DAVI TEODORO FERNANDES

Implementação de framework computacional de paralelização híbrida do
Moving Particle Semi-implicit Method para modelagem de fluidos
incompressíveis

São Paulo

2013

DAVI TEODORO FERNANDES

Implementação de framework computacional de paralelização híbrida do
Moving Particle Semi-implicit Method para modelagem de fluidos
Incompressíveis

Tese apresentada à Escola Politécnica da
Universidade de São Paulo para obtenção
do título de Doutor em Engenharia

Área de Concentração:
Engenharia Naval e Oceânica

Orientador:
Prof. Dr. Kazuo Nishimoto

São Paulo
2013

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, de julho de 2013.

Assinatura do autor _____

Assinatura do orientador _____

FICHA CATALOGRÁFICA

Fernandes, Davi Teodoro

Implementação de framework computacional de paralelização híbrida do Moving Particle Semi-implicit Method para modelagem de fluidos incompressíveis / D.T. Fernandes. -- versão corr. -- São Paulo, 2013.

212 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Naval e Oceânica.

1. Métodos numéricos em dinâmica de fluidos 2. Programação paralela I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Naval e Oceânica II. t.

Pra Senhora.¹

¹ Em memória de Da. Onófria Teodora Fernandes.

Agradecimentos

Gostaria de agradecer a todos aqueles que contribuíram de forma decisiva para a concretização deste trabalho.

Ao Prof.Dr. Kazuo Nishimoto, pela oportunidade de realizar este trabalho, pelo incentivo, pelo apoio e por me indicar os caminhos mais seguros nesta longa jornada.

Ao Prof. Dr. Cheng Liang-Yee, que coorientou este trabalho, acompanhando cada passo do seu desenvolvimento, contribuindo com críticas precisas e sugestões valiosas.

Ao Prof. Dr. José Luis Drummond Alves, pelas valiosas críticas e sugestões apresentadas durante o exame de qualificação.

A todos os amigos e colegas do TPN que muito contribuíram de forma decisiva para a realização deste trabalho.

À FUSP - Fundação de Apoio à Universidade de São Paulo, pela concessão da bolsa de doutorado e pelo apoio financeiro para a realização desta pesquisa.

Ao Tanque de Provas Numérico (TPN) da Universidade de São Paulo, por colocar à disposição o *cluster* de computadores.

Por último, manifesto um profundo reconhecimento à minha esposa Tânia e a minha filha Amanda, pelo apoio incondicional ao longo destes anos e, principalmente, por lidarem pacientemente com as constantes oscilações do meu humor, que balançavam ao ritmo da evolução dos meus esforços.

Resumo

O Tanque de Provas Numérico (TPN) é um laboratório pioneiro em hidrodinâmica aplicada e fruto de uma colaboração entre a indústria brasileira de petróleo (PETROBRAS S.A.) e as principais instituições de pesquisa do país. Seu principal objetivo é atuar como parceiro da indústria offshore e de petróleo, colaborando para a obtenção da autossuficiência da produção nacional de petróleo como uma poderosa ferramenta para projeto e análise de sistemas flutuantes de produção de óleo e gás. O coração do TPN é um *cluster* de computadores SMP que é hoje um dos maiores agrupamentos do Brasil para fins de pesquisa.

Um dos focos de atenção do TPN é a aplicação do *Moving Particles Semi-implicit Method* (MPS) na exploração de soluções para muitos problemas de Engenharia. Por trabalhar sem a necessidade do uso de malhas (método tradicional Euleriano), o método tem diversas aplicações na simulação de corpos flutuantes e na hidrodinâmica aplicada, sendo atualmente utilizado para realização de estudos sobre a influência do movimento de ondas em navios; simulações de fenômenos que envolvem fragmentações; superfícies livres; grandes deformações; dinâmica de fluidos em condições extremas, como é o caso em processos prospecção do petróleo onde muitas vezes é difícil e economicamente inviável fazer ensaios físicos.

Devido ao grande número de partículas utilizadas na simulação de sistemas complexos pelo método MPS, é necessário aproveitar de forma eficiente os recursos computacionais disponíveis para a análise de modelos com o refinamento adequado às aplicações práticas. Com tera-FLOPS disponíveis na rede *cluster* do TPN para modelagem computacional, há uma grande necessidade de uma solução computacional paralela altamente escalonável que, além disto, seja fácil de manutenção e extensibilidade.

Dentro desta linha de pesquisa, foi desenvolvida uma solução com essas características através do emprego de modernas técnicas de engenharia de software.

Abstract

The Numerical Offshore Tank (TPN) is a pioneer laboratory in applied hydrodynamics and result of collaboration between the Brazilian oil (Petrobras SA) and the major research institutions in the country. Its main purpose is to act as a partner of industry and offshore oil, contributing to the achievement of self-sufficiency of domestic oil production as a powerful tool for design and analysis of floating production systems for oil and gas. The heart of TPN is a *cluster* of SMP computers that is now one of the largest groupings of Brazil for research purposes.

One focus of attention of TPN is the application of Moving Particles Semi-implicit Method (MPS) in exploring solutions to many engineering problems. By working without the use of mesh (Eulerian traditional method), the method has several applications in the simulation of floating bodies and applied hydrodynamics, currently being used for studies on the influence of the movement of ships in waves; simulations of phenomena involving fragmentation; free surfaces, large deformations; fluid dynamics in extreme conditions, as is the case in processes where petroleum exploration is often difficult and uneconomical to do physical tests.

Due to the high number of particles used in the simulation of complex systems by the MPS method, it is necessary to efficiently take advantage of the computational resources available for the analysis of models with the refinement suitable for practical applications. With tera-FLOPS available in the TPN network *cluster* for computational modeling, there is a great need for a parallel highly scalable solution which, moreover, must be easy maintenance and extensibility.

Within this line of research, we developed a solution with these characteristics through the use of modern software engineering techniques..

Lista de Figuras

FIGURA 1-1 ‘ <i>Top 500</i> ’ SUPERCOMPUTADORES POR ARQUITETURA DE SISTEMA (1993-2012)	7
FIGURA 1-2 GRÁFICO DA FUNÇÃO PESO, EQUAÇÃO (4-3).	10
FIGURA 1-3 INTERAÇÃO ENTRE PARTÍCULAS.....	10
FIGURA 1-4 ALGORITMO DO MPS, ADAPTADO DE [11]	14
FIGURA 4-1 EXECUÇÃO DO MÉTODO ADITIVO DE SCHWARZ E DO CG EM 7 PROCESSADORES. JUSTAPOSIÇÃO DA FIGURAS 5.51 (ESQUERDA) E 5.52 (DIREITA) EXTRAÍDAS DE [65]......	36
FIGURA 4-2 SPEEDUP DAS APLICAÇÕES EM NÚMEROS DIFERENTES DE NÚCLEOS DE SCC. GRÁFICO OBTIDO POR COLORIZAÇÃO DA IMAGEM FIGURA 3 EXTRAÍDA DE [71]	38
FIGURA 5-1 TERMOS UTILIZADOS NA DESCRIÇÃO DA DECOMPOSIÇÃO DO DOMÍNIO	41
FIGURA 5-2 IDENTIFICAÇÃO DO SUBDOMÍNIO E DOS BLOCOS DA MATRIZ DE ITERAÇÃO	42
FIGURA 5-3 REPRESENTAÇÃO GRÁFICA DO ALGORITMO DE ORDENAÇÃO DE MELONI ET. AL PARA UM MODELO BIDIMENSIONAL. FIGURA EXTRAÍDA DE [74]......	44
FIGURA 5-4 REORDENAÇÃO DA MATRIZ DE INTERAÇÃO: (A) SEM REORDENAÇÃO; (B) REORDENADA PELO ALGORITMO REVERSE CUTHILL-McKEE; REORDENADA PELO ALGORITMO DE MELONI ET AL.; (D) EXEMPLO DE UMA MATRIZ DE INTERAÇÃO PARA SISTEMA CRISTALINO SEM REORDENAÇÃO. FIGURA EXTRAÍDA DE [74].	45
FIGURA 5-5 ETAPAS DA RENUMERAÇÃO DAS PARTÍCULAS	47
FIGURA 5-6 ESTRUTURA DA MATRIZ DE INTERAÇÃO COM (ESQUERDA) E SEM (DIREITA) AGRUPAMENTO DAS PARTÍCULAS POR CÉLULAS.....	48
FIGURA 5-7 SEGMENTO DA MATRIZ DE COEFICIENTES DE ITERAÇÃO REPRESENTADA COMO MATRIZ DE TRIDIAGONAL DE BLOCOS.	59
FIGURA 5-8 ESTRUTURA DE DADOS DISTRIBUÍDA NO INSTANTE $T = 0$	61
FIGURA 5-9 ESTRUTURA DE DADOS DISTRIBUÍDA NUM INSTANTE $T > 0$, APÓS SINCRONIZAÇÃO.	62
FIGURA 5-10 LISTA DE MATERIAIS DA VERSÃO SERIAL.....	63
FIGURA 5-11 LISTA DE MATERIAIS DA VERSÃO DISTRIBUÍDA	63
FIGURA 5-12 REPRESENTAÇÃO CRS DO BLOCO DIAGONAL DA MATRIZ DO SISTEMA DE EQUAÇÕES LINEARES.....	64
FIGURA 5-13 REPRESENTAÇÃO CRS DO BLOCO SUBDIAGONAL DA MATRIZ DO SISTEMA DE EQUAÇÕES LINEARES	64
FIGURA 5-14 ACESSO PARALELO AOS DADOS: N ARQUIVOS PARA N PROCESSOS.....	65
FIGURA 5-15 ACESSO PARALELO AOS DADOS: UM ARQUIVO PARA N PROCESSOS	65
FIGURA 5-16 ACESSO PARALELO AOS DADOS COM MPI/IO.	66
FIGURA 5-17 ESTRUTURA DOS ARQUIVOS DE SAÍDA PARA GRAVAÇÃO PARALELA	66
FIGURA 7-1 FLUXOGRAMA PRINCIPAL DO SIMULADOR MPS DISTRIBUÍDO.....	79
FIGURA 7-2 DIAGRAMA DAS CLASSES CSubdomain e CSimulator	82
FIGURA 7-3 FLUXOGRAMA DA CONSTRUÇÃO DO SUBDOMÍNIO	84
FIGURA 7-4 HIERARQUIA DE CLASSES DO SOLVER	93
FIGURA 7-5 FLUXOGRAMA PRINCIPAL DO SOLVER ITERATIVO	95
FIGURA 7-6 DIAGRAMA DA CLASSE GENÉRICA DE SINCRONIZAÇÃO	100
FIGURA 7-7 DIAGRAMA DA CLASSE DE SINCRONIZAÇÃO DO SOLVER	102

FIGURA 7-8 MODELO DINÂMICO DE <i>DAM BREAK</i>	104
FIGURA 7-9 DIAGRAMA DAS CLASSES PARA SINCRONIZAÇÃO DAS BORDAS DO SUBDOMÍNIOS	106
FIGURA 7-10 VALIDAÇÃO DA RENUMERAÇÃO E SINCRONIZAÇÃO, PASSO DA SIMULAÇÃO = 1	113
FIGURA 7-11 VALIDAÇÃO DA RENUMERAÇÃO E SINCRONIZAÇÃO, PASSO DA SIMULAÇÃO = 1000.....	114
FIGURA 7-12 VALIDAÇÃO DA RENUMERAÇÃO E SINCRONIZAÇÃO, PASSO DA SIMULAÇÃO = 2000.....	114
FIGURA 7-13 DIAGRAMA DAS CLASSES PARA REPRESENTAÇÃO DE VIZINHANÇAS DE PARTÍCULAS	116
FIGURA 7-14 LAYOUT DO ARQUIVO DE PARTÍCULAS PARA LEITURA EM PARALELO	127
FIGURA 7-15 TEMPO DE LEITURA PARALELIZADA DE ARQUIVO COM 22 MILHÕES DE PARTÍCULAS.....	132
FIGURA 8-1 UTILIZAÇÃO DE MEMÓRIA EM FUNÇÃO DO NÚMERO DE SUBDOMÍNIOS PARA MODELO COM 5 MILHÕES DE PARTÍCULAS.....	133
FIGURA 8-2 UTILIZAÇÃO DE MEMÓRIA EM FUNÇÃO DO NÚMERO DE SUBDOMÍNIOS PARA MODELO COM 22 MILHÕES DE PARTÍCULAS.....	134
FIGURA 8-3 COMPARAÇÃO DA INFLUÊNCIA USO DE MEMÓRIA NAS ESTRUTURAS DE CÉLULAS E VIZINHANÇAS SOBRE A TAXA DE VARIAÇÃO DO USO UTILIZAÇÃO DE MEMÓRIA TOTAL EM FUNÇÃO DO NÚMERO DE SUBDOMÍNIOS, PARA MODELO COM 22 MILHÕES DE PARTÍCULAS.	135
FIGURA 8-4 SOBREPOSIÇÃO DAS PARTÍCULAS DE SIMULAÇÕES PARALELA E SERIAL.....	136
FIGURA 8-5 POSIÇÃO DAS PARTÍCULAS, CALCULADAS EM SIMULAÇÕES SERIAL E PARALELA DO MODELO DE <i>DAM BREAK 2D</i> , SOBREPOSTAS DA PARA INSPEÇÃO VISUAL.....	138
FIGURA 8-6 EXEMPLO DE DIFERENÇAS NAS VELOCIDADES DAS PARTÍCULAS ENTRE SIMULAÇÕES SERIAL E PARALELA, COM MODELO DE <i>DAM BREAK 2D</i> , IDENTIFICADA POR INSPEÇÃO VISUAL.....	138
FIGURA 8-7 EXEMPLO DE PROBLEMA NA FRONTEIRA DE SUBDOMÍNIOS EM SIMULAÇÃO PARALELA COM QUATRO SUBDOMÍNIOS, COM MODELO DE TANQUE HIDROSTÁTICO, IDENTIFICADO POR INSPEÇÃO VISUAL.....	139
FIGURA 8-8 MODELO DINÂMICO COM 65K PARTÍCULAS, <i>DAM BREAK 2D</i>	140
FIGURA 8-9 MPS PARALELIZADO (NUM. SUBDOMÍNIO) X MPS SERIAL (MATERIAL), $T = 0,4$ s.	141
FIGURA 8-10 MPS PARALELIZADO (NUM. SUBDOMÍNIO) X MPS SERIAL (MATERIAL), $T = 0,8$ s.	141
FIGURA 8-11 MPS PARALELIZADO (NUM. SUBDOMÍNIO) X MPS SERIAL (MATERIAL), $T = 1,6$ s.	142
FIGURA 8-12 MPS PARALELIZADO (NUM. SUBDOMÍNIO) X MPS SERIAL (MATERIAL), $T = 2,8$ s.	142
FIGURA 8-13 VELOCIDADE: MPS SERIAL (SUPERIOR) X MPS PARALELIZADO (INFERIOR)	143
FIGURA 8-14 VELOCIDADE: MPS SERIAL (SUPERIOR) X MPS PARALELIZADO (INFERIOR)	144
FIGURA 8-15 <i>DAM BREAK 3D</i> PARA VALIDAÇÃO, IMAGEM EXTRAÍDA DE ISSA E VIOLEAU [26].....	145
FIGURA 8-16 ESPECIFICAÇÃO DOS SENSORES DE PRESSÃO E DE NÍVEL DO FLUIDO, IMAGEM EXTRAÍDA DE POR ISSA E VIOLEAU [26]	145
FIGURA 8-17 CONFIGURAÇÃO INICIAL DA DIVISÃO DO DOMINO NO INSTANTE INICIAL DA SIMULAÇÃO	147
FIGURA 8-18 EVOLUÇÃO DA CONFIGURAÇÃO ESPACIAL DOS SUBDOMÍNIOS, $T = 0,4$ s	148
FIGURA 8-19 EVOLUÇÃO DA CONFIGURAÇÃO ESPACIAL DOS SUBDOMÍNIOS, $T = 3,0$ s	148
FIGURA 8-20 INSTANTÂNEOS DA DISTRIBUIÇÃO DE PRESSÃO CALCULADA PELO SIMULADOR SERIAL (1.A E 1.B) E PELO SIMULADOR PARALELIZADO (2.A E 2.B).	149
FIGURA 8-21 INSTANTÂNEOS DA DISTRIBUIÇÃO DE VELOCIDADES CALCULADA PELO SIMULADOR SERIAL (1.A E 1.B) E PELO SIMULADOR PARALELIZADO (2.A E 2.B).	150

FIGURA 8-22 CORTE DO MODELO DO TANQUE DE PROVAS COM PLATAFORMA DISCRETIZADO COM 43M PARTÍCULAS (O INTERIOR DA ÁGUA FOI SUPRIMIDO DA IMAGEM).	152
FIGURA 8-23 DETALHE DA PLATAFORMA PARA O MODELO DO TANQUE DE PROVAS COM PLATAFORMA DISCRETIZADO COM 43M PARTÍCULAS.	152
FIGURA 8-24 PLATAFORMA COM 43M PARTÍCULAS: DISTRIBUIÇÃO DE VELOCIDADES 1/7.....	153
FIGURA 8-25 PLATAFORMA COM 43M PARTÍCULAS: DISTRIBUIÇÃO DE VELOCIDADES 2/7.....	154
FIGURA 8-26 PLATAFORMA COM 43M PARTÍCULAS: DISTRIBUIÇÃO DE VELOCIDADES 3/7.....	154
FIGURA 8-27 PLATAFORMA COM 43M PARTÍCULAS: DISTRIBUIÇÃO DE VELOCIDADES 4/7.....	155
FIGURA 8-28 PLATAFORMA COM 43M PARTÍCULAS: DISTRIBUIÇÃO DE VELOCIDADES 5/7.....	155
FIGURA 8-29 PLATAFORMA COM 43M PARTÍCULAS: DISTRIBUIÇÃO DE VELOCIDADES 6/7.....	156
FIGURA 8-30 PLATAFORMA COM 43M PARTÍCULAS: DISTRIBUIÇÃO DE VELOCIDADES 7/7.....	156
FIGURA 8-31 POSIÇÃO DOS SENSORES DE NÍVEL DA ÁGUA, IMAGEM EXTRAÍDA DE ISSA E VIOLEAU [26].....	157
FIGURA 8-32 EVOLUÇÃO DO NÍVEL DA ÁGUA EM H1	158
FIGURA 8-33 EVOLUÇÃO DO NÍVEL DA ÁGUA EM H2	158
FIGURA 8-34 EVOLUÇÃO DO NÍVEL DA ÁGUA EM H4	159
FIGURA 8-35 POSIÇÃO DOS SENSORES DE PRESSÃO, IMAGEM EXTRAÍDA DE ISSA E VIOLEAU [26]	159
FIGURA 8-36 EVOLUÇÃO DA PRESSÃO EM P1	160
FIGURA 8-37 EVOLUÇÃO DA PRESSÃO EM P2	160
FIGURA 8-38 EVOLUÇÃO DA PRESSÃO EM P3	161
FIGURA 8-39 EVOLUÇÃO DO TEMPO POR ITERAÇÃO EM SIMULAÇÃO LONGA PARA O MODELO COM 1M PARTÍCULAS DE RAIO 10 MM, EM SIMULAÇÕES SERIAL E DISTRIBUÍDA.	163
FIGURA 8-40 EVOLUÇÃO DO TEMPO POR ITERAÇÃO POR TAREFA NA VERSÃO SERIAL DO SIMULADOR, PARA SIMULAÇÃO DO MODELO COM 1M PARTÍCULAS DE RAIO 10 MM	164
FIGURA 8-41 EVOLUÇÃO DO TEMPO POR ITERAÇÃO NO INTERVALO DE AMOSTRAGEM PARA O MODELO COM 1M DE PARTÍCULAS DE RAIO 10 MM, EM SIMULAÇÕES COM 5 A 30 SUBDOMÍNIOS.....	165
FIGURA 8-42 EVOLUÇÃO DO TEMPO POR ITERAÇÃO NO INTERVALO DE AMOSTRAGEM PARA OS MODELOS COM 7M E 30M DE PARTÍCULAS, RAIOS DE 5 MM E 3 MM RESPECTIVAMENTE, EM SIMULAÇÕES COM 20 E 30 SUBDOMÍNIOS.	166
FIGURA 8-43 TEMPO DE EXECUÇÃO POR TAREFA COM PARALELIZAÇÃO SMP EM RELAÇÃO AO MODO SERIAL, PARA SIMULAÇÃO DO MODELO COM 1M DE PARTÍCULAS DE RAIO 10 MM.	167
FIGURA 8-44 TEMPO DE EXECUÇÃO POR TAREFA COM PARALELIZAÇÃO DISTRIBUÍDA EM RELAÇÃO AO MODO SERIAL, PARA SIMULAÇÃO DO MODELO COM 1M DE PARTÍCULAS DE RAIO 10 MM.	168
FIGURA 8-45 TEMPO DE EXECUÇÃO DO SOLVER COM PARALELIZAÇÃO DISTRIBUÍDA EM RELAÇÃO AO MODO SERIAL, PARA SIMULAÇÃO DO MODELO COM 1M DE PARTÍCULAS DE RAIO 10 MM.	169
FIGURA 8-46 TEMPO DE EXECUÇÃO POR TAREFA COM PARALELIZAÇÃO HÍBRIDA (25 SUBDOMÍNIOS) EM RELAÇÃO AO MODO SERIAL. SIMULAÇÃO DO MODELO COM 1M DE PARTÍCULAS DE RAIO 10 MM.	170
FIGURA 8-47 TEMPO DE EXECUÇÃO POR TAREFA COM PARALELIZAÇÃO HÍBRIDA EM RELAÇÃO À SIMULAÇÃO COM 15 SUBDOMÍNIO, USANDO O MODELO COM 30M DE PARTÍCULAS DE RAIO 3 MM.	171
FIGURA 8-48 SPEEDUP SIMULANDO MODELO COM 1M PARTÍCULAS DE 10 MM.	172
FIGURA 8-49 EFICIÊNCIA DA PARALELIZAÇÃO, SIMULANDO MODELO COM 1M PARTÍCULAS DE 10 MM.	173

FIGURA 8-50 SPEEDUP SIMULANDO O MODELO DE 7M PARTÍCULAS COM RAIO DE 5 MM.....	174
FIGURA 8-51 SPEEDUP SIMULANDO MODELO DE 30M PARTÍCULAS COM RAIO DE 3 MM.....	174
FIGURA 8-52 EFICIÊNCIA DA PARALELIZAÇÃO, SIMULANDO MODELOS COM 7M E 30M PARTÍCULAS.	175
FIGURA 8-53 TEMPO ESTIMADO PARA SIMULAÇÃO DE 6 SEGUNDOS DE DAM BREAK.	176
FIGURA 8-54 CONSUMO DE CPU POR TAREFA, SIMULAÇÃO COM 1M PARTÍCULAS DISTRIBUÍDAS EM 20 SUBDOMÍNIOS	178
FIGURA 8-55 EVOLUÇÃO DA DISTRIBUIÇÃO DAS PARTÍCULAS EM 8 SUBDOMÍNIOS.	179
FIGURA 8-56 USO DE CPU NA SIMULAÇÃO DO DAM BREAK 1M COM 8 SUBDOMÍNIOS	180
FIGURA 8-57 RELATÓRIO DE ANÁLISE DE CONCORRÊNCIA PELO INTEL® CONCURRENCY CHECKER	181
FIGURA G-1 ESTRUTURA DE UMA MATRIZ ESPARSA 1024x1024.....	205
FIGURA G.2 ESTRUTURA DA MATRIZ EM BANDA RESULTANTE DA REORDENAÇÃO DE LINHAS E COLUNAS DA MATRIZ DA FIGURA G.1	206
FIGURA G-3 REPRESENTAÇÃO DA MATRIZ EM BANDA DA FIGURA G.2 NA FORMA DE BLOCO TRIDIAGONAL	207
FIGURA G-4 ESQUEMA SIMPLIFICADO DE <i>CLUSTER</i> CONTENDO DOIS NÓS SMP COM ACESSO UNIFORME À MEMÓRIA	209
FIGURA G-5 ESQUEMA DE NÓ COM COM ACESSO NÃO UNIFORME À MEMÓRIA – NON UNIFORME MEMORY ACCESS (NUMA)	210

