Universidade do Porto

PARALELIZAÇÃO EM MECÂNICA DOS FLUIDOS COMPUTACIONAL USANDO HPF

Luis Manuel Alves
(Licenciado em Engenharia de Sistemas de Informação)

Dissertação submetida para a obtenção do grau de
Mestre em Métodos Computacionais em Ciências e Engenharia
pela Universidade do Porto

Dezembro, 2000

Resumo

Neste trabalho realizou-se a paralelização de um programa de Mecânica dos Fluidos Computacional, baseado no algoritmo SIMPLE, usando a linguagem HPF (*High Performance Fortran*). O HPF é uma extensão do Fortran 90 que usa o paradigma de programação *data parallel*, e permite a paralelização para sistemas SMP (*Symmetric Multiple Processing*) e *clusters* SMP.

As simulações foram realizadas no *cluster Bewolf* existente na Faculdade de Engenharia da Universidade do Porto. Este *cluster* é constituído por 23 processadores Intel PIII (o *master* com velocidade de processamento de 550 MHz e os restantes 22 com 450 MHz) ligados por uma rede ethernet dedicada. Cada processador possui um placa de 100 Mbit/s.

O HPF mostrou-se uma linguagem de fácil utilização, permitindo uma rápida adaptação de códigos sequenciais para códigos paralelos. Ao contrário de outras técnicas de paralelização, no HPF as comunicações necessárias à execução de um código paralelo são implementadas pelo compilador, pelo que a modificação do nível de paralelização (número de processadores usados) pode ser conseguida por uma simples alteração de um parâmetro no código fonte. Assim, o HPF é uma técnica de paralelização com baixos custos de manutenção, permitindo com grande facilidade a alteração do código para novas situações de cálculo.

No entanto, o HPF mostrou-se menos eficiente que por exemplo a técnica de decomposição de domínio usando *message passing*, não sendo a ferramenta ideal quando o principal objectivo da paralelização é maximizar o desempenho. Em particular, na simulação do escoamento de uma caixa bidimensional com tampa deslizante, obtiveram-se *speed-ups* de 0.42 para corridas com dois processadores e uma malha de 80×80 nós.

A maior deficiência do HPF está relacionada com os elevados tempos de comunicação que estão associados com as operações de distribuição de dados pelos processadores. No programa de cálculo usado, recorre-se ao método de resolução de sistemas de equações algébricas TDMA, que resolve os sistemas de equações ao longo das duas direcções computacionais. Devido a este facto, a paralelização óbvia do TDMA requer a utilização de uma redistribuição de dados no seu interior, o que impediu a obtenção de um bom desempenho na paralelização.

Esta situação foi ultrapassada com a verificação de que para o problema físico em estudo se obtinha convergência dos resultados utilizando uma única parte do TDMA, o que permitiu triplicar o melhor *speed-up* anteriormente obtido. Com esta modificação no TDMA foram registados valores de *speed-up* entre 1.34 e 9.09, para conjuntos de processadores entre 2 e 20 e uma malha de 800×800 nós. A maior eficiência obtida no trabalho presente foi de 67% para 2 processadores.

Abstract

In the present work the parallelisation of Computational Fluid Mechanics program, based on the SIMPLE algorithm which was implemented using the HPF language (High Performance Fortran). The HPF is a Fortran 90 extension, based in the data parallel programming paradigm, that works in SMP (Symmetric Multiple Processing) systems and SMP clusters.

The simulations were performed in the Bewolf cluster located at the Faculdade de Engenharia da Universidade do Porto. This cluster uses 23 Intel PIII processors (the master at 550 MHz and the others at 450 MHz), connected by a dedicated ethernet network with 100 Mbit/s.

The HPF revealed itself to be an easy programme language, enabling a rapid adaptation from existing sequential codes to parallel codes. Contrary to other parallelisation techniques, in HPF codes the communications are implemented by the compiler rather than by the user, and the parallelisation level (the number of processors in use) can be modified by changing a single parameter in the source code. So, the HPF is a parallelisation technique with small maintenance costs, not requiring large modifications of the source code for different situations.

However, the HPF showed itself to be less efficient than, for example, the domain decomposition technique with communications programmed using message passing languages. Thus, when the maximum performance is the final aim HPF should not be used. In our case, the simulation of the fluid flow inside a square cavity with sliding lid, the HPF produced speed-up of 0.42 for two processors using 80×80 grid.

The biggest HPF deficiency is the large communication times associated with data distribution instructions. In our code, the TDMA solver was used for solving the resulting algebraic equations in the two computational directions. The obvious parallelisation solution for the TDMA solver uses a data redistribution instruction in the algorithm (TDMA) which prevents good parallelisation results.

This difficulty was suppressed by verifying that, in the actual physical problem, the use of one single part of the solver was enough for the obtention of converged results, allowing the triplication of the previous maximum speed-up. Using this technique, the speed-up results were between 1.34 and 9.09, using between 2 and 20 processors for a 800×800 grid.

The biggest efficiency value was 67% with 2 processors.

Resumee

Ce travail comporte la parallélisation d'un programme de calcule en Mécanique des Fluides, basé sur l'algorithme SIMPLE, utilisant le langage HPF (*High Performance Fortran*). Le langage HPF est une extention du FORTRAN 90 qu'utilize le paradigme de programmation *data parallel*, et qui permet la parallélisation des systèmes SMP (*Symmetric Multiple Processing*) et *clusters* SMP.

Les simulations ètaient realisées par *cluster Bewolf* existant à la Faculdade de Engenharia da Universidade do Porto. Ce *cluster* se constitue de 23 processeurs Intel PIII (maitre avec une vitesse de 550 MHz et les 22 restants avec 450 MHz) liés par un réseaux ethernet dedicacé. Chaque processeur possède une plaque de 100 Mbit/s.

Le HPF se revèle un langage facile à programmer, permettant une adaptation rapide des codes séquentiels aux codes parallèles. Au contraire des autres techniques de parallélisation, dans le langage HPF les communications nécéssaires à l' exécution d' un code parallel sont implementées par le compilateur, pour que la modification du niveau de parallélisation (numéro de processeurs utilisés) peut être suivie par un simple changement d' un paramètre du code source. Comme cela, le langage HPF est une technique de parallélisation avec un coût de maintenance plus faible, permet facilement le changement du code pour la nouvelle situation de calcule.

Le langage HPF a montré moins de performance que, par example, la technique de décomposition de domaine utilisant *message passing*, n' étant pas le moyen idéal quant au principal objectif de la parallélisation est de maximiser la performance. En particulier, dans la simulation de l' écoulement de fluide incompressible dans une caisse avec un couvercle glissant. Dans cette caisse, nous avons un écoulement de fluide couser par le glissement du couvercle. Dans ce cas nous avons eu un calcul avec un *speed-up* égale à 0.42 dans l' utilisation de deux processeurs et un maillage de 80×80 noeuds.

Le plus grand défaillance du langage HPF est liée au temps élevé de communication entre les differents processeurs. Dans le programme de calcul utilisé, nous avons procédé à les résolution du systéme linéaire à l' aide de l' algorithme TDMA dans le cas bidimensionel. En effet, la parallélisation de TDMA demande l' utilisation de la redistribution des données au sein de l' algorithme (TDMA) même, ceci empêche la bonne performance de la parallélisation.

Cette situation était autre passé avec la vérification de la convergence du problème physique utilisant seulement une simple partie de l' algorithme TDMA. Ceci permet de triplet le meilleur *speed-up* obtenu. Avec cette modification dans l' algorithme TDMA étaient en registrées les valeurs du *speed-up* de 1.34 à 9.09 pour 2 à 20 processeurs respectivement dans le cas d' un maillage de 800×800 noeuds. La meilleur rendement (rapport entre le *speed-up* et le nombre de processeur) obtenu dans ce travail est de l' ordre de 67% pour deux processeurs.

Agradecimentos

Este trabalho decorreu sob orientação dos Doutores Fernando Aristides Castro e Alexandre Silva Lopes, a quem desde já muito agradeço, a ajuda, o trabalho e a disponibilidade dispensada.

Agradeço à Escola Superior de Tecnologia e de Gestão de Bragança toda a colaboração prestada.

Agradeço à Faculdade de Engenharia da Universidade do Porto, nomeadamente ao Centro de Computação (CICA), a disponibilização do *cluster Bewolf* para a realização de simulações.

Por último, agradeço à minha família, especialmente à minha esposa, a constante motivação, encorajamento e confiança que me incutiram.

Nomenclatura

Caracteres Romanos

A_P, A_E, etc.	coeficientes das equações algébricas
L	comprimento; comprimento de desenvolvimento de uma camada limite
P	Pressão
Re	número de Reynolds
S_n	*Speed-up*
$S_{n_{max}}$	*Speed-up* máximo
T_1	tempo de execução em modo sequencial (1 processador)
T_n	tempo de execução em modo paralelo (n processadores)
T_P^{comp}	tempo de cálculo por processador
T_P^{overh}	tempo de comunicações
T_s	tempo de execução em modo sequencial (1 processador)
U, u	componente segundo x do vector velocidade
U_i, U_j	campo de velocidade
U_{lid}	velocidade da tampa
n	número de processadores
s	componente sequencial de um algoritmo
p	componente paralelizável de um algoritmo
x^i	coordenadas de um sistema de coordenadas Cartesiano
x^j	coordenadas de um sistema de coordenadas Cartesiano

Caracteres Gregos

ϵ	eficiência de paralelização
μ	viscosidade dinâmica
ρ	massa volúmica
Φ	escalar genérico

Operadores

∂/∂	derivada parcial

Abreviaturas

ADI	*Alternate Direction Implicit*
ESS	*Earth and Space Science*
CESDIS	*Center of Excellence in Space Data and Information Sciences*

CFD	*Computacional Fluid Dynamics*
CPU	*Central Processing Unit*
DNS	*Direct Numeric Simulation*
HPC	*High Performance Computing*
HPF	*High Performance Fortran*
HPFF	*High Performance Fortran Forum*
LAPACK	*Linear Algebra Package*
LES	*Direct Numeric Simulation*
MIMD	*Multiple Instruction Multiple Data*
MPI	*Message Passing Interface*
MPP	*Massively Parallel Processors*
NOW	*Network of Workstations*
PGI	*The Portland Group, Incorporation*
PVM	*Parallel Virtual Machine*
PWIM	*Pressure-Weighted Interpolation Method*
SGI	*Silicon Graphics Incorporated*
SIMD	*Single Instruction Multiple Data*
SIMPLE	*Semi-Implicit Method for PressureLinked Equations*
SMP	*Symetric Multiple Processing*
TDMA	*Tri-Diagonal Matrix Algorithm*

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação e objectivos

O principal motivo impulsionador do desenvolvimento da computação de alta *performance*, em particular a computação paralela, é o sucesso da ciência computacional na descrição de fenómenos naturais por simulações numéricas. Assim, em particular, muitos são os problemas de interesse prático em Mecânica dos Fluidos que não podem ser resolvidos analiticamente, devido à complexidade das equações matemáticas. Para ultrapassar esta questão recorre-se a métodos numéricos e técnicas que permitem a utilização de computadores, tornando-se assim possível a resolução de um maior número de problemas. Surgiu assim um nova área da investigação, a Mecânica dos Fluidos Computacional - CFD (*Computacional Fluid Dynamics*).

As vantagens dos métodos numéricos são em geral o seu baixo custo, a rapidez na obtenção de resultados, o acesso a informação mais detalhada e a possibilidade de realizar simulações de casos em condições ideais e reais. No entanto, a qualidade dos resultados está limitada pela qualidade dos modelos matemáticos e pelas técnicas empregues, sendo muitas vezes necessário validar os resultados numéricos com ensaios experimentais.

O tempo computacional e o tamanho da memória limitam os problemas que podem ser resolvidos em computadores sequenciais. O aumento da *performance* destes computadores está limitado por factores económicos (o custo da produção das novas tecnologias) e pelas leis da Física (a informação num processador não pode deslocar-se mais rápido do que a velocidade da luz, e a distância entre caminhos da informação é limitada pelas leis dos mecanismos quanta). No entanto, estes recursos podem ser fornecidos massivamente pelo uso de supercomputadores paralelos. Surge assim, uma nova área da ciência computacional, a Computação Paralela.

Os computadores paralelos podem apresentar uma complexidade variada: uma máquina

pequena pode conter um conjunto de processadores ou uma máquina grande pode ter milhares de processadores ligados por redes de comunicação especializadas. A classificação destas arquitecturas pode tornar-se problemática, no entanto existem modelos simples de arquitecturas de computadores paralelos que transmitem as características mais importantes. Basicamente, qualquer computador paralelo consiste em três elementos principais: processadores, memórias e uma rede de comunicação entre esses elementos. Um modo típico de classificar estas arquitecturas é usar a taxinomia Flynn que as associa ao conjunto de instruções e de dados. Temos assim, as arquitecturas SIMD (*Single Instruction Multiple Data*) e MIMD(*Multiple Instruction Multiple Data*).

A arquitectura SIMD consiste num conjunto de processadores, com alguma memória local, que executam a mesma instrução em *lockstep* (isto é, sem avançar para uma nova instrução), proveniente de um processador controlador, sobre uma parte dos dados residentes nessa memória.

A arquitectura MIMD consiste num conjunto de processadores que podem executar instruções individuais. É usual subdividir-se esta arquitectura mediante a relação entre processador e memória. Assim, temos arquitecturas de memória distribuída, memória partilhada e memória partilhada virtual.

Quanto aos paradigmas da programação, dois estilos têm emergido e ganho aceitação pela comunidade de utilizadores, a programação de *message passing* e de paralelização de dados (*data parallel*).

No paradigma *message passing* um programa dividido é carregado em cada processador, normalmente numa máquina MIMD. Cada programa é escrito numa linguagem *standard*, por exemplo C ou Fortran 90, onde a manipulação dos dados é controlada por chamadas a rotinas de comunicação existentes numa biblioteca de comunicação. O programador completa o controle da distribuição de dados e comunicação entre processadores, organizando esses processos de forma a operarem colectivamente. Este paradigma permite ao programador uma grande liberdade no controle do programa, mas exige deste uma total responsabilidade de fazer o programa funcionar, mantendo os processadores ocupados e suavizando as comunicações.

A ideia subjacente ao paradigma *data parallel* é que todas as operações com *arrays* sejam executadas em paralelo. Tipicamente, um único programa controla a distribuição e operação dos dados em todos os processadores. Esta distribuição e comunicação entre processadores é feita pelo compilador, com indicações do programador. Neste paradigma, existe o conceito de transferência da responsabilidade dos detalhes de baixo nível da programação do programador para o compilador, libertando-o para se concentrar na aplicação. Podem ser várias as linguagens que implementam este paradigma, por exemplo o C ou Fortran *standard* com as respectivas extensões que executam o paralelismo, ou então linguagens *data parallel* especificas para determinada máquina.

Tipicamente, nas máquinas paralelas o problema a ser resolvido é distribuído por to-

dos os processadores disponíveis, de forma a produzir a mesma solução que em máquinas sequenciais. O objectivo de escolher uma decomposição apropriada do problema (para a arquitectura alvo e algoritmo utilizado) é fornecer um carregamento igualmente balanceado, de forma a manter os processadores ocupados e minimizar as comunicações entre processadores. Os diferentes paradigmas de programação permitem modos de controle diferentes destas características. O desenvolvimento de algoritmos para sistemas paralelos começa a ser muito importante (Elisseev, 1998).

O trabalho presente tem como objectivo a paralelização de um programa de CFD, baseado no algoritmo SIMPLE (*Semi-Implicit Method for Pressure Linked Equations*) (Patankar, 1980), recorrendo às linguagens Fortran 90 e HPF (*High Performance Fortran*). O HPF desenvolvido pelo HPFF (*High Performance Fortran Forum*), é uma linguagem que usa o paradigma de paralelização de dados (*data parallel*) para arquitecturas SIMD e MIMD. A actualidade e interesse cientifico nesta área é bem demonstrada pelo envolvimento quer da indústria, quer dos meios académicos. Nestes, incluem-se universidades Americanas, como é o caso da University of Minnesota, Rice University entre outras, e universidades Europeias, como é o caso da University of Edinburg e da University of Liverpool. O HPF também, desde 1991 tem sido motivo para a realização de vários encontros entre especialistas de todo o mundo. As primeiras discussões sobre o HPF ocorreram num destes eventos, denominado *Supercomputing' 91*.

