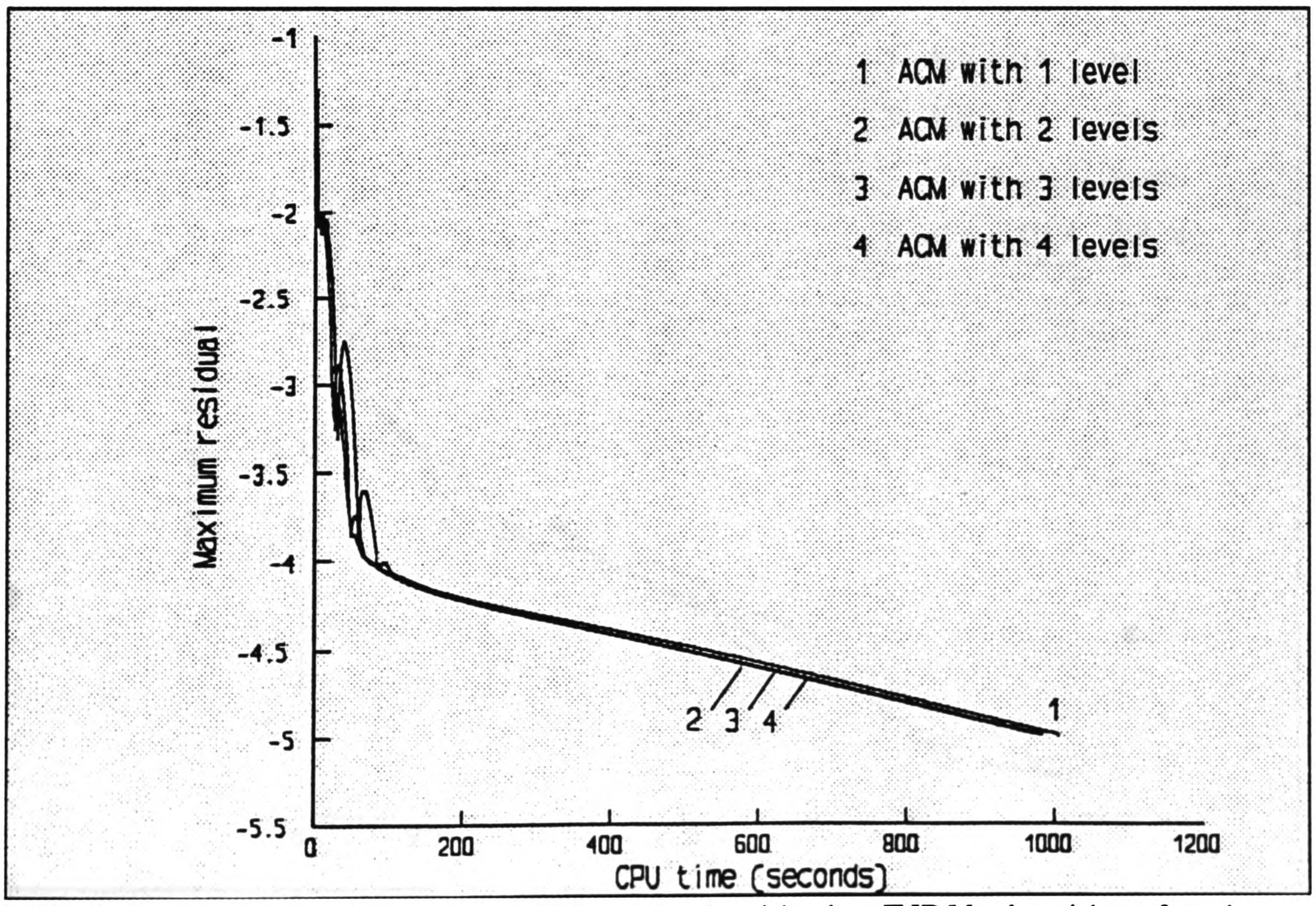


FIGURE 7.9.4-2c Using the ACM method with the JURV algorithm for the solution of the natural convection problem (Ra=10^5)

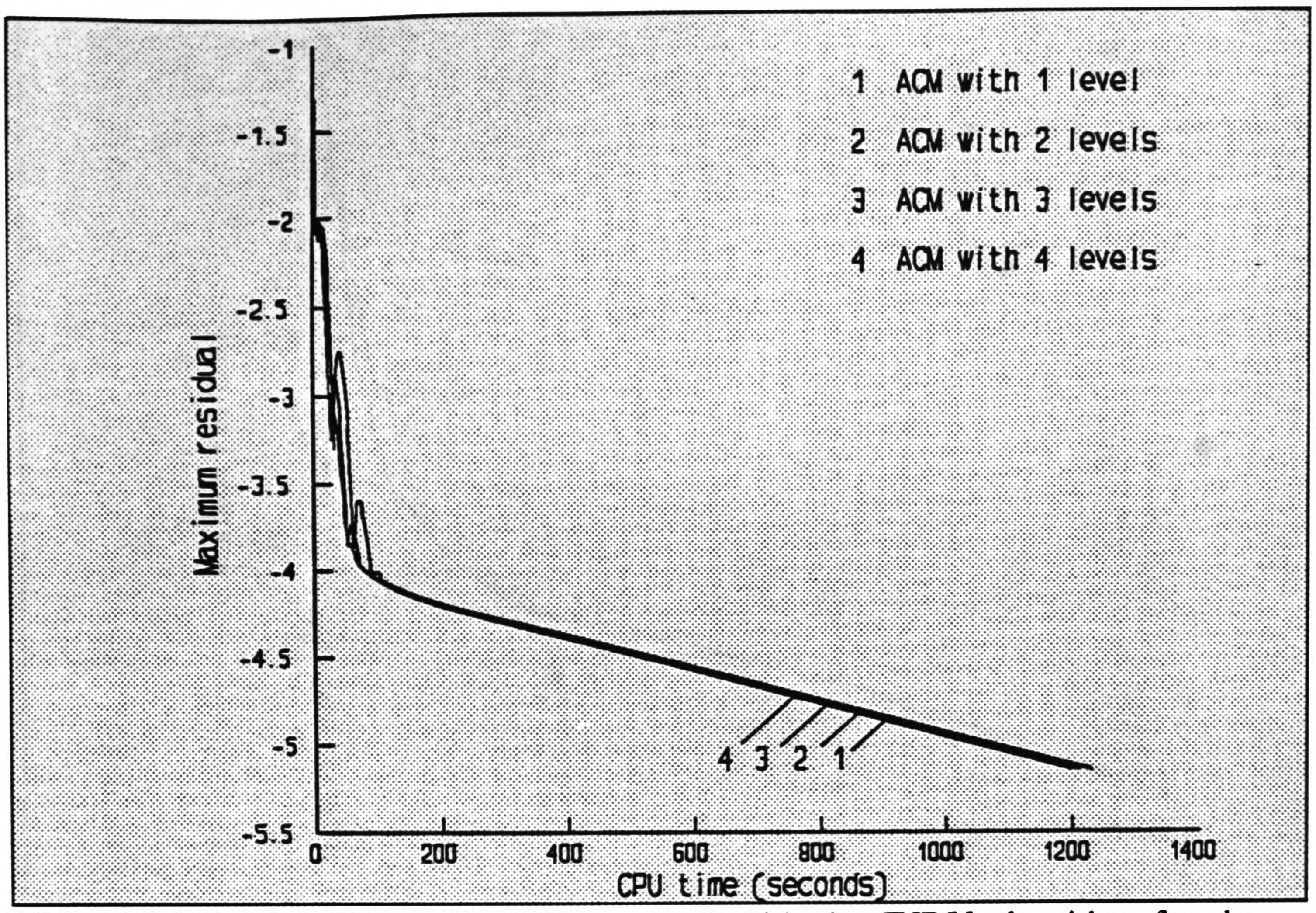


FIGURE 7.9.4-2d Using the ACM method with the JURV algorithm for the solution of the natural convection problem (Ra=10^6)

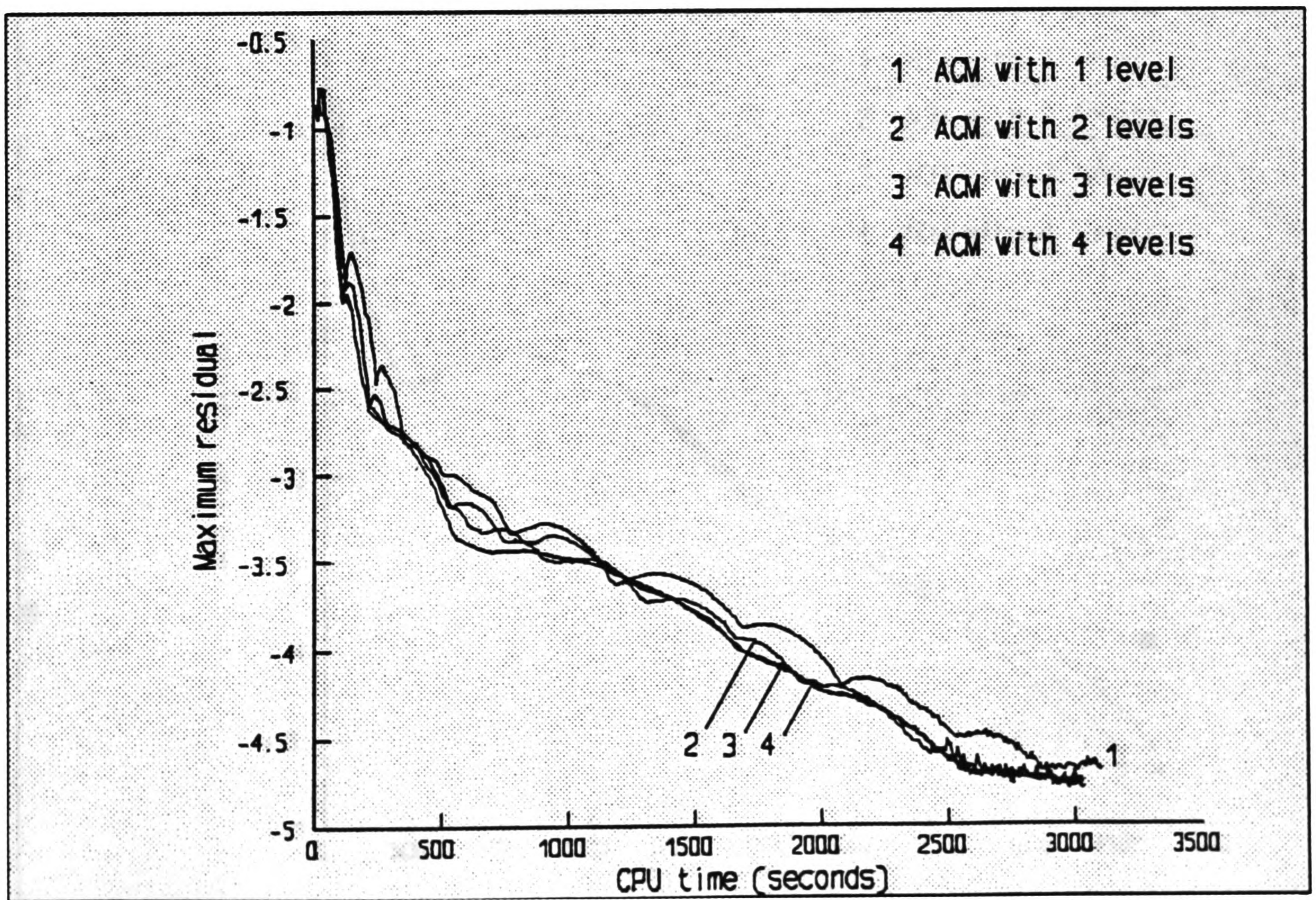


FIGURE 7.9.4-2e Using the ACM method with the JURV algorithm for the solution of the natural convection problem (Ra=10^7)