Lista de Códigos Fontes

CÓDIGO FONTE 7-1 TRECHO DO SCRIPT DE CONSTRUÇÃO DA APLICAÇÃO DISTRIBUÍDA	80
CÓDIGO FONTE 7-2 SCRIPT LINUX PARA EXECUÇÃO DA APLICAÇÃO DISTRIBUÍDA	80
CÓDIGO FONTE 7-3 LEITURA SERIAL DO ARQUIVO DE CONFIGURAÇÃO E DOS LIMITES DO DOMÍNIO (CABEÇALHO DO ARQUIVO DE PARTÍCULAS)	85
CÓDIGO FONTE 7-4 MÉTODO DE SERIALIZAÇÃO	85
CÓDIGO FONTE 7-5 CÁLCULO DA LARGURA DA BORDA	87
CÓDIGO FONTE 7-6 CÁLCULO DOS LIMITES DOS SUBDOMÍNIOS	88
CÓDIGO FONTE 7-7 ALOCAÇÃO DAS ESTRUTURAS PARA ARMAZENAMENTO DAS CONDIÇÕES INICIAIS DAS	89
CÓDIGO FONTE 7-8 INICIALIZAÇÃO DO VETOR DE RENUMERAÇÃO	90
CÓDIGO FONTE 7-9 ALOCAÇÃO FINAL DA ESTRUTURA DE PARTÍCULAS	91
CÓDIGO FONTE 7-10 CONCLUSÃO DA INICIALIZAÇÃO DO SIMULADOR DISTRIBUÍDO	92
CÓDIGO FONTE 7-11 IMPLEMENTAÇÃO DA CLASSE PARA MATRIZ ESPARSA BÁSICA	97
CÓDIGO FONTE 7-12 IMPLEMENTAÇÃO DA CLASSE PARA MATRIZ ESPARSA COMPRIMIDA POR LINHAS (CRS)	99
CÓDIGO FONTE 7-13 IMPLEMENTAÇÃO DA CLASSE GENÉRICA DE SINCRONIZAÇÃO	101
CÓDIGO FONTE 7-14 IMPLEMENTAÇÃO CLASSE PARA DA SINCRONIZAÇÃO DO SOLVER	103
CÓDIGO FONTE 7-15 DECLARAÇÃO DA ESTRUTURA DAS PARTÍCULAS DO DOMÍNIO	105
CÓDIGO FONTE 7-16 DECLARAÇÃO DA CLASSE PARA PARTÍCULAS DE BORDA	107
CÓDIGO FONTE 7-17 IMPLEMENTAÇÃO DO “EMPACOTAMENTO” DE PARTÍCULA DE BORDA	108
CÓDIGO FONTE 7-18 IMPLEMENTAÇÃO DO “DESEMPACOTAMENTO” DE PARTÍCULA DE BORDA	108
CÓDIGO FONTE 7-19 DECLARAÇÃO DA CLASSE DE SINCRONIZAÇÃO DAS PARTÍCULAS DE BORDA	109
CÓDIGO FONTE 7-20 IMPLEMENTAÇÃO CLASSE DE SINCRONIZAÇÃO DAS PARTÍCULAS DE BORDA	110
CÓDIGO FONTE 7-21 IMPLEMENTAÇÃO DO MÉTODO DE RENUMERAÇÃO DAS PARTÍCULAS	111
CÓDIGO FONTE 7-22 EXEMPLO DE CÁLCULO USANDO ACESSO DIRETO À LISTA DE VIZINHANÇAS	115
CÓDIGO FONTE 7-23 DECLARAÇÃO DA INTERFACE DA ESTRUTURA DE VIZINHANÇAS	117
CÓDIGO FONTE 7-24 DECLARAÇÃO INTERFACE DO ITERADOR DE VIZINHANÇAS	118
CÓDIGO FONTE 7-25 EXEMPLO DE CÁLCULO USANDO O ITERADOR DA LISTA DE VIZINHANÇAS	119
CÓDIGO FONTE 7-26 ALOCAÇÃO DA ESTRUTURA DE DADOS DE VIZINHANÇAS	120
CÓDIGO FONTE 7-27 ADIÇÃO DE ITENS NA ESTRUTURA DE VIZINHANÇAS	122
CÓDIGO FONTE 7-28 ITERAÇÃO DE <i>VIZINHANÇAS SMALL</i>	123
CÓDIGO FONTE 7-29 ITERAÇÃO DE <i>VIZINHANÇA LARGE</i>	124
CÓDIGO FONTE 7-30 IMPLEMENTAÇÃO DA TROCA ‘TRANSPARENTE’ DE VIZINHANÇAS	125
CÓDIGO FONTE 7-31 LEITURA PARALELIZADA DAS CONDIÇÕES INICIAIS DAS PARTÍCULAS	127
CÓDIGO FONTE 7-32 PREPARAÇÃO DOS DADOS PARA GRAVAÇÃO PARALELIZADA USANDO A BIBLIOTECA VTK	129
CÓDIGO FONTE 7-33 GRAVAÇÃO PARALELIZADA DOS DADOS DA SIMULAÇÃO USANDO A BIBLIOTECA VTK	131

Lista de Tabelas

TABELA 1 GEOMETRIA DO MODELO DE DAM BREAK 3D	146
TABELA 2 DISCRETIZAÇÕES DO MODELO DE DAM BREAK 3D	146

Lista de Abreviaturas e Siglas

2D	Bidimensional
3D	Tridimensional
API	Interface de Programação de Aplicativos (<i>Application Programming Interface</i>)
CFD	Dinâmica dos Fluidos Computacional (<i>Computational Fluid Dynamic</i>)
Cluster	Conjunto de computadores fortemente acoplados
	Unidade Central de Processamento (<i>Central Processing Unit</i>)
CPU	Originalmente usada para identificar um componente específico do computador, é usada atualmente para designar uma unidade genérica de processamento ou processador. O sentido exato é dependente do contexto.
CRS	<i>Compressed Row Storage</i>
CG	Método do Gradiente Conjugado (<i>Conjugate Gradient Method</i>)
Grid	Conjunto de computadores fracamente acoplados.
HPC	Computação de Alto Desempenho (<i>High Performance Computing</i>)
ICCG	<i>Incomplete Cholesky Conjugate Gradient Method</i>
MPI	Interface de Passagem de Mensagens (<i>Message Passing Interface</i>)

MPP	Processamento Paralelo Massivo <i>(Massively Parallel Processing)</i>
MPS	<i>Moving Particle Semi-Implicit Method</i>
$\mathcal{O}()$	Notação “big O”. Dizer que $g(n)$ é $\mathcal{O}(f(n))$ significa que, para um valor suficientemente grande de “n”, a função $g(n)$ não é maior do que uma constante multiplicada pela função $f(n)$.
OpenMP	Especificação aberta de Interface de programação de aplicativo para multiprocessamento <i>(Open Specifications for Multi Processing)</i>
PCG	Método do Gradiente Conjugado Pré-condicionado <i>(Preconditioned Conjugate Gradient)</i>
SCG	<i>Scaled Conjugate Gradient method</i>
SOR	Método de sobre relaxamento sucessivo <i>(Successive Over Relaxation Method)</i>
SMP	Multiprocessamento simétrico ou processamento paralelo em memória compartilhada. <i>(Symmetric MultiProcessing ou Shared-memory MultiProcessor)</i>
	Arquitetura de hardware consistindo de uma coleção de coleção de processadores que compartilham um banco (<i>pool</i>) de memória, com acesso aproximadamente homogêneo (ou simétrico), através de um espaço único de endereçamento. Em alguns contextos a sigla é usada para identificar processamento paralelizado em memória distribuída.

SPMD

Múltiplas cópias de um mesmo programa executando em diferentes processadores e utilizando dados diferentes.

*(Single **P**rogram **M**ultiple **D**ata)*

Lista de Símbolos

ρ	Densidade do fluído
u	Vetor velocidade de ponto do fluído
t	Instante de tempo
Δt	Incremento de tempo
P	Pressão
f	Vetor de aceleração devido às forças externas
$w(r)$	Função peso
r_e	Raio de vizinhança, ou raio efetivo: delimita a região de influência de uma partícula
r_i	Vetor posição da partícula i
m	Massa da partícula
n'	Desvio na densidade do numero de partículas
n^0	Densidade do número de partículas de referência
n^*	Densidade do número de partículas após o cálculo explícito
$\langle n \rangle_i$	Densidade do número de partículas por unidade de volume
$\langle N \rangle_i$	Número de partículas por unidade de volume
$\langle \rho \rangle_i$	Densidade do fluído por unidade de volume
u'_i	Correção da velocidade calculada explicitamente

SUMÁRIO

1	Introdução	1
1.1	DINÂMICA DOS FLUÍDOS COMPUTACIONAL NA ENGENHARIA	1
1.2	MÉTODOS NUMÉRICOS COM MALHA EM DINÂMICA DOS FLUÍDOS.....	2
1.3	MÉTODOS DE PARTÍCULAS EM DINÂMICA DOS FLUÍDOS	3
1.4	MOVING PARTICLES SEMI-IMPLICIT METHOD (MPS)	5
1.5	COMPUTAÇÃO DE ALTO DESEMPENHO (HPC)	5
1.6	FORMULAÇÃO MATEMÁTICA DO MÉTODO MPS	9
1.6.1	<i>O Algoritmo Serial do Simulador MPS</i>	14
1.7	SIMULAÇÃO DO MPS EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO	15
2	Objetivos	18
3	Organização do trabalho	19
4	Revisão bibliográfica	20
4.1	ESTRATÉGIAS COMPUTACIONAIS PARA SIMULAÇÃO POR MÉTODOS DE PARTÍCULAS	20
4.2	SOLUÇÃO NUMÉRICA DE SISTEMAS LINEARES E HPC.....	25
5	Estratégia de paralelização distribuída do MPS.....	40
5.1	DIVISÃO DINÂMICA DO DOMÍNIO COM BASE NO VETOR DE RENUMERAÇÃO DAS PARTÍCULAS	41
5.2	RENUMERAÇÃO DAS PARTÍCULAS EM MEMÓRIA DISTRIBUÍDA	44
5.3	SOLUÇÃO DO SISTEMA DE EQUAÇÕES LINEARES EM MEMÓRIA DISTRIBUÍDA.....	49
5.3.1	<i>Métodos iterativos para solução de sistemas de equações lineares</i>	49
5.3.2	<i>Seleção do método para solução do sistema de equações lineares em memória distribuída</i>	57
5.3.3	<i>O Método de Jacobi para matriz bloco tridiagonal</i>	59
5.4	DISTRIBUIÇÃO DAS ESTRUTURAS DE DADOS.....	61
5.4.1	<i>Partículas e vizinhanças</i>	61
5.4.2	<i>Lista de materiais</i>	62
5.4.3	<i>Representação das matrizes do sistema de equações lineares</i>	63
5.5	PARALELIZAÇÃO DAS OPERAÇÕES DE ENTRADA E SAÍDA	65
5.5.1	<i>Leitura paralelizada dos dados das partículas</i>	65
5.5.2	<i>Gravação dos dados da simulação</i>	66
6	Metodologia.....	67
6.1	REQUISITOS COMPUTACIONAIS DA SOLUÇÃO DE PARALELIZAÇÃO	67
6.2	PROGRAMAÇÃO PARALELA	69
6.3	O PARADIGMA DA ORIENTAÇÃO A OBJETOS	72
6.4	PROGRAMAÇÃO ORIENTADA A OBJETOS EM C++	72
6.5	O CLUSTER DO TANQUE DE PROVAS NUMÉRICO	75
6.6	FERRAMENTAS DE DESENVOLVIMENTO.....	76

7	Desenvolvimento do framework de paralelização	77
7.1	INTEGRAÇÃO DO FRAMEWORK AO SIMULADOR MPS SERIAL	80
7.2	IMPLEMENTAÇÃO DAS CLASSES SUBDOMÍNIO E SIMULADOR	82
7.2.1	<i>Leitura da configuração do modelo e dos limites do domínio.....</i>	<i>84</i>
7.2.2	<i>Cálculo de valor estimado para a largura da borda.....</i>	<i>87</i>
7.2.3	<i>Cálculo dos limites dos subdomínios.....</i>	<i>88</i>
7.2.4	<i>Alocação parcial de estrutura de dados e leitura das condições iniciais das partículas</i>	<i>89</i>
7.2.5	<i>Inicialização do vetor de renumeração</i>	<i>90</i>
7.2.6	<i>Alocação final da estrutura de partículas</i>	<i>91</i>
7.2.7	<i>Conclusão da inicialização do simulador MPS distribuído</i>	<i>91</i>
7.3	IMPLEMENTAÇÃO DO SOLVER DISTRIBUÍDO PARA MATRIZ BLOCO TRIDIAGONAL	93
7.3.1	<i>Representação das matrizes esparsas.....</i>	<i>95</i>
7.3.2	<i>Classe genérica de sincronização.....</i>	<i>100</i>
7.3.3	<i>Sincronização do solver distribuído</i>	<i>102</i>
7.3.4	<i>Validação do solver distribuído.....</i>	<i>104</i>
7.4	IMPLEMENTAÇÃO DA RENUMERAÇÃO E SINCRONIZAÇÃO DAS PARTÍCULAS EM MEMÓRIA DISTRIBUÍDA	105
7.4.1	<i>Preparação das partículas de fronteira para sincronização.....</i>	<i>107</i>
7.4.2	<i>Sincronização das partículas de borda</i>	<i>109</i>
7.4.3	<i>Reordenação das partículas em memória distribuída</i>	<i>111</i>
7.4.4	<i>Validação da renumeração e sincronização das partículas.....</i>	<i>112</i>
7.5	IMPLEMENTAÇÃO DA LISTA DE VIZINHANÇAS.....	115
7.5.1	<i>Representação de vizinhanças na versão serial</i>	<i>115</i>
7.5.2	<i>Representação de vizinhanças na versão distribuída.....</i>	<i>116</i>
7.5.3	<i>Construção de vizinhanças na versão distribuída.....</i>	<i>120</i>
7.5.4	<i>Acesso à vizinhança das partículas na versão distribuída</i>	<i>123</i>
7.6	IMPLEMENTAÇÃO DA PARALELIZAÇÃO DAS OPERAÇÕES DE ENTRADA E SAÍDA.....	126
7.6.1	<i>Paralelização da leitura dos dados do modelo.....</i>	<i>126</i>
7.6.2	<i>Paralelização da gravação dos dados da simulação</i>	<i>129</i>
7.6.3	<i>Validação.....</i>	<i>132</i>
8	Resultados	133
8.1	AVALIAÇÃO DO USO DE MEMÓRIA PELAS ESTRUTURAS DE DADOS DISTRIBUÍDAS	133
8.2	VALIDAÇÃO COM MODELO DINÂMICO 2D	136
8.2.1	<i>Modelo de dam break 2D com 65K partículas</i>	<i>140</i>
8.2.2	<i>Validação da posição das partículas.....</i>	<i>140</i>
8.2.3	<i>Validação da pressão</i>	<i>143</i>
8.2.4	<i>Validação da velocidade</i>	<i>144</i>
8.3	VALIDAÇÃO COM MODELO DINÂMICO 3D	145
8.3.1	<i>Dam break 3D com um milhão de partículas</i>	<i>147</i>
8.3.2	<i>Simulação de plataforma em tanque de provas com 43 milhões de partículas</i>	<i>152</i>

8.4	ESTUDO DE CASO COM MODELO DE DAM BREAK 3D	157
8.4.1	<i>Medidas no sensor de altura do nível da água</i>	<i>157</i>
8.4.2	<i>Medidas no sensor de pressão.....</i>	<i>159</i>
8.5	AVALIAÇÃO DE ESCALABILIDADE E DESEMPENHO	162
8.5.1	<i>Definição do intervalo de simulação para medidas de speedup.....</i>	<i>163</i>
8.5.2	<i>Avaliação do desempenho da paralelização em memória compartilhada (SMP) por tarefa</i>	<i>167</i>
8.5.3	<i>Avaliação do desempenho da paralelização em memória distribuída (DM) por tarefa.....</i>	<i>168</i>
8.5.4	<i>Avaliação do desempenho da paralelização híbrida (DM+SMP) por tarefa.....</i>	<i>169</i>
8.5.5	<i>Avaliação do speedup da paralelização híbrida (DM+SMP) em função do número de subdomínios</i>	<i>171</i>
8.5.6	<i>Comparação dos tempos de execução serial x paralelo.....</i>	<i>176</i>
8.5.7	<i>Comparação dos tempos relativos de execução por tarefa</i>	<i>178</i>
8.5.8	<i>Balanceamento de carga</i>	<i>179</i>
9	Conclusões	182
9.1.1	<i>Estratégia de decomposição.....</i>	<i>182</i>
9.1.2	<i>Metodologia orientada a objetos e C++</i>	<i>182</i>
9.1.3	<i>Resultados.....</i>	<i>183</i>
9.1.4	<i>Desenvolvimentos Futuros.....</i>	<i>186</i>
10	Bibliografia.....	187
11	Glossário	202

1 INTRODUÇÃO

Existem basicamente três formas de realizar estudos em Dinâmica dos Flúidos. A primeira é a abordagem teórica, ou analítica, que inclui a formulação de modelos matemáticos e a manipulação e solução das equações com papel e lápis. Devido à natureza complexa dos fluidos, o estudo dos fenômenos de fluxo de fluido e transferência de calor exige cálculos extremamente complexos, que, em geral, só podem ser efetivamente computados para problemas com geometrias e condições de contorno simplificadas. Embora isto restrinja a aplicabilidade direta dos métodos analíticos, eles são de grande utilidade na validação de métodos numéricos.

A segunda abordagem é a realização de experimentos em laboratório. Isto permite analisar os fenômenos em condições reais, mas é geralmente muito custoso e possibilita apenas a medição de um conjunto restrito de parâmetros do fenômeno. Além disto, muitas vezes é difícil ou impossível reproduzir as condições reais do problema, por envolver condições de risco, exceder limites normais de desempenho ou pelas dimensões do fenômeno físico.

O advento dos computadores modernos viabilizou uma terceira abordagem para o estudo destes fenômenos: a simulação computacional, através de métodos numéricos para solução dos modelos matemáticos de fluxo de fluido, denominada Dinâmica dos Flúidos Computacional (*Computational Fluid Dynamics* - CFD).

1.1 DINÂMICA DOS FLUÍDOS COMPUTACIONAL NA ENGENHARIA

Dinâmica de Fluidos Computacional (CFD) designa um grupo de técnicas para estudo do comportamento de sistemas envolvendo fluxo de fluidos, transferência de massa, transferência de calor e fenômenos associados, tais como reações químicas e fenômenos relacionados. Em CFD, são utilizados métodos numéricos para solução das equações matemáticas que regem esses processos. A técnica é muito poderosa e abrange uma ampla gama de aplicação em projetos industriais e em pesquisas científicas. Entre as aplicações em engenharia podemos destacar:

- Engenharia naval: cargas em estruturas *off-shore*, hidrodinâmica de navios.
- Hidrologia e oceanografia: fluxos em rios, estuários, oceanos.

- Aerodinâmica de aeronaves e veículos: levante e arraste.
- Turbomáquinas: passagem de fluxo por elemento rotativo, difusores etc.
- Engenharia de processo químico: mistura e separação, moldagem de polímeros.
- Engenharia civil: carga de vento e aquecimento / ventilação em ambiente externo e interno de edifícios.
- Engenharia ambiental: distribuição de poluentes e efluentes.
- Engenharia biomédica: fluxos de sangue através das artérias e veias.

A análise dos dados resultantes da aplicação de técnicas de CFD pode ser utilizada em engenharia para estudos conceituais de novos projetos, ou aperfeiçoamento de projetos existentes (redesign). Como técnica complementar aos estudos teóricos e experimentais, apresenta os seguintes benefícios:

- Redução do tempo de projeto e desenvolvimento.
- Estudo em escala real.
- Permite estudar sistemas para os quais experimentos controlados são difíceis ou até mesmo impossíveis de realizar como, por exemplo, sistemas muito grandes ou em condições perigosas ou além de seus limites normais de desempenho.
- Fornece informações com nível praticamente ilimitado de detalhes dos resultados, pois possibilita um grande controle sobre o processo físico, permite isolar fenômenos específicos para o estudo, permite examinar um grande número de locais na região de interesse e produz um conjunto completo de parâmetros de fluxo para análise.
- Complementa teste e experimentação, reduzindo o esforço total requerido no laboratório.
- As simulações são, em geral, mais baratas e os custos tendem a diminuir à com a diminuição do custo dos computadores de alto desempenho.

1.2 MÉTODOS NUMÉRICOS COM MALHA EM DINÂMICA DOS FLUÍDOS

A solução numérica de equações diferenciais parciais exige a representação da natureza contínua das equações governantes do modelo de fluxo por um sistema de

equações algébricas em forma discreta. Na formulação dos métodos numéricos atualmente mais utilizados - Método Diferenças Finitas, Método Volumes Finitos e Método dos Elementos Finitos [1] – isto é obtido pela discretização das equações do modelo através da divisão do domínio do problema em um conjunto de elementos de controle denominado malha ou grade (*grid*), e na obtenção de um sistema algébrico de equações sobre os elementos de controle (usualmente, denominados células ou nós). A malha define a precisão numérica da solução, uma vez que as variáveis são calculadas em pontos discretos definidos pela grade, influencia na taxa de convergência da solução e no custo computacional da simulação.

Estes métodos numéricos são amplamente utilizados em diversas áreas de aplicações de CFD, mas, apesar do grande sucesso, os métodos numéricos baseados em malha têm algumas limitações inerentes que limitam sua aplicação em muitos problemas de interesse prático:

- A geração de malha para representação de objetos complexos pode representar uma parte significativa do processo de simulação, ou pode até mesmo inviabilizar a aplicação do método numérico.
- A simulação de fenômenos que envolvem grandes deformações requer a introdução de procedimentos especiais de reorganização da malha, os quais são em geral, computacionalmente caros e podem introduzir imprecisões adicionais na solução [2].
- Fenômenos envolvendo fluidos não homogêneos, interfaces de material em movimento, limites deformáveis e superfícies livres também representam enormes desafios para simulações numéricas usando os métodos baseados em malha.

Esforços recentes para contornar as limitações dos métodos baseados em malha resultaram no desenvolvimento de uma nova geração de métodos numéricos, denominados métodos de partículas.