Pretende-se também obter uma comparação entre o desempenho da paralelização aqui realizada com a efectuada em Areal (1999), que recorre à técnica de decomposição de domínio e ao paradigma de *message passing* implementado em PVM (*Parallel Virtual Machine*) (Geist et al., 1994).

O trabalho presente está dividido em quatro capítulos:

- Na parte restante do capítulo 1, descrevem-se o HPF e Fortran 90. Refere-se ainda um conjunto de directivas HPF. A seguir, apresenta-se um comparação entre o HPF e outras técnicas de paralelização, recorrendo a ferramentas como é o caso do OpenMP e MPI (*Message Passing Interface*) (Forum, 1994). Faz-se uma descrição dos meios computacionais utilizados e apresenta-se uma revisão da bibliografia seleccionada.

- O capítulo dois apresenta uma descrição do programa de Mecânica dos Fluidos Computacional e as tarefas envolvidas na sua paralelização. Refere-se, com algum pormenor, a forma de paralelização das rotinas de cálculo dos coeficientes e do *solver*. Com o objectivo de avaliar o efeito da paralelização, apresentam-se os resultados e sua análise de testes efectuados a estas rotinas.

- O capítulo três apresenta uma descrição do problema físico escolhido para obter o desempenho da paralelização efectuada. Trata-se de um escoamento laminar numa cavidade com tampa deslizante. A seguir, apresentam-se os resultados e respectiva análise relativamente à paralelização deste problema, para simulações com um número de Reynolds igual a 100.

- No capítulo quatro são apresentadas as principais conclusões e sugestões para trabalho futuro.

1.2 High Performance Fortran e Fortran 90

As primeiras discussões sobre o HPF surgiram no encontro internacional *Supercomputing' 91*, organizado em 1991 na cidade de Albuquerque nos E.U.A.. Aqui, reuniram-se vários especialistas da computação, quer da indústria quer do meio académico, proporcionando a formação do HPFF. No início do ano seguinte realizou-se o primeiro encontro do HPFF em Houston, Texas nos E.U.A.. Neste ano, vários encontros se seguiram, levando à publicação da especificação HPF V. 1.0 em Maio de 1993.

Desde o início ficou claro que a filosofia de base do actual HPF devia ser associada à linguagem Fortran, uma vez que é a linguagem dominante na programação cientifica e também pelo facto dos *Fortran Optimizers* já efectuarem muita da análise necessária para gerar código paralelo eficiente.

O HPF é assim uma extensão do Fortran 90, utilizável quer em arquitecturas SIMD quer MIMD, que adiciona directivas ao código que instruem os compiladores quanto à forma de distribuição de dados e quanto ao paralelismo das instruções, sendo as trocas de mensagens (*message passing*) implementadas pelo compilador.

As instruções de HPF são introduzidas no código Fortran na forma de comentários, permitindo também a sua compilação em compiladores convencionais de Fortran 90. Alguns compiladores permitem a geração automática de código HPF não conduzindo normalmente a um código tão eficiente.

O Fortran 90 baseia-se no Fortran 77, acrescentando-lhe novas características, como é o caso dos apontadores, tipos de dados definidos pelo utilizador, módulos, subrotinas recursivas, alocação dinâmica de memória, operações com *arrays* e novas funções intrínsecas, assim como a introdução de um formato livre para o código e *interfaces* explicitas. No entanto, as características mais relevantes para a programação em paralelo são a nova forma de declaração de atribuição de *arrays* e as novas funções intrínsecas (Foster, 1995).

Outra vantagem do Fortran 90 é que todas as funções intrínsecas aplicadas a escalares podem também ser aplicadas a *arrays*, só que neste caso a função é aplicada a cada elemento do *array* (Foster, 1995).

Em 1997 o Fortran 90 foi revisto dando origem ao Fortran 95. O Fortran 95 adiciona novas características à linguagem nomeadamente, um novo ciclo FORALL, funções PURE e alguns novos procedimentos intrínsecos. Esta nova versão, vem clarificar numerosas ambiguidades do Fortran 90 *standard*.

Resumidamente, o HPF é uma extensão do Fortran que suporta três objectivos essenci-

ais, a saber, programação paralela, *performance* elevada em computadores MIMD e SIMD, com um custo não uniforme de acesso à memória e adaptação a várias arquitecturas. A declaração FORALL e muitas outras funções intrínsecas foram desenhadas essencialmente para ir ao encontro do primeiro objectivo, enquanto que as directivas de distribuição de dados e algumas outras directivas vão ao encontro do segundo objectivo. Os procedimentos extrínsecos permitem o acesso a uma programação de alto nível, o que leva ao terceiro objectivo, embora o aperfeiçoamento da *performance* usando outras características seja também possível.

A declaração FORALL, não sendo uma directiva HPF, deve ser destacada pela sua utilização na paralelização de programas. Assim, esta declaração permite atribuições mais gerais para secções de um *array*. Esta declaração, tem o seguinte formato geral:

```
FORALL (expr1, ... , exprN, condição)
     instr1
     .
     .
     .
     instrN
END FORALL
```

onde **instr** pode ser uma operação aritmética ou uma atribuição de apontadores e **expr** tem o seguinte formato geral,

```
variável = limite_inferior : limite_superior : salto
```

com salto opcional e especifica um conjunto de índices.

A declaração FORALL é executada da seguinte forma: primeiro, o lado direito da expressão é executada para todos os índices, em que essas execuções podem ser efectuadas em qualquer ordem. Segundo, as atribuições são executadas, também, em qualquer ordem. Para assegurar determinismo, a declaração FORALL não pode atribuir o mesmo elemento mais do que uma vez. Os exemplos,

```
FORALL (i = 1:m, j=1,n)
X(i,j) = i+j
END FORALL

FORALL (i = 1:m, j=1,n)
Y(i,j) = 0.0
END FORALL
```

mostram a aplicação do ciclo FORALL.

Nome da Directiva	Descrição
PROCESSORS	Define o número e organização dos processadores
DISTRIBUTE	Realiza a distribuição de dados
ALIGN	Executa o alinhamento das estruturas de dados
REALIGN	Alinha novamente as estruturas de dados
REDISTRIBRUTE	Redistribui os dados
INDEPENDENT	Usada com a declaração DO e FORALL, permite que as várias iterações sejam obtidas de forma independente
INHERIT	Usada numa subrotina, permite que esta herde as definições realizadas noutra subrotina

Tabela 1.1: Directivas mais importantes do HPF.

```
!HPF$ PROCESSORS  nome_processador (dim1, .. , dimN)
        nome_processador              nome do array de processadores
        dim1, .. , dimN               tamanho e forma do array

!HPF$ DISTRIBUTE  modo_distribuição [ONTO nome_processador] :: lista_de_arrays
        modo_distribuição             *, BLOCK ou CYCLIC
        lista_de_arrays               arrays a serem distribuídos
        nome_processador              nome do array de processadores

!HPF$ INDEPENDENT [, clausula NEW] [, clausula REDUCTION]
      NEW (lista_nomes_variáveis)
      REDUCTION (lista_variáveis_redução)
      lista_nomes_variáveis           lista de nomes de variáveis
      lista_variáveis_redução         variáveis escalares, arrays ou estruturas
```

Figura 1.1: Sintaxe das directivas PROCESSORS, DISTRIBUTE e INDEPENDENT.

1.3 Directivas de Paralelização do HPF

As directivas HPF aparecem sob a forma de comentários do tipo,

```
!HPF$ nome_da_directiva
```

permitindo também a compilação destes programas num compilador convencional. As directivas mais comuns do HPF são apresentadas na tabela 1.1.

Passamos agora a descrever as directivas PROCESSORS, DISRTIBUTE e INDEPENDENT, cuja sintaxe (Forum, 1997) é apresentada na Fig. 1.1.

Modo de distribuição	Descrição
*	Não há divisão
BLOCK(n)	Distribuição em bloco (por defeito: $n = N/P$)
CYCLIC(n)	Distribuição cíclica (por defeito: $n = 1$)

Tabela 1.2: Modo de distribuição dos dados. N representa o número de elementos de um *array* e P o número de processadores.

A directiva PROCESSORS é usada para especificar a forma e o tamanho do *array* de processadores abstractos (virtuais). Normalmente, um processador abstracto corresponde a um processador físico. Por exemplo, as instruções,

```
!HPF$  PROCESSORS procA(16)
!HPF$  PROCESSORS ProcB(4,4)
```

declaram 16 processadores, diferindo entre si na distribuição de dados pelos processadores.

A directiva DISTRIBUTE faz a distribuição de dados na memória. Esta especifica, para cada dimensão de um *array*, um mapeamento dos índices dos *arrays* pelos vários processadores. A tabela 1.2 mostra três modos de distribuição. O argumento inteiro opcional n para BLOCK e CYCLIC especifica o número de elementos num bloco.

A Fig. 1.2 mostra a utilização da directiva DISTRIBUTE para diferentes distribuições de um *array* bidimensional de tamanho 8×8 em quatro processadores. Neste exemplo, os dados mapeados no processador 1 estão representados a cinzento. Analisando esta figura verificamos que a distribuição BLOCK divide os índices do *array* em blocos iguais e contíguos de tamanho N/P, enquanto que a distribuição CYCLIC mapeia os índices por cada P processadores. Por exemplo, no primeiro caso o processador fica com 8 elementos na primeira linha e mais oito na quinta linha.

O especificador ONTO é usado para executar a distribuição através de um *array* particular de processadores. Caso nenhum *array* de processadores seja especificado, então o compilador escolhe um. A seguir, é apresentado um exemplo com uma distribuição em bloco,

```
!HPF$ PROCESSORS proc(N)
 REAL vector(100)
!HPF$ DISTRIBUTE(BLOCK) ONTO proc :: vector
```

No entanto, para resolver outro tipo de problemas o HPF permite ter um *array* de processadores e uma distribuição multidimensional dos elementos. O exemplo,

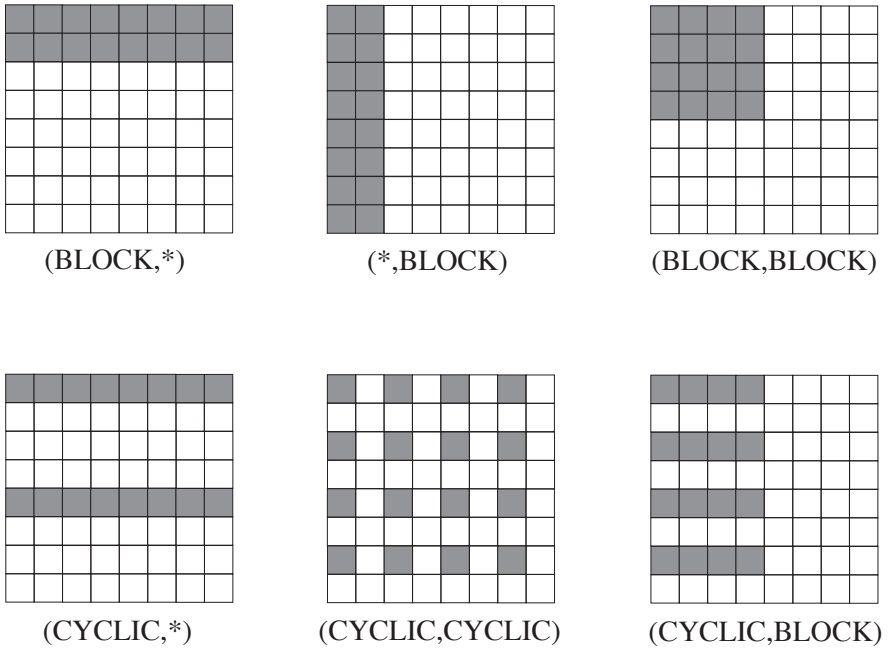

Figura 1.2: Modos de distribuição da directiva DISTRIBUTE.

```
!HPF$ PROCESSORS proc(2,2)
 REAL X(100,100)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO proc :: X
```

faz uma distribuição em duas dimensões, em que cada processador fica com um bloco de tamanho 50×50.

Um programa HPF pode revelar oportunidades adicionais para a execução em paralelo usando a directiva INDEPENDENT, de forma a assegurar que as iterações do ciclo DO poderem ser executadas independentemente, isto é, em qualquer ordem ou em concorrência sem alterar os resultados calculados. Com efeito, esta directiva altera o ciclo DO de uma declaração paralela implícita para uma declaração paralela explícita. A directiva INDEPENDENT deverá ser imediatamente precedida do ciclo DO ao qual é aplicada. Esta directiva pode ser utilizada conjuntamente com o especificador NEW. O exemplo,

```
!HPF$ INDEPENDENT
DO i = 1, n1
!HPF$ INDEPENDENT, NEW(aux1,aux2)
        DO j = 1, n2
                aux1 = B(i,j) + C(i,j)
                aux2 = B(i,j) - C(i,j)
                A(i,j) = aux1 * aux2
        ENDDO
ENDDO
```

mostra a paralelização dos dois ciclos DO com a utilização da directiva INDEPENDENT. Neste caso, a execução independente dos ciclos poderia produzir resultados errados no cálculo de **aux1** e **aux2**, bastava para isso que fossem calculados numa iteração e utilizados noutra. No entanto, o especificador NEW evita que tal aconteça. Quando se concilia a directiva INDEPENDENT com o ciclo DO, a sequência das atribuições é mantida, mas elas são distribuídas pelos vários processadores.

A clausula REDUCTION indica que a lista de variáveis de redução são alteradas num ciclo INDEPENDENT por uma série de operações que são comutativas e associativas. Os valores intermédios dessas variáveis não são usados dentro do ciclo, excepto para alterações delas próprias.

As variáveis REDUCTION fornecem um modo de acumular valores gerados num ciclo INDEPENDENT. Sem esta característica, o programador poderia guardar a informação alterada num *array* temporário cujo tamanho seria igual ao número de iterações do ciclo. O problema desta aproximação é que este *array* pode ser excessivamente grande.

Qualquer variável REDUCTION permanece protegida enquanto o ciclo DO imediatamente a seguir estiver activo. No exemplo,

```
!HPF$ INDEPENDENT, NEW(j), REDUCTION(z)
DO i = 1,20
   !HPF$ INDEPENDENT
   DO j = 1,50
      z = z + j
   ENDDO
ENDDO
```

seria incorrecto mover a clausula REDUCTION para a directiva INDEPENDENT interior. Com efeito, **z** é alterado por operações de redução (cinquenta vezes) por cada iteração do ciclo exterior, o que significa que o seu valor só estará correcto no final deste ciclo.

O HPF introduz um pequeno conjunto de funções intrínsecas, sendo as duas mais relevantes para a paralelização de um programa as funções de inquérito ao sistema, NUMBER_OF_PROCESSORS e PROCESSORS_SHAPE. Estas funções permitem fornecer ao programa informação sobre o número de processadores físicos e a topologia de ligação desses processadores. Esta informação pode ser usada para escrever programas que sejam executados num variado número de processadores e topologias. No exemplo,

```
!HPF$ PROCESSORS procA(NUMBER_OF_PROCESSORS())
integer A(SIZE(PROCESSORS_SHAPE()))
```

procA guarda um *array* de processadores com tamanho igual ao número de processadores físicos. A variável inteira **A**, usada com a função SIZE do F90, guarda um *array* de tamanho igual à dimensão do *array* de processadores físicos.