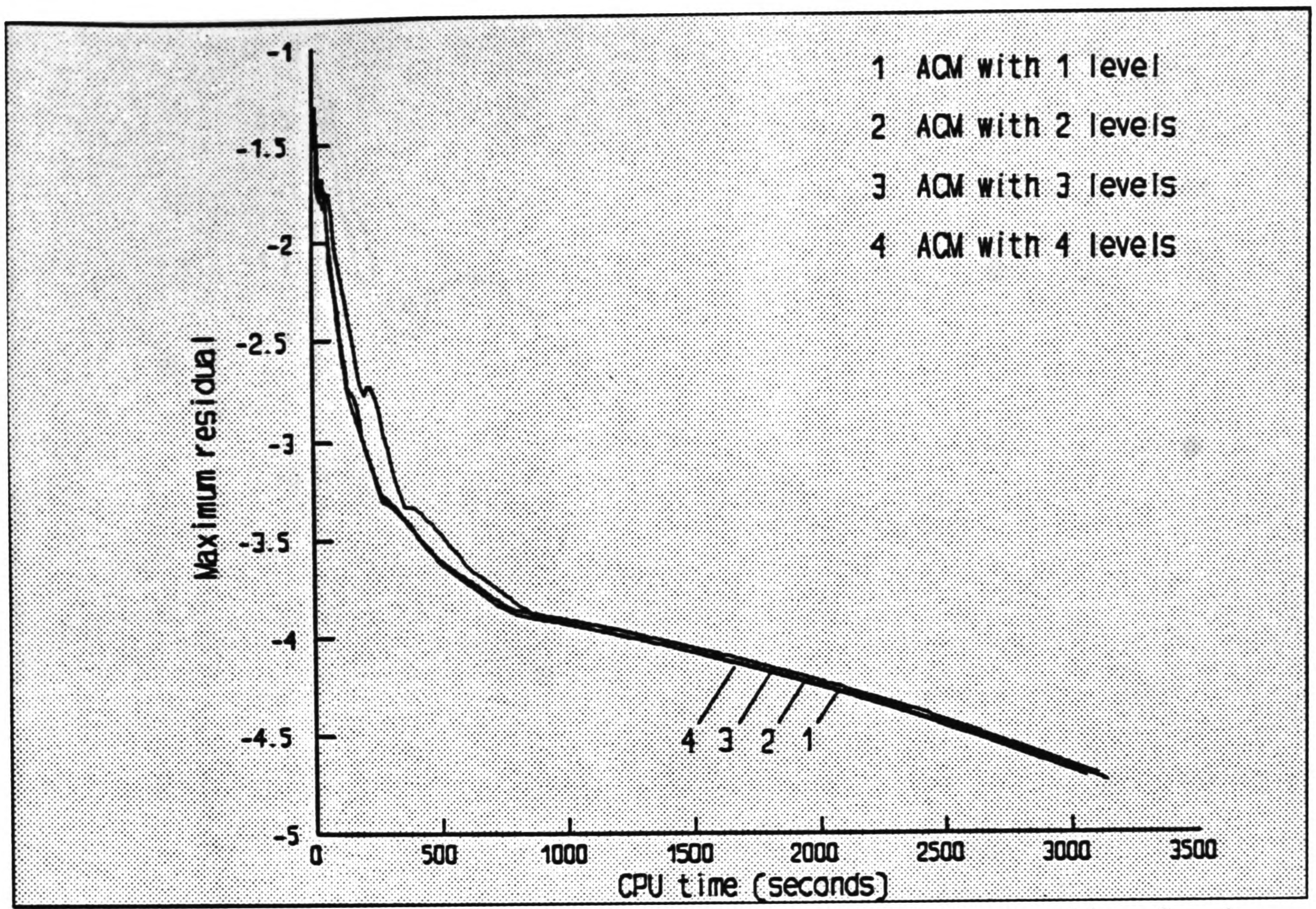


FIGURE 7.9.4-3a Using the ACM method with the LSOR algorithm for the solution of the natural convection problem (Ra=10^3)

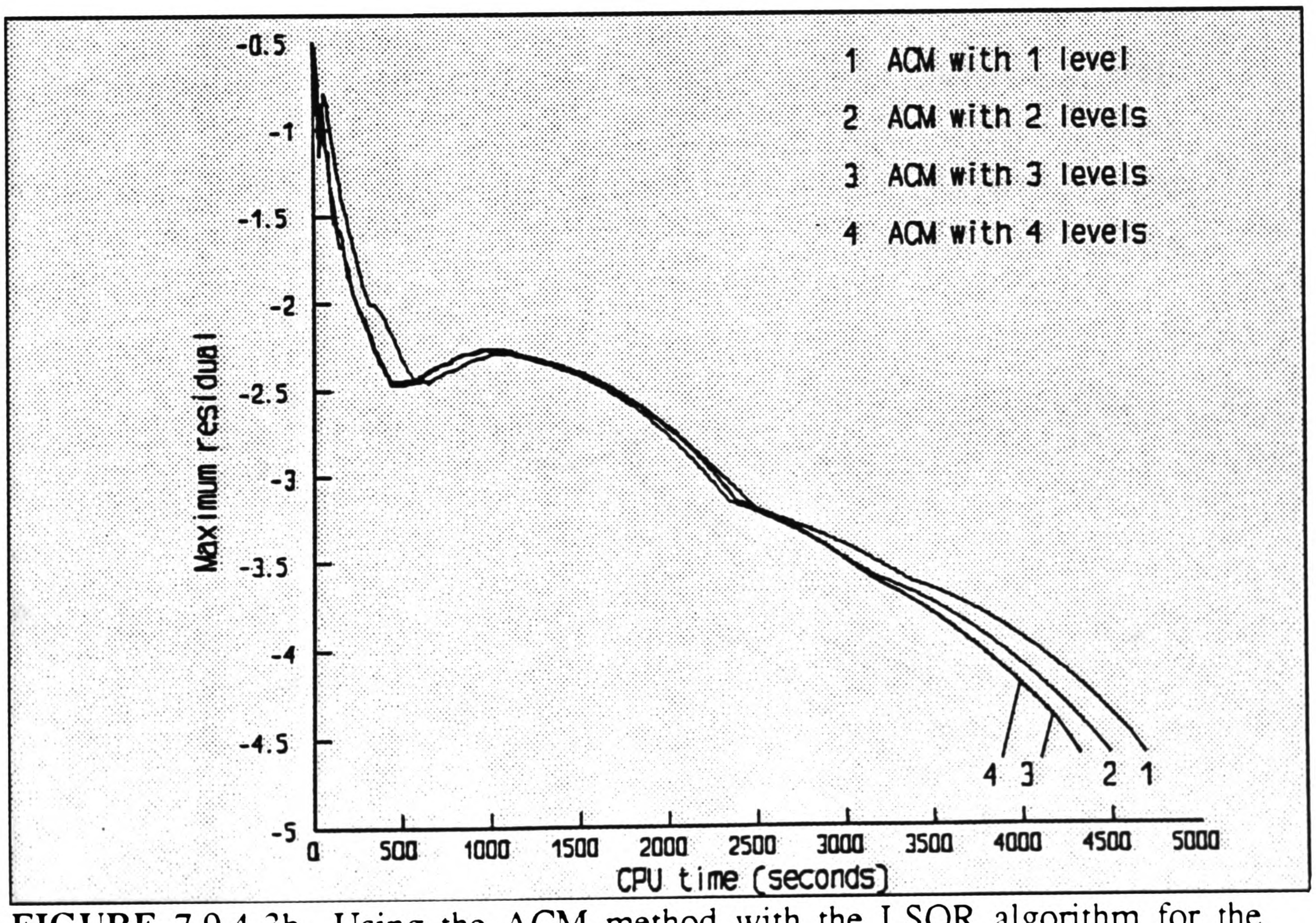


FIGURE 7.9.4-3b Using the ACM method with the LSOR algorithm for the solution of the natural convection problem (Ra=10^4)

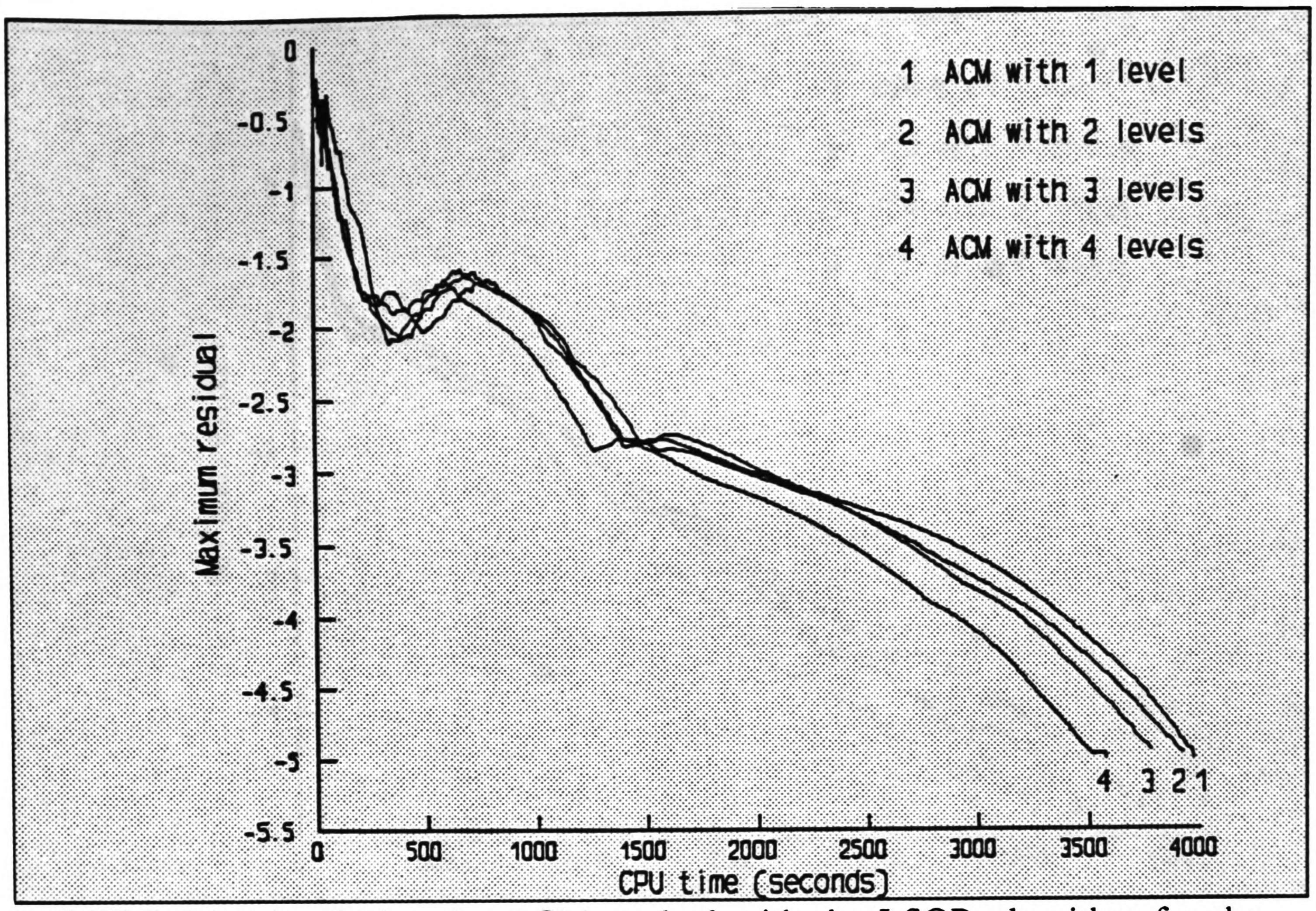


FIGURE 7.9.4-3c Using the ACM method with the LSOR algorithm for the solution of the natural convection problem (Ra=10^5)