1.3 MÉTODOS DE PARTÍCULAS EM DINÂMICA DOS FLUÍDOS

A ideia fundamental dos métodos de partículas é obter uma discretização do domínio através de um conjunto de elementos arbitrariamente distribuídos e não

conectados (partículas), contornando a limitação inerente dos métodos com malha devida à necessidade conexão entre seus nós. Uma detalhada análise dos métodos sem malha existentes pode ser encontrada no livro [3].

A maioria dos métodos de partículas utiliza uma formulação Lagrangeana para representar o estado de um sistema por um conjunto de partículas discretas, que podem ser diretamente associadas a um objeto físico discreto, ou representar uma parte de um domínio contínuo. Cada partícula possui massa, posição, velocidade e energia interna. Outras variáveis são deduzidas por interpolação ou a partir das equações constitutivas.

As principais vantagens dos métodos de partículas podem ser resumidas como:

- Como não há conectividade entre as partículas, o tratamento de grandes deformações é facilitado.
- Maior facilidade para obter a discretização de objetos com geometria complexa.
- O refinamento de uma discretização por partículas é geralmente muito mais fácil de realizar do que o refinamento de uma malha.
- É relativamente mais simples obter as características de todo o sistema físico traçando o movimento das partículas e, portanto, identificar superfícies livres, interfaces em movimento e limites deformáveis .
- A evolução temporal de variáveis de campo, em qualquer ponto do material, pode também ser obtido naturalmente.

Um exemplo típico do método de partículas é o *Smoothed Particle Hydrodynamics* (SPH) [4][5], originalmente inventado para a modelagem de fenômenos astrofísicos foi, posteriormente, estendido para simulações de sistemas de partículas discretas e sistemas contínuos. Embora o método SPH e suas variantes sejam atualmente a estratégia dominante para simulações de escoamentos de fluidos compressíveis, a aplicação destes métodos à análise do escoamento de fluidos incompressíveis ainda não está totalmente desenvolvida.

Em meados de 1990, Koshizuka et al. [6] [7] [8] propuseram o método Moving Semi-implicit Particles (MPS), como formulação alternativa ao SPH tradicional, introduzindo uma etapa implícita para correção da pressão, a fim de garantir a incompressibilidade do fluido.

1.4 MOVING PARTICLES SEMI-IMPLICIT METHOD (MPS)

O MPS é um método numérico para simulação do comportamento dinâmico de um fluido. O método consiste em representar o domínio através de partículas e, aplicando o ponto de vista Lagrangeano, resolver as equações da continuidade e de Navier Stokes. A simulação consiste em duas etapas: na primeira, explícita, são consideradas as forças externas que agem na partícula, como a gravidade, a viscosidade e a tensão superficial, enquanto que na segunda, é computada a pressão do fluido e a correção da velocidade necessária para garantir que a densidade do fluido seja constante [9] [10] [11].

Por trabalhar sem a necessidade do uso de malhas, o método tem diversas aplicações em problemas que envolvem fragmentações, superfícies livres e grandes deformações. Muito se estuda a sua aplicação na influência do movimento de ondas em navios e até mesmo o escoamento de fluidos em sistemas biológicos. O impulso para o desenvolvimento deste método advém da necessidade de se simular a dinâmica de fluidos em condições extremas como é o caso em processos prospecção do petróleo, onde muitas vezes é difícil e economicamente inviável fazer ensaios físicos.

Para que uma simulação MPS possa produzir resultados precisos e confiáveis, ela deve ser suficientemente detalhada, o que implica na utilização de um grande número de partículas e, portanto, em requisitos de processamento e memória extremamente altos. Assim, é necessário que os recursos computacionais disponíveis sejam utilizados de forma otimizada, o que estabelece um forte vínculo entre a evolução das técnicas numéricas de simulação e a evolução das técnicas de computação de alto desempenho.

1.5 COMPUTAÇÃO DE ALTO DESEMPENHO (HPC)

A Computação de Alto Desempenho (*High Performance Computing* – HPC) é usualmente designa sistemas computacionais que se destacam pela capacidade de memória, armazenamento externo e poder de processamento. Assim, trata-se de uma definição sempre em evolução, acompanhando a evolução do *hardware*, para

incluir sistemas com qualquer combinação de alta capacidade de computação e de fluxos de dados. Mais recentemente, também uma alta capacidade de computação distribuída tornou-se parte dos atributos de sistemas HPC.

As arquiteturas de sistemas de HPC têm evoluído ao longo do tempo. No início da década de 1990 predominavam as arquiteturas com Multiprocessamento Simétrico (*Symmetric Multi-Processing* - SMP) e com Processamento Paralelo Massivo (*Massive Parallel Processing* - MPP) dos supercomputadores (*supercomputers*), usados para trabalhos muito complexos, como a pesquisa nuclear e cálculo de modelos para previsão tempo.

Mais recentemente, no entanto, a popularidade destas arquiteturas tem diminuído com o surgimento de uma abordagem com melhor relação custo benefício: a computação em *cluster* (*cluster computing*). Ao contrário das arquiteturas usuais dos supercomputadores, que empregavam processadores customizados, interligados por rede de dados com arquitetura proprietária e exigiam a utilização de softwares especiais para o desenvolvimento de aplicações paralelizadas, a computação em *cluster* emprega processadores genéricos, como os da Intel e AMD, interligados por redes padrão da indústria, tais como Gigabit Ethernet [12] e InfiniBand® [13]. Também o desenvolvimento de aplicações paralelizadas em clusters pode ser realizado utilizando ferramentas padrão, como o MPI [14], para processamento distribuído, e o OpenMP [15], para processamento paralelo em memória compartilhada.

A disponibilidade de redes *cluster* de alta velocidade tem aumentado significativamente e, atualmente, fornecem a configuração de HPC mais eficiente, flexível e de baixa relação custo/benefício. De acordo com o projeto 'Top500 Supercomputers Sites' [16], em 2012, a arquitetura de *cluster* era utilizada em mais de 80% dos principais sistemas HPC, Figura 1.1.

Figura 1-1 'Top 500' Supercomputadores por Arquitetura de Sistema (1993-2012)

Nestes ambientes computacionais, com memória e processadores distribuídos, há uma forte predominância da importância da arquitetura da solução e dos algoritmos não numéricos e um dos maiores desafios é criar uma solução flexível e aberta ao desenvolvimento, que ajude a reduzir o elevado custo de implementação de códigos paralelos e, além disto, é desejável a reutilização software existente[17][18].

Como cada nó de um *cluster* computacional consiste de um número fixo de processadores que compartilham uma mesma memória, uma aplicação serial pode ser paralelizada para explorar os recursos de um único nó através de técnicas de paralelização para memória compartilhada [19]. Estas técnicas são de fácil aplicação e, em geral, a relação entre o aumento do desempenho obtido e o número de processadores alocados é bastante satisfatória. Contudo, a quantidade de memória disponível em um nó e o número fixo de processadores se tornam fatores limitantes à escalabilidade da solução.

Por outro lado, a memória disponível e a capacidade de processamento de um *cluster* correspondem à soma destes recursos disponíveis em cada um de seus nós, assim, uma aplicação cujas tarefas possam ser distribuídas entre vários nós dispõem, em princípio, de recursos praticamente ilimitados. No entanto, aplicações desta natureza devem ser programadas para utilização de memória distribuída e, em geral, usam de forma intensiva os recursos de comunicação do *cluster* para

sincronização de informações entre processadores de nós distintos. A comunicação entre processadores de um mesmo nó pode ser feita através da memória compartilhada entre os processadores, o que resulta em algum ganho no tempo de comunicação. Ainda assim, a sincronização é realizada através da troca de mensagens e, portanto, o desempenho é inferior ao obtido pela paralelização com acesso direto aos dados compartilhados.

Para o efetivo aproveitamento dos recursos computacionais de um *cluster* é necessária uma combinação efetiva das duas técnicas de paralelização acima descritas (paralelização híbrida): as tarefas da aplicação devem ser distribuídas de forma balanceada entre os nós e os requisitos de comunicação para sincronização entre nós devem ser minimizados; em cada nó a tarefa recebida deve ser distribuída entre os processadores através de técnicas de paralelização para memória compartilhada, evitando assim a necessidade de comunicação entre os processos de um mesmo nó [20].

O desenvolvimento de uma solução para simulação numérica, com as características acima, depende essencialmente da identificação de propriedades do modelo matemático em estudo que possam ser exploradas para elaboração uma estratégia de paralelização híbrida.

1.6 FORMULAÇÃO MATEMÁTICA DO MÉTODO MPS

Nos parágrafos seguintes, apresenta-se a formulação matemática do método MPS destacando as duas propriedades do método que foram fundamentais na formulação do framework de paralelização do MPS. Uma refere-se à força de interação de curto alcance, tal como definido na formulação original do método, enquanto a outra resulta da introdução de uma compressibilidade limitada proposta por Ikeda et al.[21].

- **As equações governantes**

As equações governantes são expressas por leis de conservação de massa e quantidade de movimento:

$$\frac{D\rho}{Dt} = -\rho(\nabla \cdot u) = 0 \quad (1-1)$$

$$\frac{Du}{Dt} = -\frac{1}{\rho}\nabla P + f \quad (1-2)$$

onde, ρ denota a densidade, P a pressão, u a velocidade, f aceleração devido às forças externas e $\frac{D}{Dt}$ é a derivada lagrangeana.

- **Modelagem da Interação entre as partículas**

A primeira das duas características do método MPS exploradas pelo framework de processamento paralelo é a condição de forças de curto alcance: cada partícula interage com as partículas de uma vizinhança coberta por uma função peso $w(r)$, onde r é a distância entre duas partículas. A função peso empregado por Koshizuka em [6] é

$$w(r) = \begin{cases} \frac{r_e}{r} - 1 & \text{if } 0 < r \leq r_e \\ 0 & \text{if } r_e < r \end{cases} \quad (1-3)$$

A Figura 1-2 ilustra um exemplo de função peso, dada pela equação (1-3).

Figura 1-2 Gráfico da função peso, equação (4-3).

Como a área que é coberta por esta função peso é delimitada pelo raio de interação r_e , uma partícula interage apenas com um conjunto limitado de partículas vizinhas, Figura 1-3.

Figura 1-3 Interação entre partículas

Assim, a matriz de iteração é esparsa e pode ser rearranjada para uma forma de bloco tridiagonal ^[G.11] que pode ser facilmente distribuída, como será explicado no Capítulo 5.

- **Densidade número de partículas**

Para a partícula i , o número de densidade da partícula $\langle n \rangle_i$ é definido como:

$$\langle n \rangle_i = \sum_{j \neq i} w(|r_j - r_i|) \quad (1-4)$$

Nesta equação, a contribuição da partícula i a em si mesma não é considerada. O número de partículas em uma unidade de volume $\langle N \rangle_i$ é dado por:

$$\langle N \rangle_i = \frac{\langle n \rangle_i}{\int_V w(r) dv} \quad (1-5)$$

O denominador da equação (1-5) é a integral da função peso de toda a região exceto uma porção central ocupada pela partícula i . Assumindo que as partículas têm a mesma massa m , a densidade do fluido na partícula i , $\langle \rho \rangle_i$, é proporcional à densidade do número de partículas:

$$\langle \rho \rangle_i = m \langle N \rangle_i = \frac{m \langle n \rangle_i}{\int_V w(r) dv} \quad (1-6)$$

Assim, para um fluxo incompressível, a densidade do fluido é necessariamente constante, o que é equivalente a estabelecer uma densidade do número de partículas constante [7], este valor constante é denotado por n^0 .

- **A Modelagem do Gradiente**

Um vetor gradiente entre duas partículas i e j , que possuem quantidades escalares ϕ e ϕ_j , nas coordenadas r_i e r_j pode ser obtido por:

$$\frac{(\phi_j - \phi_i)(r_j - r_i)}{|r_j - r_i|^2} \quad (1-7)$$

O vetor gradiente para uma função escalar ϕ para quaisquer duas partículas pode ser obtido pelo cálculo da média ponderada do gradiente entre a partícula i e as partículas vizinhas j , usando a função peso:

$$\langle \nabla \phi \rangle_i = \frac{d}{n^0} \sum_{j \neq i} \left[\frac{(\phi_j - \phi'_i)}{|r_j - r_i|^2} (r_j - r_i) w(|r_j - r_i|) \right] \quad (1-8)$$

onde d é o número de dimensões espaciais e ϕ'_i é calculado como $\min(\phi_j)$ para qualquer j satisfazendo $w(|r_j - r_i|) \neq 0$ para melhorar a estabilidade do método. Este modelo de gradiente é aplicado ao termo gradiente de pressão no método de MPS.

- **Modelagem do Laplaciano**

Para modelar o operador Laplaciano, um problema de difusão transitório é considerado. Uma parte da quantidade de ϕ_i é distribuída para as partículas j da vizinhança utilizando a função de ponderação:

$$\langle \nabla^2 \phi \rangle_i = \frac{2d}{n^0 \lambda} \sum_{j \neq i} (\phi_j - \phi_i) w(|r_j - r_i|) \quad (1-9)$$

Onde o parâmetro λ é introduzido de modo que o aumento da variância seja consistente com a da solução analítica [8]:

$$\lambda = \frac{\sum_{j \neq i} w(|r_j - r_i|) |r_j - r_i|^2}{\sum_{j \neq i} w(|r_j - r_i|)} \quad (1-10)$$

Este modelo de Laplaciano é conservativo, uma vez que a quantidade perdida pela partícula i é exatamente igual à recebida pela partícula j .

- **Modelagem da incompressibilidade**

Para garantir a incompressibilidade a densidade de partículas deve ser constante. Quando as partículas se deslocam, um novo número de densidade de partículas n^* é calculado para cada partícula. Se este se desvia de n^0 por n' , n^* é corrigido de acordo com a equação:

$$n^* + n' = n^0 \quad (1-11)$$

onde n' é o valor da correção. Esta correção está relacionada com a correção da velocidade u' por meio da equação de conservação da massa:

$$\frac{1}{\Delta t} \frac{n'}{n^0} = -\nabla \cdot u' \quad (1-12)$$

O valor da correção da velocidade u' é deduzido a partir do termo implícito da equação do gradiente de pressão (1-2):

$$u' = -\frac{\Delta t}{\rho} \nabla P^{n+1} \quad (1-13)$$

A equação de Poisson para a pressão pode ser obtida substituindo as equações (1-11) (1-12) na equação (1-13):

$$\langle \nabla^2 P^{n+1} \rangle_i = -\frac{\rho}{\Delta t^2} \frac{\langle n^* \rangle_i - n^0}{n^0} \quad (1-6)$$

O lado direito da equação acima representa o desvio do número de densidade de partículas em relação ao valor constante n^0 . O lado esquerdo da equação é discretizado pelo modelo Laplaciano, equação (1-9). Expandindo o lado esquerdo da equação (1-7) e rearranjando os termos, obtém-se um sistema de equações lineares com matriz simétrica:

$$\sum_{j \neq i} (P_j^{n+1} - P_i^{n+1}) w(|r_j - r_i|) = - \frac{\lambda \rho (\langle n^* \rangle_i - n^0)}{2d \Delta t^2} \quad (1-8)$$

Em 1999, Ikeda et al. [21] sugeriram considerar a densidade do fluido como uma função linear da pressão e reescrever o lado direito da equação (1-9) como:

$$- \frac{1}{\Delta t^2} \left(\frac{\rho (\langle n^* \rangle_i - n^0)}{n^0} - \frac{P_i^{n+1}}{c^2} \right) \quad (1-11)$$

onde c é a velocidade do som.

Como apontado por Tanaka e Masunaga[22], embora isto introduza uma compressibilidade limitada no Modelo MPS, a simulação torna-se mais estável porque os elementos da diagonal principal da matriz de coeficientes são maiores, isto é, a matriz resultante é diagonal dominante. Esta propriedade confere bom condicionamento à matriz do sistema de equações lineares da pressão sendo, assim, a segunda característica do método MPS que garante o sucesso da implementação do método numérico iterativo utilizado no framework para solução do sistema de equações lineares.

1.6.1 O Algoritmo Serial do Simulador MPS

A implementação da versão serial algoritmo do MPS foi objeto da dissertação de mestrado em Engenharia Naval pela a Escola Politécnica da USP de Marcio Tsukamoto [11]. A Figura 1-4 apresenta o fluxograma do programa principal. Esta figura foi adaptada de [11] ajustando-se as referencias às equações para este trabalho.

Figura 1-4 Algoritmo do MPS, adaptado de [11]

1.7 SIMULAÇÃO DO MPS EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO

Para que as simulações com o MPS representem adequadamente o experimento físico, o raio das partículas deve ser suficientemente pequeno e, portanto, o número de partículas n deve ser grande. Com n da ordem de dezenas milhões, o número de elementos não nulos da matriz de interação $A(n,n)$ é superior a 10^9 nos modelos 3D.

Para acelerar a construção da matriz de interação utiliza-se o raio efetivo para limitar o número de partículas consideradas no modelo de interação (construção da lista de vizinhanças). Como a posição relativa das partículas é modificada com o movimento destas e de suas vizinhas, é necessário reconstruir a lista de vizinhanças a cada iteração[11][23].

Tipicamente duas tarefas são responsáveis por cerca de 70% do tempo de processamento: a construção da lista de vizinhança das partículas e a solução do sistema de equações lineares, a qual se torna mais onerosa à medida que a matriz de interação fica mais desestruturada devido ao movimento das partículas.

Técnicas de decomposição do domínio computacional e a utilização de estrutura de dados especiais para representação da lista de vizinhanças têm sido utilizadas como forma de diminuir o tempo de simulação do modelo [24][25][26][23]. No entanto, a solução do sistema de equações lineares ainda apresenta desafios: enquanto escalabilidade de métodos diretos de solução é limitada devido à redução da esparsidade da matriz, a escalabilidade de métodos iterativos ainda esbarra no alto custo da comunicação necessária para sincronização entre os subdomínios [27].

Em modelos matemáticos cuja evolução temporal não inclui a solução de um sistema de equações lineares a cada passo de iteração, como, por exemplo, em Dinâmica Molecular [28], a utilização da decomposição em subdomínios é relativamente simples, obtendo-se a formulação computacional paralelizada com poucas alterações na formulação serial original. Além disto, como todas as operações são realizadas sobre dados locais aos subdomínios e os requisitos de comunicação para sincronização periódica das fronteiras são moderados, é possível conseguir altos ganhos em desempenho e escalabilidade.

Em modelos matemáticos nos quais um sistema de equações lineares deve ser resolvido a cada passo de iteração, a solução deste sistema é o principal foco dos esforços de paralelização. Nestes casos duas abordagens têm sido adotadas, com diferentes níveis de sucesso.

Para sistema de equações lineares de dimensões moderadas, a solução, por métodos diretos, pode ser obtida com estratégias simples de reestruturação da matriz através renumeração das partículas, resultando em soluções com baixos requisitos de comunicação, porém com limitações severas na escalabilidade, devido à largura da banda da matriz resultante. Estratégias mais elaboradas de redução da banda da matriz[29] [30], aplicadas a cada iteração aumentam consideravelmente a complexidade da solução computacional sem ganhos equivalentes em escalabilidade.

Uma segunda abordagem é a utilização de algoritmos iterativos para a solução do sistema de equações lineares. Neste caso os requisitos de comunicação se tornam o principal limitante da escalabilidade da solução. Como forma de minimizar a comunicação entre subdomínios, e melhorar as escalabilidade da solução, diversas estratégias de renumeração das partículas e atualização de fronteiras foram estudadas [31][23] [32][33]. Estas soluções embora sejam capazes de simular modelos com alguns milhões de partículas, não apresentam um ganho em escalabilidade e desempenho compatível com o aumento na complexidade computacional da solução.

Um fator limitante do desempenho e da escalabilidade, comum às abordagens acima, refere-se à distribuição homogênea de carga entre os processadores (*Load Balancing*). Em modelos cuja distribuição espacial das partículas varia pouco ao longo da simulação, a divisão do domínio em subdomínios homogêneos no início da simulação pode ser suficiente para garantir o balanceamento de carga. Nos casos em que essa variação é significativa, a divisão do domínio é recalculada periodicamente segundo algum critério, como propriedades geométricas ou centro de massa dos subdomínios [33][34].

Nesta tese foi desenvolvida uma estratégia de decomposição de domínio para processamento distribuído do método MPS, que procura combinar algumas das melhores características de cada uma das abordagens descritas:

- a) Como nos modelos de Dinâmica Molecular, todas as operações são realizadas sobre dados locais, enquanto que as alterações introduzidas para a solução paralelizada ficam restritas a pontos específicos da solução serial.
- b) Para minimizar os requisitos de comunicação foi empregada uma estratégia de renumeração das partículas, comum nas soluções com métodos diretos, que permite a obtenção de uma matriz bloco tridiagonal ^[G.11].
- c) Para garantir a escalabilidade da implementação, a solução do sistema de equações lineares bloco tridiagonal é calculada por método iterativo.
- d) Como forma de garantir o balanceamento de carga ao longo de toda a simulação, foi criada uma estratégia de decomposição do domínio, com base na renumeração das partículas, que estabelece a distribuição homogênea das partículas entre os subdomínios na inicialização da simulação. Deste modo é possível acompanhar dinamicamente a variação da distribuição espacial das partículas entre os subdomínios, mantendo fixo o tamanho destes.

Como resultado da estratégia adotada, cada um dos algoritmos da solução - como renumeração das partículas; construção da vizinhança das partículas; construção e solução do sistema de equações lineares local e cálculo da parte explícita - atuam de forma independente sobre dados locais, permitindo que, em cada caso, sejam combinadas técnicas de otimização e de paralelização em memória compartilhada mais adequadas.

2 Objetivos

O objetivo deste trabalho é a implementação de um framework computacional de paralelização híbrida ^[G.4] para o Simulador MPS do TPN, com alto grau de escalabilidade, que utilize de modo efetivo os recursos de processamento paralelo disponíveis em clusters SMP ^[G.20].

Como o simulador MPS é utilizado em uma ampla gama de aplicações, outro importante objetivo é desenvolver o framework com grande flexibilidade, permitindo que modificações nos métodos numéricos e nos modelos físicos utilizados no MPS sejam facilmente implementadas, aplicativos de pré e pós-processamento disponíveis sejam aproveitados e a integração com bibliotecas numéricas, que apresentem funcionalidades e desempenho adequados ao projeto, seja simplificada.

3 ORGANIZAÇÃO DO TRABALHO

No capítulo 4 é apresentada uma revisão bibliográfica dividida em duas partes. Na primeira, é apresentada uma revisão dos trabalhos encontrados na literatura relacionados à paralelização de Métodos de Partículas, com ênfase no método MPS. Na segunda parte, o foco é a solução de sistemas lineares muito grandes utilizando processamento distribuído.

No capítulo 5 são apresentadas as principais características da solução desenvolvida e sua relevância ao tema em estudo.

No capítulo 6 é descrita a metodologia de desenvolvimento, baseada na tecnologia de hardware disponível e nas técnicas e de engenharia de software.

No capítulo 7 são apresentadas as etapas do desenvolvimento, com ênfase nos algoritmos que implementam a paralelização distribuída: solução do sistema linear, renumeração de partículas, sincronização e distribuição das estruturas de dados.

No capítulo 8 são apresentados os resultados dos testes de validação da solução.

O capítulo 9 apresenta uma análise dos resultados obtidos e as conclusões deste trabalho.