1.4 Comparação do HPF com outras técnicas de Paralelização

Nesta secção, apresentamos uma breve comparação entre dois estilos de programação paralela, a programação *data parallel* e a *message passsing*. Assim, recorre-se no primeiro caso, às ferramentas de paralelização HPF e OpenMP e no segundo caso, à técnica de paralelização por decomposição de domínio usando MPI(*Message Passing Interface*). Abordaremos ainda, as vantagens e desvantagens e quando usar cada uma das diferentes ferramentas de paralelização (PGI Group, 2000).

O HPF é uma linguagem de alto nível, que foi definida em 1993, para permitir o desenvolvimento de aplicações *data parallel* em Fortran portáteis para ambos os sistemas de computadores paralelos, de memória partilhada e de memória distribuída. A utilização de directivas que permitem executar tarefas em paralelo já é efectuada há alguns anos em sistemas como por exemplo o CRAY C90. Por outro lado, nos últimos anos tem-se verificado avanços na área dos sistemas de computação de alta *performance* de memória partilhada escalável. Como resultado, um consórcio de empresas liderado pela SGI (*Silicon Graphics Incorporated*) desenvolveu um possível *standard*, denominado OpenMP, uma linguagem de alto nível, para paralelização em sistemas de memória partilhada baseada nas implementações mais recentes para o CRAY.

O HPF apresenta suporte para sistemas SMP (*Symetric Multiple Processing*), *clusters*/MPP (*Massively Parallel Processors*) e *clusters* SMP, ao passo que o OpenMP apenas é utilizado em sistemas SMP. Em relação à *performance*, o HPF apresenta desempenho razoável em *clusters* SMP, e bom em sistemas SMP e *clusters*. O OpenMP revela desempenho muito bom em sistemas SMP (PGI Group, 2000).

Quer o HPF quer o OpenMP, não implementam qualquer tipo de suporte ao *interface input/output* paralelo. Por outro lado, apenas o HPF poderá invocar rotinas do MPI, mas também, de entre estas ferramentas de paralelização, é o único que não pode ser invocado a partir do MPI.

Quando pretendemos paralelizar uma aplicação, por vezes não é fácil escolher qual a técnica a usar. Assim, algumas considerações úteis devem ser tidas em conta. Por exemplo, devemos usar o HPF quando pretendermos ter *performances* idênticas em *clusters* e sistemas SMP e/ou quando queremos construir uma aplicação escalável a partir de um pequeno esboço. Quando existem rotinas de bibliotecas ou algoritmos em MPI, muitas vezes fazendo reutilização a partir do HPF ou quando queremos simplicidade do modelo. E ainda, quando os custos de programação e manutenção da aplicação são importantes.

Por último, o OpenMP deve ser usado quando apenas forem utilizados sistemas SMP. Quando existir uma grande necessidade de fornecer rapidamente um protótipo paralelo ou quando o custo de programação deverá ser pequeno. E ainda, quando existir uma grande aplicação anterior que pretendemos paralelizar rapidamente para um número pequeno de

processadores SMP ou quando a escalabilidade para 8-16 processadores não é essencial (PGI Group, 2000).

A paralelização dos programas de cálculo pode também ser obtida recorrendo à técnica de decomposição de domínio, como em Carvalho (1995); Areal (1999). Nesta técnica, o domínio físico é dividido em vários subdomínios, sendo cada subdomínio resolvido num processador. Os programas executados nos vários processadores são idênticos, diferindo na porção de dados sobre o qual operam e nas comunicações que efectuam. As comunicações entre os vários processadores podem ser programadas recorrendo a linguagens como o PVM e MPI.

Sendo o MPI a ferramenta de *message passing* mais usada, apresenta-se a seguir uma breve caracterização desta. O MPI é escalável, portátil e implementa uma linguagem de programação de baixo nível comparativamente ao HPF e OpenMP. Deste modo, na utilização do MPI, o programador tem maior controle sobre a distribuição e comunicação dos dados. Este maior controle sobre as instruções paralelas, devolve ao programador uma maior responsabilidade na execução das aplicações. E ainda, o tempo de aprendizagem deste método de programação é muito mais demorado.

O MPI apresenta suporte para sistemas *clusters* SMP, SMP e *clusters*/MPP, revelando desempenho razoável, bom e muito respectivamente (PGI Group, 2000). E ainda, o MPI implementa um suporte ao *interface input/output* paralelo. O MPI deve ser usado quando o custo de programação e manutenção não é decisivo, somente é importante a *performance* a obter.

Uma das desvantagens da decomposição de domínio consiste na necessidade de reprogramar o código sempre que se pretenda uma nova subdivisão do domínio, ou, de uma maneira geral, sempre que se pretenda efectuar o estudo de um novo caso físico, cujas particularidades exigem normalmente uma subdivisão diferente das já efectuadas.

Existe outro aspecto que pode ser considerado como uma limitação, que resulta da criação de condições fronteira, artificiais, nas superfícies onde os domínios físicos são divididos. Devido à dificuldade de aplicar correctamente condições fronteira no interior dos domínios de cálculo, os programas que usam esta abordagem necessitam de utilizar regiões de sobreposição entre subdomínios que podem, em alguns casos possuir dimensões consideráveis (Areal, 1999), de forma a garantir convergência.

Verifica-se também que a introdução destas condições fronteira artificiais provoca perturbações nos campos de pressão e velocidade durante o processo de convergência. Embora se tratem de fenómenos que não se manifestam nos resultados finais de um processo iterativo referente à simulação de escoamentos estacionários, podem inviabilizar a sua utilização em programas de simulação transiente como é o caso da *Large Eddy simulation* (LES) e *Direct Numeric Simulation* (DNS), que são exactamente o tipo de programas de Mecânica dos Fluidos Computacional que mais podem beneficiar das técnicas de paralelização devido ao seu elevado peso computacional.

Algumas das vantagens desta abordagem consistem num maior controlo sobre o balanceamento da carga numa rede não homogénea de computadores, que pode ser exercido através de uma escolha criteriosa da dimensão dos dados a distribuir a cada processador, assim como um maior controlo sobre as comunicações. Como resultado destas características, o desempenho de códigos paralelizados recorrendo à técnica de decomposição de domínio é elevado.

A tendência no futuro, é usar-se uma mistura das diferentes técnicas de paralelização, tirando partido do melhor que cada uma contem. Actualmente, já existem aplicações desenvolvidas usando por exemplo, OpenMP e MPI. No entanto, embora se consiga *performances* elevadas com estas aplicações, a sua programação "ainda" é muito difícil e trabalhosa, os custos de programação e manutenção são elevados e também por enquanto não são portáteis.

1.5 Meios computacionais utilizados

Com vista ao processamento paralelo, este trabalho envolveu a utilização de hardware e software específicos. Assim, efectuaram-se testes num duplo Pentium III Xeon@550 MHz e no *cluster Beowulf* existente no centro de cálculo da Faculdade de Engenharia da Universidade do Porto. Este *cluster* é constituído por 23 processadores (o *master* com velocidade de processamento de 550 MHz e os restantes 22 com 450 MHz) ligados por uma rede ethernet dedicada. Cada processador possui um placa de 100 Mbit/s. Ambos os sistemas correm o sistema operativo Linux.

É de referir que a primeira máquina *Bewolf* foi construída nos E.U.A. em 1994. Thomas Sterling e Don Becker no desenvolvimento da sua investigação no CESDIS (*Center of Excellence in Space Data and Information Sciences*) sob o projecto ESS (*Earth and Space Science*) construíram um *cluster* com 16 processadores DX4 ligados por uma rede *Ehernet* a que deram o nome de *Beowolf*.

O projecto *Beowolf* resulta essencialmente da combinação de dois factores: o aperfeiçoamento dos microprocessadores e o ganho custo/*performance* obtido pela tecnologia de rede. Uma característica importante dos *clusters Beowolf* é que as alterações no tipo e velocidade dos processadores e da tecnologia de rede não alteram o modelo de programação.

Atendendo à maturidade e robustez do Linux, do *software* GNU e da padronização da programação *message passing* via PVM e MPI, os programadores podem ter a garantia de que os seus programas irão correr em *clusters Bewolf* futuros, independentemente dos processadores e da rede.

Os *clusters Beowolf* fornecem fornecem às Universidades, muitas vezes com recursos limitados, uma excelente plataforma para ensinar computação paralela a um custo relativamente reduzido. Desta forma, este tipo de computação pode tornar-se acessível a um

maior número de pessoas.

A diferença entre um *cluster Beowolf* e uma NOW (*Network of Workstations*) é subtil, mas apresenta particularidades significativas. A principal característica que distingue os dois tipos de plataforma é que os nós de um *cluster* são dedicados para o *cluster*. Assim, alivia os problemas de carregamento balanceado, pois a *performance* dos nós individuais não estão sujeitos a factores externos. Também, desde que a rede de interligação seja isolada da rede externa, o carregamento da rede é somente determinado pelas aplicações que correm no *cluster*, permanecendo todos os nós sob jurisdição administrativa do *cluster*. Por exemplo, a rede de interligação do *cluster* não é visível do mundo exterior e por conseguinte a única autenticação necessária entre processadores é feita por razões de integridade do sistema. Outro exemplo, o *software Beowolf* fornece o identificador (ID) de todos os processos. Isto permite um mecanismo, em que o processo de um nó pode enviar sinais para outro processo de outro nó do sistema, tudo dentro do domínio do utilizador.

Actualmente, os *clusters Beowolf* são reconhecidos pela comunidade HPC (*High Performance Computing*) como uma plataforma disponível para desenvolver programação paralela.

O *software* usado foi a *release* 3.1 dos compiladores e ferramentas PGI IA-32 da Portland Group, Incorporated. Fundamentalmente, usaram-se os compiladores PGF90 e PGHPF para compilações exclusivas de Fortran 90 e Fortran 90 com directivas HPF respectivamente. Com vista a uma melhor eficiência na paralelização, recorreu-se também a um *profiler* denominado PGPROF Profiler 3.1-3.

1.6 Revisão Bibliográfica

A investigação realizada na área da Computação Paralela tem suscitado interesse e reconhecimento cada vez maior desta disciplina. Para isso, muito tem contribuído a diminuição dos preços de *hardware* e o desenvolvimento de novas linguagens e bibliotecas paralelas.

A ligação de estações de trabalho através de redes rápidas (100 Mbit/s) formando *clusters* têm permitido alargar o ensino da Computação Paralela a uma comunidade mais alargada. Este tipo de solução, permite o desenvolvimento de aplicações paralelas num modelo de memória distribuída a um custo reduzido.

A tabela 1.3 mostra o número de ocorrências por palavra chave, obtida a partir das principais bases de dados de referência. A pesquisa foi efectuada recorrendo ao *site* da biblioteca da Faculdade de Engenharia da Universidade do Porto, usando o *software Web-SPIRS* versão 4.2.

A tabela 1.4 mostra as diferentes áreas de aplicação das bases de dados de referência pesquisadas. Assim, analisando as tabelas 1.3 e 1.4 podemos ficar com uma ideia geral do

Palavra Chave	Base de Dados	Número de Ocorrências
Computational Fluid Dynamics	INSPEC	3611
	ISMEC	1975
	ICONDA	182
	ERIC/ISA	28
	GEOREF	12
High Performance Fortran	INSPEC	412
	ERIC/ISA	7
	GEOREF	2

Tabela 1.3: Pesquisa em bases de dados de referência.

Base de Dados	Áreas de aplicação
GEOREF	Geologia, Geofísica, Geoquímica, Hidrologia, Mineralogia, Petrologia, Sismografia e Estratigrafia
ICONDA	Engenharia Civil, Planeamento Urbano e Regional, Arquitectura e Construção
INSPEC	Física, Engenharia Electrotécnica, Electrónica, Telecomunicações Informática, Tecnologia de Controlo e de Informática
ISA	Ciências de Informação
ISMEC	Engenharia Mecânica, Engenharia da Produção e Gestão

Tabela 1.4: Áreas de aplicação das bases de dados de referência.

número de trabalhos e áreas de aplicação da CFD e HPF.

Desde a sua criação, o HPF tem sido utilizado como uma linguagem *data parallel* por vários investigadores. Assim, uma pesquisa no *site* acima, recorrendo à base de dados *Ei Compendex* usando neste caso o software *Dialog* resultou em 30 ocorrências. Tratavam-se na sua totalidade, de *abstracts* de dissertações de Doutoramento e Mestrado. As áreas de investigação destes trabalhos apresentavam grande diversidade, quer na Engenharia quer nas Ciências.

Quanto à Mecânica dos Fluidos Computacional Paralela (*Parallel Computational Fluid Dynamics*), diversos trabalhos têm sido desenvolvidos nos últimos anos. Com o objectivo de discutir e aprofundar a investigação nesta área, várias conferências têm sido realizadas (Ecer et al., 1996) e (Schiano et al., 1997). Assim, optamos por apenas incluir as publicações nesta área que tiveram o HPF como base no desenvolvimento dos algoritmos paralelos.

Por exemplo (Hawick et al., 1995) apresentam uma avaliação da linguagem HPF na implementação de algoritmos para aplicações CFD em computadores de alta *performance*.

A sua investigação centrou-se em métodos implícitos, como é o caso do algoritmo ADI (*Alternate Direction Implicit*), métodos de matrizes densas, como é o caso do método *panel* e métodos de matrizes esparsas, como é o caso do gradiente conjugado. A atenção destes autores incidiu sobre malhas regulares, desde que pudessem ser representadas por uma definição HPF existente.

Ainda noutra publicação, (Hawick and Fox, 1994) apresentam uma discussão da linguagem de programação HPF *data parallel* como uma ajuda à Engenharia de *Software* e como um ferramenta para a exploração de sistemas de computação de alta *performance* para aplicações CFD. Este trabalho, apresenta uma discussão acerca do uso de funções intrínsecas, directivas de distribuição de dados e instruções paralelas explicitas para optimização da *performance*, atendendo à minimização dos requisitos de comunicação e portabilidade. Os autores usaram um método implícito, como é o caso do algoritmo ADI, para ilustrar as principais características da linguagem. Analisaram malhas regulares, desde que pudessem ser eficientemente representadas por uma definição HPF existente, e malhas irregulares que iriam influenciar a definição revista do HPF-2.

Outros investigadores (Bogucz et al., 1994), avaliam a linguagem HPF como candidata à implementação de *software* CFD em sistemas de computadores de arquitectura paralela. Nesta publicação, os autores revêm as características principais da linguagem HPF e discutem os fluxos de algoritmos comuns, gerais à classe de códigos de CFD. As estruturas do HPF são apresentadas como forma conveniente de implementar uma variedade de algoritmos CFD usados, incluindo os *solvers* de diferenças finitas e volume finito que usam malhas regulares. Os investigadores concluem que, outros algoritmos CFD, incluindo aproximações multi-malha, multi-bloco e malhas não estruturadas, são mais convenientemente expressos usando extensões à linguagem HPF inicial.

Para o algoritmo SIMPLE, algoritmo CFD estudado, várias têm sido as estratégias de paralelização implementadas. Assim, Lewis and Brent (1993) efectuam uma paralelização funcional, distribuindo cada uma das equações de transporte a um processador diferente. Portanto, esta abordagem não é escalável, pois o número de processadores é igual ao número de equações de transporte.

A paralelização do algoritmo SIMPLE desenvolvida por Carvalho (1995) e Areal (1999), recorrem à técnica de decomposição de domínio e ao paradigma de *message passing* implementado em PVM.

O presente trabalho distingue-se por utilizar uma abordagem diferente de paralelização. Usou-se o paradigma de programação *data parallel*, recorrendo ao HPF. Fundamentalmente, este método de programação assenta na distribuição por vários processadores de *arrays* de valores.

Capítulo 2

Tarefas de Paralelização

2.1 Descrição do programa de Mecânica dos Fluidos Computacional

O programa de cálculo aqui desenvolvido tem por base a versão sequencial usada em (Areal, 1999), que implementa um método numérico em variáveis primitivas com formulação de volumes finitos para malhas estruturadas, não-desfasadas (Rhie and Chow, 1983; Raithby et al., 1988) e coordenadas Cartesianas. O acoplamento pressão–velocidade é resolvido com o algoritmo SIMPLE de Patankar and Spalding (1972).