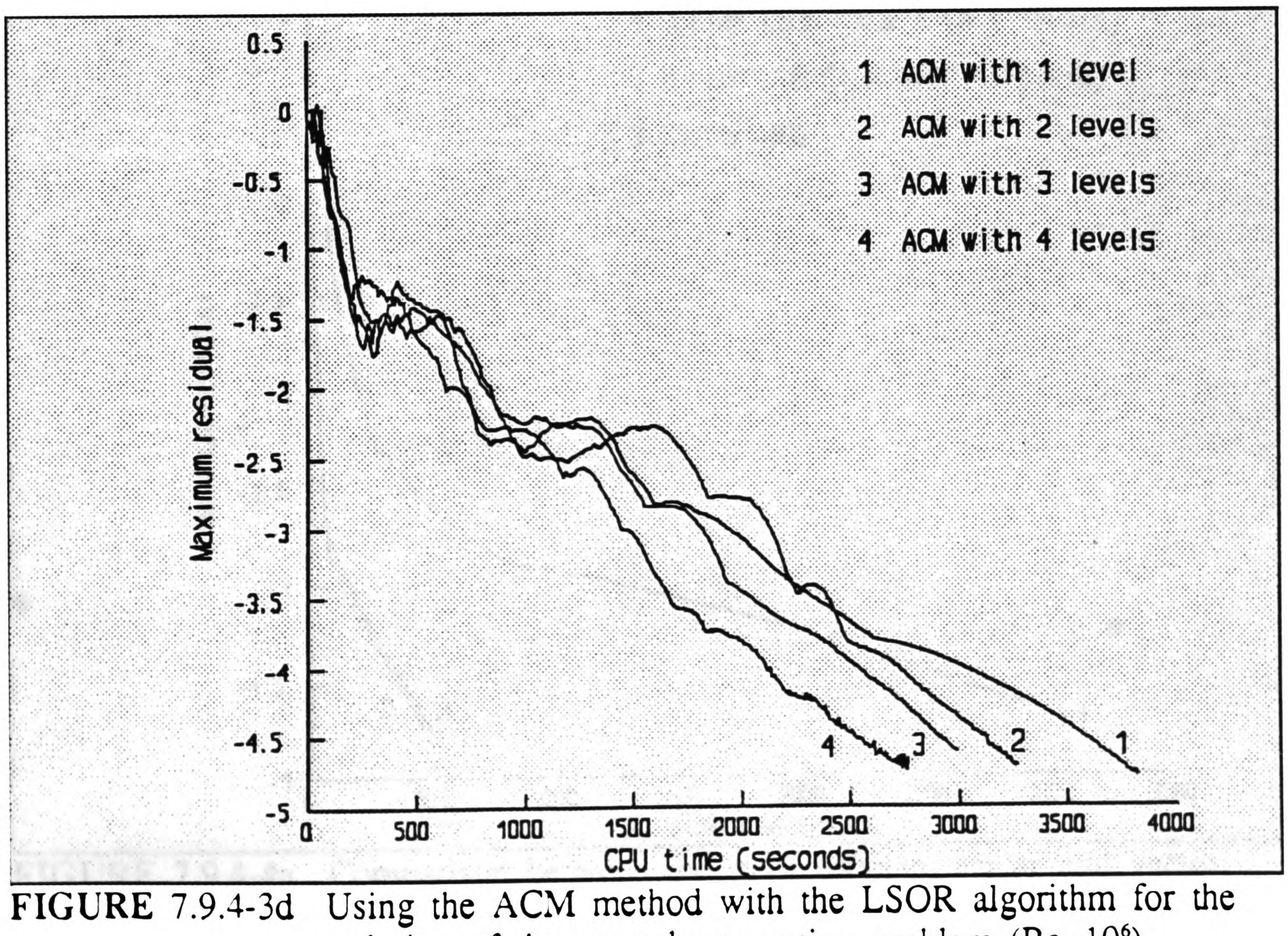


FIGURE 7.9.4-3d Using the ACM method with the LSOR algorithm for the solution of the natural convection problem (Ra=10^6)

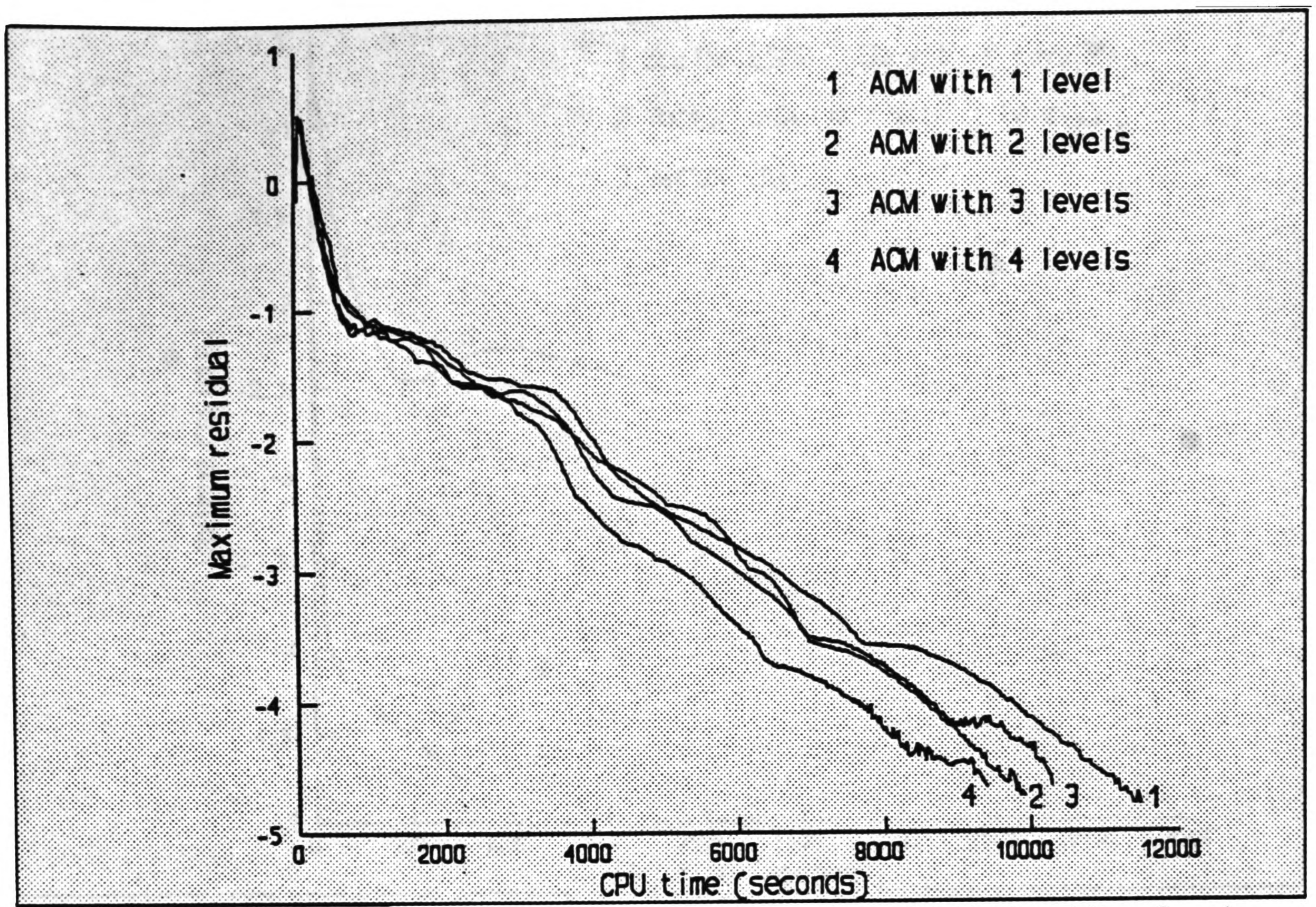


FIGURE 7.9.4-3e Using the ACM method with the LSOR algorithm for the solution of the natural convection problem (Ra=10^7)

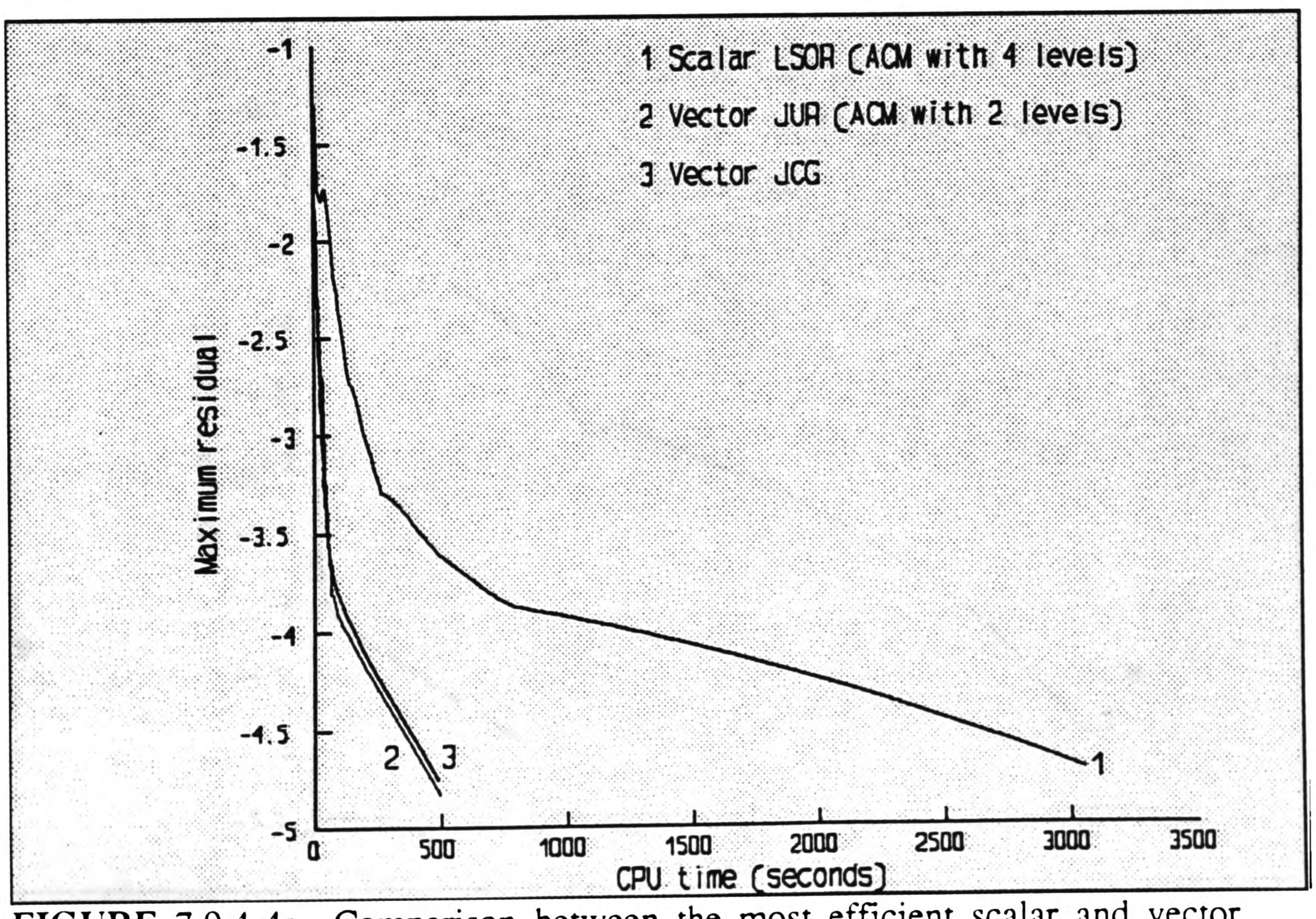


FIGURE 7.9.4-4a Comparison between the most efficient scalar and vector algorithm for the solution of the natural convection problem (Ra=10^3)