Por último, devido ao caráter interdisciplinar deste estudo, foi incluído um glossário entendido, que apresenta uma breve descrição de termos específicos da Álgebra Linear Computacional empregados neste estudo. Nas descrições de propriedades computacionais de matrizes, incluídas neste apêndice, procurou-se destacar sua importância na formulação de algoritmos.

4 REVISÃO BIBLIOGRÁFICA

4.1 ESTRATÉGIAS COMPUTACIONAIS PARA SIMULAÇÃO POR MÉTODOS DE PARTÍCULAS

O MPS é um método numérico de natureza *lagrangiana* para aplicação em problemas de escoamento de fluidos incompressíveis. O MPS teve origem em trabalhos pioneiros de Koshizuka e Oka [35][36].

O foco inicial dos autores foi o aperfeiçoamento do método original, visando resolver problemas de estabilidade numérica e a aplicabilidade do modelo a fenômenos físicos mais complexos, ainda assim, já neste período de consolidação do método tiveram início os esforços para melhorar seu desempenho computacional.

Em 1996, Koshizuka et al. [35] desenvolveram modelos para os operadores gradiente e laplaciano, além de um modelo para o tratamento de superfície livre. Os autores apresentaram, também, resultados de testes para aperfeiçoamento dos valores para o coeficiente de superfície livre e para o raio de vizinhança.

Em 1998, Koshizuka et al. [36] aplicaram o método MPS à simulação de ondas geradas e quebradas em uma praia. Até aquele momento, não havia simulação em três dimensões com o método MPS, pois o atrito artificial devido ao movimento desordenado das partículas introduzia erros na simulação. Os autores apresentaram, também, uma otimização para o cálculo da vizinhança que reduzia o tempo de processamento de $\mathcal{O}(N^2)$ para $\mathcal{O}(N^{1.5})$, onde N é o número de partículas.

Nos anos seguintes, com as bases do método melhor estabelecidas e com o aumento do interesse por simulações 3D, de maior custo computacional, aumentou o interesse de pesquisas focadas principalmente no aumento da escalabilidade e melhora do desempenho computacional do método, através do aperfeiçoamento de algoritmos existentes e a introdução de algoritmos paralelizados.

Em 1999, Vertanen [37] desenvolveu uma implementação paralelizada para o método SPH. Da perspectiva atual trata-se de implementação bastante limitada, usando somente modelos 2D com alguns milhares de partículas. No entanto, o autor além de inovar na metodologia de desenvolvimento, utilizando-se de modelagem

orientada a objetos, descreve de forma muito instrutiva suas dificuldades em transformar uma solução inicialmente serial em uma versão paralelizada. O autor relata como certas dificuldades enfrentadas no desenvolvimento paralelizado não têm similar no desenvolvimento serial, com destaque para a identificação de erros. De particular interesse foi um fenômeno que a versão paralelizada apresentava após alguns milhares de passos de simulação: o surgimento e proliferação aleatórios de “buracos” no interior do fluido. A investigação deste problema, segundo o autor, resultou na implementação de diversas ferramentas essenciais para monitoração de detalhes da simulação e mostrou a importância da execução de testes de longa duração desde os estágios iniciais do desenvolvimento.

Em 2003, Premože et al. [9] apresentou os resultados de simulações realizadas pelo método MPS utilizando recursos gráficos mais avançados. Os autores concluíram que o gargalo das simulações é a solução da equação de Poisson de pressão e sugeriram que este problema poderia ser contornado paralelizando-se a solução da equação. Além disso, concluíram que o uso de esquemas de interpolação mais sofisticado poderia melhorar a precisão dos resultados.

Em 2004, Yao et al. [23] utilizaram um lista encadeada de células para representação da vizinhança das partículas e conseguiram reduzir o tempo de cálculo da vizinhança para $\mathcal{O}(N)$, onde N é o número de partículas.

Em 2006, Tsukamoto[11] utilizou paralelização em memória compartilhada, como forma de acelerar o método MPS, para realizar a simulação de corpos flutuantes em ondas altamente não lineares, obtendo-se ganhos significativos de desempenho em relação à versão serial.

Ainda em 2006, Shibata e Koshizuka [38] simularam, em três dimensões, o impacto de água marítima em um convés utilizando o método MPS. A pressão de impacto da água no convés foi determinada e comparada com os resultados experimentais obtidos por um modelo bidimensional. O movimento do fluido, a elevação da superfície e a pressão de impacto da água no deck estavam de acordo com os resultados experimentais. Neste trabalho os autores verificaram a necessidade de desenvolvimento de novas técnicas para a aceleração da determinação da vizinhança, pois o modelo tridimensional era composto por 328.452 partículas.

Utilizando uma estratégia de divisão estática do domínio do problema, os autores conseguiram realizar a busca de vizinhanças com tempo de processamento de $\mathcal{O}(N)$, onde N é o número de partículas.

Em 2008, Ikari e Gotoh [32] comparam dois métodos de decomposição do problema: decomposição baseada nas partículas (*particles decomposition*) e decomposição do domínio (*domain decomposition*) e verificaram que a aplicação com decomposição de domínio tende a executar os cálculos em menos tempo que a aplicação utilizando decomposição por partículas.

Em 2009, Gotoh [33] desenvolveu uma versão do MPS para execução paralelizada, denominada 3D CMPS-HS, combinando técnicas de decomposição de domínio com fronteiras dinâmicas, recalculadas periodicamente com base no centro de massa de cada subdomínio, para melhorar o balanceamento de carga nos processadores e um processo de reestruturação da matriz de pré-condicionador para realização do processo de *forward/backward* do método de Conjugate Gradient em paralelo. Neste trabalho foram comparados os resultados de desempenho de dois métodos para solução do sistema de equações lineares: PICCG-RP (*Parallelized Incomplete Cholesky Conjugate Gradient with Renumbering Process*) e SCG (*Scaled Conjugate Gradient*). Os autores concluíram que a versão 3D CMPS-HS foi capaz de simular satisfatoriamente os modelos estudados, mas que ainda seriam necessários diversos aperfeiçoamentos ao modelo paralelizado: melhorar a precisão e eficiência computacional, através do desenvolvimento e aplicação de esquemas numéricos mais precisos e consistentes para os modelos de operador diferencial, bem como para a integração do tempo; melhorar o desempenho através do uso de solucionadores (*solvers*²) mais apropriados e rápidos e de estratégias de decomposições de domínio dinâmico mais eficiente tanto para problemas 2D quanto 3D.

² O termo inglês *solver*, que pode ser traduzido com solucionador, é comumente usado em computação numérica para designar uma implementação computacional de algoritmo para solução de sistemas de equações lineares. Para maior concisão deste texto, o termo *solver* será utilizado com esta conotação.

Em 2010, Iribe et al. [27] apresentaram resultados da simulação do MPS paralelizado para um *cluster* de PC's. Neste trabalho eles identificaram que o gargalo da paralelização do *solver* iterativo em memória distribuída é o custo computacional da comunicação entre os subdomínios. Para minimizar a comunicação os autores testaram um sofisticado processo de renumeração das partículas baseado em “pacotes” e no uso de uma lista de comunicação. Com estas técnicas conseguiram aumentar a velocidade de comunicação e realizaram uma simulação de tsunami com seis milhões de partículas em 237 horas. Os autores desenvolveram um processo eficiente de rotulagem das partículas baseado em um algoritmo de células ligadas, para reduzir o tempo de computação para uma simulação de dinâmica molecular por um fator de três. Eles mostraram que a melhoria do desempenho deve-se ao cumprimento eficiente de ambos os princípios localidade espacial e temporal, resultado da proximidade dos rótulos associados aos átomos que interagem. Finalmente, eles mostraram que o procedimento de reordenação apresentado pode ser usado para elaborar um eficiente esquema de decomposição unidimensional para processamento.

O histórico dos esforços em melhorar a escalabilidade e o desempenho do MPS mostra a importância e a complexidade desta tarefa. Em linhas gerais, as soluções analisadas se baseiam em duas estratégias básicas: a decomposição dinâmica do domínio e a utilização de *solver* iterativo. A primeira como forma de preservar o balanceamento de carga e a segunda devido a limitações de escalabilidade dos métodos diretos para solução de sistemas lineares.

A paralelização do MPS tem evoluído gradualmente através da combinação de algoritmos não numéricos cada vez mais sofisticados para o gerenciamento da configuração dinâmica dos domínios (construção da vizinhança das partículas, sincronização das fronteiras, etc.), minimização da comunicação na solução do sistema de equações lineares e reorganização da matriz de coeficientes de interação.

Nos trabalhos analisados as diferentes estratégias implementadas combinam, em grau variado, a soluções do problema de modelagem física / geométrica da decomposição do domínio do modelo com o problema matemático da solução de

sistemas lineares, resultando em soluções computacionais com unidades fortemente interligadas. Em soluções com esta característica o ganho em desempenho é em geral associado a um custo muito alto no tempo de desenvolvimento.

Para simulação de fluxo de fluídos por métodos numéricos com malhas, cuja história de desenvolvimento remonta à década de 1960, as estratégias básicas da modelagem computacional dos problemas numéricos estão mais bem estabelecidas e, com isto, muitos projetos de software tem focado a formulação de soluções mais gerais e flexíveis. O emprego de técnicas modernas de engenharia de software a estes problemas, para identificação das características matemáticas do modelo relevantes à solução computacional paralelizada, e a abstração dos objetos computacionais, tem resultado em soluções flexíveis e expansíveis [39] [40] [41]. No desenvolvimento do framework de paralelização do simulador MPS utilizou-se destas técnicas para combinar algumas das estratégias mais bem sucedidas da abordagem usual de paralelização do MPS.

4.2 SOLUÇÃO NUMÉRICA DE SISTEMAS LINEARES E HPC³

Embora o estudo de soluções numéricas para sistema de equações lineares anteceda em muito o desenvolvimento dos computadores, podendo ser encontrado em trabalhos pioneiros de Gauss e Jacobi ainda no século XIX [42]. Foi com o advento dos computadores, na década de 1950, que teve início uma explosiva atividade de pesquisa na área, tanto no nível teórico quanto no desenvolvimento de algoritmos. Atualmente, com a diversidade de algoritmos conhecidos e de bibliotecas computacionais disponíveis para uso imediato, selecionar a estratégia mais adequada para solução do problema de interesse é uma tarefa complexa. Muitos fatores influenciam esta escolha, sendo que para este estudo são de particular importância a dimensão e estrutura da matriz do sistema de equações lineares.

Os métodos para solução de sistemas lineares são tradicionalmente divididos em duas classes: os métodos diretos ^[G.14], que resolvem o sistema de equações lineares em um número finito de passos e, na ausência de erros de arredondamento, calculam a solução exata do problema; e os métodos iterativos ^[G.15], que geram uma sequência de soluções que convergem para a solução exata.

Os métodos diretos são muito robustos e têm sido utilizados com sucesso para solução de uma ampla classe de sistemas lineares com matriz densa. Quando a matriz é esparsa ^[G.9], em geral é utilizada uma etapa prévia de reordenação da matriz [43].

Embora diversas técnicas de reordenação da matriz tenham sido propostas para estender a aplicabilidade do método à matrizes esparsas de maior dimensão, a necessidade de resolver sistemas lineares cada vez maiores, tornou a utilização destes métodos cada vez mais inviável devido a duas limitações intrínsecas aos métodos diretos:

³ Na literatura internacional sobre métodos numéricos, os algoritmos estudados neste trabalho são, tradicionalmente, identificados pela abreviatura originada do nome em inglês. Nos textos em língua portuguesa, os algoritmos básicos são, eventualmente, identificados por abreviaturas derivadas do nome em português, não existindo, contudo, uma forma padrão de referência às suas variantes. Por simplicidade notacional e compatibilidade com a literatura internacional, optamos por utilizar neste estudo, apenas as abreviaturas tradicionais originadas dos nomes em inglês.

- a) A necessidade de matrizes intermediárias com menor esparsidade que a matriz original. Para sistemas lineares muito grandes, a memória requerida para armazenar estas novas entradas pode ser proibitiva, mesmo considerando-se o armazenamento distribuído [43].
- b) A complexidade computacional dos algoritmos de fatoração, que para matrizes densas, é proporcional ao cubo da dimensão da matriz. Assim, mesmo considerando que matrizes esparsas possam ser fatoradas em um número muito menor de operações por linha ou coluna que as matrizes densas, explorar a esparsidade da matriz, para otimizar o processamento distribuído, impõe grandes desafios para administrar os cálculos não estruturados, de forma a garantir a utilização eficiente dos processadores bem como um balanceamento de carga satisfatório [44].

Assim, uma vez que será necessário resolver numericamente sistemas lineares muito grandes, neste estudo procuramos identificar a opção mais viável entre os métodos iterativos.

Métodos iterativos clássicos, como Jacobi e Gauss-Seidel, Seção 5.3.1.1, foram inicialmente estudados, como alternativa aos métodos diretos, para a solução de grandes sistemas lineares com matriz esparsa diagonalmente dominante ^[G.12]. Estes métodos exigiam uma quantidade limitada de memória, mas apresentavam uma taxa de convergência insatisfatória e, portanto, um custo computacional alto. O empenho em acelerar a convergência destes métodos resultou na invenção, no início da década de 1950, do Método do Gradiente Conjugado – *Conjugate Gradient* (CG), Seção 5.3.1.2, para solução de sistemas lineares com matrizes simétricas positivas definidas ^[G.13]. Mais de duas décadas de estudos teóricos e técnicos, resultaram em uma extensa classe de métodos, os Métodos do Gradiente Conjugado Pré-condicionados – *Preconditioned Conjugate Gradient* (PCG), Seção 5.3.1.3, que se tornaram a estratégia padrão para solução numérica de sistemas lineares com matrizes simétricas positivas definidas [45].

A partir do início da década de 1970, com o surgimento dos primeiros computadores com capacidade de processamento paralelo, houve uma mudança na ênfase das pesquisas sobre o método do gradiente conjugado: a construção de algoritmos otimizados para as diferentes arquiteturas de processamento paralelo disponíveis

tornou-se predominante, em detrimento do estudo teórico sobre os efeitos da paralelização sobre o comportamento das respectivas versões seriais [46].

Como as principais arquiteturas de computadores operavam com memória compartilhada, a atenção estava em melhorar o desempenho realizando operações matemáticas sobre grandes blocos de dados, o que levou a elaboração de sofisticados algoritmos com base em operações vetor-vetor e matriz-vetor, culminando, no final do século passado, com a formulação de algoritmos baseados em operações sobre matrizes de blocos [44].

O desempenho dos algoritmos propostos, em geral, apresentava ótimos resultados para os sistemas lineares com matrizes da ordem de 10^5 .

A partir da década de 1990, com o surgimento de novas arquiteturas para computação paralela, baseadas em grupos homogêneos (*cluster*) ou grupos heterogêneos (*grid*) de computadores, o custo de expandir a capacidade de memória bem como de processamento deixava de ser a principal restrição à dimensão do problema que podia ser estudado por métodos numéricos. Como as técnicas tradicionais, baseadas na realização de um conjunto limitado de operações sobre grandes blocos de dados, não eram adequadas, porque, embora os recursos globais do grupo fossem virtualmente ilimitados, cada computador do grupo tinha relativamente pouca memória e grande capacidade de processamento. Assim, estas arquiteturas exigiram o desenvolvimento de novas técnicas de paralelização, usualmente denominadas Computação Distribuída, implicando em novos desafios técnicos e teóricos.

As principais estratégias utilizadas em computação distribuída baseiam-se em técnicas de divisão do domínio. Para problemas definidos por equações diferenciais parciais, estas técnicas caracterizam-se pela reformulação do sistema de equações diferenciais parciais original como um sistema equivalente de equações diferenciais parciais acopladas, definidas sobre os subdomínios, e impondo condições de contorno que garantam a igualdade da solução na fronteira de subdomínios adjacentes [47]. Para sistemas lineares esparsos, a representação na forma de matriz de blocos ^[G.11] pode ser vista como uma técnica de decomposição de domínio, considerando cada bloco da diagonal como um subdomínio, e impondo a

condição de igualdade da solução nos blocos fora da diagonal associados a subdomínios adjacentes [48].

O desenvolvimento de algoritmos para processamento distribuído aumentou em uma ordem de grandeza a dimensão dos sistemas lineares que podiam ser resolvidos com sucesso pelo método do Gradiente Conjugado, mas sistemas lineares com matrizes da ordem de 10^7 representavam ainda um desafio considerável [33], [27].

Enquanto os avanços obtidos na paralelização eram dificultados pela formulação essencialmente sequencial do PCG, uma nova classe de algoritmos iterativos, formulados para solução distribuída de sistemas lineares, passou a dividir o foco das pesquisas a partir do início dos anos 1980. Estes métodos, denominados Multimalhas - *Multigrid* (MG), foram propostos, inicialmente, apenas para solução de sistemas de equações lineares esparsos associados a modelos definidos em malha[49][50]. Posteriormente, foram propostas generalizações dos métodos MG independentes da geometria da malha. Esta nova classe de algoritmos passou a ser denominada Multimalhas Algébrico - *Algebraic Multigrid* (AMG), enquanto a formulação tradicional passou a ser denominada Multimalhas Geométrico – *Geometric Multigrid* (GMG), Seção 5.3.1.4. Embora os algoritmos AMG possam ser utilizados para solução de sistemas de equações lineares em geral, obter uma configuração do método que disponibilize todo seu potencial ainda depende fortemente do conhecimento da geometria da malha.

Isto foi comprovado pelos resultados expressivos obtidos por Baker et al. [51]. Em 2011 os autores estudaram escalabilidade de um *solver* AMG customizado para três arquiteturas distintas de *cluster* de computadores. Os autores foram capazes de resolver sistema de equações lineares da ordem de 10^9 incógnitas, associados a um problema de difusão 3D definido sobre uma malha com 8 regiões distintas. Para obter o máximo desempenho do *solver* em cada arquitetura de *hardware*, a especificação das submalhas era otimizada em função da arquitetura de processamento de cada *cluster*. Com esta estratégia os autores foram capazes de estudar modelos configurados de forma ótima tanto para processamento usando exclusivamente memória distribuída, quanto para processamento combinando memória distribuída e memória compartilhada (paralelização híbrida). Apesar do sucesso conseguido na solução de sistemas lineares tão grandes, os autores

concluíram que dois fatores ainda limitam a escalabilidade de métodos AMG: as implementações totalmente distribuídas necessitam trocar um número muito grande de mensagens curtas, o que torna a capacidade de comunicação um fator crítico na escalabilidade dos métodos AMG, de forma análoga ao que ocorre com o método do Gradiente Conjugado. Por outro lado, o uso da memória compartilhada, para reduzir a comunicação é dificultado pela extrema complexidade de lidar com códigos híbridos em plataformas com múltiplos processadores (*multicore*). Além disto, os autores verificaram que a flexibilidade conseguida ao formular os operadores de agregação, Seção 5.3.1.4, usando apenas elementos da matriz, em oposição à formulação geométrica, introduzia um limitante adicional à escalabilidade do método algébrico: a matriz de coeficientes se torna menos esparsa com a diminuição da resolução da malha⁴. Deste modo, afirmaram os autores, um elemento conectado à apenas seis vizinhos para uma dada resolução de malha, pode facilmente ter 40 vizinhos numa resolução menor. Considerando que, nos modelos tridimensionais baseados em partículas a vizinhança de um elemento da matriz de coeficientes (partícula), na resolução inicial do modelo, pode ter até 64 elementos (partículas) em modelo 3D, a progressão crescente no número de vizinhos pode ser um limitante crítico à escalabilidade do método.

Em 2006, Chow et al. [52] realizaram um levantamento detalhado das limitações dos métodos multigrid geométrico e algébrico, bem como das propostas elaboradas para contornar as restrições à paralelização.

Em 2007, na sua tese de doutorado pela Escola Politécnica da USP [53], Pereira analisou limitação crítica à escalabilidade do AMG, associada à natureza essencialmente sequencial do processo de escolhas dos elementos da matriz de coeficientes original para compor as matrizes associadas às malhas de menor resolução (Aglomerção). Pereira propôs uma estratégia para paralelização deste processo, através uma técnica de análise de sinais denominada *Wavelets*.

Ao contrario dos problemas com malhas não homogêneas, cuja estrutura pode ser utilizada para otimização do *solver* Multimalhas Algébrico, como fizeram Baker et al.

⁴ Embora nos métodos de Multigrid Algébrico a hierarquia de malhas seja substituída por uma hierarquia de matrizes, sempre que possível, faremos referência às operações sobre malhas, por estas serem muito mais intuitivas que as respectivas operações algébricas sobre matrizes.

[51], para os modelos de partículas, a configuração ótima do *solver* deve ser conseguida exclusivamente a partir da análise da matriz de coeficientes. Nestes casos, o *solver* é programado como “caixa preta” e pode ser utilizado, em princípio, para solução de sistemas lineares em geral. A implementação de *solvers* com estas características tem sido foco de intensa pesquisa resultando na disponibilização de bibliotecas com *solvers* Multimalhas Algébricos para uso em processamento distribuído. Estes *solvers* podem ser utilizados como alternativa aos *solvers* tradicionais, baseados no método do gradiente.

Em 2009 George et al. [54] realizaram um extenso estudo experimental com as bibliotecas Trilinos [55], HyPre [56] e PetSc [57], correspondendo, no total, a sete implementações de métodos Gradiente Conjugado Precondicionado e duas implementações de métodos Multigrid Algébrico. Foram utilizadas no estudo dezenas de matrizes simétricas positivas e definidas, com dimensão da ordem de 104 até 106, com exceção de uma com dimensão da ordem de 107 (42M). As matrizes foram selecionadas em distintas áreas de aplicação, como análise estrutural, dinâmica dos fluídos, simulação de circuitos. Entre estas matrizes apenas seis, incluindo a de maior ordem, eram diagonais dominantes e, portanto, mais similares às matrizes de coeficientes do método MPS.

Para as duas matrizes menores, com dimensões da ordem de 104 a 105, o tempo para solução dos sistemas lineares, com processamento distribuído em 64 processadores, foi inferior a um segundo, com desempenho ligeiramente superior dos solvers AMG. Para as três matrizes com dimensão da ordem de 106, na mesma configuração de processamento distribuído, os tempos de simulação, variaram entre 2 e 12 segundos, com desempenho nitidamente superior dos solvers PCG.

Os resultados para a matriz com dimensão da ordem de 107 apontaram sérios problemas de escalabilidade dos solvers analisados. Por um lado, apesar do bom desempenho apresentado com matrizes da ordem de 106, nenhum dos solvers PCG foi capaz de resolver o sistema de equações lineares com o processamento distribuído. Além disto, como um dos solvers PCG, quando configurado para execução no modo sequencial, obteve sucesso na solução deste sistema, o insucesso do modo distribuído deve ser atribuído, principalmente, a limitações de

escalabilidade do método PCG e não à dimensão, propriedade numérica ou estrutural da matriz.

Por outro lado, apenas um dos solvers AMG foi capaz de resolver este sistema de equações lineares, mas com desempenho insatisfatório: foram necessários mais de 6 minutos para reduzir o erro relativo a 10^{-5} .