Este programa calcula os campo de velocidade e pressão para escoamentos estacionários sem transferência de calor, bidimensionais, laminares e incompressíveis, cujas equações fundamentais consistem na equação de continuidade e equações de transporte de quantidade de movimento. Para um sistema de coordenadas Cartesiano essas equações podem ser escritas na forma,

$$\frac{\partial(\rho U_i)}{\partial x^i} = 0, \tag{2.1}$$

$$\rho\frac{\partial\left(U_j U_i\right)}{\partial x^j} = -\frac{\partial P}{\partial x^i} + \frac{\partial}{\partial x^j}\left[\mu\left(\frac{\partial U_i}{\partial x^j} + \frac{\partial U_j}{\partial x^i}\right)\right], \tag{2.2}$$

em que U_i representa as componentes do vector velocidade num sistema de coordenadas Cartesiano, x^i representa as direcções do sistema de coordenadas Cartesiano, μ é a viscosidade molecular, ρ é a densidade e P a pressão. A convenção de Einstein é aplicada a índices repetidos.

Neste programa, a discretização das equações fundamentais de quantidade de movimento e continuidade produz equações algébricas da forma,

$$A_P\Phi_P = A_N\Phi_N + A_S\Phi_S + A_E\Phi_E + A_W\Phi_W + S_\Phi, \tag{2.3}$$

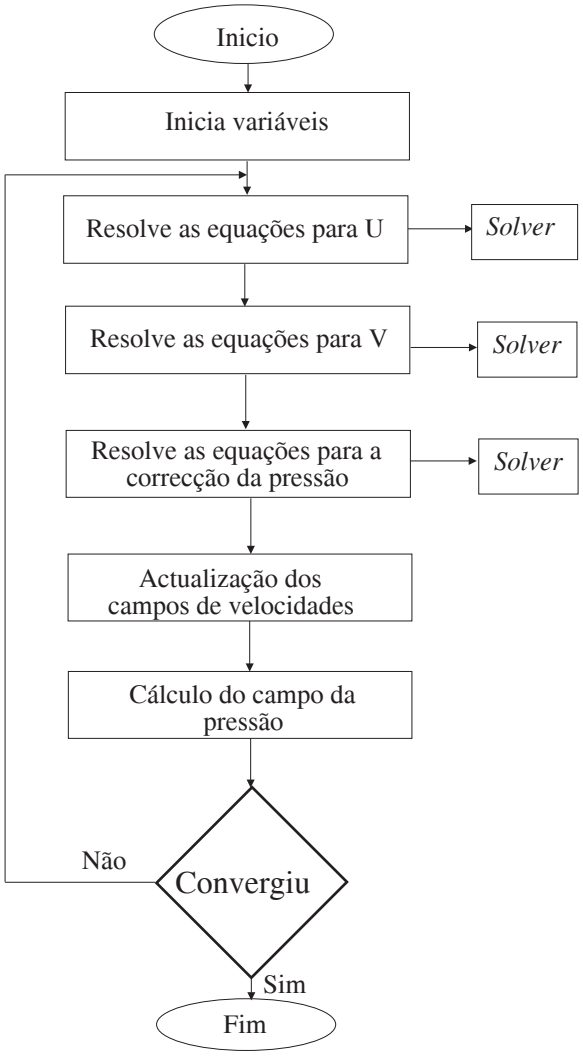

Figura 2.1: Algoritmo SIMPLE.

onde os coeficientes A armazenam as contribuições devidas aos volumes de controlo vizinhos ($North$, $South$, $East$ e $West$), sendo S_Φ um termo fonte. Por cada volume de controlo são produzidas três equações algébricas, duas para as componentes do campo de velocidade ($\Phi = U$ e $\Phi = V$) e uma equação para o campo de pressão ($\Phi = P$). Os termos convectivos das equações de quantidade de movimento são discretizados usando o esquema híbrido (Patankar, 1980).

O uso de malhas não-desfasadas juntamente com a formulação em variáveis primitivas para escoamentos incompressíveis dá origem a oscilações irrealistas no campo de pressão (por exemplo por Patankar (1980); Miller and Schmidt (1988)), quando nos modelos usados para obter as velocidades nas faces dos volumes de controlo não intervém o campo de pressão. Para ultrapassar este problema, o programa utilizado implementa o método PWIM (*Pressure-Weighted Interpolation Method*), com a introdução dos coeficientes de relaxação

Nome da subrotina	Descrição
MAIN	Programa principal que invoca várias subrotinas de cálculo e apresenta os resultados
INIVAR	Inicia várias estruturas de dados (*arrays*)
GRID	Geração da malha e parâmetros associados
CALCU	Cálculo da componente x do vector velocidade
CALCV	Cálculo da componente y do vector velocidade
CALCP	Cálculo da correcção de pressão
LSOLVR	Resolve sistemas de equações algébricas lineares (*solver*)

Tabela 2.1: Subrotinas do programa SIMPLE.

de acordo com as sugestões de Miller and Schmidt (1988), que interpola equações algébricas da quantidade de movimento para nós vizinhos da face onde se pretende obter o valor da velocidade, fazendo assim intervir o gradiente de pressão na obtenção das velocidades nas faces.

As equações algébricas de quantidade de movimento são resolvidas sequencialmente em ordem às componentes do campo de velocidade. No final deste passo obtém-se um campo de velocidade que não satisfaz conservação de massa. Após a resolução da equação algébrica de continuidade, previamente transformada numa equação algébrica para uma correcção de pressão, o campo de pressão e o campo de velocidade são corrigidos à custa da correcção de pressão. Estes passos repetem-se até se obter convergência em todas as equações algébricas. Os sistemas de equações algébricas lineares são resolvidos pelo método TDMA (*Tri-Diagonal Matrix Algorithm*) ao longo das duas direcções computacionais. Este algoritmo é apresentado na Fig. 2.1.

A primeira tarefa desenvolvida neste trabalho foi a divisão em subrotinas do programa de cálculo, baseado no algoritmo acima representado. A tabela 2.1 identifica as subrotinas que constituem o programa de cálculo.

Efectuada a separação do programa em subrotinas, procedeu-se à conversão do código de F77 para F90. Basicamente, esta conversão envolveu a transformação do formato fixo do Fortran 77 para o formato livre do Fortran 90 e a modificação das declarações das variáveis.

As secções seguintes descrevem a paralelização do programa de cálculo.

2.2 Estratégia de paralelização do programa de cálculo

A paralelização do programa de cálculo envolveu quatro fases, nomeadamente,

1. Identificação dos ciclos a paralelizar
2. Se necessário, reescrita de partes do código
3. Distribuição de variáveis pelos processadores e introdução de outras instruções HPF
4. Teste (comparação de tempos e resultados)

Na primeira fase, os ciclos foram analisados para determinar as possibilidades de paralelização.

A segunda fase representa as ocasiões em que se tornou necessário proceder a alterações no código, de forma a permitir a paralelização, eliminando o máximo possível de dependências e/ou melhorando a eficiência da paralelização. Nesta fase surgiram-nos as seguintes situações que tivemos de resolver:

- Manter os ciclos DO sem recurso a paralelização;
- Manter os ciclos DO, acrescentando-lhes a directiva INDEPENDENT;
- Substituir os ciclos DO por ciclos FORALL.

A terceira fase representa, por um lado, a identificação das variáveis (*arrays*) que aparecem nos primeiros membros das atribuições e o modo de as distribuir, e por outro, a implementação da melhor combinação de directivas HPF para a paralelização do programa. A melhor escolha era encontrada após os testes realizados na quarta fase.

Na quarta fase, a fase de testes, comparam-se os tempos obtidos na execução das versões paralela e sequencial do programa de cálculo, de forma a obter um indicador do desempenho da paralelização. Obviamente, são também comparados os resultados para evitar situações de erro.

As quatro fases acima descritas podem constituir um processo iterativo, que termina quando se maximiza o desempenho da porção de código paralelizada. Este procedimento deverá ter em conta dois limites para a quantificação do desempenho, nomeadamente o tempo de execução da versão sequencial do programa e o tempo que se obteria caso a eficiência da paralelização fosse unitária, em que o tempo de execução corresponderia ao tempo da versão sequencial dividida pelo número de processadores utilizados.

2.3 Estratégia de distribuição de dados

A forma de distribuição de dados mais eficiente será aquela que minimiza o número de redistribuições dos dados por iteração global do programa de cálculo.

O *solver* utilizado, TDMA, resolve sequencialmente os sistemas de equações algébricas ao longo de linhas e colunas, razão pela qual as distribuições mais eficientes são do tipo (*,BLOCK) e (BLOCK,*), respectivamente.

Assim, tendo em mente a minimização das operações de redistribuição, a escolha da distribuição de dados a usar nas rotinas de cálculo dos coeficientes ficou restrita a uma destas duas, tendo-se optado pela distribuição (*,BLOCK), que foi aquela que produziu menores tempos de cálculo. Este tipo de distribuição de dados implica a utilização de um *array* unidimensional de processadores.

2.4 Paralelização das rotinas de cálculo dos coeficientes

As directivas HPF para definição do *layout* dos processadores virtuais e distribuição de dados nesse *layout*, PROCESSORS e DISTRIBUTE respectivamente, foram especificadas no ficheiro **hpf.def** a seguir apresentado.

```
!HPF$ PROCESSORS proc(n)
!HPF$  DISTRIBUTE(*,BLOCK) ONTO proc :: ap,an,as,ae,aw,su,&
du,dv,bl,dex,dny,ue,vn,u,v,p,pp,vis
```

Com excepção da rotina *solver*, este ficheiro é incluído em todas as rotinas do programa de cálculo, usando a instrução **include** do Fortran. Este estilo de programação permitiu-nos alterações rápidas quer do número e disposição dos processadores, quer do tipo de distribuição de dados.

Em todas as rotinas de cálculo de coeficientes, usou-se uma disposição unidimensional para os n processadores abstractos, designado por **proc**. Os *arrays* **ap**, **an**, ..., etc, reais e bidimensionais, são partidos na segunda dimensão (colunas) através da especificação (*,BLOCK) da instrução DISTRIBUTE.

De acordo com a descrição efectuada na secção 2.1, as rotinas de cálculo dos coeficientes são a CALCU, CALCV e CALCP que correspondem ao cálculo da componente x e y do vector velocidade e ao cálculo da correcção de pressão respectivamente. Dada a semelhança do cálculo matemático subjacente, o código das rotinas CALCU e CALCV é semelhante. Deste modo, resulta uma forma de paralelização também semelhante, e consequentemente apenas apresentamos a descrição da paralelização das rotina CALCU e CALCP.

Passamos agora a apresentar a paralelização da rotina CALCU. O código original desta rotina apresentava dois ciclos DO encadeados para o cálculo dos coeficientes **an**, **as**, **ae**, **aw** e **bl**. A seguir, mostra-se a porção de código original referente apenas ao coeficiente **an**,

```
      DO i=2,nim1
```

```
        DO j=2,njm1

*         Get the areas for each cell face

          arean=sew(i)
        ...
*         Compute Convective Fluxes through each cell face

          cn=density*vn(i,j)*arean
        ...
*         Compute Difusive Fluxes through each cell face

          visn=cintnp(j)*vis(i,j)+cintnn(j)*vis(i,j+1)
        ...
          dn=visn*arean/dynp(j)
        ...
          an(i,j)=dn-.5*cn
          IF(an(i,j).LT.0.0) an(i,j)=0.0
        ...
        END DO
      END DO
```

Depois de analisados, verificou-se que existiam variáveis intermédias que criavam fortes dependências para o cálculo destes coeficientes. No entanto, porque era possível, procedeu-se à eliminação destas dependências e a seguir implementou-se um ciclo FORALL conjuntamente com a directiva INDEPENDENT. O código resultante para a expressão, por exemplo, do coeficiente **an**, passou a ter o seguinte aspecto,

```
! ciclo 1
!
!HPF$ INDEPENDENT
FORALL (j=2:njm1, i=2:nim1)

!         Assemble the coefficients

an(i,j)=max(-density*vn(i,j),(cintnp(j)*vis(i,j)+ &
                cintnn(j)*vis(i,j+1))/dynp(j)-.5*density*vn(i,j),0.)*sew(i)
        ...
END FORALL
```

cuja atribuição foi obtida pela substituição directa das variáveis utilizadas no código original na expressão,

```
an(i,j)=max(-cn,dn-.5*cn,0)
```

Um procedimento idêntico foi efectuado para o ciclo DO encadeado que efectua o cálculo dos coeficientes **su**, **ap**, **du** e os resíduos. No entanto, devido à impossibilidade de eliminar totalmente as dependências, houve a necessidade de implementar um ciclo FORALL para os coeficientes **su**, **ap**, **du** e um ciclo DO encadeado para o cálculo dos resíduos. Neste caso, mostra-se o código original,

```
DO i=2,nim1
        DO j=2,njm1
*         Residual Calculation
 ap(i,j)=ap(i,j)+an(i,j)+as(i,j)+ae(i,j)+aw(i,j)
          du(i,j)=du(i,j)/ap(i,j)
          resor=an(i,j)*u(i,j+1)+as(i,j)*u(i,j-1)+ae(i,j)*u(i+1,j)+
     1    aw(i,j)*u(i-1,j)-ap(i,j)*u(i,j)+su(i,j)
          resid=resid+ABS(resor)
*         Under-relaxation
          ap(i,j)=ap(i,j)/urelax
          su(i,j)=su(i,j)+(1-urelax)*ap(i,j)*u(i,j)
          du(i,j)=du(i,j)*urelax
        END DO
      END DO
```

e o código resultante da paralelização, respectivamente.

```
! ciclo 2
!
!HPF$ INDEPENDENT
  forall(j=2:njm1,i=2:nim1)
     su(i,j)=bl(i,j)+sns(j)*((cintww(i)*p(i-1,j)+cintpw(i)*p(i,j))&
     -(cintep(i)*p(i,j)+cintee(i)*p(i+1,j)))
     ap(i,j)=(an(i,j)+as(i,j)+ae(i,j)+aw(i,j))
     du(i,j)=sns(j)/(an(i,j)+as(i,j)+ae(i,j)+aw(i,j))
  end forall
!
!
! ciclo 3
!
!HPF$ INDEPENDENT, reduction(resid)
  do j=2,nim1
!HPF$ INDEPENDENT
     do i=2,nim1
```

```
        resid=resid+abs(an(i,j)*u(i,j+1)+as(i,j)*u(i,j-1)+ae(i,j)*u(i+1,j)+&
        aw(i,j)*u(i-1,j)-ap(i,j)*u(i,j)+su(i,j))
     end do
  end do
!        Under-relaxation
!
! ciclo 4
!
!HPF$ INDEPENDENT
  do j=2,njm1
!HPF$ INDEPENDENT
     do i=2,nim1
        ap(i,j)=ap(i,j)/urelax
        su(i,j)=su(i,j)+(1-urelax)*ap(i,j)*u(i,j)
        du(i,j)=du(i,j)*urelax
     end do
  end do
```

Para melhorar a *performance*, incluiu-se a directiva INDEPENDENT imediatamente antes de cada ciclo DO. Por se tratar de uma variável acumulativa, **resid** é incluída na cláusula REDUCTION.

Calculados os valores para os coeficientes, a rotina CALCU invoca o *solver* para efectuar a resolução do sistema de equações algébricas lineares. Por sua vez, o *solver* devolve as soluções do sistema através da passagem de parâmetros.