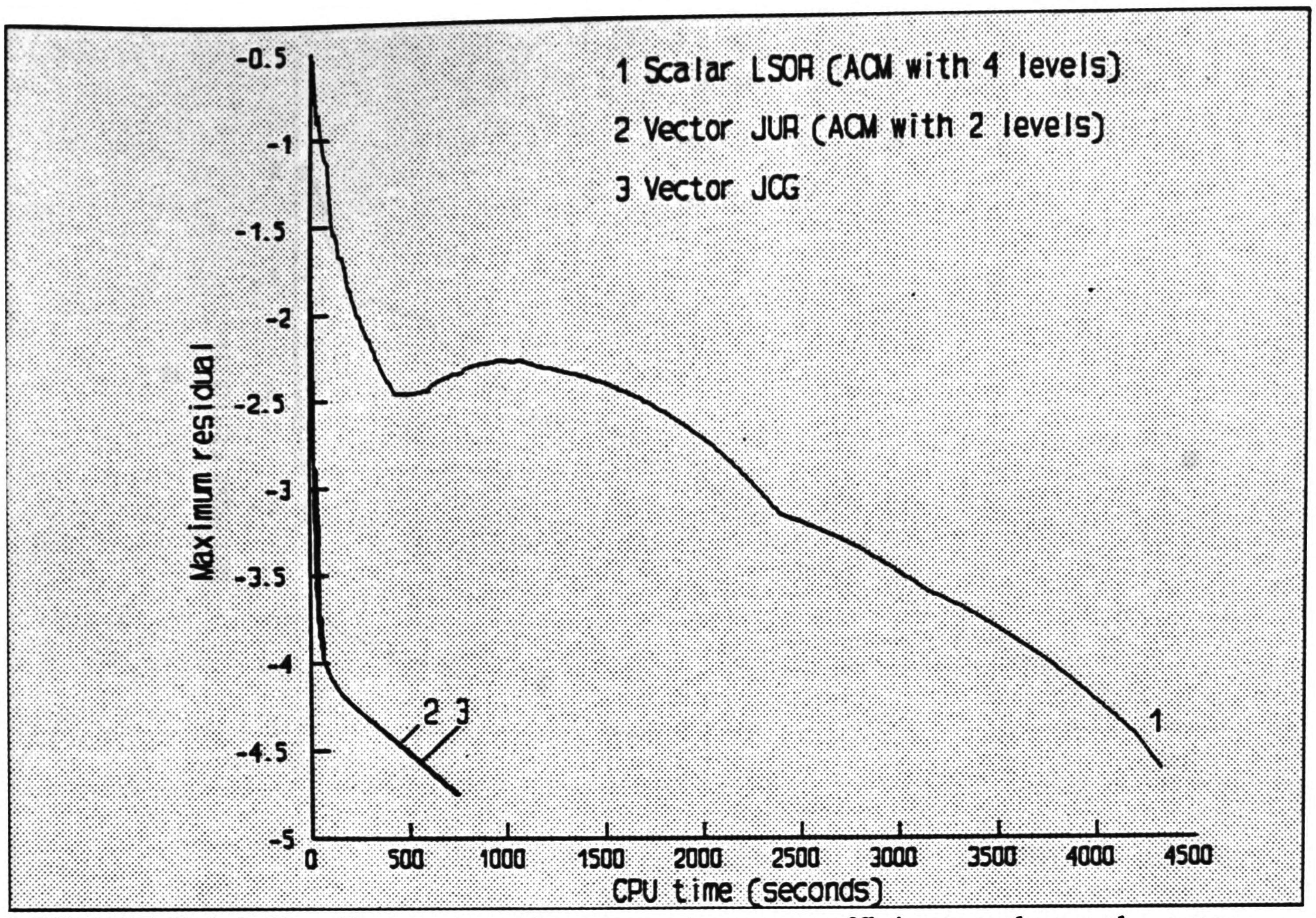


FIGURE 7.9.4-4b Comparison between the most efficient scalar and vector algorithm for the solution of the natural convection problem ($Ra=10^4$)

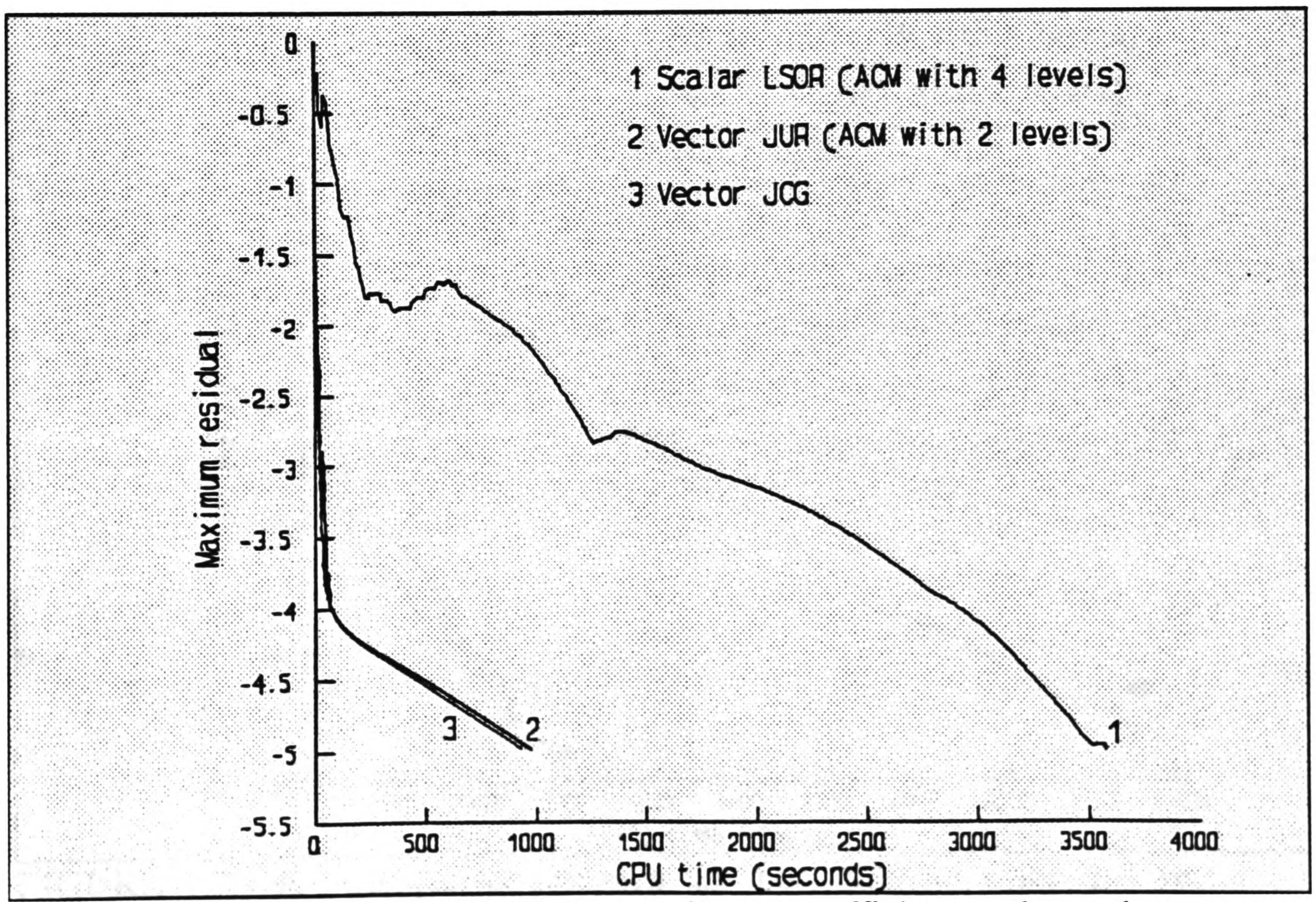


FIGURE 7.9.4-4c Comparison between the most efficient scalar and vector algorithm for the solution of the natural convection problem ($Ra=10^5$)

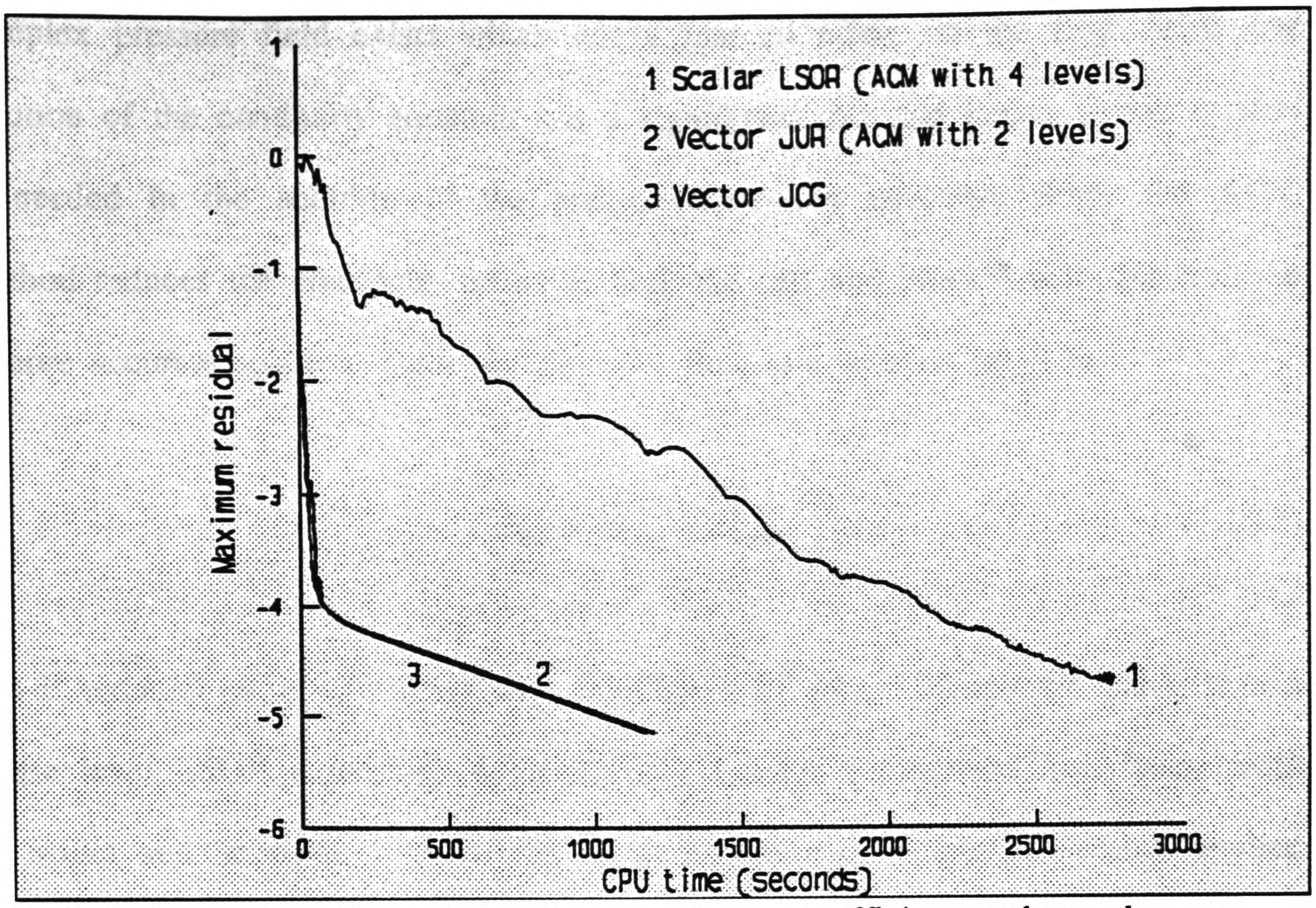


FIGURE 7.9.4-4d Comparison between the most efficient scalar and vector algorithm for the solution of the natural convection problem ($Ra=10^6$)