Com resultados insatisfatórios das bibliotecas de *solvers* PCG e AMG, restava, apenas, a opção de desenvolver um *solver* adaptado às necessidades específicas do presente estudo. Embora, para processamento distribuído, os métodos AMG sejam potencialmente superiores aos métodos PCG, identificar uma estratégia eficiente para formulação dos operadores dos métodos AMG poderia, eventualmente, consumir o tempo total disponível para este estudo. Assim, foi necessário revermos literatura sobre os algoritmos iterativos distribuídos, com um enfoque diferente, que pudesse revelar alguma alternativa que tivesse sido indevidamente negligenciada.

A história dos métodos iterativos para solução distribuída de sistemas lineares, resumida em [42] e, em particular, dos métodos Multimalhas [52], mostra que desde o advento da computação distribuída, a linha principal de pesquisa de algoritmos para solução numérica de sistemas lineares tem sido a elaboração de estratégias para paralelização de algoritmos sequenciais, mesmo com alguma perda em suas características mais desejáveis. A formulação destes algoritmos deve considerar estratégias para sincronização de resultados intermediários e consolidação do resultado final, além de avaliar as vantagens do uso de estruturas especializadas para a representação distribuída dos dados. Assim, mesmo o método de Jacobi, em que cada elemento da solução aproximada pode ser calculado de forma independente, a versão distribuída do algoritmo é muito mais complexa que sua contraparte sequencial. A diferença de complexidade é ainda mais evidente quando a dependência no cálculo dos elementos da solução não pode ser totalmente eliminada, como no caso dos métodos PCG. A elaboração de estratégias para minimizar esta dependência, preservando as características desejáveis do algoritmo sequencial, é principal o foco dos esforços na formulação dos algoritmos. Estas estratégias impõem, em geral, um crescente custo computacional que acaba por superar os benefícios da distribuição do processamento.

Poucos autores se propuseram a estudar uma linha de estratégias diametralmente oposta: como os recursos de processamento disponíveis em ritmo crescente poderiam ser usados para compensar a baixa taxa de convergência do método Jacobi, levando o tempo de processamento a níveis aceitáveis. Estes estudos procuraram, inicialmente, analisar se a relação entre as taxas de convergência dos algoritmos iterativos clássicos, Jacobi, Gauss-Seidel e SOR (Successive Overrelaxation Method), Seção 5.3.1.1, era preservada após a paralelização distribuída destes algoritmos.

A taxa de convergência do algoritmo de Jacobi é sempre a menor das três, enquanto a do algoritmo SOR pode superar as demais, se for usado como valor ótimo de sobre-relaxamento (Optimal SOR)[56]. Enquanto o algoritmo de Jacobi é essencialmente paralelo, os demais são essencialmente sequenciais. No entanto, para sistemas lineares com matriz esparsa, a atualização de todos os valores de uma iteração pode, muitas vezes, ser realizada com um número razoavelmente pequeno de passos, utilizando-se, para isto, técnicas de paralelização baseadas no particionamento de matrizes [57]. Nestes casos, a convergência superior dos demais métodos sugere que estes sejam preferíveis ao método de Jacobi.

No entanto, em 1973, Traub [58] estudou experimentalmente a solução numérica de sistemas lineares tridiagonais, utilizando versões paralelizadas desses algoritmos. Com exceção do método de Jacobi, o autor explorou a estrutura tridiagonal das matrizes estudadas na estratégia de paralelização dos demais algoritmos.

Os resultados de desempenho foram quase o inverso do que seria esperado se o experimento fosse realizado com as respectivas versões seriais. Estes resultados foram sintetizados por Traub como abaixo:

- SOR Paralelo é superior a Gauss-Seidel paralelo.
- Nenhum destes algoritmos é competitivo com Jacobi ou Gauss Paralelo.
- Para matriz fortemente diagonalmente dominante, Jacobi é um pouco superior ao Gauss paralelo.
- Para matrizes com diagonal moderadamente dominante, Gauss paralelo é aproximadamente igual ao Jacobi.
- Para matriz com muito pouca dominância diagonal, Gauss paralelo é muito superior ao Jacobi.

Em 1987, van der Vorst [57] analisou as limitações de técnicas padrão de reorganização da matriz do sistema de equações lineares, como *Recursive Doubling*, *Cyclic Reduction*, e *Divide and Conquer* [59], como recurso para paralelização do método do Gradiente Conjugado. Ele utilizou a representação em blocos para matrizes em banda, a fim de elaborar uma técnica mais efetiva de paralelização para matrizes tridiagonais de blocos. O algoritmo proposto foi derivado de um algoritmo para fatoração LU de matrizes tridiagonais considerando matrizes-bloco no lugar de escalares. Embora a estratégia proposta, por representar matrizes em banda como matrizes de blocos, seja uma ferramenta poderosa para a identificação de oportunidades de paralelização em algoritmos sequenciais, os desempenhos dos algoritmos paralelizados não foram comparados com a versão sequencial do método CG para análise do ganho obtido.

Em 1988, Smart e White [60] analisaram teoricamente a taxa de convergência dos métodos de Jacobi e Gauss-Seidel. Usando a formulação matricial destes métodos, Seção, eles demonstraram que, para uma grande classe de matrizes $\rho(M_{GS})^r < \rho(M_{GS})$, onde ρ é o raio espectral da matriz e r é o número de passos intermediários necessários para computar uma iteração do método de Gauss-Seidel, ao contrário do esperado, com base nos resultados para os respectivos algoritmos seriais, o método Jacobi paralelizado é assintoticamente mais eficiente que o método GS paralelizado.

Em 1989, Tsitsiklis et al. [56] estenderam os resultados de Smart e White para algoritmos com iterações monotonicamente decrescente e para uma classe de algoritmos iterativos intermediários entre os métodos de Jacobi e Gauss-Seidel. Além disto, após testarem implementações de variantes do método de Gauss-Seidel e do método de Jacobi concluíram que a variante de Jacobi é preferível também para estes casos mais gerais e destacaram que estes resultados podem ter implicações práticas interessantes.

Em 1989, Bertsekas e Tsitsiklis [61] analisaram o efeito do uso de comunicação assíncrona em algoritmos numéricos paralelizados. Eles concluíram que, embora em várias circunstâncias métodos assíncronos possam ser uma alternativa desejável, as propriedades matemáticas de iterações assíncronas são bastante diferentes das de sua contraparte síncrona e, mesmo que uma teoria bastante abrangente tenha sido

desenvolvida, ainda há certas questões (com relação à taxa de convergência e de condição de termino de iterações assíncronas) que não estão satisfatoriamente respondidas.

Em 1993, Koester [62] realizou uma abrangente revisão da literatura sobre métodos diretos e iterativos para solução de sistema de equações lineares com matrizes bloco diagonal com borda. O foco principal do trabalho foi a análise do desempenho dos métodos diretos, fatoração LU e fatoração de Cholesky, e do método iterativo de Gauss-Seidel para solução de sistemas lineares com matrizes positivas definidas com diagonal dominante.

Koester enfatizou que o desenvolvimento de algoritmos paralelos eficientes para matrizes esparsas exige mais do que apenas a implementação de versões paralelas dos algoritmos sequenciais para matrizes esparsas, uma vez que todo paralelismo deve ser identificado na estrutura da matriz esparsa e, para isto, são necessárias, em geral, as seguintes etapas de pré-processamento da matriz:

- a) Ordenação da matriz original para obtenção de uma matriz de blocos;
- b) Fatoração simbólica da matriz, para minimizar o *fill-in* ^[G.3] nos métodos diretos, ou ordenação por técnicas de *multi-coloring*, para explicitar o paralelismo dos cálculos nos métodos iterativos.
- c) Balanceamento de carga da matriz para distribuir uniformemente os cálculos entre processadores.

Em geral, o tempo para realização destas etapas de pré-processamento, superou em muito o tempo da etapa de solução do sistema de equações lineares, sendo que em alguns dos exemplos estudados, o tempo de pré-processamento foi uma ordem de grandeza maior que o tempo de solução do sistema. Ainda assim, Koster concluiu que, para matriz diagonal dominante e com uma estimativa adequada do valor inicial da solução, o speedup⁵ do método iterativo poderia superar os métodos diretos em até 10 vezes para sistemas lineares muito grandes. É importante salientar que os sistemas lineares estudados por Koster tinham apenas alguns milhares de incógnitas.

⁵ Speedup, $S(n, p)$, é a razão do tempo de execução do algoritmo serial $T(n)$ pelo tempo de execução do algoritmo paralelo $T(n, p)$, $S(n, p) = T(n)/T(n, p)$ onde n é a “dimensão” do problemas e p o número de processadores.

Em 1996, Yamakawae et al.[63] apresentaram uma modificação do método do gradiente conjugado para resolver problemas de programação quadrática de larga escala que pode ser efetivamente implementado para memória distribuída. O método aplica o método de Jacobi para matrizes em blocos e executa a iteração do gradiente conjugado para resolver subproblemas de programação quadrática associada aos respectivos blocos. Os resultados numéricos indicaram que esta abordagem é particularmente eficaz quando os problemas a serem resolvidos têm alguma estrutura de bloco. Além disto, os autores ressaltaram que a modificação do método de Jacobi, para operação com matrizes em bloco deve aumentar a taxa de convergência do método.

Em 2001, B. Boudewijn et al. [64] estudaram a solução numérica de sistemas lineares de grande porte com matrizes altamente esparsas, no máximo 12 entradas não nulas por linha/coluna. Foram comparadas implementações paralelas dos métodos de Jacobi e Gradiente Conjugado. Usando memória em disco para armazenamento de resultados intermediários, foi possível resolver sistemas lineares com dimensões variando de 500 mil a mais de 700 milhões, executando em um *cluster* de estações de trabalho com 24 nós. Para os sistemas lineares menores, com até 54 milhões de incógnitas, o desempenho do método CG foi até 25% superior ao do método de Jacobi. No entanto, os sistemas lineares maiores, com mais de 700 milhões de incógnitas, somente puderam ser resolvidos pelo método de Jacobi, pois apesar da alta esparsidade das matrizes, o método do gradiente necessitava manter na memória principal vetores numéricos além da capacidade disponível.

Em 2004 Martinotto[65], realizou um estudo sobre a aplicação de métodos de decomposição de domínios para solução de sistemas lineares. Seus resultados mostraram o impacto negativo dos requisitos de comunicação sobre o desempenho e a escalabilidade do método CG, como mostra a Figura 4-1, onde foram justapostas as Figuras 5.51 e 5.52 de [65], cada barra horizontal corresponde ao tempo despendido em processamento, para realizar uma iteração do método. As cores foram utilizadas para diferenciar entre computação numérica e cada uma das operações de comunicação.

Figura 4-1 Execução do método aditivo de Schwarz e do CG em 7 processadores.

Justaposição da figuras 5.51 (esquerda) e 5.52 (direita) extraídas de [65]

A análise destas figuras mostra claramente que no método Aditivo de Schwarz⁶ ao custo de comunicação tem pouca influência no tempo total da iteração. Por outro lado, uma parte significativa do tempo de uma iteração do método CG, é despendida em operações de comunicação, principalmente em operações de *reduce*, necessária ao cálculo do produto entre vetores armazenados em memória distribuída. Além dos métodos exemplificados acima, Martinotto comparou, ainda, outras estratégias de solução de sistemas lineares baseadas em decomposição de domínio, e, entre suas conclusões destacamos uma que é de particular interesse a este estudo:

“Já o método aditivo de Schwarz apresentou resultados melhores. Esse método mostrou-se ser uma boa alternativa para a resolução de sistemas de equações em paralelo, apresentando, inclusive, uma escalabilidade maior que métodos paralelizados por decomposição de dados.”

Em 2005, Mehmood et al.[66] usaram um conjunto de 24 computadores fracamente acoplados, com 2 processadores cada, para fazer análise de estado estacionário de Cadeias de Markov de Tempo Contínuo. Os autores usaram o método de Jacobi para resolver sistemas lineares com até 1,25 bilhão de incógnitas.

⁶ O método de Jacobi para matrizes de blocos corresponde a um caso limite dos métodos de Aditivo de Schwarz para subdomínios sem sobreposição de fronteiras.

Em 2010, Raim [67] usou um solver CG paralelo para solução de sistemas lineares tridiagonais, com até 1,1 bilhão de incógnitas, para analisar o desempenho de um cluster de computadores com 64 nós com 8 processadores cada.

Para resolver o maior sistema de equações lineares tridiagonal, Raim [67] precisou alocar todos os 62 nós para executar o solver CG. Usando um processador por nó, o solver CG demorou 320 minutos para resolver o sistema com 1,1 bilhões de incógnitas e 3,3 bilhões de entradas. Por outro lado, Mehmood et al. [66], usando o solver Jacobi, executando em 48 processadores, conseguiram resolver um sistema de equações lineares com 1,3 bilhões de incógnitas e 5 vezes mais entradas em um tempo apenas 50% maior (482 minutos).

Embora não seja viável uma comparação rigorosa, entre os desempenhos do método CG e o de Jacobi, para solução de sistemas lineares com dezenas de milhões de incógnitas, os resultados de Raim [67] e Mehmood et al. [66] permitem inferir que o desempenho do método de Jacobi distribuído pode ser superior ao CG distribuído, para problemas desta ordem de grandeza.

Em 2006, Teresco et al. [68] realizaram uma detalhada revisão dos algoritmos de decomposição dinâmica de domínio, focando principalmente a eficiência no balanceamento de carga de processamento e concluíram que, apesar de ter sido desenvolvida uma ampla gama estratégias, a eficiência dos algoritmos disponíveis ainda depende fortemente das características do problema.

Em 2008, Bylinda e Bylinda [69] propuseram uma combinação dos métodos iterativos de Jacobi e Gauss-Seidel para a solução de grandes sistemas de equações. Com o algoritmo proposto, implementado para execução em *cluster* de computadores, foram resolvidos sistemas lineares esparsos, resultantes de um modelo markoviano para um mecanismo de controle de congestionamento, com até 30 milhões de incógnitas.

Em 2009, Wang e Gobbert [70] avaliaram o uso de comunicação assíncrona na implementação do método de Jacobi para matrizes de blocos, como forma de compensar a velocidade relativamente lenta da comunicação em relação à velocidade de cálculo. Ao contrário do esperado, os resultados mostraram que o desempenho da versão assíncrona era apenas 5% superior ao da versão síncrona.

Em 2012, Tottoni et al. [71] realizaram um estudo do desempenho do chip de 48 núcleos da Intel (Single-Chip Cloud Computer - SCC Intel). Utilizando ferramentas capazes de monitorar detalhadamente o consumo de energia, a temperatura, a utilização dos canais de comunicação e dos circuitos de memória, os autores avaliaram o desempenho do SCC executando cinco aplicações com requisitos computacionais bem distintos: simulação de dinâmica molecular, solução de um quebra-cabeça de xadrez, um algoritmo de ordenação e os algoritmos Jacobi e CG. Eles concluíram que todas as aplicações tem boa escalabilidade, com exceção do CG devido à necessidade comunicação global, com muitas mensagens de tamanho pequeno. O gráfico da Figura 4-2 mostra a evolução do speedup das cinco aplicações em função do número de processadores utilizados, com exceção do algoritmo CG, cuja implementação só podia ser executada em 2^n processadores, para $n = 1, 2, \dots, 5$. A Figura 4-2 foi obtida colorindo os elementos gráficos associados aos *solvers* na Figura 3 de.

Figura 4-2 Speedup das aplicações em números diferentes de núcleos de SCC. Gráfico obtido por colorização da imagem figura 3 extraída de [71]

Os resultados confirmaram a boa escalabilidade do método de Jacobi, inferior apenas ao do algoritmo de ordenação, enquanto o método CG apresentou a pior escalabilidade dos cinco algoritmos avaliados. Com base no monitoramento dos

canais de comunicação, os autores concluíram que a comunicação global de mensagens presente no método CG é a causa da baixa escalabilidade do método.

Com o crescimento explosivo da capacidade de processamento em *cluster*, devido à tendência a uniformização das arquiteturas e a padronização de técnicas de programação distribuída, é possível encontrar farta literatura sobre métodos eficientes para solução de sistemas lineares de grande porte. Entretanto, a grande maioria das matrizes de coeficientes nestes casos, tem origem em métodos com malhas e são, portanto, estruturalmente simples e bem definidas. Embora haja um interesse crescente na solução de sistemas lineares de grande porte, com matrizes esparsas estruturalmente mais complexas como, por exemplo, as resultantes da aplicação do método MPS e de alguns modelos de processos estocásticos, a literatura específica ainda é bastante escassa. Mesmo assim, formaram uma base sólida para a formulação de uma estratégia de solução do sistema de equações lineares do método MPS, perfeitamente integrada à estratégia de decomposição de domínio adotada, de grande escalabilidade e desempenho satisfatório.

5 ESTRATÉGIA DE PARALELIZAÇÃO DISTRIBUÍDA DO MPS

Para simulação do Método MPS com modelos da ordem de dezenas de milhões de partículas é claramente necessária uma estratégia de divisão do problema que permita distribuir os dados e o processamento de forma balanceada entre os recursos computacionais existentes. Em particular, é necessária uma forma efetiva de tratar a solução do sistema de equações lineares que, para problemas dessa magnitude, é responsável pelo maior custo computacional.

Para abordar problemas dessa magnitude é preciso primeiro estabelecer o modelo de decomposição de domínio a ser adotado, considerando a necessidade de minimizar a comunicação entre os subdomínios e, principalmente, garantir uma distribuição uniforme dos requisitos de memória e de processamento entre os processadores disponíveis. A estratégia atualmente mais usada para isto é divisão geométrica do domínio baseada no cálculo de centros de massa [32]. A atualização dos limites dos subdomínios é realizada periodicamente de modo a preservar, tanto quanto possível, uma uniformidade no número de partículas em cada subdomínio.

Para solução do sistema de equações lineares são relatadas diversas experiências com variações do método do Gradiente Conjugado, um método essencialmente sequencial, mas que devido a particularidades da matriz de coeficientes de iteração, pode ser adaptado para proporcionar algum grau de paralelização.

Os esforços mais recentes para paralelização do MPS esbarram em dois problemas: a manutenção da distribuição uniforme de partículas entre os subdomínios (balanceamento de carga) e redução da comunicação necessária para solução do sistema de equações lineares pelos métodos iterativos.

5.1 DIVISÃO DINÂMICA DO DOMÍNIO COM BASE NO VETOR DE RENUMERAÇÃO DAS PARTÍCULAS

Como forma de abordar ambos os problemas, foi formulada uma abordagem para o modelo de decomposição de domínio que, ao invés de utilizar uma decomposição geométrica dinâmica, recalculada periodicamente, utiliza o vetor de renumeração das partículas para dividir o domínio em subdomínios com o mesmo número de partículas, conforme ilustrado na Figura 3-1. Esta é uma formulação não geométrica para a decomposição de domínio, isto é, as fronteiras são estabelecidas de forma virtual pela divisão do vetor de renumeração de partículas, junto com as partículas referenciadas, em partes iguais entre os processadores.

Figura 5-1 Termos utilizados na descrição da Decomposição do Domínio

Com a renumeração de partícula, a matriz dos coeficientes é em banda [72]. A utilização do vetor de renumeração das partículas permite associar uma estrutura de bloco tridiagonal a esta matriz.

Na Figura 5-2, os números ordinais S_i^n , S_f^n , B_i^n , B_f^n determinam, simultaneamente, os limites do subdomínio n , o blocos da matriz de coeficientes do sistemas linear local (bloco matricial da diagonal) e as matrizes associadas à sobreposição das bordas do subdomínio (blocos matriciais fora da diagonal). Com esta formulação, as fronteiras

de cada subdomínio são definidas pelos limites de uma sequência fixa de números ordinais, em vez de através de divisão uma divisão baseada na posição do conjunto de partículas do domínio. Em modelos dinâmicos, com a posição das fronteiras calculas com base na posição das partículas, é necessário reconstruir periodicamente a divisão do domínio para preservar o balanceamento de carga.

Esta divisão é realizada apenas no início da simulação, como na decomposição estática do domínio. Além disto, na renumeração de partículas, necessária a cada passo de tempo garantir a reestruturação da matriz, o vetor de renumeração de partículas é recalculado. Como o conjunto de partículas de um subdomínio é determinado pela posição ordinal destas (partículas referenciadas em intervalo do vetor de renumeração), as partículas pertencentes ao subdomínio variam, seguindo a variação da posição das partículas ao longo da simulação, como na decomposição dinâmica de domínio.

Figura 5-2 Identificação do subdomínio e dos blocos da matriz de interação

Além disso, como o número de partículas em cada subdomínio é preservado, os subdomínios são de tamanho fixo e não diferem por mais de uma partícula.

Deste modo, podemos formular um algoritmo para solução do sistema de equações lineares com base em algoritmos para solução de matrizes bloco tridiagonais. Neste algoritmo cada subdomínio resolve localmente o sistema correspondente ao seu bloco da diagonal e faz as correções associadas às fronteiras com os subdomínios adjacentes, representadas pelos blocos fora da diagonal. Globalmente, um laço controla os desvios e verifica a condição de parada. Os subdomínios adjacentes comunicam entre si apenas o segmento da solução local associado aos elementos da fronteira.

5.2 RENUMERAÇÃO DAS PARTÍCULAS EM MEMÓRIA DISTRIBUÍDA

Algoritmos de reordenação de matrizes simétricas esparsas têm sido utilizados para acelerar a solução de sistemas lineares por métodos diretos ou para introduzir algum grau de paralelismo em algoritmos iterativos que, de outra forma, seriam essencialmente seriais [43] [44]. Algoritmos tradicionais de reordenação, como Reverse Cuthill-McKee, Gibbs-King e Sloan Ordering, baseiam-se na análise do grafo associado à matriz, onde cada nó do grafo corresponde a uma entrada [73], o que torna impraticável a aplicação destes algoritmos a matrizes de grande porte, pois o custo computacional desta análise depende do quadrado da dimensão da matriz.

Em 2007, Meloni et al. [74], desenvolveram uma estratégia alternativa para reestruturação da matriz de interação de modelos baseados em partículas que prescinde da análise do grafo da matriz, utilizando, para isto, um processo eficiente de rotulagem das partículas através de um algoritmo células ligadas. O objetivo principal dos autores era aumentar a localidade dos dados, isto é, otimizar os algoritmos para processamento vetorial, assim, a fim de melhorar a eficácia da reordenação eles não impuseram restrição que estrutura final da matriz fosse em banda. Em linhas gerais o algoritmo proposto se baseia na divisão do domínio em um conjunto homogêneo de células e na identificação das partículas que pertence a cada célula, de modo análogo ao utilizado por Yao [23] para otimização da construção de vizinhanças. Uma vez concluída a rotulação, as partículas de cada célula são reordenadas, como ilustrado Figura 5-3 extraída de [74].

Figura 5-3 Representação gráfica do algoritmo de ordenação de Meloni et. al para um modelo bidimensional. Figura extraída de [74].

Segundo os autores, além de reordenar a matriz em um tempo inferior ao algoritmo Reverse Cuthill-McKee, o algoritmo proposto permitiu, também, a implementação mais eficiente dos métodos de cálculo do modelo, pois, embora a estrutura final da matriz de interação não resultasse em banda, a estrutura resultou muito mais agrupada, o que possibilitou um melhor aproveitamento dos recursos de *cache* de memória devido à proximidade dos dados. O painel (c) da Figura 5-4 ilustra a estrutura final da matriz de interação ilustrada no painel (a). Na mesma figura, os painéis (b) e (d) ilustram respectivamente, a estrutura e banda da matriz após a reordenação pelo algoritmo Reverse Cuthill-McKee e a estrutura fortemente agrupada de um sistema cristalino, que, segundo os autores, reforça a adequação do método proposto para aglutinar as entradas da matriz.