Os dois últimos ciclos DO encadeados para o cálculo das variáveis **dex** e **ue** a seguir apresentadas,

```
*     Update cell face velocities

      nim2=ni-2
      DO i=2,nim2
        DO j=2,njm1
          dex(i,j)=cintep(i)*du(i,j)+cintee(i)*du(i+1,j)
          ue(i,j)=dex(i,j)*(p(i,j)-p(i+1,j))+(1.-urelax)*ue(i,j)
     1    +cintep(i)*(an(i,j)*u(i,j+1)+as(i,j)*u(i,j-1)+ae(i,j)
     2    *u(i+1,j)+aw(i,j)*u(i-1,j)+bl(i,j))/ap(i,j)
     3    +cintee(i)*(an(i+1,j)*u(i+1,j+1)+as(i+1,j)*u(i+1,j-1)
     4    +ae(i+1,j)*u(i+2,j)+aw(i+1,j)*u(i,j)+bl(i+1,j))/ap(i+1,j)
        END DO
      END DO
```

apresentam dependências entre as expressões, pelo que não é possível implementar um ciclo FORALL, optando-se por introduzir a directiva INDEPENDENT combinada com a cláusula NEW,

```
! ciclo 5
!
!HPF$ INDEPENDENT, NEW(J)
  do j=2,njm1
!HPF$ INDEPENDENT, NEW(I)
     do i=2,nim2
        dex(i,j)=cintep(i)*du(i,j)+cintee(i)*du(i+1,j)
        ue(i,j)=(cintep(i)*du(i,j)+cintee(i)*du(i+1,j))*&
             (p(i,j)-p(i+1,j))+(1.-urelax)*ue(i,j)&
             +cintep(i)*(an(i,j)*u(i,j+1)+as(i,j)*u(i,j-1)+ae(i,j)&
             *u(i+1,j)+aw(i,j)*u(i-1,j)+bl(i,j))/ap(i,j)&
             +cintee(i)*(an(i+1,j)*u(i+1,j+1)+as(i+1,j)*u(i+1,j-1)&
             +ae(i+1,j)*u(i+2,j)+aw(i+1,j)*u(i,j)+bl(i+1,j))/ap(i+1,j)
     end do
  end do
```

que declara os índices **j** e **i** como variáveis privadas do ciclo. Assim, evita-se o fluxo de valores destas variáveis antes e depois do ciclo DO, e mais importante ainda, em cada iteração. Com esta cláusula, é assegurado um determinado comportamento para o programa que evita o cálculo de resultados errados.

Vamos agora apresentar a paralelização efectuada em CALCP. O código original desta rotina implementa duas estruturas de repetição DO encadeadas que efectuam o cálculo dos coeficientes **an**, **as**, **ae**, **aw** e iniciação das variáveis **su** e **ap**. À semelhança do que acontecia em CALCU e CALCV, o código original desta rotina,

```
 DO i=2,nim1
        DO j=2,njm1

          arean=sew(i)
          areas=sew(i)
          areae=sns(j)
          areaw=sns(j)

          cn=density*vn(i,j)*arean
          cs=density*vn(i,j-1)*areas
          ce=density*ue(i,j)*areae
          cw=density*ue(i-1,j)*areaw
```

```
          smp=cn-cs+ce-cw

          resid=resid+ABS(smp)

          an(i,j)=density*dny(i,j)*arean
          as(i,j)=density*dny(i,j-1)*areas
          ae(i,j)=density*dex(i,j)*areae
          aw(i,j)=density*dex(i-1,j)*areaw
          su(i,j)=-smp
          ap(i,j)=0.0

        END DO
      END DO
```

também revelava dependências no cálculo das várias atribuições. Para além disso, a variável **resid**, tratando-se de um acumulador, teria de ter um tratamento especial. Por este motivo, tal como em CALCU, optou-se por isolar esta variável e efectuar o seu cálculo dentro de dois ciclos DO encadeados. O ciclo DO exterior foi paralelizado com uma directiva INDEPENDENT conjugada com a cláusula REDUCTION para a variável **resid**, ao passo que o ciclo DO interior foi paralelizado apenas coma a directiva INDEPENDENT. O código resultante, é mostrado a seguir,

```
!HPF$ INDEPENDENT, reduction(resid)
  do j=2,nim1
!HPF$ INDEPENDENT
     do i=2,nim1
        resid=resid+ABS((density*vn(i,j)*sew(i))-(density*vn(i,j-1)*sew(i))+&
             (density*ue(i,j)*sns(j))-(density*ue(i-1,j)*sns(j)))
     end do
  end do
```

Em relação aos coeficientes, após a eliminação de algumas variáveis, que levaram à eliminação de dependências, implementou-se um ciclo FORALL conjuntamente com a directiva INDEPENDENT. O código paralelo produzido é apresentado a seguir,

```
!HPF$ INDEPENDENT, NEW(j,i)

  FORALL (j=2:njm1, i=2:nim1)

     an(i,j)=density*dny(i,j)*sew(i)
     as(i,j)=density*dny(i,j-1)*sew(i)
     ae(i,j)=density*dex(i,j)*sns(j)
```

```
      aw(i,j)=density*dex(i-1,j)*sns(j)
      su(i,j)=-((density*vn(i,j)*sew(i))-(density*vn(i,j-1)*sew(i))+&
           (density*ue(i,j)*sns(j))-(density*ue(i-1,j)*sns(j)))
      ap(i,j)=0.0

  END FORALL
```

Também aqui, foi usada a cláusula NEW para os índices **j** e **i** pelas mesmas razões apontadas anteriormente.

O cálculo do *array* **ap** e iniciação a zero de **pp** que originalmente eram efectuados em ciclo DO separados,

```
      DO i=2,nim1
        DO j=2,njm1
          ap(i,j)=ap(i,j)+an(i,j)+as(i,j)+ae(i,j)+aw(i,j)
        END DO
      END DO

      DO i=2,nim1
        DO j=2,njm1
          pp(i,j)=0.0
        END DO
      END DO
```

passaram também,

```
!HPF$ INDEPENDENT, NEW(i)
  FORALL (i=1:ni)
     pp(i,1)=pp(i,2)
     pp(i,nj)=pp(i,njm1)
  END FORALL
```

a ser calculados por um ciclo FORALL conjuntamente com uma directiva INDEPENDENT que inclui uma cláusula NEW, mas agora só referente à variável **i**.

Tal como em CALCU, calculados os valores para os coeficientes, a rotina CALCP invoca o *solver* para efectuar a resolução de equações algébricas lineares. Através da passagem de parâmetros, o *solver* recebe os coeficientes e devolve as soluções das equações.

Os dois ciclos DO seguintes do código original,

```
      DO i=1,ni
```

```
        pp(i,1)=pp(i,2)
        pp(i,nj)=pp(i,njm1)
      END DO

      DO j=1,nj
        pp(1,j)=pp(2,j)
        pp(ni,j)=pp(nim1,j)
      END DO
```

para o cálculo da variável **pp**, segundo **i** e **j** respectivamente, foram paralelizados recorrendo também a um ciclo FORALL conjuntamente com uma directiva INDEPENDENT. Esta directiva HPF, para o código resultante,

```
!HPF$ INDEPENDENT, NEW(i)
  FORALL (i=1:ni)
     pp(i,1)=pp(i,2)
     pp(i,nj)=pp(i,njm1)
  END FORALL

!HPF$ INDEPENDENT, NEW(j)
  FORALL (j=1:nj)
     pp(1,j)=pp(2,j)
     pp(ni,j)=pp(nim1,j)
  END FORALL
```

inclui também uma cláusula NEW para os índices do *array*. Como se vem demonstrando, sempre que possível implementou-se um ciclo FORALL, pois revelava melhor desempenho.

No caso seguinte, para os dois ciclos DO encadeados que efectuam o cálculo das variáveis **u**, **v**, **ue** e **ve** já não era possível a implementação de um ciclo FORALL, pois continham dependências que não se podiam eliminar. O código original é mostrado a seguir,

```
     DO i=2,nim1
        DO j=2,njm1

          u(i,j)=u(i,j)+du(i,j)*((cintww(i)*pp(i-1,j)+cintpw(i)*pp(i,j))
     1                     -(cintee(i)*pp(i+1,j)+cintep(i)*pp(i,j)))
          v(i,j)=v(i,j)+dv(i,j)*((cintss(j)*pp(i,j-1)+cintps(j)*pp(i,j))
     1                     -(cintnn(j)*pp(i,j+1)+cintnp(j)*pp(i,j)))

          IF (i.LT.nim1) ue(i,j)=ue(i,j)+dex(i,j)*(pp(i,j)-pp(i+1,j))
          IF (j.LT.njm1) vn(i,j)=vn(i,j)+dny(i,j)*(pp(i,j)-pp(i,j+1))
```

```
        END DO
      END DO
```

Assim, mais uma vez, optou-se por implementar uma directiva INDEPENDENT combinada com a cláusula NEW para o ciclo DO exterior e apenas a directiva sem qualquer cláusula para o ciclo DO interior. O código paralelizado passou a ter o seguinte aspecto,

```
!HPF$ INDEPENDENT, NEW(i,j)
  do j=2,njm1
!HPF$ INDEPENDENT
     do i=2,nim1
        u(i,j)=u(i,j)+du(i,j)*((cintww(i)*pp(i-1,j)+cintpw(i)*pp(i,j))&
             -(cintee(i)*pp(i+1,j)+cintep(i)*pp(i,j)))
        v(i,j)=v(i,j)+dv(i,j)*((cintss(j)*pp(i,j-1)+cintps(j)*pp(i,j))&
             -(cintnn(j)*pp(i,j+1)+cintnp(j)*pp(i,j)))

        if (i < nim1) ue(i,j)=ue(i,j)+dex(i,j)*(pp(i,j)-pp(i+1,j))
        if (j < njm1) vn(i,j)=vn(i,j)+dny(i,j)*(pp(i,j)-pp(i,j+1))

     end do
  end do
```

Por último, o cálculo da variável **p** e iniciação a zero de **pp**, que originalmente era efectuado por dois ciclos DO encadeados,

```
      DO i=1,ni
        DO j=1,nj
          p(i,j)=p(i,j)+urelax*(pp(i,j)-ppref)
          pp(i,j)=0
        END DO
      END DO
```

foi paralelização da seguinte forma,,

```
!HPF$ INDEPENDENT, NEW(j)
  do j=1,nj
!HPF$ INDEPENDENT, NEW(i)
     do i=1,ni
        p(i,j)=p(i,j)+urelax*(pp(i,j)-ppref)
        pp(i,j)=0.
     end do
  end do
```

2.5 Paralelização do *solver*

Da discretização das equações de quantidade de movimento resultam equações que dão origem a matrizes bandeadas pentadiagonais, que são resolvidas pelo método TDMA. O *solver* é constituído por duas partes que efectuam a resolução do sistema de equações algébricas ao longo das direcções computacionais i e j, tratando-se das mesmas variáveis a distribuir num e noutro caso.

Inicialmente, optou-se por incluir também no *solver* o ficheiro **hpf.def** descrito na secção anterior. Só que neste caso, a distribuição dos dados não era adequada para ambas as partes da rotina. Após uma análise detalhada, verificou-se que a rotina completa não era eficiente, pelo que, houve a necessidade de efectuar a esta um tratamento especial.

Ainda nesta fase, implementou-se um mapeamento dinâmico dos dados para paralelização do *solver*. Assim, imediatamente antes dos ciclos que efectuam a resolução do sistema de equações algébricas lineares segundo a direcção i, por imposição do HPF, definiram-se os *arrays* a distribuir, usando a directiva DYNAMIC. A seguir, imediatamente antes dos ciclos que efectuam a resolução do sistema de equações algébricas lineares segundo a direcção j, estes *arrays* foram redistribuídos recorrendo à directiva REDISTRIBUTE. Deste modo, era possível um remapeamento dos dados em tempo de execução. Pretendia-se assim, uma redistribuição dos dados de (BLOCK,*) para (*,BLOCK).

Aqui verificou-se que os tempos associados à redistribuição dos dados eram elevados, pelo que se optou por separar a rotina original em duas partes, de forma a poder testar cada uma das partes separadamente. Assim, a distribuição dos dados para cada subrotina passou a ser efectuada com a directiva DISTRIBUTE no início de cada uma delas. A primeira destas subrotinas, LSOLVRHa, efectua a resolução do sistema de equações algébricas ao longo da direcção i, com distribuição (BLOCK,*).

```
subroutine LSOLVRHa(nsweep,imin,jmin,ni,nj,ae,aw,an,as,ap,su,phi)

!HPF$ PROCESSORS solv(2)
!HPF$ DISTRIBUTE(BLOCK,*) onto solv :: a,c,phi,ap,ae,aw,an,as,su
```

A segunda, LSOLVRHb, efectua a resolução do sistema de equações algébricas ao longo da direcção j, com distribuição (*,BLOCK).

```
subroutine LSOLVRHb(nsweep,imin,jmin,ni,nj,ae,aw,an,as,ap,su,phi)

!HPF$ PROCESSORS solv(2)
!HPF$ DISTRIBUTE(*,BLOCK)  onto solv  :: a,c,phi,ap,ae,aw,an,as,su
```

Desta forma, poderíamos invocar LSOLVRHa, LSOLVRHb, ou ambas, nas várias rotinas de cálculo dos coeficientes, o que permitiu uma maior diversidade de testes a efectuar.

Aqui note-se que, as variáveis distribuídas são *arrays* bidimensionais passados como parâmetros das subrotinas. Associada a uma distribuição diferente dos dados, houve a necessidade de definir uma variável para o vector de processadores. Isto explica-se pelo facto do HPF não permitir que se utilize a directiva DISTRIBUTE sem antes ter especificado o *array* de processadores abstractos. No nosso caso, optamos por criar um novo nome para este *array*, que denominamos **solv**. No exemplo apresentado, este *array* especifica dois processadores.

O código sequencial original, que a seguir apresentamos para uma das partes do *solver* (LSOLVRHa), apresentava grande dependência para o cálculo dos vários termos dos *arrays*.

```
      DO n=1,nsweep
        a(jminm1)=0
        DO i=imin,nim1
          c(jminm1)=phi(i,jminm1)
          DO j=jmin,njm1
            a(j)=an(i,j)
            b(j)=as(i,j)
            c(j)=ae(i,j)*phi(i+1,j)+aw(i,j)*phi(i-1,j)+su(i,j)
            d(j)=ap(i,j)
            term=1/(d(j)-b(j)*a(j-1))
            a(j)=a(j)*term
            c(j)=(c(j)+b(j)*c(j-1))*term
          END DO
          DO j=njm1,jmin,-1
            phi(i,j)=a(j)*phi(i,j+1)+c(j)
          END DO
        END DO
      END DO
```

Assim, eliminaram-se as variáveis **b** e **d** porque apenas apareciam do lado direito das atribuições e reorganizaram-se as restantes. Para isso, recorreu-se a um ciclo FORALL combinado com uma directiva INDEPENDENT para inicialização das variáveis **a** e **c**. Note-se ainda, a passagem de vector a *array* destas duas variáveis, de forma a permitir a paralelização dos cálculos.

```
!HPF$ INDEPENDENT
  forall(i=1:ni)
     a(i,jminm1)=0.
     c(i,jminm1)=phi(i,jminm1)
  end forall

   do n=1,nsweep
```

```
!HPF$ INDEPENDENT, NEW(i,j,term)
     do i=imin,nim1
        do j=jmin,njm1
           c(i,j)=ae(i,j)*phi(i+1,j)+aw(i,j)*phi(i-1,j)+su(i,j)
           term=1./(ap(i,j)-as(i,j)*a(i,j-1))
           a(i,j)=an(i,j)*term
           c(i,j)=(c(i,j)+as(i,j)*c(i,j-1))*term
        enddo
        do j=njm1,jmin,-1
           phi(i,j)=a(i,j)*phi(i,j+1)+c(i,j)
        enddo
     enddo
  enddo
```

Por ser impossível reduzir completamente o número de dependências entre as variáveis, mantiveram-se os ciclos DO. Apenas foi acrescentada uma directiva INDEPENDENT conciliada com uma cláusula NEW para as variáveis **i**, **j** e **term**, esta para evitar cálculos errados.