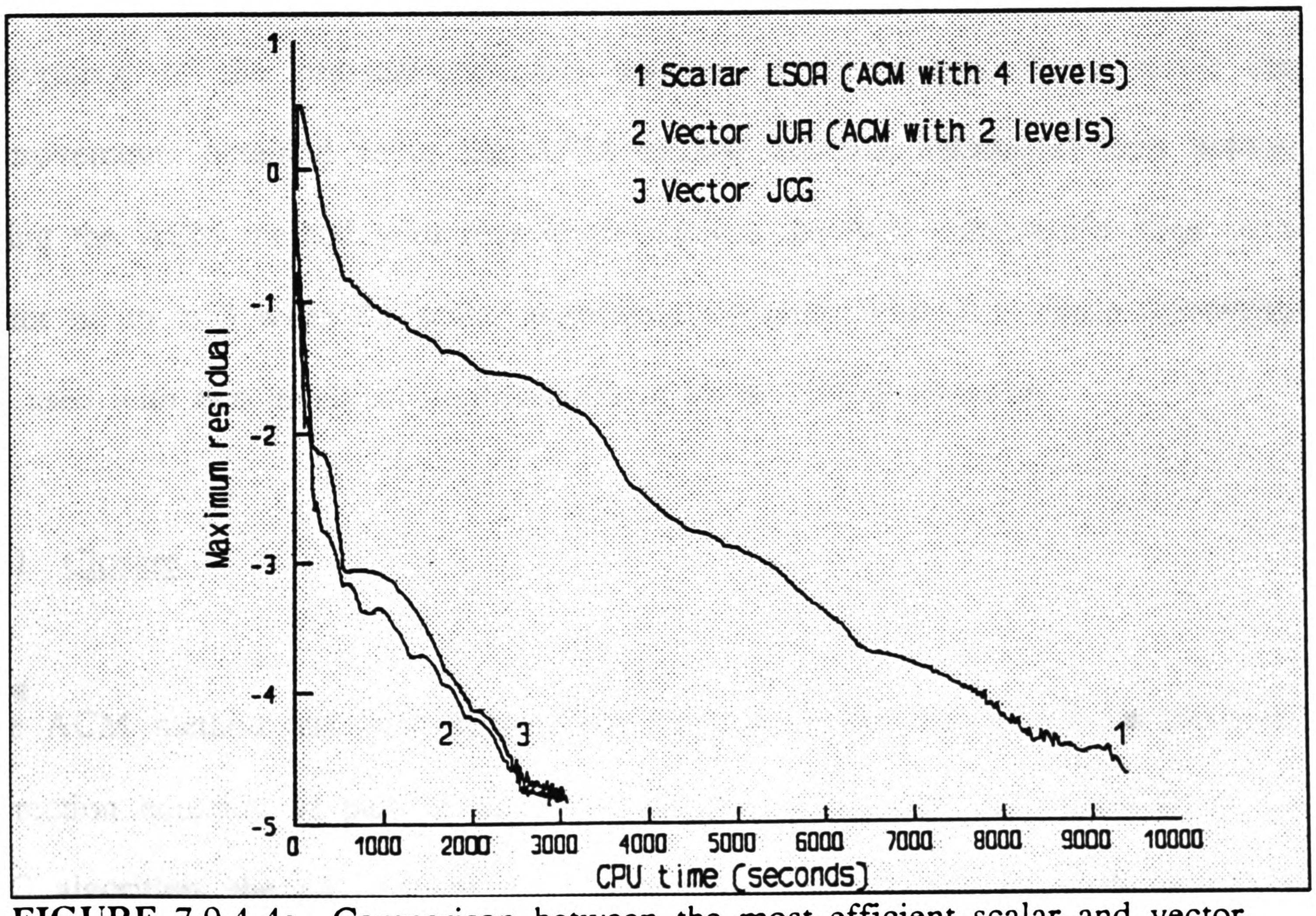


FIGURE 7.9.4-4e Comparison between the most efficient scalar and vector algorithm for the solution of the natural convection problem ($Ra=10^7$)

complex pressure field exists which has a marked effect on the flow field. The solution of the continuity equation has a significant role and hence greater control is needed in the solution of the pressure-correction equation. Since the ACM method reduces the continuity errors at both the cell and block levels this leads to a more accurate solution. Defining the mass flow rate balance ($\dot{m}_{bal}$) as

$$\dot{m}_{bal} = \dot{m}_{inlet} - \dot{m}_{outlet} \tag{7.10-1}$$

this is reduced more effectively with four levels of the ACM method rather than a single level LSOR algorithm (figure 7.10-1). Similar conclusions have been made by Miller and Schmidt [1988] about the performance of multigrid methods applied to open and closed flow problems.

The latter two problems which involve additional scalar equations show that the importance of the pressure-correction solution is diminished. As a result the improvements obtained using the ACM method are less substantial than before. Using the ACM method with four levels reduces the total computation time by a factor of up to 3 when the scalar algorithms are used. When the vector algorithms are used with two levels an improvement of up to 15% is obtained.

7.11 Closure

The ACM method has been successfully applied to the solution of the pressure-correction equation as part of the SIMPLE procedure. With the exception of the JCG algorithm, the method has shown potential in enhancing the performance of classical iterative algorithms (such as JUR and LSOR), and is particularly effective

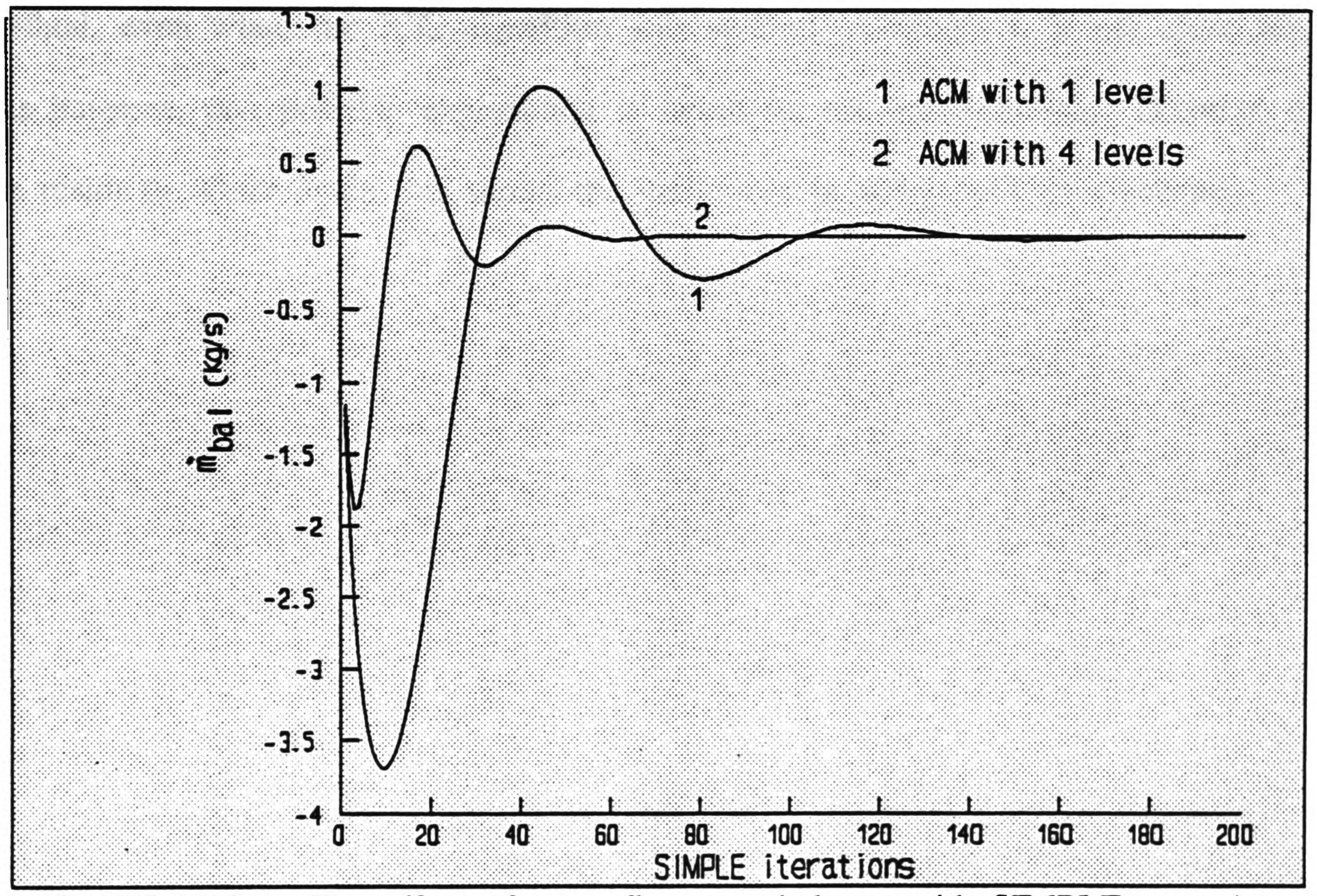


FIGURE 7.10-1 The effect of mass flow rate balance with SIMPLE procedure iterations. The LSOR algorithm is used with the ACM method in problems where there is a strong coupling. The ACM method is less effective when it is vectorised and performs best with two levels. However, as the grid is refined it is likely that more levels will be necessary to improve the efficiency of the method. Despite this, the vectorised ACM method gives improvements of up to 8.5 over the best scalar algorithm when u, v, p are solved, and when scalars are introduced into the computation a factor of 4.5 is achieved.

An unsuccessful attempt was made to combine the JCG algorithm with the ACM method, even when the JCG algorithm was relaxed. Although it is not clear why this happens, one possible explanation is based on the non-monotonic decrease of the residuals. Once again this shows the limitations in using such an algorithm.

CHAPTER EIGHT

8.0 CONCLUSIONS

A detailed investigation was carried out into the use of pipeline vector processors for the solution of CFD problems. The study clearly shows that significant reductions in computation time are possible for the solution of both linear and non-linear problems. This has been made possible because careful consideration is given to all aspects of the solution strategy.

Initially, it seemed as though the vectorisation of the linear equation solvers would be the major obstacle to overcome, so a number of algorithms were used to obtain the solution of Poisson's equation. The resulting linear system of equations were solved using the MASSCOMP VA-1 pipeline processor. In general, the results show that near optimal use can be made of the pipeline architecture, with possible factors of improvement of up to 90. This is not surprising since nearly all the scalar components of the code can be re-designed to fully exploit the pipeline processor.