Figura 5-4 Reordenação da matriz de interação: (a) sem reordenação; (b) reordenada pelo algoritmo Reverse Cuthill-McKee; reordenada pelo algoritmo de Meloni et al.; (d) exemplo de uma matriz de interação para sistema cristalino sem reordenação. Figura extraída de [74].

Meloni et al, afirmam, também, terem utilizado o algoritmo de ordenação proposto na concepção de uma estratégia muito eficiente para decomposição unidimensional do domínio, baseada no paradigma de passagem de mensagens. Neste caso, o domínio da simulação é dividido em fatias ao longo de uma das suas bordas e as partículas pertencentes a cada fatia são atribuídas a um determinado elemento de

processamento. Esta ideia foi retomada por Iribe et al. [27] para desenvolvimento de uma estratégia de redução da comunicação em computação paralela do método MPS. O algoritmo proposto em [27] parágrafo 4.1, está reproduzido abaixo (a utilização de marcadores e o realce de alguns trechos foram com negrito foram introduzidos para facilitar a referência ao texto, além disto, foram suprimidas as considerações sobre a orientação horizontal das linhas por não serem relevantes no presente contexto):

- Em primeiro lugar, **o domínio de análise é dividido em pequenos subdomínios por linhas verticais** (denominados *buckets*) em relação a um eixo especificado.
- Em seguida, um número de *bucket* é calculado a partir das coordenadas da partícula, bem como o número de partícula é registrado no *bucket*. A correspondência entre o número da partícula e o número do *buckets* é gravado. Esta operação é repetida para todas as partículas.
- As partículas registradas em cada o *bucket* são representadas na direção de um eixo, e então ordenadas sequencialmente.
- **Após a renumeração, as partículas são então divididas e atribuídas a cada núcleo.** Este agrupamento ‘localiza’ as partículas calculadas em cada processador e reduz a quantidade de comunicação.

Dois pontos devem ser ressaltados no algoritmo acima. Em primeiro lugar, o passo exige a determinação de linhas verticais para dividir o domínio, o que, para problemas com variação espacial da posição das partículas ao longo da simulação, obriga o recálculo destas linhas com base no centro de massa das partículas ou alguma outra estratégia equivalente. Em segundo lugar, no passo final, a distribuição das partículas entre os processadores é realizada após a ordenação, o que sugere a necessidade de acesso global a todas as partículas em um único banco de memória ou, então, a utilização de alguma estratégia de processamento distribuído não explicitada pelos autores.

Assim, para abordar simultaneamente as referidas limitações da estratégia proposta em [27], optamos por desenvolver uma estratégia renumeração (ordenação) que

considera a distribuição prévia das partículas entre subdomínios e realiza a reordenação sem necessitar de acesso global às mesmas.

Algoritmos genéricos de ordenação para memória distribuída possuem sua própria estratégia de distribuição dos elementos a ser ordenados [75], o que dificulta a aplicação destes algoritmos quando os dados estão previamente distribuídos. Deste modo optou-se por desenvolver um algoritmo adaptado às características específicas da estratégia adotada para paralelização do MPS.

Considerando que, em cada passo da simulação o movimento de uma partícula é inferior ao raio efetivo, as partículas de um subdomínio devem permanecer no mesmo subdomínio ou migrar para uma região delimitada de um subdomínio adjacente, foi proposto o algoritmo de ordenação em memória distribuída ilustrado na Figura 5-5. Este algoritmo é definido pelos seguintes passos, Seção 7.4:

1. Cada subdomínio realiza a ordenação das partículas de seu interior.
2. Sincronização das regiões de fronteira à “esquerda” dos subdomínios.
3. Sincronização das regiões de fronteira à direita dos subdomínios.
4. Cada subdomínio realiza a renumeração das partículas das regiões de fronteiras à esquerda.
5. De modo análogo ao passo anterior, realiza a renumeração da região de fronteira à direita de cada subdomínio.

Figura 5-5 Etapas da renumeração das partículas

Este algoritmo possui três importantes características:

- **Minimização da comunicação:** apenas dados da região de fronteira são comunicados entre os subdomínios adjacentes e esta operação é realizada uma única vez. Na rotina de renumeração, a rotina de sincronização é chamada uma única vez entre as etapas de renumeração.
- **Ordenação de dados locais:** as duas etapas de ordenação são realizadas sobre os dados disponíveis localmente ao subdomínio, de modo que podem ser utilizados os algoritmos mais eficientes disponíveis para ordenação serial ou paralelizados para memória compartilhada.
- **Redução da banda:** como resultado da renumeração das partículas sem o uso reagrupamento por células, como proposto em [74], a matriz de interação apresenta estrutura em banda mais estreita, mesmo quando as células são definidas em função do raio de interação, conforme ilustrado na Figura 5-6.

Figura 5-6 Estrutura da matriz de interação com (esquerda) e sem (direita) agrupamento das partículas por células

5.3 SOLUÇÃO DO SISTEMA DE EQUAÇÕES LINEARES EM MEMÓRIA DISTRIBUÍDA

Em Álgebra Linear Computacional são utilizados os mesmos conceitos da Álgebra Linear, no entanto como o foco principal é a formulação de algoritmos, a caracterização da estrutura das matrizes, isto é a distribuição dos elementos não nulos, e suas propriedades numéricas são de fundamental relevância. Isso deu origem a termos e siglas para caracterizar matrizes, bem como para identificar classes de algoritmos numéricos.

Devido ao caráter interdisciplinar deste estudo, alguns destes termos podem não ser familiares ao leitor. Assim, optamos incluir como apêndice um glossário estendido com definição dos termos mais relevantes. A definição de termos referentes a propriedades matriciais é seguida de considerações sobre sua importância na formulação de algoritmos. De modo análogo, na definição de termos referentes a algoritmos, procuramos destacar as características relevantes à solução de sistemas lineares de grande porte e ao processamento distribuído.

5.3.1 Métodos iterativos para solução de sistemas de equações lineares

5.3.1.1 Estacionários (ou Clássicos)

O método de **Jacobi** para solução de um sistema de equações lineares de dimensão n , na forma $Ax = b$, é comumente definido pela relação de iteração:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), i = 1, 2, \dots, n \quad (5-1)$$

Na equação (5-1) os valores da iteração corrente $x_i^{(k+1)}$ dependem apenas de valores da iteração anterior $x_j^{(k)}$, assim, todos os componentes da iteração corrente podem ser calculados simultaneamente e, portanto, o método é totalmente paralelizável. De forma análoga é definido o método de **Gauss-Seidel**:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j < i} a_{ij} x_j^{(k+1)} - \sum_{j > i} a_{ij} x_j^{(k)} \right), i = 1, 2, \dots, n \quad (5-2)$$

O método SOR (*Successive Over Relaxation*) é definido introduzindo-se um fator de sobre relaxamento ω na iteração de Gauss-Seidel:

$$x_i^{(k+1)} = (1 - \omega)x_i^{(k)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j < i} a_{ij}x_j^{(k+1)} - \sum_{j > i} a_{ij}x_j^{(k)} \right), i = 1, 2, \dots, n \quad (5-3)$$

A escolha do valor ótimo de ω depende de propriedades da matriz do sistema de equações lineares. Nos casos em que é possível escolher um valor ótimo de ω , o método é denominado *Optimal SOR*.

Nas equações (5.2) e (5.3), os valores da iteração corrente $x_i^{(k+1)}$ dependem também de $x_j^{(k+1)}$ para $j < i$, que são da iteração corrente. Esta dependência torna estes métodos essencialmente sequenciais, exceto quando aplicados na solução de sistemas lineares com estrutura especial.

As equações de iteração dos métodos estacionários podem também ser expressas na forma matricial considerando-se a decomposição da matriz do sistema de equações lineares $Ax = b$ na forma $A = D - L - U$, onde L , D e U são, estritamente, triangular inferior, diagonal e triangular superior [76].

O passo de iteração de Jacobi na forma matricial pode ser obtido a partir das seguintes transformações no sistema de equações lineares:

$$\begin{aligned} Ax &= b \\ Dx &= (L + U)x + b \\ x &= D^{-1}(L + U)x + D^{-1}b \end{aligned} \quad (5-4)$$

E a iteração do método de Jacobi deriva diretamente de (5.4):

$$\begin{aligned} x^{(k+1)} &= M_J x^{(k)} + D^{-1}b \\ \text{com } M_J &= D^{-1}(L + U) \end{aligned} \quad (5-5)$$

A matriz M_J , definida na equação (5-5), é denominada matriz de iteração do método de Jacobi. De forma análoga, o passo de iteração de Gauss-Seidel na forma matricial pode ser obtido a partir das seguintes transformações no sistema de equações lineares:

$$\begin{aligned}
Ax &= b \\
(D - L)x &= Ux + b \\
x &= (D - L)^{-1}Ux + (D - L)^{-1}b
\end{aligned} \tag{5-6}$$

E a iteração do método de Gauss-Seidel e a respectiva matriz de iteração resultam diretamente da equação (5-6):

$$\begin{aligned}
x^{(k+1)} &= M_{GS} x^{(k)} + (D - L)^{-1}b \\
M_{GS} &= (D - L)^{-1}U
\end{aligned} \tag{5-7}$$

A inversão da matriz M_{GS} impõe uma dependência entre os valores calculados na interação corrente, como fica evidente na versão estendida da iteração de Gauss-Seidel:

$$x^{(k+1)} = D^{-1}Lx^{(k+1)} + D^{-1}Ux^k + D^{-1}b \tag{5-8}$$

Analogamente obtêm-se a expressão matricial para as iterações dos demais métodos estacionários. Nas equações acima, o raio espectral das matrizes de iteração M_J e M_{GS} determinam a taxa de convergência dos respectivos métodos.

5.3.1.2 O Método do Gradiente Conjugado - *Conjugate Gradient* (CG)

O método de gradientes conjugado é usado para resolver os sistemas lineares quando a matriz dos coeficientes é SPD. Uma excelente exposição da estratégia que levou à formulação do CG está disponível em [44]. Além disto, muitos livros de computação científica e análise numérica do método apresentam os pormenores técnicos do método, como, por exemplo, o capítulo 10.2 de [77], uma referência clássica na área. Assim, nos limitaremos a uma descrição concisa do método, destacando, principalmente, suas propriedades mais relevantes ao presente estudo.

O CG é formulado como uma estratégia para minimização do funcional quadrático q :

$$\begin{aligned}
q(x) &= \frac{1}{2}xAx - x \circ b \\
\nabla q(x) &= Ax - b
\end{aligned} \tag{5-9}$$

Onde A é uma matriz $n \times n$ SPD, x e b são vetores coluna n -dimensionais, e o operador $\circ$ denota o produto escalar de vetores. Assim, se a solução do sistema de equações lineares $Ax = b$ for única, resolver o sistema é equivalente a minimizar o gradiente do funcional q .

O método do gradiente gera aproximações sucessivas da solução através de uma estratégia de minimização: partindo de uma estimativa inicial do mínimo do funcional q , o método prossegue escolhendo novas direções de pesquisa definida por vetores n -dimensionais $p^1, p^2 \dots$,

Assim, se p^k é a nova direção de pesquisa, e x^{k-1} é a estimativa anterior da solução, então, a nova estimativa x^k é calculada escolhendo um escalar α que segundo que satisfaça a condição:

$$\min_{\alpha} q(x^{k-1} + \alpha p^k) \quad (5-10)$$

A nova direção de pesquisa é calculada com base no resíduo r^k resolvendo

$$\begin{aligned} \min_{p^k \in W} \|p^k - r^k\| \\ \text{com } r^k = b - Ax^k \end{aligned} \quad (5-11)$$

onde r^k é o resíduo da iteração k e W é o complemento ortogonal de $\text{Span}(Ap^1, Ap^2, \dots, Ap^{k-1})$.

Pode ser mostrado que o valor de p^k calculado como:

$$\begin{aligned} p^k &= r^{k-1} + \beta^k p^{k-1} \\ \text{onde } \beta^k &= \frac{r^{k-1} \circ r^{k-1}}{r^{k-2} \circ r^{k-2}} \end{aligned} \quad (5-12)$$

minimiza a equação (5-11) e que o valor de α calculado como:

$$\alpha^k = \frac{r^{k-1} \circ r^{k-1}}{p^k \circ s^k} \quad (5-13)$$

minimiza a equação (5-10). Assim, o valor de x^k , que minimiza a equação (5-10) e, portanto, define a solução aproximada do sistema de equações lineares no passo k , é dado por:

$$x^k = x^{k-1} + \alpha p^k \quad (5-14)$$

Uma versão sequencial do método de gradiente conjugado está esquematizada no algoritmo abaixo:

Método de Gradiente Conjugado

```

k = 0;
x0 = 0;
r0 = b;
while ||xk||2 > max_dev and k < max_iter
    k = k + 1
    if k = 1
        p1 = r0
    else
        
$$\beta^k = \frac{r^{k-1} \circ r^{k-1}}{r^{k-2} \circ r^{k-2}} \quad \text{minimiza } \|p^k - r^{k-1}\|$$

        
$$p^k = r^{k-1} + \beta^k p^{k-1}$$

    end
    
$$s^k = A p^k$$

    
$$\alpha^k = \frac{r^{k-1} \circ r^{k-1}}{p^k \circ s^k} \quad \text{minimiza } q(x^{k-1} + \alpha p^k)$$

    
$$x^k = x^{k-1} + \alpha p^k$$

    
$$r^k = r^{k-1} - \alpha s^k$$

end
x = xk

```

Neste algoritmo, as operações que mais demandam recursos computacionais são as operações de produto matriz-vetor e produto escalar.

Em linhas gerais, o algoritmo CG para processamento distribuído é formado, essencialmente, pela mesma sequência de passos, mas, neste caso, a implementação das operações matriz-vetor e produto escalar devem considerar que os dados estão distribuídos entre os processadores [78].

O cálculo do produto matriz-vetor pode requerer informações adicionais de processadores vizinhos, nestes casos, a quantidade de dados depende da esparsidade da matriz e da topologia específica de processadores⁷.

O cálculo do produto interno de dois vetores exige uma comunicação global entre todos os processadores. Após o cálculo parcial do produto escalar, com dados locais a cada processador, é necessário somar estes valores para obter o valor global e, em seguida, o valor global deve ser enviado a todos os processadores, esta

⁷ A topologia de processadores é definida pela forma como são numerados. As formas mais comuns são a numeração sequencial simples (unidimensional) e a forma bidimensional, sendo que no primeiro caso o tamanho da vizinhança é 2 e, no segundo caso 8.

operação é usualmente denominada redução. Em processamento paralelo baseado no padrão MPI, esta operação é realizada através da função `MPI_allreduce`, que calcula uma soma e a distribui para todos os processadores [14].

Embora a taxa de convergência esperada do método CG seja muito superior à dos métodos clássicos, a opção por este método deve considerar, além dos requisitos de comunicação mencionados, os requisitos de memória para armazenamento da matriz de coeficientes, além de quatro vetores de dimensão n , ou das respectivas frações, no caso do algoritmo distribuído [79].

5.3.1.3 Método do Gradiente Conjugado Pré-condicionado - *Preconditioned Conjugate Gradient (PCG)*

A forma de aumentar a taxa de convergência do CG para solução de um sistema de equações lineares $Ax = b$, consiste em construir uma matriz P , tal que o número de condicionamento da matriz $P^{-1}A$ seja menor que o da matriz A e resolver o sistema equivalente $P^{-1}Ax = P^{-1}b$. Como a escolha ótima, $P^{-1} = A^{-1}$, equivale a resolver o sistema de equações lineares, o objetivo é a identificação de estratégias para construção de aproximações da inversa de A , que possam ser construídas eficientemente, em geral utilizando de variantes dos métodos diretos e dos métodos iterativos clássicos [77].

A matriz P , denominada pré-condicionador da matriz A , é escolhida com base nos seguintes critérios [78]:

- a) A matriz P deve ser facilmente construída;
- b) Deve ser possível eficientemente a solução do sistema de equações lineares $Py = d$ para qualquer vetor d .
- c) No caso do PCG distribuído, o algoritmo para a solução de $Py = d$ deve ter paralelismo suficiente para permitir uma implementação eficiente em arquiteturas de processamento paralelo com grande número de processadores;
- d) O número de condição da matriz $P^{-1}A$ dever ser o menor possível comparado com o de A .

Uma versão sequencial do método de gradiente conjugado pré-condicionado está esquematizada no algoritmo abaixo:

Método de Gradiente Conjugado Pré – condicionado

```

 $k = 0;$ 
 $x^0 = 0;$ 
 $r^0 = b;$ 
while  $\|x^k\|^2 > \text{max\_dev}$  and  $k < \text{max\_iter}$ 
     $k = k + 1$ 
    if  $k = 1$ 
        solves  $Pz^0 = r^0$ 
         $p^1 = z^0$ 
    else
         $\beta^k = \frac{r^{k-1} \circ z^{k-1}}{r^{k-2} \circ r^{k-2}}$  minimiza  $\|p^k - r^{k-1}\|$ 
         $p^k = z^{k-1} + \beta^k p^{k-1}$ 
    end
     $s^k = Ap^k$ 
     $\alpha^k = \frac{r^{k-1} \circ z^{k-1}}{p^k \circ s^k}$  minimiza  $q(x^{k-1} + \alpha p^k)$ 
     $x^k = x^{k-1} + \alpha p^k$ 
     $r^k = r^{k-1} - \alpha s^k$ 
    solves  $Pz^k = r^k$ 
end
 $x = x^k$ 

```

Além das considerações feitas a respeito método CG na Seção 5.3.1.2, é importante ressaltar que o no caso do PCG é necessário resolver o sistema de equações lineares com a matriz de coeficiente P e há um aumento nos requisitos de memória para armazenamento da matriz P e de mais um vetor de dimensão n .

5.3.1.4 Métodos Multimalha – Multigrid (MG)

A motivação para formulação dos métodos MG veio da análise do espectro de frequência do erro de aproximação dos métodos clássicos. Esta análise mostrou que, enquanto os erros de frequência muito baixa eram responsáveis pela lenta taxa de convergência destes métodos, os erros com frequência comparável à resolução da malha eram eliminados em poucos passos de iteração.

Os métodos MG foram desenvolvidos para tirar vantagem deste fato, considerando que, erros de baixa frequência para uma dada resolução da malha, serão tratados como erros de alta frequência em uma malha de menor resolução (*coarse grid*) e, portanto, amortecidos em poucas iterações.

Nos métodos MG é necessário construir uma hierarquia de malhas de menor resolução, baseadas na malha original. O algoritmo iterativo utilizado para amortecer o desvio da solução, para cada resolução de malha contida nesta hierarquia, é denominado **Suavizador**, enquanto o processo de aplicação de algumas iterações deste algoritmo é denominado **Relaxamento** (*relaxation*).

O processo de definir uma malha de menor resolução envolve a definição de três operadores:

- a) **Aglomerção** (*Agglomeration*) é o operador usado para combinação de diversos nós da malha original.
- b) **Restrição** (*Restriction*) usado para interpolação do erro entre uma malha e outra de menor resolução

Prolongamento (*Prolongation*), ou Extensão, que realiza a operação inversa da restrição, isto é, injeção do erro de uma malha em outra de maior resolução.

Para obter uma solução do sistema de equações lineares, os métodos MG realizam, recursivamente, um número pré-estabelecido de ciclos. Um ciclo típico inicia com a malha mais refinada. A solução obtida neste nível é, então, transferida para a próxima malha de resolução mais grosseira. Após alguns ciclos de relaxamento em uma dada resolução de malha, a solução é então transferida por restrição, à próxima malha na sequência de resolução decrescente, e assim sucessivamente até que seja atingida a malha de menor resolução. Neste ponto, a solução é obtida com máxima precisão, eventualmente por algum método direto, e transferida por prolongamento, segundo a sequência crescente de resolução das malhas, até o nível de maior resolução.

Para malhas regulares, a definição destes operadores, assim como a paralelização dos métodos Multigrid, não apresenta maiores dificuldades, e a aplicação do método, nas mais diversas áreas, foi realizada com sucesso para modelos cada vez maiores [51][52]. No entanto, para malhas não homogêneas e, principalmente, para modelos definidos sem uma malha de suporte, como os modelos de partículas, é necessário formular os operadores acima com base apenas nos elementos da matriz de coeficientes do sistema de equações lineares.

Os estudos de formulações alternativas ao MG tradicional, que fossem independentes da configuração da malha do modelo, deram origem a uma nova classe de métodos denominados Multigrid Algébricos - *Algebraic Multigrid* (AMG) [49]. Com isto, a classe dos métodos MG que exploram propriedades da malha de suporte do modelo para definição estes operadores, passou a ser denominada métodos Multimalhas Geométricos – *Geometric Multigrid* (GMG).

5.3.2 Seleção do método para solução do sistema de equações lineares em memória distribuída

Os métodos diretos baseados em fatorações triangulares têm proporcionado os algoritmos mais robustos e confiáveis para a resolução de sistemas lineares densos. Isto se deve ao fato de o seu tempo para solução do sistema de equações lineares ser previsível, quer a priori, quer após a reordenação da matriz de. Por outro lado, para soluções de sistemas com matriz esparsa, o *fill in* pode ser substancial e com isto, também, o tempo de execução[80]. Para simular modelos MPS com dezenas de milhões de partículas, sistemas lineares com bilhões de entradas devem ser resolvidos. Neste caso, o requisito de memória é simplesmente proibitivo.

No caso de sistemas lineares com matriz simétrica definida positiva, o Gradiente Conjugado é o método mais amplamente utilizado[32][81][21]. No entanto, apesar da eficiência da implementação serial do CG, sua escalabilidade para implementações distribuídas é reduzido devido às exigências de comunicação para resolver sistemas lineares muito grandes[67] [71].

A limitação de escalabilidade do CG é ainda mais crítica quando particularidades da estrutura da matriz de coeficientes não podem ser exploradas para otimização do algoritmo, como é o caso das implementações disponíveis nas bibliotecas numéricas para solução numérica de sistemas lineares em ambientes de processamento distribuído [54] [65].

Embora escalabilidade seja um fator de fundamental importância para seleção do método numérico empregado na solução de sistemas lineares de grande porte, ela informa apenas a razão entre os desempenhos do método em duas configurações de recursos de processamento. Assim, um método pode ter ótima escalabilidade, mas necessitar de uma quantidade de recursos proibitiva para atingir um

desempenho aceitável. Portanto, a escolha entre métodos numéricos deve considerar, também, a relação entre o desempenho destes métodos em situações equivalentes, tanto de dados de entrada e de recursos computacionais utilizados.

Embora não seja viável uma comparação rigorosa, como proposto acima, entre os desempenhos do método CG e o de Jacobi, para solução de sistemas lineares com dezenas de milhões de incógnitas, os resultados de Raim [67] e Mehmood et al. [66] permitem inferir que o desempenho do método de Jacobi distribuído pode ser superior ao CG distribuído, para problemas desta ordem de grandeza.