Por ser semelhante à subrotina descrita, não incluímos neste trabalho a descrição pormenorizada da subrotina LSOLVRHb. No entanto, mostra-se a seguir a versão paralela desta rotina.

```
!HPF$ INDEPENDENT
  forall(j=1:nj)
     a(iminm1,j)=0.
     c(iminm1,j)=phi(iminm1,j)
  end forall

  do n=1,nsweep
!HPF$ INDEPENDENT, NEW(j,i,term)
     do j=jmin,njm1
        do i=imin,nim1
           c(i,j)=an(i,j)*phi(i,j+1)+as(i,j)*phi(i,j-1)+su(i,j)
           term=1./(ap(i,j)-aw(i,j)*a(i-1,j))
           a(i,j)=ae(i,j)*term
           c(i,j)=(c(i,j)+aw(i,j)*c(i-1,j))*term
        enddo
        do i=nim1,imin,-1
           phi(i,j)=a(i,j)*phi(i+1,j)+c(i,j)
        enddo
     enddo
  enddo
```

2.6 Testes

Existem vários modelos para caracterizar a *performance* dos algoritmos paralelos. No entanto, são três os modelos mais utilizados (Hwang, 1993). A lei de Amdahl que é baseada num carregamento ou tamanho de problemas fixo, a lei de Gustafson que é aplicada a problemas escaláveis, onde o tamanho do problema aumenta com o aumento do tamanho da máquina, e o modelo de Sun e Ni que é baseado em problemas escaláveis limitados pela capacidade de memória.

Os conceitos chave para a avaliação de um algoritmo paralelo são o *speed-up* e a eficiência. Assim, entenda-se por *speed-up* a razão entre o tempo de execução de um programa com um processador e o tempo de execução com n processadores. Formalmente,

$$S_n = \frac{T_1}{T_n}, \tag{2.4}$$

onde S_n é o valor do *speed-up* para n processadores, T_1 o tempo de execução (tempo de relógio de parede) para um processador e T_n o tempo de execução para n processadores.

Esta definição é muitas vezes codificada como a lei de Amdahl, que pode ser exposta da seguinte forma: se a componente sequencial de um algoritmo for $1/s$ do tempo de execução de um programa, então o máximo *speed-up* possível que pode ser obtido num computador paralelo é s (Foster, 1995). Por exemplo, se a componente sequencial é 5 por cento então o máximo *speed-up* que pode ser obtido é 20.

Esta lei tem por base o facto de todo o problema poder ser decomposto em duas componentes, uma paralelizável e outra sequencial (não paralelizável). Assim, podemos apresentar a lei de Amdahl da seguinte forma,

$$S_n = \frac{1}{s + \frac{p}{n}}, \tag{2.5}$$

onde s representa a componente sequencial e p a paralelizável. Calculando o limite da equação (2.5) quando n tende para infinito, obtemos,

$$S_{n_{max}} = \lim_{n \to \infty} \frac{1}{s + \frac{p}{n}} = \frac{1}{s}, \tag{2.6}$$

que mostra que, o *speed-up* do sistema fica limitado assimptoticamente ao inverso da componente sequencial.

O outro conceito chave, importante na avaliação de qualquer algoritmo paralelo, é a eficiência. Esta pode ser definida com sendo a razão entre o *speed-up* e o número n de processadores do sistema.

$$\epsilon = \frac{S_n}{n} \tag{2.7}$$

Normalmente, a eficiência é analisada em termos de percentagem e dá-nos uma ideia da taxa de utilização da capacidade de processamento instalada.

Para determinar o desempenho de paralelização das várias partes do programa de cálculo, realizaram-se testes às rotinas de cálculo de coeficientes e ao *solver*. Estes testes serviram para obter valores de S_n e ϵ para várias combinações de malhas de cálculo e número de processadores, independentemente do problema físico.

Os tempos de execução para o programa paralelo e sequencial foram obtidos através do comando **time** do sistema operativo Unix. Este comando faz medições do tempo como um *wall clock time*, isto é, como um relógio de parede.

Na compilação do código foram utilizadas opções de forma a optimizar os recursos do compilador. Assim, para a compilação do código na forma sequencial utilizou-se a seguinte linha de comando:

```
pgf90 -O4 -o nome_ficheiro_executável nome_ficheiro_fonte
```

e para a compilação do código na forma paralela utilizou-se:

```
pghpf -O4 -o nome_ficheiro_executável nome_ficheiro_fonte
```

Recorde-se que a compilação do código na forma sequencial, usando o compilador **pgf90** ignora as directivas de paralelização, produzindo um executável puramente sequencial.

2.6.1 Testes às rotinas de cálculo de coeficientes

O desempenho da paralelização do cálculo de coeficientes foi inferida realizando dois testes. Um dos teste consistiu em analisar o desempenho da rotina de cálculo de coeficientes da componente horizontal do vector velocidade (rotina CALCU) e um segundo teste foi realizado utilizando as três rotinas de cálculo de coeficientes (CALCU + CALCV + CALCP).

Nos dois casos as rotinas em teste não fizeram chamadas à rotina de resolução de sistemas de equações algébricas lineares, sendo assim só observado o desempenho dessas rotinas.

Testes à rotina CALCU

Esta rotina possui 5 ciclos, tendo-se obtido valores de S_n e ϵ para a execução isolada do primeiro ciclo, do primeiro e segundo ciclo, do primeiro ao terceiro ciclo, do primeiro ao quarto ciclo e para a totalidade da rotina. Estes testes foram efectuados executando 1000

n	Malha 200 × 200		400 × 400		800 ×800		1200 × 1200	
(total)	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$
2	1.33	66.5	1.16	58.2	1.39	69.6	0.91	45.5
4	2.31	57.7	1.88	47.1	2.42	60.6	2.33	58.3
8	3.79	47.4	3.02	37.7	3.92	49.0	3.92	49.0
16	5.33	33.3	4.31	27.0	5.84	36.5	6.19	38.7
20	5.04	25.2	4.72	23.6	6.76	33.8	7.06	35.3
(1 ciclo)								
2	1.53	76.3	1.40	70.1	1.66	83.0	1.27	63.6
4	2.93	73.2	2.83	70.8	3.26	81.5	2.97	74.2
8	4.66	58.3	5.57	69.7	6.29	78.7	5.85	73.2
16	7.14	44.6	8.24	51.5	11.90	74.4	11.21	70.0
20	8.41	42.0	9.39	47.0	14.44	72.2	13.63	68.2
(ciclos 1–2)								
2	1.42	70.8	1.33	66.5	1.58	79.1	0.84	42.2
4	2.66	66.6	2.54	63.6	3.12	78.0	2.68	67.1
8	4.68	58.5	5.47	68.4	6.00	75.5	5.27	65.8
16	6.92	43.2	10.05	62.8	11.44	71.5	9.98	62.3
20	8.28	41.4	9.67	48.4	13.38	66.9	12.07	60.4
(ciclos 1–3)								
2	1.24	62.1	1.15	57.6	1.40	69.8	0.77	38.6
4	2.19	54.8	2.19	54.8	2.72	68.0	2.32	58.0
8	3.86	48.3	4.68	58.5	5.37	67.1	4.58	57.3
16	5.45	34.1	8.26	51.6	9.99	62.5	8.82	55.1
20	6.26	31.3	9.60	48.0	11.75	58.7	10.20	51.0
(ciclos 1–4)								
2	1.24	62.0	1.17	58.5	1.42	70.8	0.81	40.4
4	2.27	56.8	2.23	55.7	2.77	69.2	2.35	58.9
8	4.04	50.5	4.81	60.1	5.50	68.1	4.72	59.0
16	5.85	36.6	8.60	53.7	10.00	62.5	8.99	52.2
20	6.59	33.0	8.91	44.6	12.18	60.9	10.63	53.1

Tabela 2.2: S_n e ϵ obtidos na paralelização da rotina CALCU. Resultados obtidos executando 1000 iterações.

chamadas à rotina CALCU, usando malhas de 200×200, 400×400, 800×800 e 1200×1200 e para n =2, 4, 8, 16 e 20 processadores. Os resultados são apresentados na tabela 2.2.

Analisando os resultados da tabela 2.2 referentes à paralelização de toda a rotina, dados referentes a (total), pode-se verificar que o maior valor de *speed-up*, $S_n = 7.06$, é obtido com $n = 20$ para a malha de maior dimensão, 1200×1200. É também para esta malha que ocorre o menor valor de *speed-up*, $S_n = 0.91$, para $n = 2$.

Em relação à eficiência, o maior valor obtido, $\epsilon = 69.6\%$, ocorreu para uma malha de 800×800 e $n = 2$. Por outro lado, o menor valor de eficiência, $\epsilon = 23.6\%$, foi obtido para uma malha de 400×400 e $n = 20$.

Analisando ainda os resultados da tabela 2.2 referentes à paralelização de toda a rotina, pode-se verificar que para praticamente todas as malhas os valores de S_n aumentam com o aumento do número de processadores. A excepção ocorre na malha de 200×200 na passagem de $n = 16$ para 20, onde se regista uma diminuição de S_n. Esta redução de S_n pode ser compreendida recorrendo a um modelo simplificado (Elisseev, 1998). Neste modelo, o tempo obtido na execução paralela é decomposto no tempo de cálculo por processador, denominado T_P^{comp}, e no tempo devido a comunicações, denominado T_P^{overh}, conjungando-se estes tempos com o tempo da execução paralela de acordo com a equação (2.8),

$$T_P = T_P^{comp} + T_P^{overh}. \tag{2.8}$$

O tempo dispendido por processador numa execução paralela com n processadores é,

$$T_P^{comp} = \frac{T_S}{n}, \tag{2.9}$$

resultando após manipulação, tendo em conta a definição de S_n (equação 2.5),

$$S_n = \frac{n}{1 + n\frac{T_P^{overh}}{T_S}}. \tag{2.10}$$

Através deste modelo verifica-se que as condições óptimas de paralelização, o crescimento aproximadamente linear de S_n com n, são encontradas quando,

$$n\frac{T_P^{overh}}{T_S} \ll 1. \tag{2.11}$$

Este modelo mostra que quanto mais rápida for a execução do programa sequencial maior será a dificuldade de obter S_n elevados. Esta conclusão é relevante para compreender o facto de a degradação de S_n se ter verificado na malha menos densa, tendo por trás o facto de que com a redução da dimensão dos dados a tratar os programas são executados de uma forma mais eficiente.

O resultado de S_n está também dependente de T_P^{overh}. Com o aumento de n a quantidade de informação a ser trocada por processador não aumenta, embora aumente o número de processadores que necessitam de comunicar, contribuindo isto para uma redução da taxa de transferência de informação na rede de comunicações. Compreende-se assim o facto de a degradação de S_n se ter verificado na malha menos densa e para o maior valor de n usado.

Analisando ainda a tabela 2.2, mas agora retendo a atenção na evolução da paralelização dos vários ciclos (ver secção 2.4), podemos retirar as seguintes ilações. É na paralelização do primeiro ciclo, que usa uma instrução FORALL, que é obtido o melhor desempenho. Por exemplo, é neste caso que se obtêm os valores mais elevados de *speed-up* e eficiência, que são respectivamente $S_n = 14.44$ e $\epsilon = 83.0\%$. Estes valores foram obtidos para a malha de 800×800 para $n = 20$ e $n = 2$, respectivamente.

Em relação à paralelização dos dois primeiros ciclos, dados referentes a (ciclos 1-2), podemos afirmar que, de uma modo geral, ocorreu uma ligeira degradação do desempenho, o que permite concluir que o segundo ciclo que usa igualmente uma instrução FORALL, é pior paralelizado que o primeiro. A excepção ocorreu para a malha de 200×200 com $n = 8$ e para a malha de 400×400 com $n = 16$ e $n = 20$, onde se registaram aumentos, embora moderados, de *speed-up* e eficiência.

Quanto à paralelização dos três primeiros ciclos, dados referentes a (ciclos 1-3), podemos referir que, ocorreu sem excepção, uma degradação de todos os valores de S_n e de ϵ. Por exemplo, é neste caso que se obteve o menor valor de *speed-up* de entre todos, $S_n = 0.77$, ocorrido para uma malha de 1200×1200 com $n = 2$ processadores. Para este resultado contribui o facto do terceiro ciclo conter uma cláusula REDUCTION, que requer comunicações adicionais entre processadores.

Em relação à paralelização dos quatro primeiros ciclos, dados referentes a (ciclos 1-4), podemos afirmar que ocorreu, sem excepção, uma melhoria, embora moderada, de todos os valores de *speed-up* e da eficiência. Conclui-se que o quarto ciclo tem boas características de paralelização, pelo facto de não haver necessidade dos vários processadores comunicarem dados entre si. Isto ocorre porque só são usados elementos das matrizes com índices (i, j). Contudo, continua a existir um afastamento considerável destes valores com os obtidos na paralelização do primeiro ciclo, paralelizado com a instrução FORALL.

Por último, se compararmos a paralelização efectuada para os primeiros quatro ciclos e toda a rotina, podemos inferir do desempenho da paralelização do quinto ciclo. Aí verificamos que é neste ciclo que se dá a maior quebra de eficiência, obtendo-se em alguns casos reduções de ϵ da ordem dos 50 %. Para isto contribuiu o facto de ser o ciclo com maior necessidade de comunicações, devido a ter um maior número de variáveis com índices fora dos abrangidos pelos processadores locais.

Como pode observar-se ainda, de uma maneira geral, a eficiência da paralelização diminui com o aumento do número de processadores. Esta conclusão, será reforçada nas secções seguintes e terá por base, rigorosamente, a mesma justificação.

A partir das equações (2.7) e (2.10) resulta,

$$\epsilon = \frac{1}{1 + n\frac{T_P^{overh}}{T_S}}, \tag{2.12}$$

donde podemos concluir que, com o aumento do número de processadores n dá-se uma diminuição da eficiência, a não ser que o valor de T_P^{overh} diminua também para compensar esse aumento. De facto, na realidade tal não acontece, pois com o aumento do número de processadores a tendência é o aumento do *overhead* das comunicações, resultando numa degradação da eficiência. Assim, para a mesma malha, geralmente a eficiência diminui com o aumento do número de processadores.

Podemos concluir ainda que, de um modo geral, a eficiência para o mesmo conjunto de processadores aumenta com o aumento da dimensão do problema. A explicação imediata para este facto tem a haver com o tempo de CPU (*Central Processing Unit*), isto é, para malhas grandes o tempo gasto na execução das instruções em paralelo compensa o *overhead* das comunicações, provocando um aumento da eficiência com a dimensão do problema.

Globalmente, os melhores resultados na paralelização de CALCU foram obtidos com a malha 800×800. A justificação para esta situação, prende-se com a distribuição dos dados pelos processadores. Assim, é encontrado um tamanho ideal para a dimensão do problema, para o qual o HPF existente responde de um modo mais eficiente.

Testes à rotina CALCU, CALCV e CALCP

As rotinas CALCU, CALCV e CALCP referem-se ao cálculo do vector x e y do vector velocidade e ao cálculo da correcção de pressão respectivamente. Os resultados dos testes foram obtidos para 1000 chamadas a CALCU, CALCV e CALCP, sem invocar a rotina do *solver*. Pretende-se com isto, testar o desempenho da paralelização destas rotinas.

A tabela 2.3 mostra os valores para testes efectuados, usando malhas de 200×200, 400×400, 800×800 e 1200×1200 e para n =2, 4, 8, 16 e 20 processadores. Analisando os resultados desta tabela, pode-se verificar que o maior valor de *speed-up*, $S_n = 10.72$, é obtido com $n = 20$ para a malha de 800×800. O menor valor de *speed-up*, $S_n = 0.82$, ocorreu para uma malha de 400×400 para $n = 2$.

Em relação à eficiência, o maior valor obtido, $\epsilon = 62.2\%$, ocorreu para uma malha de 800×800 para $n = 2$. O menor valor de eficiência, $\epsilon = 19.6\%$, foi obtido para uma malha de 200×200 para $n = 20$.