The work has been extended to the solution of the steady state, incompressible Navier-Stokes equations in two dimensions. The resulting non-linear system of equations were solved using the SIMPLE procedure. An analysis of the distribution of computation effort revealed that the solution of the pressure-correction equation accounted for 65-75% of the total, and as a result, a number of linear equation solvers were vectorised. Although there was a significant reduction in the time taken to solve the pressure-correction equation, the overall reduction in computation time was a factor of 3. In retrospect this was a naive approach, the structure of the SIMPLE procedure was then considered on a more generic level, and an approach was taken to fully exploit the vectorisation. In doing so, it was possible to efficiently vectorise up to 98% of the procedure, and the speed-up factors which were obtained were well supported throughout with theoretical estimates obtained using Amdahl's law. In the test cases studied, improvements in speed between 6 and 29 have been achieved for isothermal problems. The complexity of the problems was then increased to include scalar equations such as temperature and turbulence, and speed-up factors between 5 and 11 were obtained for turbulent problems, where at least 85% of the code was vectorised. These improvements are a practical measure of how much faster an efficient vectorised code is over an efficient scalar code, so the algorithms used in each case are not necessarily the same.

As well as reducing the time taken to execute the SIMPLE procedure by vectorisation, the solution time for the pressure-correction equation is also reduced by means of a multigrid method. The results show that there is some mileage in using such methods, and this is likely to become more obvious as the grid size is increased. On the whole, the results from the test cases studied are very encouraging and give reason to believe that careful vectorisation of the solution procedure can lead to very worthwhile savings in computation time.

The existing formulation can easily be modified to include compressible fluid flow simulations. This would usually consist of an explicit expression which can be updated in a similar fashion to that used for the effective viscosity quantity.

The extension to three dimensions is also seen as straightforward, and either the NEAT or whole-field approach can be used. If the NEAT approach is adopted then the process involves the solution of a two-dimensional slab of nodes rather than a one-dimensional line (when solving a two-dimensional problem). Therefore, the NEAT approach is seen as a natural extension of the existing two-dimensional process. The effort involved in the solution of a two-dimensional slab is comparable to that of the test problems studied here. However, there are two main disadvantages to the NEAT approach. The first is the necessary housekeeping of data, because the solution of a two-dimensional slab will require information from the two neighbouring slabs. The second is more of a vectorisation problem, the largest vector operation which can be performed is only of length n^2 (assuming an n^3 grid). A better proposition may be the whole-field approach. Although it has the disadvantage of requiring a larger amount local memory, it does not require any special housekeeping of the data. Furthermore, an efficient implementation of the solution procedure can be carried out with the largest vector operation being of length n^3.

The vectorising compilers present in today's supercomputers have improved considerably over the years, and are thought by some to have reached a mature state. However, this work has also shown that to exploit pipeline vector architectures fully then a high degree of programmer interaction is still necessary. It is also believed that the techniques employed by the programmer in re-structuring a code for vectorisation on one machine will not necessarily benefit other machines. Ideally, a suite of tools are needed which will attempt to re-structure the scalar code for a given machine. As part of the vectorisation process the machine can be characterised using parameters such as $n_{1/2}$ and r_{∞}, and by considering factors such as the number of processors, and the ability to perform chaining, recursion etc, this information can be used to decide the appropriate vectorisation strategy.

This research has also exposed the limitations of such pipeline architectures. In particular, the speed of the scalar processor is still the major limiting factor. Although the speed of pipeline processors continue to improve and become more affordable (highlighted by the recent launch of the Intel-i860 pipeline processor with a vector rating of 66Mflops), it is necessary to provide a scalar processor with the speed to maintain a good balance between the two.

Multiprocessor systems can overcome this limitation provided the raw power can be harnessed. The systems are made up of processors linked together in some topology and present the programmer with a different form of parallelism. The parallelism can exist at different levels, for example, microtasking and multitasking (Seager [1986]) present in some CRAY systems, or geometrical and farming approaches (Cross et al [1989]) present in transputer-based systems. To date, some of the most promising results for CFD computations have been achieved on transputer-based systems using a geometrical partition strategy (Hockney and Jesshope [1988]), where each processor performs computations on a subset of the entire domain. It is likely that the popularity of multiprocessor systems will continue to grow as more sophisticated tools become available.

It could be concluded that there is a need for both pipeline and multiprocessor architectures to achieve optimal performance. The combination of pipeline processing power together with the simultaneous execution of a number of these processors appears to be a very exciting prospect. This will greatly benefit CFD practitioners, allowing problems of real importance to be modelled at a reasonable cost and in a fraction of the original processing time.

Finally, whilst there is still much work to be done, it is hoped that the present work will stimulate future research into pipeline and multiprocessor-pipeline systems.

<u>REFERENCES</u>

ADAMS L M & JORDAN H F (1986)
'Is SOR Color-Blind'.
SIAM Journal Sci. Stat. Comput., **7**, pp490-506.

AMDAHL G M (1967)
'Validating the Single Processor Approach to Achieving Large Scale Computing Capabilities'.
AFIPS Conference Proceedings, **30**, pp483-485.

BACK L H & ROSCHKE E J (1972)
'Shear-Layer Flow regimes and Wave Instabilities and Re-attachment Lengths Downstream of an Abrupt Circular Channel Expansion'.
Journal of Applied Mechanics, **39**, Trans. ASME Series 7, pp677.

BAKHVALOV N S (1966)
'On the Convergence of a Relaxation Method With Natural Constaraints on the Elliptic Operator'.
USSR Comput. Math. Math. Phys., **6**, pp101-135.

BORIS J P & WINSOR N K (1982)
'Vectorised Computation of Reactive Flows'.
<u>Parallel Computations</u>. Editor G. Rodrigue. Academic Press.

BORREL M, MONTAGNE J L, NERON M, VEUILLOT J & VUILLOT A M (1985)
'Implementation of 3D Explicit Euler Codes on a CRAY-1S Vector Computer', pp47-65.
<u>The Efficient Use of Vector Computers with Emphasis on CFD.</u> Editors W. Schonauer and W. Gentzsch. Vieweg Publishers.

BOZMAN J D & DALTON C (1973)
'Numerical Study of Viscous Flows in a Cavity'.
Journal of Computational Physics, **12**, pp348.

BRANDT A (1977)
'Multi-Level Adaptive Solutions to Boundary-Value Problems'.
Mathematics of Computation, **31**, 333-390.

BRANDT A (1979)
'Multi-Level Adaptive Computations in Fluid Dynamics'.
Proceedings of Fourth AIAA CFD Conference, 100-108.

BRANDT A (1982)
'*Multigrid Solutions to Staedy-State Compressible Navier-Stokes Equations*'.
Computing Methods in Applied Sciences and Engineering. Editors R. Glowinski and J.L. Lions, North-Holland, INRIA.

BRANDT A, DENDY J E & RUPPEL H (1980)
'The *Multigrid Method for Semi-Implicit Hydrodynamics Codes*'.
Journal of Computational Physics, **34**, 348-370.

BRANDT A & DINAR N (1977)
'*Multigrid Solution to Elliptic Flow Problems*', pp53-147.
Numerical Methods in Partial Differential Equations, Editor S. Parter.

BURDEN R L, FAIRES J D & REYNOLDS A C (1981)
Numerical Analysis. (2nd Edition). Prindle, Weber and Schmidt Publishers, Boston, Massachusetts.

BURGGRAF O R (1966)
'*Analytical and Numerical Studies of the Structure of Steady Separated Flows*'.
Journal of Fluid Mechanics, **24**, pp113-151.

BURNS A D, WILKES N S, JONES I P & KIGHTLEY J R (1986)
'*FLOW3D: Body-Fitted Coordinates*'.
Harwell Report AERE-R 12262.

CONCUS P, GOLUB G H & MEURANT G (1985)
'*Block Preconditioning for the Conjugate Gradient Method*'.
SIAM Journal Sci. Stat. Comput., **6**, pp220-252.

CONCUS P, GOLUB G H & O'LEARY (1975)
'*A Generalised Conjugate Gradient Method for the Numerical Solution of Elliptic Partial Differential Equations*'.
Lawrence Berkeley Laboratory Publishers. LBL-4604, Berkeley, California.

CONTE S D & DE BOOR C (1980)
Elementary Numerical Analysis and Algorithmic Approach. (3rd Edition). McGraw-Hill Kogakusha Limited.

CROSS M, JOHNSON S & CHOW P (1989)
'*Mapping Enthalpy-Based Solidification Algorithms onto Vector and Parallel Architectures*'.
Applied Mathematical Modelling, **13**, pp702-709.

DAVIES R (1987)
Vector Accelerators programmer's manual. Order No. M-VA1-PM, Massachusetts.

DENHAM M K & PATRICK M A (1974)
'Laminar Flow Over a Downstream-Facing Step in a Two-Dimensional Flow Channel'.
Trans. Instn. Chem. Engrs., **52**, pp361-367.

DE VAHL DAVIS G (1983)
'Natural Convection of Air in a Square Cavity: A Bench Mark Numerical Solution'.
Int. Journal of Numerical Methods in Fluids, **3**, pp249-264.

DE VAHL DAVIS G & JONES I P (1983)
'Natural Convection in a Square Cavity: A Comparison Exercise'.
Int. Journal of Numerical Methods in Fluids, **3**, pp227-248.

DE VAHL DAVIS G & MALLINSON G D (1976)
'An Evaluation of Upwind and Central Difference Approximations by a Study of Recirculating Flow'.
Computational Fluids, **4**, pp24-43.

DONGARRA J J, BUNCH J R, MOLER C B & STEWART G W (1979)
<u>LINPACK User's guide</u>. SIAM Publications, Philadelphia.

DUBOIS P F, GREENBAUM A & RODRIGUE G H (1979)
'Approximating the Inverse of a Matrix for use in Iterative Algorithms for Vector Processors'.
Computing, **22**, pp257-268.

FEDORENKO R P (1961)
'A Relaxation Method for Solving Elliptic Difference Equations'.
USSR Comput. Math. Math. Phys., **1**, pp1092-1096.

FENNEL D (1988)
<u>Investigation into the King's Cross Underground Fire</u>. Department of Transport, Her Majesty's Stationary Office.

FLYNN M J (1966)
'Very High-Speed Computing Systems'.
Proceedings of the IEEE, **54**, pp1901-1909.

FLYNN M J (1972)
'Some Computer Organizations and Their Effectiveness'.
IEEE trans. on Computers, **c-21**, pp948-960.

FUCHS L (1983)
'New Relaxation Methods for Incompressible Flow Problems', pp627-640.
Numerical Methods in Laminar and Turbulent Flow, Editors C. Taylor et al, Pineridge Press, Swansea.

FUJINO S, TAMURA T & KUWAHARA K (1989)
'Application of the RAINBOW SOR Technique to Fluid Flow Analysis in the 3D Generalised Curvilinear Coordinate System'.
Proceedings of 6th Int. Conference on Numerical Methods in Laminar and Turbulent Flow, Swansea.

GASKELL P H & LAU A K C (1988)
'Curvature Compensated Convective Transport: SMART, a New Boundedness Preserving Transport Algorithm'.
Int. Journal of Numerical Methods in Fluids, **8**, pp617-641.

GASKELL P H & WRIGHT N G (1988)
'Multigrids Applied to a Solution Technique for Recirculating Flow Problems', pp51-65.
Simulations and Optimisation of Large Systems, Editor A. Osiadacz, IMA conference Series, Clarendon Press, Oxford.

GENTZSCH W (1987)
'A Fully Vectorizable SOR Variant'.
Parallel Computing, **4**, pp349-353.

GHIA U, GHIA K N & SHIN C T (1982)
'Solution of the Incompressible Navier-Stokes Equations by a Coupled Strongly-Implicit Multi-Grid Method'.
Journal of Computational Physics, **48**, pp387-411.

GORSLINE G W (1980)
Computer Organization. Englewood Cliffs, N.J. Prentice-Hall.