Para explorar as melhores características de ambos os métodos, a reconhecida excelência de CG para o SPD matriz de até alguns milhões de incógnitas e a facilidade da paralelização e melhor escalabilidade do método de Jacobi, optou-se por desenvolver um método híbrido, Jacobi / Gradiente Conjugado com base nas estratégias propostas em [69] e [63]. Para garantir a escalabilidade, a solução global do sistema de equações lineares é calculada pelo método de Jacobi, como em [69] e, a fim de acelerar a solução de sistemas lineares locais, o método de Gauss-Seidel utilizado em [63] foi substituído pelo algoritmo do CG paralelizado memória compartilhada, estratégia análoga à [63].

Em geral, algoritmos iterativos paralelizados para solução de sistemas lineares realizam, inicialmente, uma análise da distribuição das entradas na matriz de coeficientes para, com isto, identificar uma reestruturação da matriz que diminua a intercomunicação e equilibre a distribuição de carga entre os processadores. Esta análise pode ser eventualmente necessária para cálculo da matriz pré-condicionadores.

Entretanto, devido às características da matriz de coeficientes de iteração do modelo MPS após a renumeração, a solução do sistema de equações lineares pode prescindir desta etapa de análise: como matriz de interação é SPD, não há necessidade de reestruturação da matriz por questões de estabilidade numérica ou de convergência; como o domínio é dividido com base na estrutura em banda da matriz interação, esta pode ser distribuída de modo que cada subdomínio constrói o respectivo bloco do sistema de equações lineares, operando apenas sobre dados locais. Além disto, um balanceamento de carga quase homogêneo resulta da divisão

da matriz em partes iguais.

5.3.3 O Método de Jacobi para matriz bloco tridiagonal

O sistema de equações lineares definidas pelas equações (1-10) e (1-11) pode ser escrita como $AX = B$, onde X é o vetor de pressão, B é definido pela equação (1-11). A é uma matriz esparsa simétrica com diagonal dominante e, portanto, o método de Jacobi converge [45]. Com a renumeração de partículas, a matriz dos coeficientes é em banda ^[G.10] a qual pode ser associada a uma estrutura de bloco tridiagonal [72], como mostrado na Figura 5-7.

Figura 5-7 Segmento da matriz de coeficientes de iteração representada como matriz de tridiagonal de blocos.

A partir da estrutura de tridiagonal em blocos associada à matriz A , o algoritmo de Jacobi é definido pela equação:

$$\begin{aligned}
 A_1 X_1^{(k+1)} &= B_1 - \begin{bmatrix} 0 \\ A_1^r X_1^r(k) \end{bmatrix} \\
 A_n X_n^{(k+1)} &= B_n - \begin{bmatrix} A_n^l X_n^l(k) \\ 0 \\ A_n^r X_n^r(k) \end{bmatrix} \text{ para } 1 < n < p \\
 A_p X_p^{(k+1)} &= B_p - \begin{bmatrix} A_p^l X_p^l(k) \\ 0 \end{bmatrix}
 \end{aligned} \tag{5-3}$$

onde p é o número de subdomínios (dimensão da matriz de blocos), k é o número

da iteração, n é o número do subdomínio, l e r referem-se às bordas esquerda e direita, respectivamente. Este algoritmo está otimizado no sentido de que as dimensões das matrizes de blocos de fora da diagonal, A_n^l e A_n^r , e dos vetores de incógnitas associados, X_n^l e X_n^r , são reduzidas à largura da banda.

5.4 DISTRIBUIÇÃO DAS ESTRUTURAS DE DADOS

5.4.1 Partículas e vizinhanças

A representação das partículas distribuídas, como na versão serial, consiste de uma lista de vetores, um vetor para cada propriedade das partículas. Apesar desta semelhança de forma, existem profundas diferenças no modo como as partículas são identificadas e distribuídas na estrutura.

Enquanto na versão serial a posição de cada partícula é estática e serve de identificação para a partícula, na versão distribuída a associação é dinamicamente definida pelo do vetor de renumeração e a identificação da partícula é feita através de uma propriedade adicional.

A estrutura de dados para armazenamento das partículas, formada fisicamente de vetores contínuos, é logicamente composta de quatro regiões. A configuração inicial destas regiões da estrutura depende da posição das partículas, como é mostrado na Figura 5-8. A primeira região, em azul, contém as partículas internas ao subdomínio; a segunda região, em cinza, é reservada para a representação de partículas nos casos de simulações com *inflow* / *outflow* e as duas últimas regiões, em amarelo e verde, armazenam as partículas das fronteiras esquerda e a direita do subdomínio.

Figura 5-8 Estrutura de Dados Distribuída no Instante $t = 0$.

Ao contrário do que pode sugerir a Figura 5-8, não existe uma identificação dos limites destas regiões, os elementos que compõe cada região são determinados dinamicamente pelo conteúdo do vetor de índices da renumeração, como ilustra a representação da mesma estrutura num instante $t > 0$ da simulação, Figura 5-9.

Figura 5-9 Estrutura de dados distribuída num instante $t > 0$, após sincronização.

De forma análoga à estrutura de partículas, a estrutura de dados distribuída para representação da vizinhança das partículas, utiliza uma lista de vetores das propriedades da vizinhança, Seção 7.5. Como a vizinhança é utilizada apenas para as partículas internas ao subdomínio, o tamanho dos vetores da vizinhança é igual ao tamanho do subdomínio mais o tamanho da área reservada para *inflow/outflow*.

A associação entre uma partícula e a sua vizinhança é recalculada, também, com base no conteúdo do vetor de índices da renumeração.

5.4.2 Lista de materiais

Na versão serial a lista de Materiais é representada por uma estrutura contendo um vetor para cada tipo de material. O tamanho do vetor para cada material é determinado pelo número de partículas do material em questão, mais a área reserva para os casos de simulações com *inflow / outflow*, Figura 5-10.

Figura 5-10 Lista de Materiais da Versão Serial

Na versão distribuída esta estratégia não pôde ser adotada, pois cada subdomínio só tem conhecimento de um número reduzido de partículas, internas e de fronteira, e a composição do subdomínio pode variar ao longo da simulação, por isso optou-se por uma lista de vetores encadeados, Figura 5-11.

Figura 5-11 Lista de Materiais da Versão Distribuída

Nesta representação o tamanho de cada vetor varia em função do número de partículas de cada tipo existentes no subdomínio em cada passo da simulação.

5.4.3 Representação das matrizes do sistema de equações lineares

Para compatibilidade com os algoritmos numéricos da biblioteca MKL-Intel [82], a matriz do bloco diagonal do sistema linear de equações é armazenada no formato de matriz esparsa comprimida por linhas, *Compressed Row Storage* - CRS [82]. Para a representação na referência local, números de linhas e colunas são recalculados com base na posição inicial do subdomínio, como ilustrado na Figura 5-12. Nesta figura, *IA*, *JA* e *VA* são respectivamente: vetor de apontadores para o início da linha, vetor com os números das colunas com entradas e vetor com os valores de entradas. Como a matriz é simétrica, são armazenadas apenas as entradas abaixo da diagonal.

Figura 5-12 Representação CRS do bloco diagonal da matriz do sistema de equações lineares

Os blocos subdiagonais são armazenados utilizando três vetores: IA armazena o número da linha, JA o número da coluna e VA o valor da entrada. Neste caso, para conversão à referência local dos blocos subdiagonais à esquerda e à direita são utilizadas as posições iniciais das respectivas bordas.

Figura 5-13 Representação CRS do bloco subdiagonal da matriz do sistema de equações lineares

5.5 PARALELIZAÇÃO DAS OPERAÇÕES DE ENTRADA E SAÍDA

5.5.1 Leitura paralelizada dos dados das partículas

Ao contrário do arquivo de configuração do modelo, o arquivo com os dados das partículas, nos modelos com dezenas de milhões de partícula, é muito grande e, em geral, deve ser lido por dezenas de subdomínios que necessitam de partes distintas do arquivo. Neste caso a leitura paralelizada passa a ser um requisito fundamental. A forma trivial de dividir um arquivo entre N subdomínios é dividir o arquivo original em N arquivos distintos. A principal característica a favor deste modelo de paralelização, ilustrado na Figura 5-14, é que nenhuma comunicação entre processos ou coordenação necessária. Por outro lado, é necessário gerar um lote de arquivos de entrada diferente toda vez que variar o número de subdomínios.

Figura 5-14 Acesso paralelo aos dados: N arquivos para N Processos

Uma alternativa a este método de paralelização em que todos os subdomínios têm acesso ao arquivo todo, ilustrado na Figura 5-15, exige um projeto muito bem elaborado para coordenação do acesso ao arquivo que não cause bloqueio.

Figura 5-15 Acesso paralelo aos dados: um arquivo para N Processos

Uma terceira alternativa, que se ajusta perfeitamente ao caso em que os subdomínios necessitam de partes distintas do arquivo, é disponibilizada pela biblioteca MPI/IO, que é parte do MPI padrão 2. Com as funções desta biblioteca é possível estabelecer qual segmento do arquivo (visão) será lida pelos subdomínios, Figura 5-16.

Figura 5-16 Acesso paralelo aos dados com MPI/IO.

Para utilizar esta alternativa, foi necessário introduzir na fase de pré-processamento uma etapa de ordenação do arquivo de modo que as partículas de cada subdomínio sejam previamente agrupadas. Além disto, todos os registros do arquivo devem ser do mesmo tamanho.

5.5.2 Gravação dos paralelizada dos dados da simulação

Para gravação paralelizada dos dados da simulação utilizando a biblioteca VTK os arquivos de saída foram organizados como ilustra a Figura 5-17.

Figura 5-17 Estrutura dos arquivos de saída para gravação paralela

A cada intervalo de tempo previamente definido pelo usuário, conjuntos de parâmetros da simulação denominados “instantâneos” são gravados em arquivos de saída para posterior análise com ferramentas de pós-processamento. Estes arquivos, mais um arquivo de índice por processo, são gravados em uma mesma pasta e diferenciados por um sufixo contendo o número do processo (subdomínio) e, no caso dos instantâneos, um número do passo da simulação.

6 METODOLOGIA

Adicionar paralelismo a um algoritmo serial acrescenta novas dificuldades de implementação, devido à necessidade de comunicação entre os processadores e de sincronização dos passos do algoritmo.

O paralelismo introduz, também, alto grau de não determinismo, o que torna o desenvolvimento de programas paralelizados mais sujeito a erros e a depuração uma tarefa muito mais complexa.

Um problema exclusivo do desenvolvimento de programas paralelos corretos é a introdução de não determinismo (*nondeterminacy* [83]) introduzido pelo entrelaçamento de instruções: um subconjunto de instruções de um algoritmo, quando executadas em paralelo, podem ser intercaladas em diferentes sequência, que dependem, entre outros fatores, do que mais está sendo executado na máquina, quantos processadores estão sendo utilizados, qual o estado inicial da máquina, o leiaute de entrada na memória e seu efeito sobre a *cache* de página ou diferentes atrasos em redes de comunicação. Um programa pode funcionar corretamente em para números ímpares de processadores, mas incorretamente caso contrário. Este não determinismo restringe eficácia do uso de testes de regressão de código [84] para a prevenção de erros e, também, torna a depuração muito mais difícil, uma vez que não há previsibilidade sobre a repetição de um determinado erro [85].

Na Seção 6.2 são apresentadas as principais características dos sistemas paralelos e as ferramentas de paralelização utilizadas nestes ambientes. Em seguida, as Seções 6.3 e 6.4 apresentam as metodologias de projeto e desenvolvimento utilizadas para dar conta da tarefa de implementar, com qualidade, um sistema computacional flexível e escalonável para simulações de modelos hidrodinâmicos utilizando o modelo MPS.

6.1 REQUISITOS COMPUTACIONAIS DA SOLUÇÃO DE PARALELIZAÇÃO

Para que a solução de paralelização do MPS fosse bem sucedida foi necessário considerar os seguintes requisitos computacionais:

- **Compatibilidade:** O simulador MPS desenvolvido no TPN tem sido utilizado há vários anos, isto garante às principais rotinas do sistema um alto grau de confiabilidade. Deste modo o principal requisito do framework era que estas rotinas pudessem ser utilizadas com o mínimo possível de alterações a fim de não comprometer a confiabilidade do simulador.
- **Extensibilidade:** Atualmente cerca de uma dezena de projetos utiliza o simulador MPS para estudos que necessitam de modificações em funções específicas do simulador ou ainda a introdução de novas funcionalidades. Era, portanto, imprescindível que o framework pudesse suportar estas extensões de forma transparente, isto é, não exigindo do usuário conhecimentos além dos necessários atualmente para elaboração de simulações específicas com o MPS.
- **Escalabilidade:** Para o aproveitamento efetivo dos recursos do *cluster*, o framework precisava incorporar paralelismo em pelo menos três níveis de processamento:
 - A paralelização da parte explícita do simulador, conseguida pela restrição dos cálculos em cada subdomínio apenas a elementos internos ou de fronteira [19], [86], [87].
 - A paralelização da reordenação das partículas para obtenção da matriz de banda com técnicas de ordenação serial e paralela, e identificação da solução adequada ao esquema de divisão de domínio adotado [88], [86], [89].
 - A paralelização da solução do sistema de equações lineares obtida através de técnicas de solução de sistemas lineares representados por matrizes em banda [90], [91], [92], [93], [29], [94].
- **Balanceamento de carga:** além dos itens acima, um requisito fundamental para escalabilidade do framework era definir uma estratégia de divisão de domínio que garantisse o balanceamento da utilização de processamento em cada nó da rede.

6.2 PROGRAMAÇÃO PARALELA

O tipo de processamento paralelo utilizado neste trabalho pode ser definido como o uso simultâneo de múltiplos processadores para reduzir o tempo de execução de um programa. Outras formas de paralelismo, como no nível de bits, isto é, na implementação de uma instrução, ou no nível de instrução - múltiplas instruções de máquina sendo executadas simultaneamente - não são relevantes ao problema em estudo.

Essa forma de processamento paralelo requer máquinas com múltiplos processadores e/ou um *cluster*, sistemas operacionais capazes de suportar e de gerenciar esses processadores e aplicações que explorem essa forma de paralelismo. Existem três tipos de arquiteturas de hardware para esta forma de processamento paralelo [19]:

Arquiteturas de Memória Compartilhada (SMP): máquinas de memória compartilhada possuem uma memória global acessível a todos os processadores. Cada processador possui alguma memória local e o mesmo sistema operacional controla todos os processadores. Os processadores operam de maneira independente, porém apenas um processador por vez pode escrever em um endereço da memória compartilhada. A principal característica de sistemas de memória compartilhada é que o tempo de acesso à memória é independente do processador, embora isto não elimine a disputa por acessos a memória, mesmo para operações de leitura. A vantagem desta arquitetura é a facilidade no uso e a rapidez na transferência de dados entre processadores. Entretanto o número de processadores é limitado pela complexidade do sistema de acesso simultâneo à memória, resultando em baixa escalabilidade, que é a principal limitação dessa arquitetura.

Arquiteturas de Memória Distribuída (Cluster): máquinas de memória distribuída não compartilham memória, cada processador possui sua própria memória, invisível a todos os outros processadores. Os processos de uma aplicação paralela utilizam troca de mensagens para obter ou alterar os dados nas memórias dos outros processos. A vantagem dessa arquitetura é sua escalabilidade, maior que em máquinas com memória compartilhada, exatamente por não possuir acesso

simultâneo à memória. Além disso, cada processador acessa sem interferência sua memória local. A principal desvantagem é o elevado custo computacional de comunicação entre os processadores.

Arquiteturas Híbridas (SMP Cluster): trata-se de um conjunto de nós de processamento, onde cada nó é uma máquina com memória compartilhada. Os processadores do nó compartilham uma memória local comum e cada nó da rede pode compartilhar dados de memória com outro nó da rede através de troca de mensagens, [20].

Atualmente duas ferramentas se tornaram o padrão “de fato” para a implementação de aplicações paralelas nestas arquiteturas:

Message Passing Interface (MPI): é o padrão da indústria para programação em sistemas de memória distribuída, é baseado no modelo de troca de mensagens, com controle explícito do paralelismo. Os processos só são capazes de ler e escrever em sua respectiva memória local. Os dados são copiados para a memória local usando chamadas de sub-rotinas. A norma de interface de programação de aplicativos – *Application Programming Interface* (API) - do modelo MPI define um conjunto de funções e procedimentos que implementam o modelo de passagem de mensagens[14].

Características do padrão MPI:

- Códigos MPI podem ser executados tanto em arquitetura de memória distribuída como de memória compartilhada.
- Implementações genéricas de MPI têm sido portadas para quase todos os sistemas.
- Particularmente adaptáveis para paralelismo em *cluster*.
- Existe um grande número de bibliotecas MPI otimizadas.
- Cada processo tem sua própria memória local.
- Os dados são copiados através de mensagens entre memórias locais e são enviados e recebidos via chamadas explícita de sub-rotinas.

Open Specifications for Multi-Processing (OpenMP): é um padrão da indústria para programação em sistema de memória compartilhada [15]. Trata-se de uma

combinação de diretivas de compilação, biblioteca de rotinas e variáveis de ambiente, que é utilizada para especificar o paralelismo em máquinas de memória compartilhada. As diretivas são adicionadas ao código para especificar ao compilador a presença de uma região a ser executada em paralelo, juntamente com algumas instruções sobre o modo como a região deve ser paralelizada [95].

Características do padrão OpenMP:

- Códigos OpenMP executam em máquinas de memória compartilhada.
- Bastante portátil.
- Utiliza diretrizes que ajudam o compilador a paralelizar o código.
- Cada processo vê a mesma memória global, mas tem a sua própria memória local.
- Trocas implícitas de mensagens.
- Alto nível de abstração (superiores ao MPI).

A eficácia do paralelismo depende de como a computação sequencial é dividida entre os processos que compõe o programa paralelo. Esta divisão é conhecida como partição do problema. As principais formas de distribuir a solução de um problema são: a partição de tarefas e a partição de domínio. Nesta última, os dados do problema são divididos entre os processos.

Na partição de tarefas, as tarefas que compõe a solução do problema são divididas entre os processos. Ou seja, na partição de domínio cada processo executa a mesma computação em parte do conjunto de dados, enquanto na partição de tarefas cada processo executa uma tarefa sobre todos os dados. Um fator importante para a eficácia da paralelização, independente do modo de partição do problemas, é garantir que a carga de trabalho seja bem distribuída entre os processadores. Este fator é denominado balanceamento de carga

Em teoria, um modelo de memória compartilhada, como OpenMP, deveria oferecer uma estratégia mais eficiente de paralelização. Assim, uma combinação dos paradigmas de memória compartilhada e paralelização por passagem de mensagem dentro da mesma aplicação (modo híbrido de programação) pode prestar um serviço mais eficiente do que a estratégia de paralelização apenas por troca de mensagens

[96], [86]. Diversos trabalhos têm sido realizados para avaliar uma utilização destes dois paradigmas, individualmente e em conjunto [97], [96], [98].

6.3 O PARADIGMA DA ORIENTAÇÃO A OBJETOS

A abordagem orientada a objetos fornece um meio eficaz para a concepção de sistemas científicos complexos a qual permite projetar abstrações do problema e o desenvolvimento de aplicações que podem cumprir estritamente os requisitos de desempenho, embora ainda exista a alguma necessidade de pesquisas para aprimoramento das técnicas orientadas a objetos para aplicações de alto desempenho, [99].

A reutilização de software científico, adicionando componentes a um software existente, a fim de ser capaz de rapidamente testar novas funcionalidades, ganhou muita atenção e vários trabalhos científicos têm abordado esta questão, [100], [101], [102], [55]. A multiplicidade de métodos para resolver numericamente equações lineares, equações integrais e equações diferenciais parciais, combinado com uma multiplicidade de modelos físicos, necessita de um ambiente software em que tanto os métodos numéricos quanto os modelos matemáticos possam ser facilmente trocados.

O paradigma da orientação a objetos, como descrito em [103], adequa-se perfeitamente a essas necessidades, oferecendo um conjunto de características chaves para enfrentar a complexidade, permitindo flexibilidade e reutilização de código: encapsulamento, herança, polimorfismo e ligação dinâmica.

6.4 PROGRAMAÇÃO ORIENTADA A OBJETOS EM C++

Linguagens orientadas a objetos permitem a expressão completa das três ideias fundamentais da programação orientada para objeto[104]: objetos, hierarquia de classes, e polimorfismo:

Objetos: são definidos pelos seus atributos (dados) e métodos (funções de comportamento). Objetos generalizam o conceito de sub-rotinas, acrescentando os atributos, que por sua vez, podem influenciar o resultado de uma função do objeto (comportamento). Cada objeto é instanciado a partir de uma definição de classe

(especificação dos dados e da função do objeto). Todos os objetos da mesma classe têm operações comuns, mas (em geral) estados distintos.

Hierarquias de classes: organizam as classes para reutilização, permitindo que classes de um nível mais baixo da hierarquia herdem operações de classes de níveis superiores: operações não precisam ser recodificadas para as classes com componentes comuns.

Polimorfismo: permite que diferentes tipos de objetos, que compartilham certas propriedades ou comportamentos, sejam usados em código que só opera sobre estas propriedades ou comportamentos. Por exemplo, enquanto a herança suporta o uso de vetores bidimensionais para construir matrizes, tensores, ou malhas, o polimorfismo permite-nos escrever uma função para multiplicar duas matrizes, sejam eles armazenados como filas adjacentes ou com colunas adjacentes, ou em outro formato.

Descrever cálculos em termos de objetos e classes traz importantes benefícios:

- **Expressão direta:** objetos são metáforas naturais tanto de objetos físicos como de entidades abstratas. Expressar cálculos em termos de objetos reduz a diferença entre conceito e programa.
- **Maleabilidade:** bons programas evoluem, evolução é mais fácil quando as modificações são locais. Objetos, combinando dados com funções de manipulação destes dados, permitem controlar localmente o acesso a estes dados.
- **Extensibilidade:** usando herança, novos objetos e seu comportamento podem ser definidos como alterações incrementais e ampliações dos objetos existentes.
- **Abstração:** usando polimorfismo, semelhanças entre os objetos podem ser expressas no programa, permitindo-nos escrever código em termos de semelhanças sem ter em conta as diferenças.

C++ é uma linguagem orientada a objetos muito flexível e poderosa, [105],[104],[106]. Sem dúvida pacotes numéricos bem projetados podem se beneficiar de seus recursos, contudo, deve-se ter cuidado ao desenvolver aplicações de alto desempenho usando C++, uma vez que alguns dos recursos desta

linguagem podem resultar em perda de desempenho [104], [107]. Existem boas referências que discutem o impacto sobre o desempenho de recursos como sobrecarga de operadores (overload), polimorfismo dinâmico, etc. [99] [108].

6.5 O CLUSTER DO TANQUE DE PROVAS NUMÉRICO

O *cluster* do TPN é um *cluster* Sun, composto 192 nós (Sun Blade x6275). Os nós são configurados como:

- 4 servidores com sistema de arquivos de alto desempenho Lustre™ [109], com capacidade de armazenamento de 30 TB; um servidor de arquivos NFS [110] com capacidade de armazenamento de 10 TB, servidor para compilação; 2 *master nodes* (ou *master servers*);
- 184 nós de processamento dedicados. Cada nó possui dois processadores Intel® Xeon® Processor X5560 [111] (8M Cache, 2.80 GHz, 6.40 GT/s Intel® QPI, 24 GB de Memória RAM).