Pode-se verificar também que para todas as malhas os valores de S_n aumentam com o aumento do número de processadores, o que revela a existência de escalabilidade da paralelização. A Fig. 2.2 mostra graficamente os resultados desta tabela.

n	Malha							
	200 × 200		400 × 400		800 ×800		1200 × 1200	
	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$
2	0.98	48.9	0.82	40.8	1.24	62.2	0.83	41.6
4	1.73	43.3	1.57	39.2	2.43	60.8	1.67	41.8
8	2.77	34.6	3.71	46.4	4.75	59.4	3.28	40.9
16	3.67	22.9	6.06	37.9	8.82	55.1	6.34	39.6
20	3.93	19.6	6.91	34.6	10.72	53.6	7.37	36.8

Tabela 2.3: S_n e ϵ obtidos na paralelização das rotinas CALCU, CALCV e CALCP. Resultados obtidos executando 1000 iterações.

Comparando os resultados das tabelas 2.2 e 2.3, que mostram os resultados da paralelização de CALCU e CALCU, CALCV e CALCP respectivamente, verifica-se um maior desempenho para CALCU, CALCV e CALCP nos valores mais elevados de n, com excepção da malha de 200 × 200 onde o desempenho diminui. A melhoria mais significativa ocorreu na malha de 800 × 800 para $n = 20$, onde S_n e ϵ aumentaram cerca de 63 %.

Conclui-se que a malha de 800 × 800 é aquela onde a eficiência da paralelização é superior, como ocorreu em CALCU. Pode-se também concluir que a rotina CALCP possui ciclos cuja paralelização foi efectuada de forma mais eficiente que em CALCU e CALCV, devendo-se este facto a uma maior utilização de ciclos FORALL.

2.6.2 Testes à rotina do *solver*

O *solver* testado refere-se à rotina LSOLVRHb (também designado por *solver* b) e efectua a resolução do sistema de equações algébricas lineares ao longo da direcção computacional j, com distribuição (*,BLOCK). A razão por só efectuar o teste numa parte do *solver* teve como objectivo evitar operações de redistribuição de dados, de forma a permitir a análise isolada do desempenho da paralelização dos ciclos e comunicações locais.

Aqui convém distinguir as comunicações associadas à execução de um ciclo em paralelo, onde os vários processadores comunicam entre si os dados junto às fronteiras das partições dos dados (designadas aqui por comunicações locais), com as comunicações envolvidas nas operações de redistribuição, onde todos os dados armazenados nas memórias dos processadores são novamente distribuídas.

Os testes foram efectuados executando 1000 chamadas à rotina *solver* b, usando malhas de 200 × 200, 400 × 400, 800 × 800 e 1200 × 1200 e para n =2, 4, 8, 16 e 20 processadores. Analisando os resultados da tabela 2.4, pode-se verificar que o maior valor de *speed-up*, $S_n = 9.37$, é obtido com $n = 20$ para a malha de 800 × 800. O menor valor, $S_n = 1.14$, ocorreu para uma malha de 1200 × 1200 e $n = 2$.

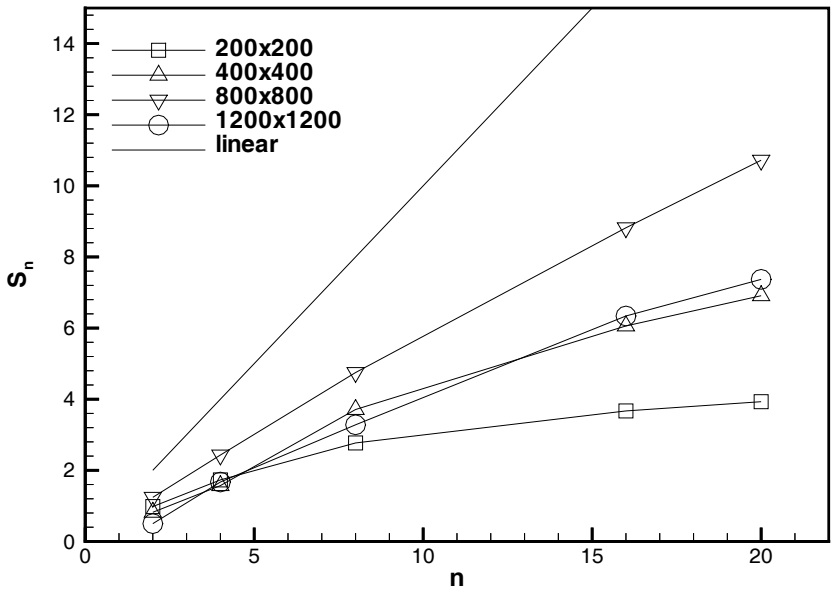

Figura 2.2: S_n obtido na paralelização das rotinas CALCU, CALCV e CALCP. Resultados obtidos executando 1000 iterações.

Em relação à eficiência, o maior valor obtido, $\epsilon = 79.7\%$, ocorreu para uma malha de 800×800 e $n = 2$. O menor valor, $\epsilon = 17.5\%$, foi obtido para uma malha de 200×200 e $n = 16$.

Na tabela 2.4 pode-se verificar que, com excepção do que ocorre na malha de 1200×1200 para $n = 20$, os valores de S_n aumentam com o aumento do número de processadores, o que revela a existência de escalabilidade da paralelização. A Fig. 2.3 mostra graficamente os resultados desta tabela.

Comparando as tabelas 2.3 e 2.4, que mostram a paralelização das rotinas de cálculo dos coeficientes e do *solver*, respectivamente, verifica-se que ocorreu uma melhoria do desempenho da paralelização para todas as malhas com a utilização de 2 e 4 processadores. Para a malha de 1200×1200 ocorreu uma melhoria para todos os conjuntos de processadores testados. Os restantes testes revelaram um menor desempenho da paralelização.

Para 2, 4 e 20 processadores, os maiores valores de *speed-up* foram obtidos para uma malha de 800×800. No entanto, para 8 e 16 processadores o melhor *speed-up* foi obtido para uma malha de 1200×1200. Isto permite concluir que, parcialmente a malha de 800×800 continua a revelar um bom desempenho de paralelização.

De uma forma geral, as melhores eficiências foram obtidas para a malha de 800×800. A excepção ocorre para uma malha de 1200×1200, usando $n = 16$, onde o valor da eficiência

n	Malha							
	200 × 200		400 × 400		800 ×800		1200 × 1200	
	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$
2	1.42	71.0	1.41	70.3	1.59	79.7	1.14	57.1
4	1.83	45.8	2.21	55.2	2.76	69.0	2.64	65.9
8	2.75	34.4	3.14	39.3	4.36	54.5	4.47	55.9
16	2.79	17.5	4.64	29.0	6.59	41.2	8.16	51.0
20	3.57	17.9	6.26	31.3	9.37	46.8	7.67	38.4

Tabela 2.4: S_n e ϵ obtidos na paralelização do *solver*. Resultados obtidos executando 1000 iterações.

é substancialmente superior a todas as outras registadas. Os valores mais altos de eficiência foram obtidos para $n = 2$, exceptuando a malha 1200 × 1200 onde ocorreu para $n = 4$. Quanto aos valores mais baixos, eles ocorreram para $n = 16$, exceptuando também o caso da malha 1200 × 1200.

2.6.3 Conclusões

Os testes efectuados às rotinas de cálculo de coeficientes e ao *solver* permitem concluir,

1. A malha de 800 × 800 é a que apresenta maior eficiência de paralelização.
2. O desempenho da paralelização aumenta com o aumento da dimensão do problema. A excepção verificou-se para a malha de 1200 × 1200, que apresentou, de uma forma geral, piores resultados que a malha de 800 × 800. As razões para o sucedido estão relacionadas com o crescimento dos *overheads* de comunicação que degradam a eficiência da paralelização.
3. De acordo com o ponto anterior, o HPF mostrou-se uma ferramenta de paralelização escalável, até ao limite de uma malha de 800 × 800 e $n = 20$. Estes limites são no entanto dependentes da arquitectura, em grande medida através da eficiência das comunicações.
4. Não existem diferenças substanciais na eficiência da paralelização obtida no cálculo de coeficientes e no *solver*. Verificou-se que o *solver* favorece a obtenção de maior *speed-up* para os valores mais reduzidos do número de processadores, acontecendo o contrário para os valores mais elevados de n.
5. De entre as rotinas de cálculo de coeficientes, a rotina CALCP foi a que proporcionou melhor resultados da paralelização. Isto deve-se à maior utilização de ciclos com a instrução FORALL.

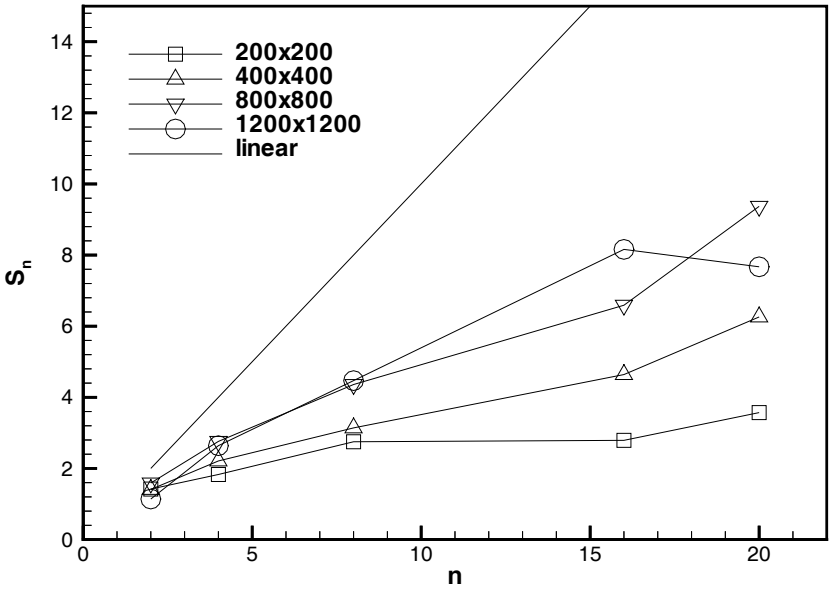

Figura 2.3: S_n obtido na paralelização do *solver*. Resultados obtidos executando 1000 iterações.

Capítulo 3

Simulação do escoamento numa caixa com tampa deslizante

3.1 Introdução

O problema escolhido para obter o desempenho da paralelização consistiu na simulação do escoamento laminar numa cavidade com tampa deslizante. A razão da opção por este escoamento prende-se com a facilidade em colocar correctamente o problema físico, uma vez que neste escoamento as condições fronteira para o campo de velocidade são conhecidas. Existem também trabalhos que podem ser usados para a comparação do desempenho da paralelização nesta geometria, como por exemplo Areal (1999).

Para este escoamento, representado de forma simplificada na Figura 3.1, foram efectuadas simulações para um número de Reynolds de 100. O número de Reynolds é baseado na dimensão da caixa (quadrada) e definido como,

$$Re = \frac{\rho U_{lid} L}{\mu} \tag{3.1}$$

A pressão nas paredes da caixa foram obtidas considerando a existência de um gradiente normal nulo, como em Areal (1999),

$$\frac{\partial P}{\partial n} = 0, \tag{3.2}$$

onde n representa a direcção normal à parede.

Durante a fase de obtenção dos resultados aqui apresentados, verificou-se que as operações de redistribuição de dados, que ocorrem entre as duas partes do *solver* utilizado (TDMA), eram demasiado demoradas, provavelmente devido à arquitectura usada (*Bewolf*), tornando a eficiência da paralelização muito reduzida.

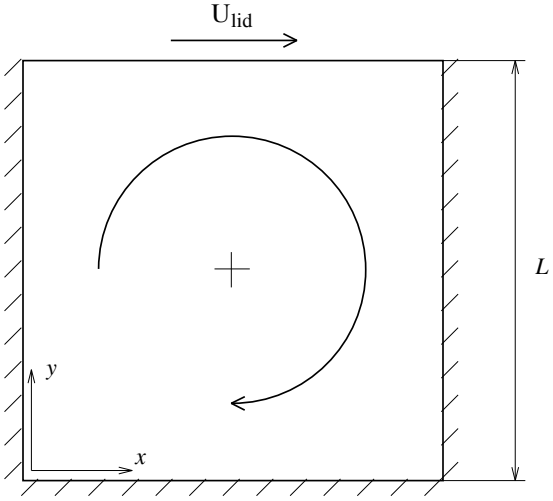

Figura 3.1: Caixa com tampa deslizante.

Tendo em mente o tempo reduzido para a realização deste trabalho, optou-se por não tentar a implementação de outros *solvers*, tendo-se decidido verificar se a utilização de uma única parte do *solver* (LSOLVRHa ou LSOLVRHb) era suficiente para produzir convergência dos resultados. Na realidade, e para o caso físico aqui testado, a utilização do LSOLVRHb além de proporcionar convergência, produziu tempos de cálculo inferiores aos obtidos com o *solver* completo, pelo que se decidiu também apresentar esses resultados.

Neste capítulo são apresentados resultados obtidos com n =2, 4, 8, 16 e 20 processadores para malhas de 80×80, 200×200, 400×400, 800×800 e 1200×1200 nós. A malha de 80×80 foi incluída para permitir uma comparação directa com os resultados obtidos em Areal (1999).

3.2 Resultados obtidos

O padrão de linhas de corrente, obtido na simulação para uma malha de 80×80 nós e $n = 2$, está representado na Figura 3.2. Estes resultados são iguais aos obtidos com outros valores de n e são também iguais aos obtidos em Areal (1999) para a mesma malha de cálculo.

Nas secções seguintes são apresentados resultados obtidos com o *solver* completo (secção 3.2.1) e com uma parte do *solver* (secção 3.2.2). O capítulo termina com a apresentação das principais conclusões deste capítulo.

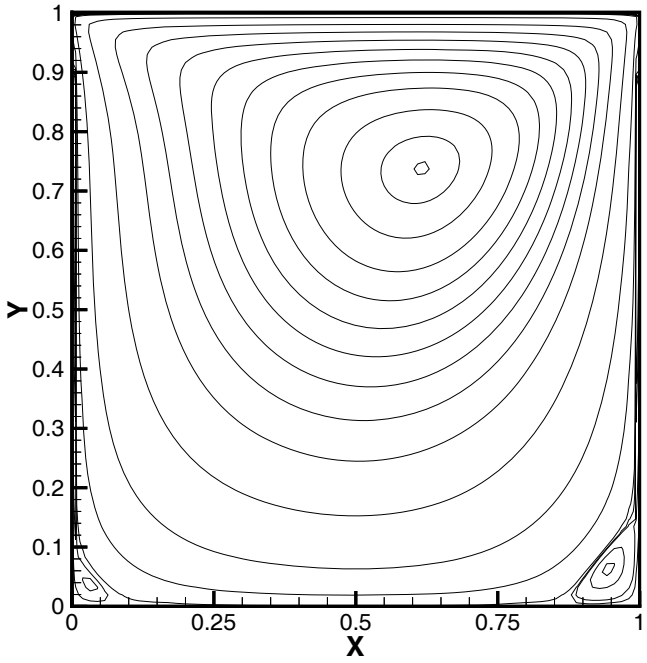

Figura 3.2: Linhas de corrente para uma malha de 80×80 nós e $Re = 100$.

3.2.1 Resultados obtidos com o *solver* completo

A tabela 3.1 mostra os resultados obtidos para uma malha de 80×80 nós. Na primeira coluna apresentam-se os resultados usando o HPF, e na segunda e terceira colunas os resultados usando a técnica de decomposição de domínio, implementada em PVM, obtidos em Areal (1999) a partir de um PC duplo Pentium II@350 MHz e uma máquina SGI Power Challenge 8000/75MHz, respectivamente.

Da análise dos resultados apresentados na tabela 3.1, onde se podem ver os valores reduzidos do *speed-up* obtidos no trabalho presente, conclui-se que para malhas destas dimensões (80×80) a utilização do HPF não é eficiente. Por este facto, a malha de menor dimensão usada na restante parte deste trabalho é de 200×200.