HACKBUSCH W (1978)
'On the Multi-grid Method Applied to Difference Equations'.
Computing, **20**, pp291-306.

HAGEMAN L A & YOUNG D M (1981)
Applied Iterative Methods. Academic Press, New York.

HAN T, HUMPHREY J A C & LAUNDER B E (1981)
'A Comparison of Hybrid and Quadratic-Upstream differencing in High Reynolds Number Elliptic Flows'.
Computer Methods in Applied Mechanics and Engineering, **29**, pp81-95.

HANDLER W (1977)
'The impact of Classification Schemes on Computer Architectures'.
Proceedings Int. Conf. on Parallel Processing, pp7-15.

HARLOW F H & WELCH J E (1965)
'Numerical Calculation of Time-Dependent Viscous Incompressible Flow of a Fluid with a Free Surface'.
Physics of Fluids, **8**, pp2182-2189.

HEMKER P W, KETTLER R & WESSELING P (1983)
'Multigrid Software for the Solution of Elliptic Problems on Rectangular Domains: MG00 (Release 1)'.
GMD-Studien 70.

HEMKER P W, WESSELING P & ZEEUW P H (1984)
'A Portable Vector-Code for Autonomous Multigrid Modules'.
<u>PDE SOFTWARE: Modules, Interface and Systems</u>. Editors B. Enquist and T. Smedsaas. North-Holland, Amsterdam-New York.

HEMKER P W & de ZEEUW P M (1985)
'Some Implementations of Multigrid Linear System Solvers'.
<u>Multigrid Methods for Integral and Differential Equations</u>. Editors D.J. Paddon and H. Holstein. Oxford University Press, London

HESTENES M R & STIEFEL E (1952)
'Methods of Conjugate Gradients for Solving Linear Systems'.
Journal Res. Nat. Bur. Standards, **49**, pp409-436.

HOCKNEY R W (1965)
'A Fast Direct Solution of Poisson's Equation Using Fourier Analysis'.
Journal Assoc. Comput. Mach., **12**, pp95-113.

HOCKNEY R W (1977)
'Supercomputer Architecture'.
Infotech State of the Art Conference: Future systems, pp65-93.

HOCKNEY R W & JESSHOPE C R (1981)
<u>Parallel Computers. Architectures, Programming and Algorithms</u>. Adam Hilger Ltd.

HOCKNEY R W & JESSHOPE C R (1988)
Parallel Computers 2. Architectures, Programming and Algorithms. Adam Hilger Ltd.

HOLTER W (1985)
'A Vectorised Multigrid Solver for the Three-Dimensional Poisson Equation'.
Presented at the Second Copper Mountain Conference on Multigrid Methods, Copper Mountain, CO.

HUTCHINSON B R & RAITHBY G D (1986)
'A Multigrid Method Based on the Additive Correction Strategy'.
Numerical Heat Transfer, **9**, pp511-537.

IEROTHEOU C S (1987)
'High Level Subroutines for the MASSCOMP MC5400 Vector Accelerator'.
Thames Polytechnic, London.

IEROTHEOU C S, RICHARDS C W & CROSS M (1988)
'Vector Methods for Computational Procedures for Fluid Flow'.
Presented at the 12th IMACS World Congress on Scientific Computation, Paris.

IEROTHEOU C S, RICHARDS C W & CROSS M (1989a)
'Vectorization of the SIMPLE Solution Procedure for CFD Problems - Part I: A Basic Assessment'.
Applied Mathematical Modelling, **13**, pp524-529.

IEROTHEOU C S, RICHARDS C W & CROSS M (1989b)
'Vectorization of the SIMPLE Solution Procedure for CFD Problems - Part II: The Impact of Using a Multigrid Method'.
Applied Mathematical Modelling, **13**, pp530-536.

IRIBARNE A, FRANTISAK F, HUMMEL R L & SMITH J W (1972)
'An Experimental Study of Instabilities and Other Flow Properties of a Laminar Pipe Jet'.
AICHE Journal, **18**, pp689.

ISSA R I (1986)
'Solution of the Implicitly Discretised Fluid Flow Equations by Operator-Splitting'.
Journal of Computational Physics, **62**, pp40-65.

JANG D S, JETLI R & ACHARYA S (1986)
'Comparison of the PISO, SIMPLER and SIMPLEC Algorithms for the Treatment of the Pressure-Velocity Coupling in Steady Flow Problems'.
Numerical Heat Transfer, **10**, pp209-228.

JENNINGS A (1985)
Matrix Computation for Engineers and Scientists. J Wiley and Sons.

JONES I P (1979)
'A Comparison Problem for Numerical Methods in Fluid Dynamics: the "Double-Glazing" Problem', pp338-348.
Numerical Methods in Thermal Problems. Editors R.W. Lewis and K. Morgan. Pineridge Press, Swansea, U.K.

JONES I P, KIGHTLEY J R, THOMPSON C P & WILKES N S (1985)
'FLOW3D, a Computer Code for the Prediction of Laminar and Turbulent Flow, and Heat Transfer: RELEASE 1'.
Harwell Report AERE-R 11825.

JORDAN T L (1974)
'A New Parallel Algorithm for Diagonally Dominant Tridiagonal Matrices'.
Los Alamos Scientific Laboratory Report.

JORDAN T L (1981)
'A Guide to Parallel Computation and Some Experiences'.
LANL Report LA-UR-81-247, Los Alamos National Laboratory, Los Alamos, NM.

KAPITZA H & EPPEL D (1987)
'A 3-D Poisson Solver Based on Conjugate Gradients Compared to Standard Iterative Methods and its Performance on Vector Computers'.
Journal of Computational Physics, **68**, pp474-484.

KASCIC M J Jr (1979)
'Vector Processing on the CYBER 200'.
Infotech State of the Art Report. Supercompters, **2**, pp237-270.

KERSHAW D S (1978)
'The Incomplete Cholesky-Conjugate Gradient Method for the Iterative Solution of Systems of Linear Equations'.
Journal of Computational Physics, **26**, pp43-65.

KIGHTLEY J R & JONES I P (1985)
'A Comparison of Conjugate Gradient Preconditionings for Three-Dimensional Problems on a CRAY-1'.
Comput. Phys. Comm., **37**, pp205-214.

KIGHTLEY J R & THOMPSON C P (1987)
'On the Performance of Some Rapid Elliptic Solvers on a Vector Processor'.
SIAM Journal Sci. Stat. Comput., **8**, pp701-714.

KINCAID D R, OPPE T C & YOUNG D M (1986)
'Vectorised Iterative Methods for Partial Differential Equations'.
Communications in Applied Numerical Methods, **2**, pp289-296.

KOPPENOL P J (1985)
'Simulating 3D Euler Flows on a CYBER 205 Vector Computer', pp71-92.
The Efficient Use of Vector Computers with Emphasis on CFD. Editors W. Schonauer and W. Gentzsch. Vieweg Publishers.

KORDULLA W (1984)
'Vectorisation of Algorithms in Compuattional Fluid Dynamics on the CRAY-1 Vector Computer', pp157-171
Vectorisation of Computer Programs with Applications to CFD. Editor W. Gentzsch. Vieweg Publishers.

LAI C H & LIDDELL H M (1987)
'A Review of Parallel Finite Element Methods on the DAP'.
Applied Mathematical Modelling, **11**, pp330-340.

LAMBIOTTE J & VOIGT R G (1975)
'The Solution of Tridiagonal Linear Systems on the CDC STAR-100 Computer'.
ACM Trans. on Mathematical Software, **1**, pp308-329.

LATIMER B R & POLLARD A (1985)
'Comparison of Pressure-Velocity Coupling Solution Algorithms'.
Numerical Heat Transfer, **8**, pp635-652.

LAUNDER B E & SPALDING D B (1974)
'The Numerical Computation of Turbulent Flows'.
Computer Methods in Applied Mechanics and Engineering, **3**, pp269-289.

LAUNDER B E, REECE G J & RODI W (1975)
'Progress in the Development of a Reynolds Stress Turbulence Closure'.
Journal of Fluid Mechanics, **68**, pp537-566.

LAWSON C, HANSON R, KINCAID D & KROGH F (1979)
'Basic Linear Algebra Subprograms for Fortran Usage'.
ACM Trans. Math. Software, **5**, pp308-371.

LEONARD B P (1979)
'A Stable and Accurate Convective Modelling Procedure Based on Quadratic Upstream Interpolation'.
Computational Methods in Applied Mechanics and Engineering, **19**, pp59-98.

LONSDALE G (1988)
'Solution of a Rotating Navier-Stokes Problem by a Non-Linear Multigrid Algorithm'.
Journal of Computational Physics, **74**, pp177-190.

LONSDALE R D & WEBSTER R (1989)
'The Application of Finite Volume Methods for Modelling Three-Dimensional Incompressible Flow on an Unstructured Mesh'.
Proceedings of 6th Int. Conference on Numerical Methods in Laminar and Turbulent Flow, Swansea.

MACAGNO E O & HUNG T K (1967)
'Computational and Experimental Study of a Captive Annular Eddy'.
Journal of Fluid Mechanics, **28**, pp43-64.

MASDEN N K & RODRIGUE G H (1976)
'A Comparison of Direct Methods for Tridiagonal Systems on the CDC STAR-100'.
Report UCRL-76993, Lawrence Livermore National Laboratory, Livermore, California.

MASSCOMP (1984)
Reference Manual. Order No. 075-00123-00-00, Rev. B, Massachusetts.

MEIJERINK J A & van der VORST H A (1977)
'An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix'.
Mathematics of Computation, **31**, pp148-162.

MELHEM R & GANNON D (1987)
'Toward Efficient Implementation of Preconditioned Conjugate Gradient Methods on Vector Supercomputers'.
Int. Journal of Supercomputer Applications, **1**, pp70-98.

MEURANT G (1984)
'The Block Preconditioned Conjugate Gradient Method on Vector Computers'.
BIT, **24**, pp623-633.

MILLER T F & SCHMIDT F W (1988)
'Evaluation of a Multilevel Technique Applied to the Poisson and Navier-Stokes'.
Numerical Heat Transfer, **13**, pp1-26.

O'LEARY D P (1984)
'Ordering Schemes for Parallel Processing of Certain Mesh Problems'.
SIAM Journal of Sci. Stat. Comput., 5, pp620-632.

PATANKAR S V (1980)
Numerical Heat Transfer and Fluid Flow. Hemisphere, Washington DC.

PATANKAR S V (1981)
'*A Calculation Procedure for Two-Dimensional Elliptic Situations*'.
Numerical Heat Transfer, **4**, pp409-425.

PATANKAR S V & SPALDING D B (1972)
'*A Calculation Procedure for Heat, Mass and Momentum Transfer in Three-Dimensional Flows*'.
Int. Journal of Heat and Mass Transfer, **15**, pp1787-1806.

PATEL M K (1987)
'*On the False-Diffusion Problem in the Numerical Modelling of Convection-Diffusion Processes*'.
PhD Thesis, Thames Polytechnic, London.

PATEL M K, CROSS M & MARKATOS N C (1988)
'*An Assessment of Flow Oriented Schemes for Reducing False Diffusion*'.
Int. Journal for Num. Methods in Eng., **26**, pp2279-2304.

PHILLIPS R E & SCHMIDT F W (1984)
'*Multigrid Techniques for the Numerical Solution of the Diffusion Equation*'.
Numerical Heat Transfer, **7**, pp251-268.

PHILLIPS R E & SCHMIDT F W (1985a)
'*Multigrid Techniques for the Solution of the Passive Scalar Advection-Diffusion Equation*'.
Numerical Heat Transfer, **8**, pp25-43.

PHILLIPS R E & SCHMIDT F W (1985b)
'*A Multilevel-Multigrid Technique for Recirculating Flows*'.
Numerical Heat Transfer, **8**, pp573-594.

POLLARD A (1980)
'*Entrance and Diameter Effects on the Laminar Flow in Sudden Expansions*'.
Momentum and Heat Transfer Processes in Recirculating Flows. Editors Launder and Humphreys, **13**, pp21-26.

PRAKASH C & PATANKAR S V (1985)
'*A Control-Volume Based Finite-Element Method for Solving the Navier-Stokes Equations using Equal-Order Velocity-Pressure Interpretation*'.
Numerical Heat Transfer, **8**, pp259-280.

PUN W M & SPALDING D B (1976)
'A General Computer Program for Two-Dimensional Elliptic Flows'.
Report No. HTS/76/2, Imperial College, London.

RADICATI di BROZOLO G & VITALETTI M (1987)
'Conjugate Gradient Subroutines for the IBM 3090 Vector Facility'.
IBM Technical Report ICE-0010.

RAITHBY G D (1976)
'Skew Upstream Differencing Schemes for Problems involving Fluid Flows'.
Computation Methods in Applied Mechanical Engineering, **9**, pp153.

RAITHBY G D & SCHNEIDER G E (1979)
'Numerical Solution of Problems in Incompressible Fluid Flow: Treatment of the Velocity-Pressure Coupling'.
Numerical Heat Transfer, **2**, pp417-440.

RAITHBY G D & SCHNEIDER G E (1980)
Erratum. Numerical Heat Transfer, **3**, p513.

RAMAMORTHY C V & LI H F (1977)
'Pipeline Architecture'.
Comput. Surv., **9**, pp61-102

RHIE C M & CHOW W L (1983)
'Numerical Study of the Turbulent Flow Past an Airfoil with Trailing Edge Separation'.
AIAA Journal, **21**, pp1525-1532.

RILEY J J & METCALF R W (1980)
'Direct Numerical Simulation of the Turbulent Wake of an Asymmetric Body'.
Turbulent Shear Flows II. Editors Bradbury et al. Springer-Verlag.

RIZZI A & THERRE J P (1985)
'Vector Algorithm for Large-Memory CYBER 205 Simulations of Euler Flows', pp93-116.
The Efficient Use of Vector Computers with Emphasis on CFD. Editors W. Schonauer and W. Gentzsch. Vieweg Publishers.

ROACHE P J (1976)
Computational Fluid Dynamics. Hermosa Publishers. Albuquerque, New Mexico.

RODRIGUE G & WOLITZER D (1981)
'Incomplete Block Cyclic Reduction'.
International Symposium on Parallel Computation, Newark, Delaware.

ROSTEN H I & SPALDING D B (1986)
PHOENICS - Beginners Guide and User Manual. CHAM TR/100 Report.

RUSSEL R M (1978)
'The CRAY-1 Compter System'.
Communications in ACM, **21**, pp63-72.

SCHNEIDER G E & RAW M J (1987)
'Control Volume finite-Element Method for Heat Transfer and Fluid Flow Using Colocated Variables - 1. Computational Procedure'.
Numerical Heat Transfer, **11**, pp363-390.

SCHNEIDER G E & ZEDAN M (1981)
'A Modified Strongly Implicit Procedure for the Numerical Solution of Field Problems'.
Numerical Heat Transfer, **4**, pp1-19.

SCHONAUER W & SCHNEPF E (1988)
'FIDISOL: A Black-Box Solver for Partial Differential Equations'.
Parallel Computing, **6**, pp635-648.

SCHWAMBORN D (1984)
'Vectorisation of an Implicit Finite Difference Method for the Solution of the Laminar Boundary-Layer Equations', pp195-216.
Vectorisation of Computer Programs with Applications to CFD. Editor W. Gentzsch.
Vieweg Publishers.

SEAGER M K (1986)
'Overhead Considerations for Parallelizing Conjugate Gradient'.
Communications in Applied Numerical Methods, **2**, pp273-279.

SETTARI A & AZIZ K (1973)
'A generalization of the Additive Correction Methods for the Iterative Solution of Matrix Equations'.
SIAM Journal of Numerical Analysis, **10**, pp506-521.

SHORE J E (1973)
'Second Thoughts on Parallel Processing'.
Comput. and Elect. Engng. **1**, pp95-109.

SIVALOGANATHAN S & SHAW G J (1988a)
'A Multigrid Method for Recirculating Flows'.
Int. Journal for Numerical Methods in Fluids, **8**, pp417-440.

SIVALOGANATHAN S & SHAW G J (1988b)
'On the Smoothing Properties of the SIMPLE Pressure-Correction algorithm'.
Int. Journal for Numerical Methods in Fluids, **8**, pp441-461.

SMITH G D (1969)
Numerical Solution of Partial Differential Equations. Oxford University Press, London.

SONNEVELD P, WESSELING P & de ZEEUW P M (1985)
'Multigrid and Conjugate Gradient Methods as Convergence Acceleration Techniques'.
Multigrid Methods for Integral and Differential Equations. Editors D.J. Paddon and H.Holstein, Oxford University Press, London.

SPALDING D B (1972)
'A Novel Finite Difference Formulation for Differential Expressions Involving Both First and Second Derivatives'.
Int. Journal for Numerical Methods in Engineering, **4**, pp551-559.

SPALDING D B (1976)
'Basic Equations of Fluid Mechanics and Heat and Mass Transfer and Procedures for their Solution'.
Report No. HTS/76/6, Mechanical Engineering Dept., Imperial College, London.

SPALDING D B (1980)
'Mathematical Modelling of Fluid Mechanics, Heat Transfer and Chemical-Reaction Process'.
Report No. HTS/80/1, Mechanical Engineering Dept., Imperial College, London.

SPRADLEY L W, STALNAKER J F & RATLIFF A W (1981)
'Solution of the Three-Dimensional Navier-Stokes Equations on a Vector Processor'.
AIAA Journal, **19**, pp1302-1308.

STONE H L (1968)
'Iterative Solution of Implicit Approximations of Multidimensional Partial Differential Equations'.
SIAM Journal of Numerical Analysis, **5**, pp530-558.

STONE H S (1973)
'An efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations'.
Journal of ACM, **20**, pp27-38.

SWARZTRAUBER P N (1979)
'A Parallel Algorithm for Solving General Tridiagonal Equations'.
Mathematics of Computation, **33**, pp185-199.

THOMAS L H (1949)
'Elliptic Problems in Linear Difference Equations Over a Network'.
Watson Scientific Computing Lab. Report. Columbia Univ., New York.

TIMIN T & ESMAIL M N (1983)
'A Comparative Study of Central and Upwind Difference Schemes Using the Primitive Variables'.
Int. Journal of Methods in Fluids, **3**, pp295-305.

TRAUB J F (1973)
'Iterative Solution of Tridiagonal Systems on Parallel or Vector Computers'.
Complexity of Parallel Numerical Algorithms. Academic Press, New York.

van der VORST H A (1982)
'A Vectorizable Variant of Some ICCG Methods'.
SIAM Journal Sci. Stat. Comput., **3**, pp350-356.

van der VORST H A (1986)
'The Performance of FORTRAN Implementations for Preconditioned Conjugate Gradients on Vector Computers'.
Parallel Computing, **3**, pp49-58.

Van DOORMAAL J P & RAITHBY G D (1984)
'Enhancements of the SIMPLE Method for Predicting Incompressible Fluid Flows'.
Numerical Heat Transfer, **7**, pp147-163.

VANKA S P (1986)
'Block-Implicit Multigrid Solution of Navier-Stokes Equations in Primitive Variables'.
Journal of Computational Physics, **65**, pp138-158.

VANKA S P & MISENGADES K P (1987)
'Vectorized Multigrid Fluid Flow Calculations on a CRAY X-MP/48'.
Int. Journal for Numerical Methods in Fluids, **7**, pp635-648.

VARGA R S (1962)
Matrix Iterative Analysis. Prentice-Hall, Englewood Cliffs, N.J.

WANG H H (1981)
'A Parallel Method for Tridiagonal Equations'.
ACM Transactions on Mathematical Software, **7**, pp170-183.

WANG Y, HE J & ZANG B Q (1989)
'A Calculation Procedure for Steady Two-Dimensional Elliptic Flows'.
Int. Journal for Numerical Methods in Fluids, **9**, pp609-617.

WESSELING P (1982)
'A Robust and Efficient Multigrid Method'.
Multigrid Methods. Editors W. Hackbusch and U. Trottenberg. Springer-Verlag, Berlin-New York.

WHITEWAY J (1979)
'A Parallel Algorithm for Solving Tridiagonal Systems'.
DAP Newsletter 3, Queen Mary College, London.

YOUNG D M (1971)
Iterative Solution of Large Linear Systems. Academic Press, New York.

APPENDIX 2.9.3

```
      SUBROUTINE TPADDFVV(A,B,C,n)
C
C AUTHOR:  C.S.IEROTHEOU
C DATE:    20/10/1986
C DESCRIPTION:
C    This subroutine will use the VA to add the vector A to B and store
C    the result in C. The VA memory is 32000 32-bit words so it can
C    accomodate at most 2 vectors up to 16000 in length. If n > 16000
C    then we must split the vector into chunks of 16000.
C    NB: This version does not account for non-contiguous vectors.
C
      INTEGER n,IDX,PTR,MAXSIZ,VECLEN
      REAL A(n),B(n),C(n)

      PTR=1
      MAXSIZ=16000
      VECLEN=MIN(n,MAXSIZ)
C
C    Load in chunk of vector A and B, each of size VECLEN, into AP memory.
C
100   IDX=MAPLODFV(A(PTR),4,0,1,VECLEN)
      IDX=MAPLODFV(B(PTR),4,VECLEN,1,VECLEN)
C
C    Since the MATH and DMA routines can run in parallel,all vector chunks are
C    loaded before the addition is carried out. Do this using MAPSYNC routine.
C
      CALL MAPSYNC(IDX)
C
C    Now carry out arithmetic
C
      IDX=MAPADDFVV(0,1,VECLEN,1,0,1,VECLEN)
C
C    Ensure addition is finished before storing result to host memory.
C
      CALL MAPSYNC(IDX)
      IDX=MAPSTRFV(0,1,C(PTR),4,VECLEN)
      CALL MAPSYNC(IDX)
C
C    Update PTR and VECLEN. Check to see if all chunks have been processed.
C
      PTR=PTR+VECLEN
      IF(PTR.LE.n)THEN
        VECLEN=MIN(n-PTR+1,MAXSIZ)
        GOTO 100
      ENDIF
C
C    Finished addition. Ensure all other operations have been completed.
C
      CALL MAPBWAITRBE()
      RETURN
      END
```

Consider the simple vector operation of the addition of two vectors (A and B) of length n, and the storage of the result in a third vector (C). The vector accelerator can be used in one of two ways to carry out the addition:

(i) using the low-level MASSCOMP RTL routines

(ii) using the high-level routine

Code fragments for the two approaches are given below:

(i) MASSCOMP RTL routines

```
              .
              .
              .
IDX=MAPLODFV(A,4,0,1,n)
IDX=MAPLODFV(B,4,n,1,n)
CALL MAPSYNC(IDX)
IDX=MAPADDFVV(0,1,n,1,0,1,n)
CALL MAPSYNC(IDX)
IDX=MAPSTRFV(0,1,C,4,n)
CALL MAPSYNC(IDX)
CALL MAPBWAITRBE()
              .
              .
              .
```

(ii) the high-level routine

```
              .
              .
              .
CALL TPADDFVV(A,B,C,n)
              .
              .
              .
```