Analisando os resultados de S_n e ϵ obtidos para as malhas de maior densidade (tabela 3.2), podemos verificar o pobre desempenho da paralelização. O maior valor de *speed-up* foi apenas de $S_n = 3.37$, obtido com a malha de 800×800 nós para $n = 16$ processadores. Os piores resultados foram obtidos com a malha de 200×200, onde o *speed-up* máximo foi de $S_n = 1.17$ para $n = 16$.

Em termos de eficiência, o melhor resultado foi de $\epsilon = 45\%$, mesmo assim inferior a 50%, obtido para $n = 2$ na malha de 800×800. A eficiência chegou a apresentar valores

n	HPF		PVM PC		PVM SGI	
	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$
2	0.42	20.8	1.50	75.0	1.60	80.0
4	0.37	9.3	-	-	2.60	65.0

Tabela 3.1: S_n e ϵ obtidos para uma malha de 80×80 usando HPF vs técnica de decomposição de domínio.

n	Malha							
	200×200		400×400		800×800		1200×1200	
	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$
2	0.64	32.0	0.66	33.0	0.90	45.0	0.66	33.1
4	0.87	21.8	1.02	25.5	1.38	35.0	1.16	29.0
8	1.11	13.9	1.67	20.9	2.28	28.5	1.95	24.4
16	1.17	7.3	2.21	13.8	3.37	21.0	3.17	19.8
20	0.92	4.6	1.64	8.2	2.94	15.0	2.86	14.3

Tabela 3.2: S_n e ϵ obtidos com a utilização do *solver* completo. Resultados obtidos executando 100 iterações para a malha de 1200×1200 e 1000 para as restantes malhas.

de $\epsilon = 4.6\%$ na malha de 200×200 e $n = 20$.

De qualquer forma, uma das características mais desejadas nas técnicas de paralelização, que consiste no aumento da eficiência com o aumento da dimensão do problema, foi aqui obtida para malhas entre 200×200 e 800×800. Este limite na obtenção do *speed-up* depende unicamente da eficiência das comunicações, como é por exemplo mostrado em (Elisseev, 1998) através de um modelo simplificado, estando como tal intimamente relacionado com a arquitectura usada.

Face aos valores de S_n e ϵ obtidos no Capítulo 2, pode-se concluir que o *solver* utilizado, *solver* completo, não é adequado para a utilização do HPF na arquitectura usada, devido aos tempos exigidos nas operações de redistribuição entre partes do *solver*. A implementação de outros *solvers* não foi no entanto aqui efectuada devido ao tempo reduzido para a realização deste trabalho.

Prevêm-se no entanto melhores resultados na paralelização de programas que fazem simulações 3D, devido à maior porção de código por rotina de cálculo.

Observando a Figura 3.3, onde se representam graficamente os resultados da tabela 3.1, podemos concluir que para todas as malhas existe um aumento de *speed-up* até 16 processadores. Assim, para a arquitetura usada, foi atingido o ponto de *speed-up* máximo, a

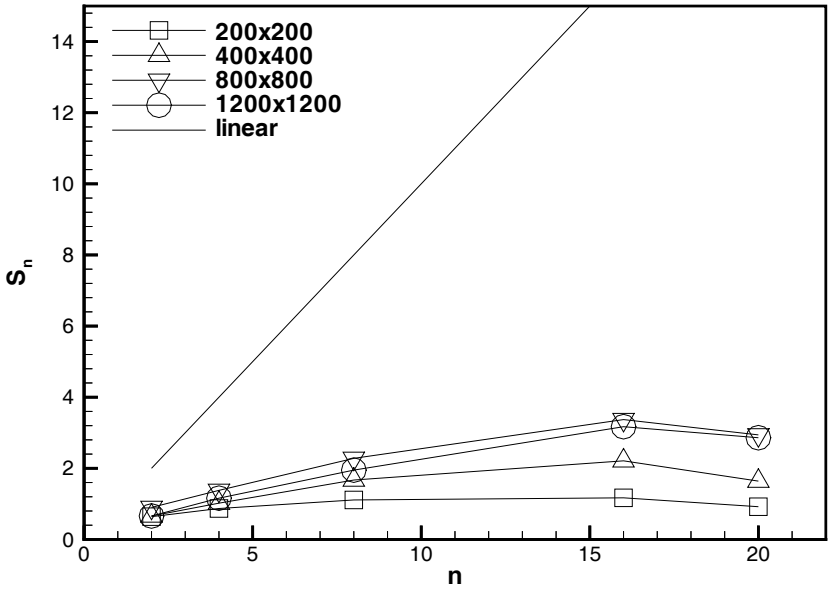

Figura 3.3: S_n obtido com a utilização do *solver* completo.

partir do qual o aumento do número de processadores não aumenta o desempenho.

Como em Elisseev (1998), também aqui ocorreu um afastamento da linha ideal com o aumento do número de processadores, o que implica uma degradação da eficiência de paralelização.

3.2.2 Resultados obtidos com uma parte do *solver*

A utilização da parte b do solver (LSOLVRHb) deu origem a resultados de melhor qualidade, como pode ser observado na tabela 3.3 e na Figura 3.4. Aqui, o maior *speed-up* foi de $S_n = 9.09$, obtido na malha de 800×800 com $n = 20$, a que corresponde uma eficiência de $\epsilon = 45\%$. Este resultado é cerca de três vezes superior ao maior *speed-up* obtido na secção anterior.

Para esta melhoria significativa contribuiu o facto de coeficientes e *solver* utilizarem a mesma distribuição (*,BLOCK), o que minimizou as operações de distribuição de dados.

Uma vez mais, verificou-se um aumento na eficiência da paralelização para malhas entre 200×200 e 800×800. Observando a Figura 3.4 conclui-se que para todas as malhas existe um aumento de *speed-up* até 20 processadores, não tendo sido atingidas as condições de *speed-up* máximo. O afastamento do comportamento ideal, comportamento linear, é menor que no caso da utilização do *solver* completo, apresentado na secção anterior.

n	Malha							
	200 × 200		400 × 400		800 ×800		1200 × 1200	
	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$	S_n	$\epsilon(\%)$
2	1.20	60.1	1.03	51.5	1.34	67.0	0.79	39.4
4	2.21	55.3	1.91	47.6	2.53	63.0	2.09	52.2
8	3.44	43.0	4.20	53.0	4.62	58.0	3.77	47.1
16	4.93	30.8	7.17	44.8	8.05	50.3	6.43	40.2
20	5.03	25.2	7.90	40.0	9.09	45.0	7.34	36.7

Tabela 3.3: S_n e ϵ obtidos com a utilização do *solver* b. Resultados obtidos executando 1000 iterações.

A grande vantagem da utilização do HPF em relação à utilização da técnica de decomposição de domínio reside nos baixos custos de implementação. Por exemplo, para alterar o código desenvolvido em Areal (1999) de forma a testar uma malha de 800×800 para 20 processadores, seriam necessários dias, ao passo que usando HPF uma alteração deste tipo demora minutos. Este maior tempo de implementação deve-se fundamentalmente à necessidade de especificação das comunicações nas partições de domínio.

3.2.3 Conclusões

Nesta secção, apresentam-se as principais conclusões da paralelização do programa de cálculo,

1. Para problemas de pequena dimensão, a paralelização usando HPF revelou-se ineficiente, tendo-se obtido *speed-ups* de 0.42 com dois processadores e uma malha de 80×80 nós. Conluido-se assim, que o HPF deverá ser usado para para problemas de grande dimensão, onde a maximização da eficiência de paralelização não seja o principal objectivo.

2. A maior deficiência do HPF está relacionada com os elevados tempos de comunicação que estão associados com as operações de distribuição e redistribuição de dados pelos processadores. A grande vantagem da utilização do HPF, por exemplo em relação à técnica de decomposição de domínio utilizando *message passing*, reside nos baixos custos de implementação.

3. Em todas as simulações efectuadas, a malha de 800×800 nós foi a que apresentou uma maior eficiência de paralelização. Com a utilização de uma parte do *solver*, *solver* b, o máximo valor do *speed-up* ($S_n = 9.09$) foi cerca de três vezes superior ao melhor resultado obtido com o *solver* completo ($S_n = 3.37$).

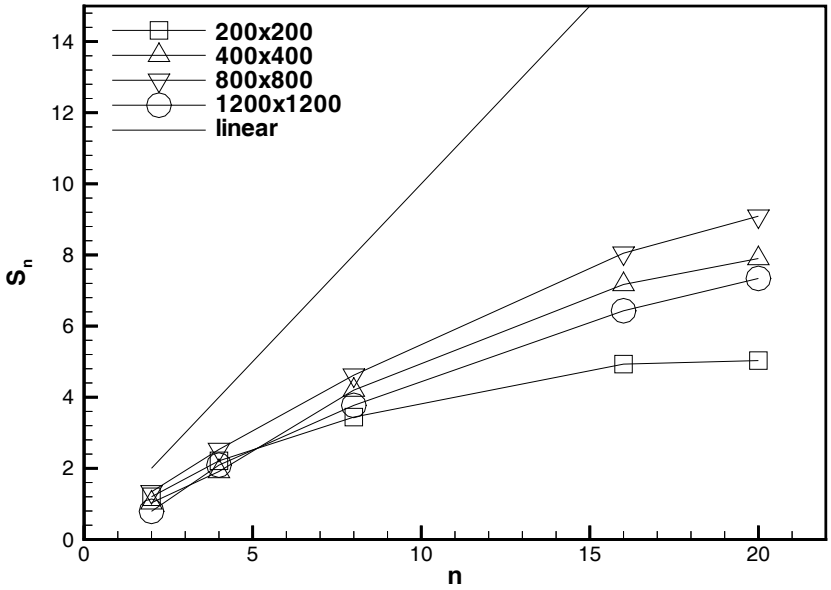

Figura 3.4: S_n obtido com a utilização do *solver* b.

4. Com a utilização do *solver* completo registou-se um aumento dos *speed-ups* até 16 processadores. Usando apenas uma parte do *solver*, registou-se um aumento contínuo dos *speed-ups* até 20 processadores.

Capítulo 4

Conclusões e Sugestões para trabalho futuro

4.1 Conclusões

Nesta secção apresentam-se as principais conclusões obtidas neste trabalho, onde se realizou a paralelização de um programa de Mecânica dos Fluidos Computacional, baseado no algoritmo SIMPLE, usando a linguagem HPF.

1. O HPF é uma linguagem de fácil utilização, devido ao facto de as comunicações necessárias à execução de um código paralelo serem implementadas pelo compilador. Esta linguagem permite assim uma rápida adaptação de códigos sequenciais para códigos paralelos, sem alteração da precisão numérica dos resultados. A alteração da densidade de paralelização (alteração do número de processadores a usar), faz-se com uma simples modificação de um parâmetro no código fonte.

2. Dos testes realizados às várias rotinas do programa de cálculo, verificou-se que foi a rotina CALCP que proporcionou melhor resultados de paralelização. Isto deve-se à maior utilização de ciclos com a instrução FORALL.

3. Comparando o HPF existente com a técnica de decomposição de domínio usando *message passing*, conclui-se que o HPF não é a ferramenta ideal de paralelização quando o principal objectivo é maximizar o desempenho. Em particular, na simulação do escoamento de uma caixa bidimensional com tampa deslizante, obtiveram-se *speed-ups* de 0.42 para corridas com dois processadores e uma malha de 80×80 nós.

4. A maior deficiência encontrada no HPF está relacionada com os elevados tempos de comunicação que estão associados com as operações de distribuição e redistribuição

de dados pelos processadores. No *solver* utilizado, TDMA, é necessário recorrer a uma operação de redistribuição de dados, o que tornou a paralelização ineficiente.

5. Essa situação foi ultrapassada com a verificação de que para o problema físico em estudo se obtinha convergência dos resultados utilizando uma única parte do TDMA, o que permitiu triplicar o melhor *speed-up* anteriormente obtido. Com esta modificação no TDMA foram registados valores de *speed-up* entre 1.34 e 9.09, para conjuntos de processadores entre 2 e 20 e uma malha de 800×800 nós. A maior eficiência obtida no trabalho presente foi de 67% para 2 processadores.

6. Para a arquitetura usada, um cluster *cluster Bewolf*, os melhores resultados foram obtidos com uma malha de 800×800 nós. Com a utilização do *solver* completo registou-se um aumento do *speed-up* até 16 processadores. Usando apenas uma parte do *solver* registou-se um aumento do *speed-up* até 20 processadores, não tendo sido atingido o ponto de *speed-up* máximo.

4.2 Sugestões para trabalho futuro

O programa paralelo implementado deverá ser testado, usando outros compiladores de HPF e/ou arquitecturas de computadores. Os *speed-up* obtidos deverão ser comparados com os obtidos nesta tese.

A paralelização usando HPF deverá ser aplicada a códigos tridimensionais, onde se esperam melhores resultados devido à maior porção de código por rotina.

Deverá efectuar-se um estudo que permita a selecção de *solvers* com melhores características para a utilização com o HPF, nomeadamente, *solvers* que não requeiram, ou minimizem, as operações intermédias de redistribuição de dados pelos processadores.

Bibliografia

P. M. Areal. Paralelização do algoritmo SIMPLE para Mecânica dos Fluidos Computacional através de subdomínios com sobreposição. Master's thesis, Faculdade de Engenharia da Universidade do Porto, 1999.

E. A. Bogucz, G. C. Fox, T. Haupt, K. A. Hawick, and S. Ranka. Preliminary evaluation of high performance fortran as a language for computational fluid dynamics. *Syracuse University*, 1994.

L. M. R. Carvalho. Paralelização de um programa de cálculo em Mecânica dos Fluidos Computacional. Master's thesis, Faculdade de Engenharia da Universidade do Porto, 1995.

A. Ecer, J. Periaux, N. Satofuka, and S. Taylor. *Parallel Computational Fluid Dynamics: Implementations and Results using Parallel Computers*. North Holland - Elsevier, 1996.

V. V. Elisseev. Parallelization of three-dimensional spectral laser-plasma interaction code using high performance fortran. *Computers in Physics*, 12:173–180, 1998.

H. P. F. Forum. *High Performance Fortran Language Specification*. Rice University, 1997.

Message Passing Interface Forum. A message-passing interface standard. *International Journal of Supercomputer Applications*, 3, 1994.

I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley Publishing Company, 1995.

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Network Parallel Computing*. Scientific and Engineering Computation. MIT Press, 1994.

K. A. Hawick, E. A. Bogucz, A. T. Degani, G. C. Fox, and G. Robinson. Cfd algorithms in high performance fortran. *NPAC Technical Report SCCS 638, Paper 95-1752, 12th AIAA CFD Conference*, 1995.

K. A. Hawick and G. C. Fox. Exploiting high performance fortran for computational fluid dynamics. *NPAC Technical Report SCCS 661*, 1994.

K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill International Editions, 1993.

A. Lewis and A. Brent. A comparison of coarse and fine grain parallelization strategies for the simple pressure correction algorithm. *International Journal of Numerical Methods in Fluids*, 16:891–914, 1993.

T. F. Miller and F. W. Schmidt. Use of a pressure-weighted interpolation method for the solution of the incompressible Navier-Stokes equations on a nonstaggered grid system. *Numerical Heat Transfer*, 14:212–233, 1988.

S. V. Patankar. *Numerical Heat Transfer and Fluid Flow.* Hemisphere Publishing Corporation, 1980.

S. V. Patankar and D. B. Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal of Heat and Mass Transfer*, 15:1787–1806, 1972.

PGI Group. *Programming Tutorial. PGI's Singapure Linux Conference Parallel.* The Portland Group, 2000.

G. D. Raithby, R. V. Elliott, and B. R. Hutchinson. Prediction of three-dimensional thermal discharge flows. *Journal of Hydraulics Engineering.*, 104:721–737, 1988.

C. M. Rhie and W. L. Chow. Numerical study of the turbulent flow past an airfoil with trailing edge separation. *AIAA Journal*, 21:1525–1532, 1983.

P. Schiano, A. Ecer, J. Periaux, and N. Satofuka. *Parallel Computational Fluid Dynamics: Implementations and Results using Parallel Computers.* North Holland - Elsevier, 1997.