Todos os nós são conectados através de Infiniband QDR (quad data rate) [13] com taxa de transferência de 40 GB/s e latência de 100 nano segundos.

O sistema operacional instalado no *cluster* é a distribuição Enterprise-class Linux CentOS 5.4 [112].

6.6 FERRAMENTAS DE DESENVOLVIMENTO

As principais ferramentas utilizadas no desenvolvimento do framework foram:

- Compilador C++: *Intel(R) C++ Compiler Professional Edition 11.1 for Linux OS* [113].
- Biblioteca para solução numérica de sistemas lineares: *MKL - Intel® Math Kernel Library 10.3 for Linux OS* [82].
- Biblioteca MPI para processamento distribuído: *MPICH2 x86_64 v. 1.4.1p1: High-performance and Widely Portable MPI* [114].
- Biblioteca de classes C++ para geração de arquivos para pós-processamento: *Visualization Toolkit (VTK)* [115].
- Gerenciador de recursos de rede para execução de programas distribuídos: *Simple Linux Utility for Resource Management SLURM 2.2.5* [116].
- Ferramenta de construção de software (*make*): *SCons: A software construction tool* [117].
- Processador para análise, validação, serialização e manipulação de XML, escrito em C++: *Xerces-C++ XML parser* [118].

7 DESENVOLVIMENTO DO FRAMEWORK DE PARALELIZAÇÃO

Na estruturação do framework procurou-se estabelecer uma partição funcional das tarefas de modo que os algoritmos não numéricos, associados às tarefas de gerenciamento da decomposição de domínio, e algoritmos numéricos associados ao cálculo das equações do modelo MPS, fossem independentes entre si. Além disto, procurou-se também construir uma solução distribuída formada, quase exclusivamente, por uma combinação de tarefas executadas sobre dados locais em cada subdomínio.

A combinação dessas características resultou em uma solução de paralelização com os seguintes benefícios:

- **Reutilização de código:** as rotinas numéricas do modelo serial puderam ser aproveitadas com poucas alterações;
- **Paralelização híbrida:** os principais algoritmos operam sobre dados locais, utilizando estratégias de processamento para memória compartilhada. Estes algoritmos são em geral mais simples e eficientes que os algoritmos para processamento em memória distribuída.
- **Desenvolvimento ágil:** testes do funcionamento geral do framework puderam ser antecipados, implementando-se inicialmente soluções simplificadas para cada tarefa e transferindo o investimento em aperfeiçoamentos das soluções locais, como otimização e paralelização em memória compartilhada, para após a validação da estratégia de paralelização para memória distribuída.
- **Flexibilidade:** futuras evoluções poderão ser introduzidas no modelo numérico sem impacto significativo no código de gerenciamento e vice-versa.

Para apresentação da estrutura geral do framework optou-se pela utilização de diagramas de classes, como é usual em projetos orientados a objetos. Os algoritmos mais relevantes, utilizados na construção dos subdomínios, na estruturação dos dados distribuídos, e nas demais operações específicas do processamento distribuído, são apresentados pelo respectivo código fonte em C++ comentado. O uso de fluxograma, comum na descrição de código serial, não se mostrou adequada

na maioria dos casos, assim sua utilização ficou restrita à apresentação do algoritmo geral do método MPS e ao algoritmo principal do solver.

A integração do framework de paralelização no algoritmo serial do MPS não acarretou nenhuma mudança significativa nos procedimentos de cálculo, com exceção da forma de percorrer a vizinhança de uma partícula. As principais alterações se resumem a uma etapa de inicialização do framework de paralelização e a introdução de um passo de renumeração das partículas no laço principal do simulador. Na Figura 7-1, que apresenta o fluxograma geral da versão distribuída do simulador MPS, estas alterações estão indicadas pelo sombreado mais escuro.

As alterações necessárias na etapa de inicialização do framework de paralelização são apresentadas na Seção 7.5. A descrição detalhada dos algoritmos de cálculo, como desenvolvidos para aversão serial, esta disponível na dissertação de mestrado de Tsukamoto [11].

Figura 7-1 Fluxograma principal do simulador MPS distribuído

7.1 INTEGRAÇÃO DO FRAMEWORK AO SIMULADOR MPS SERIAL

Para integração do framework à versão serial do MPS foi necessário acrescentar ao *script* de construção da aplicação [117] referências à biblioteca MPI e à biblioteca do gerenciador de recursos distribuídos Slurm [116]. Um exemplo destas referências, elaboradas para a estrutura de diretórios das respectivas instalações no cluster do TP, está ilustrado no Código fonte 7-1.

```
//    Diretórios da instalação da biblioteca MPICH
env.Append(LIBPATH=['/usr/lib64/mpich2/lib'])
env.Append(INCLUDE=['/usr/include/mpich2-x86_64'])

//    Diretórios da instalação da biblioteca SLURM
env.Append(LIBPATH=['/opt/slurm/lib'])

//    links as bibliotecas
env.Append(LIBS=['pmi', 'm', 'mpich'])
env.Append(LINKFLAGS='-Wl,-rpath,/usr/lib64/mpich2/lib:/opt/slurm/lib:./lib')
```

Código fonte 7-1 Trecho do script de construção da aplicação distribuída

Além disto, para execução distribuída no cluster do TPN é necessário usar um *script* de comandos Linux para alocação dos nós do cluster e execução aplicação pelo Slurm, ilustrado no Código fonte 7-2.

```
#
#    nmpi = número de nós
#    nomp = número de processadores por nó
#    model = arquivo XML de configuração do modelo
#    dMPS = nome do programa executável do simulador MPS distribuído
#
export OMP_NUM_THREADS=$nomp

srun --ntasks=$nmpi --cpus-per-task=$nomp --ntasks-per-node=1 --exclusive ./dMPS $model
```

Código fonte 7-2 Script Linux para execução da aplicação distribuída

Os principais parâmetros para especificação de como os recursos do cluster devem ser alocados pelo Slurm são:

- `--exclusive`: o nó alocado à uma tarefa não pode ser compartilhado com outras tarefas em execução.

- `--ntasks-per-node`: especifica o número máximo de tarefas que pode ser invocado em cada nó. Deve ser igual a 1 para aplicações de paralelização híbrida, MPI+OpenMP, onde apenas um par MPI *task/rank* deve ser atribuído a cada nó, permitindo que a parte OpenMP utilize todo o paralelismo presente no nó.
- `--cpus-per-task`: número de processadores alocados para cada tarefa. Na forma especificada apenas reforça que todos os processadores do nó devem estar disponível para a parte OpenMP.

7.2 IMPLEMENTAÇÃO DAS CLASSES SUBDOMÍNIO E SIMULADOR

A classe `CSubdomain` contém os atributos e métodos necessários ao controle da decomposição de domínio, enquanto a classe `CSimulator` contém os métodos numéricos utilizados para simulação do modelo MPS. A Figura 7-2 ilustra os principais elementos destas classes.

Figura 7-2 Diagrama das classes CSubdomain e CSimulator

Atributos da classe `CSubdomain`:

- `nprocs` e `rank`: número de processadores e identificação do subdomínio, respectivamente.
- `Bi`, `Si`, `Sf`, `Bf`: identificam os limites do subdomínio.
- `part`, `prop`, `sim` e `Isort`: lista de partículas, tabela de propriedades das partículas, parâmetros da simulação e vetor de renumeração, respectivamente.
- `neighborhood`: instância da classe de vizinhança das partículas.
- `model`: apontador para o objeto com os dados globais da simulação.
- `leftBorder` e `rightBorder`: apontadores para as bordas esquerda e direita do subdomínio, respectivamente.

- `syncBorder`: instância da classe `CSyncBorder` utilizada para sincronização das bordas.
- `solver`: apontador para uma instância da classe `CBlockSolver` utilizada para solução do sistema de equações lineares.

Para ilustrar as diferentes responsabilidades das classes `CSubdomain` e `CSimulator`, foram destacados alguns métodos destas classes. Estes métodos, assim como alguns métodos omitidos, foram apresentados nas seções anteriores deste capítulo. A classe `CSimulator` foi definida como derivada da classe `CSubdomain` para simplificar as referências aos elementos do subdomínio necessários nos métodos numéricos.

A inicialização do simulador foi alterada para contemplar uma etapa de construção dos subdomínios, que antecede a leitura do arquivo de partículas. Isto foi necessário para permitir o gerenciamento da alocação de memória para a estrutura distribuída de representação das partículas. Além disto, foi incluída no laço (*loop*) de iteração uma etapa de renumeração das partículas. O fluxograma da Figura 7-3 apresenta os passos da etapa da construção dos subdomínios e a descrição dos passos mais dos procedimentos mais importantes dos algoritmos de cada etapa é apresenta nas seções subsequentes.

Figura 7-3 Fluxograma da construção do subdomínio

7.2.1 Leitura da configuração do modelo e dos limites do domínio

Os parâmetros de configuração do modelo para o simulador serial são descritos com detalhes por Tsukamoto [11]. Inicialmente estes parâmetros eram representados em um arquivo texto não estruturado, posteriormente foi elaborada uma estruturação XML e desenvolvida uma função, `xmlReader()`, para leitura dos dados com base no *parser* XML Xerces-C++ [118]. Esta função foi incorporada à versão distribuída sem alteração usando a função de serialização, Código fonte 7-3 (1).

De forma análoga, a leitura dos limites do domínio, definidos no cabeçalho do arquivo de partículas, Seção 0, utiliza a função de serialização para controle do acesso ao arquivo de partículas, Código fonte 7-3 (2).

```

void CSubdomain::Initialize(int argc, char *argv[])
{
(1) |Serialize(this, &CSubdomain::xmlReader);
(2) |Serialize(this, &CSubdomain::modelReadBoundary);
    ...

```

Código fonte 7-3 Leitura serial do arquivo de configuração e dos limites do domínio (cabeçalho do arquivo de partículas)

O método de serialização, `Serialize()`, implementa um semáforo simples para sequenciamento do acesso à qualquer recurso computacional requerido pelo método implementado pelo parâmetro `PMethod method`, Código fonte 7-4, onde `PMethod` é definido como: `typedef void (CSubdomain::*PMethod)();`, isto é, `PMethod` é um apontador para um método da classe `CSubdomain` que não retorna nenhum valor.

```

void Serialize(CSubdomain *me, PMethod method)
{
    char message = '1';
(1) | int source = rank-1;
    | int dest = rank+1;
    | int count = 1;
    | int tag = 0;
    | MPI_Status status;

    if (rank == 0) {
(2) | (me->*method)();
    | MPI_Send(&message, count, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    else {
(3) | MPI_Recv(&message, count, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
    | (me->*method)();

    if (rank < nprocs-1)
        rc = MPI_Send(&message, count, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
}

```

Código fonte 7-4 Método de serialização

Os principais passos deste algoritmo são:

1. Com base no próprio `rank`, o subdomínio, determina o vizinho à esquerda `source`, do qual ficará aguardando uma mensagem, e o vizinho a direita `dest`,

para o qual irá enviar uma mensagem após a execução do método especificado no parâmetro `method`.

2. O subdomínio 0 executa o método, sem aguardar mensagem alguma, e envia uma mensagem para o subdomínio.
3. O demais subdomínios aguardam uma mensagem antes da execução do método e, com exceção do último, após a execução do método enviam uma mensagem para o subdomínio a direita.

7.2.2 Cálculo de valor estimado para a largura da borda

Para computar uma estimativa da largura da borda o método `ComputeBorderWidth()` executa os seguintes passos:

Calcula o número máximo de partículas do domínio do modelo em cada eixo.

Se o modelo for 2D, atribui a `w` o número máximo de partículas na direção do eixo oposto ao de ordenação, `axes_order[i] == 1`.

Se o modelo for 3D, atribui `w` o número de partículas necessárias para formar um retângulo cujos lados são definidos pelo o número máximo de partículas na direção dos eixos opostos ao de ordenação.

Estima a largura da banda, `border_size`, considerando o número de partículas necessárias para preencher uma região com largura aproximada de duas vezes o raio efetivo grande. Nos casos em que, devido à geometria do modelo, o valor estimado é muito maior que o valor efetivo, obtidos durante a simulação, o usuário pode reiniciar a simulação informando uma estimativa mais adequada.

```
void CSubdomain::ComputeBorderWidth()
{
(1)  nx = (modelBoundary->xmax - modelBoundary->xmin) / sim->dPart + 1;
    ny = (modelBoundary->ymax - modelBoundary->ymin) / sim->dPart + 1;
    nz = (modelBoundary->zmax - modelBoundary->zmin) / sim->dPart + 1;

(2)  int w;
    if (sim->dim == 2) {
        if (axes_order[0] == 1) w = (ny == 0 ? nz : ny);
        if (axes_order[1] == 1) w = (nx == 0 ? nz : nx);
        if (axes_order[2] == 1) w = (nx == 0 ? ny : nx);
(3)  }
    Else {
        if (axes_order[0] == 1) w = ny * nz;
        if (axes_order[1] == 1) w = nx * nz;
        if (axes_order[2] == 1) w = nx * ny;
    }
    border_size = (long)(1.2 * w * (1.0 + sim->reL / sim->dPart));
}
```

Código fonte 7-5 Cálculo da largura da borda

7.2.3 Cálculo dos limites dos subdomínios

A determinação dos limites dos subdomínios, B_i , S_i , S_f e B_f , deve contemplar os casos em que o número de partículas não é múltiplo do número de subdomínios. Nestes casos, os subdomínios iniciais terão uma partícula a mais. Estes limites são computados pelo método `ComputeBoundaries()`, Código fonte 7-6, através das seguintes etapas de cálculo:

1. Inicialmente atribui ao subdomínio o número mínimo de partículas, `minSize`, assumindo uma distribuição homogênea das mesmas entre os subdomínios.
2. Se o número do subdomínio, `rank`, é menor que o resto da divisão do número de partículas do modelo, `modelSize`, pelo número de subdomínios, `nprocs`, soma um ao número de partículas.
3. Calcula o início do subdomínio, S_i , somando o tamanho de todos os subdomínios à esquerda do subdomínio corrente.
4. Calcula o final do subdomínio, S_f , considerando tamanho do subdomínio corrente. Os valores obtidos para S_i , S_f e a largura da borda, `border_size`, são, então, utilizados para computar os limites das bordas, B_i e B_f .
5. Calcula o número máximo de partículas com base no tamanho do subdomínio, das bordas e do percentual de reserva `sim->reserve`.

```

void CSubdomain::ComputeBoundaries()
{
(1) | int minSize = int(floor((double)modelSize / (double)nprocs));
    | int m = modelSize % minSize;
    | sim->nPart = minSize;
(2) | if(rank < m) sim->nPart += 1;

    | Si = 0;
(3) | for (int i = 0; i < rank; i++){
    |     Si += minSize;
    |     if (i < m) Si++;
    | }
(4) | Sf = Si + sim->nPart-1;
    | Bi = Si;
    | Bf = Sf;
    | if (Bf + Nb <= modelSize) Bf += border_size;
    | if (Bi > Nb) Bi -= border_size;
(5) |
    | sim->nPartMax = sim->nPart * (1.0+sim->reserve/100.0) + 2* border_size;
    | }

```

Código fonte 7-6 Cálculo dos limites dos subdomínios

7.2.4 Alocação parcial de estrutura de dados e leitura das condições iniciais das partículas

Para leitura das condições iniciais das partículas em modo distribuído é necessário alocar um buffer de memória temporário com tamanho suficiente para conter o segmento do arquivo de partículas correspondente ao subdomínio. Por isto, somente a parte da estrutura de partículas, referente às condições definidas no arquivo, é construída na memória, Código fonte 7-7.

```
void CSubdomain::AllocGrid()
{
    long nSize = sim->nPartMax;
    makeIntArray(&part->idMat, nSize);
    makeVector(&part->r, nSize);           // Position
    makeVector(&part->u, nSize);           // Velocity
    makeRealArray(&part->p, nSize);        // Pressure
    makeRealArray(&part->pndS[0], nSize);  // Particle Number Density
    makeRealArray(&part->Temp, nSize);     // Particle Temperature
}
```

Código fonte 7-7 Alocação das estruturas para armazenamento das condições iniciais das

Após a alocação parcial da estrutura de partículas, a leitura distribuída das condições iniciais é feita pelo método `ParallelReadGridFile()` apresentada na Seção 7.6.

7.2.5 Inicialização do vetor de renumeração

O vetor de renumeração, `Isort`, e seu inverso, `Ipart`, são inicializados pelo método `initIsort()`, conforme mostrado no Código fonte 7-8. Neste algoritmo é importante ressaltar que as apenas partículas do interior do subdomínio recebem um identificador, `part->partID` (1), enquanto que os identificadores das partículas de fronteira somente serão conhecidos após as etapas de sincronização. Os parâmetros `border_left_start` e `border_right_start` determinam a posição inicial na estrutura de partículas reservada para as partículas das bordas esquerda e direita respectivamente.

```
void CSubdomain::initIsort()
{
    // initialize Isort for interior particles
    for(long i = 0; i < sim->nPart; i++)
    {
        long is = Si + i;
        Isort[is] = i;
        Ipart[i] = is;
        part->partID[i] = is;
    }
    // initialize Isort for left border particles
    if (rank > 0)
    {
        for(long k = 0; k < border_size; k++)
        {
            long is = Sf - border_size + k;
            long i = border_left_start + k;
            Isort[is] = i;
            Ipart[i] = is;
        }
    }
    // initialize Isort for right border particles
    if (rank < nprocs-1)
    {
        for(long k = 0; k < border_size; k++)
        {
            long is = Sf + 1 + k;
            long i = border_rigth_start + k;
            this->Isort[is] = i;
            this->Ipart[i] = is;
        }
    }
}
```

Código fonte 7-8 Inicialização do vetor de renumeração

7.2.6 Alocação final da estrutura de partículas

Uma vez liberada a memória utilizada para leitura das condições iniciais das partículas, a alocação da memória para as demais propriedades é realizada pelo método `allocParticlesState()`, como mostra o Código fonte 7-9.

```
void CSubdomain::allocParticlesState()
{
    long nSize = sim->nPartMax;
    makeIntArray(&part->index, nSize); // Particle State
    makeVector(&part->rn, nSize); // Position prediction
    makeVector(&part->dr, nSize); // Position correction
    makeVector(&part->un, nSize); // Velocity prediction
    makeVector(&part->du, nSize); // Velocity correction
    makeVector(&part->FiST, nSize); // Surface tension force
    makeRealArray(&part->pndL[0], nSize); // Particle Number Density
    makeRealArray(&part->pndS[1], nSize); // Particle Number Density
    makeRealArray(&part->pndL[1], nSize); // Particle Number Density
    makeRealArray(&part->pMin, nSize); // Minimum pressure
    makeRealArray(&part->pPrev, nSize); // Pure pressure in previous step
    makeIntArray(&part->bc, nSize); // Boundary condition

    for (long i = 0; i < nSize; i++)
    {
        part->rn.x[i] = part->r.x[i];
        part->rn.y[i] = part->r.y[i];
        part->rn.z[i] = part->r.z[i];
        part->bc.v[i] = noFreeSurf;
    }
}
```

Código fonte 7-9 Alocação final da estrutura de partículas

7.2.7 Conclusão da inicialização do simulador MPS distribuído

Uma vez conhecidas todas as ‘dimensões’ do subdomínio podem ser criadas instâncias das duas classes restantes que requerem grande quantidade de memória, a vizinhança e o solver, e realizadas as etapas de conclusão da inicialização, como mostrado no Código fonte 7-1:

1. Criação da instância da vizinhança, `Neigh`.
2. Criação da instância do solver, `solver`.
3. Sincronização das bordas. Nesta etapa da inicialização cada subdomínio contém apenas as partículas do seu interior, que foram lidas, já ordenadas,

7.3 IMPLEMENTAÇÃO DO SOLVER DISTRIBUÍDO PARA MATRIZ BLOCO TRIDIAGONAL

A solução do sistema de equações lineares envolve dois passos de características numéricas bem distintas: a solução do sistema associado ao bloco da diagonal que, na versão atual, utiliza o *solver* da biblioteca Intel MKL [82], e a atualização dos elementos associados aos blocos fora da diagonal, que envolve apenas produtos de matrizes por vetores e a sincronização destes elementos entre os subdomínios. Para que a classe *CBlockSolver*, Figura 7-4, fosse independente da forma como as matrizes esparsas são armazenadas e permitir a avaliação de diversas estratégias para cada uma destas tarefas, as operações foram definidas por uma interface básica, independente da representação interna.

Figura 7-4 Hierarquia de classes do *solver*

Os principais atributos da classe *CBlockSolver* são:

- *CBlockSolver()*: inicializa as variáveis do *solver* e cria as instâncias das matrizes utilizadas.
- *band*: largura da banda.
- *A*, *X*, *B*: sistema de equações lineares local, seção 6.3.4.

- AL, AR: submatrizes correspondentes aos blocos inferior (*Left-block*) e superior (*Right-block*) da diagonal, conforme notação da seção 6.3.4.
- XL, XR: vetores correspondentes elementos de fronteira da solução inferior (*Left-block*) e superior (*Right-block*) da diagonal, conforme notação da seção 6.3.4.
- leftBoundary, rightBoundary: apontadores para instâncias da classe CBoundarySolution que armazena os segmentos da solução e do termo constante associados à sobreposição das fronteiras, (seção 5.7.5)
- syncEngine: classe de sincronização utilizada para sincronizar as fronteiras do sistema de equações lineares.

Os principais métodos da classe *CBlockSolver* são:

- Config(): configuração do *solver*.
- calcMatrixCoeff(): construção do sistema de equações lineares local, isto é, a matriz de coeficientes de iteração (A) e o termo constante (B).
- Execute(): solução do sistema de equações lineares.

Na classe CBasicMatrix os elementos da matriz esparsa são armazenados em uma lista de instâncias da classe SBasicSparseMatrixElement e na classe CSparseMatrixCRS os elementos da matriz esparsa são armazenados segundo o padrão CRS (Compressed Row Storage). A inicialização destas estruturas é feita através do *constructor*⁸ de cada classe. Ambas as classes implementam a mesma interface, cujos principais métodos são:

- Add(): insere um novo elemento na matriz esparsa
- Reset(): reinicializa as estruturas de dados
- MultiplyByVector(): multiplica um vetor pela matriz

⁸ Construtor em linguagens de programação orientadas a objeto é um método chamado assim que uma nova instância do objeto é criada. Tal método geralmente é responsável pela alocação de recursos necessários ao funcionamento do objeto além da definição inicial das variáveis de estado (atributos).

Para solução do sistema de equações lineares conforme método de Jacobi para matrizes bloco tridiagonais, conforme apresentado na Seção 5.3.3, foi implementada uma classe, cujo fluxo de dados do método principal está ilustrado na Figura 7-5. Este método consiste de um laço principal onde a condição de parada é controlada pelo subdomínio 0, através da monitoração dos desvios calculados em cada subdomínio.

Figura 7-5 Fluxograma principal do *solver* iterativo

Internamente a este laço cada subdomínio resolve localmente o sistema de equações lineares definido pelo bloco da diagonal. Na versão atual do framework, a solução do sistema de equações lineares é realizada pela biblioteca matemática MKL-Intel [82], que pode ser utilizada tanto no modo serial quanto paralelizada em memória compartilhada.

Após a sincronização dos subdomínios e atualização dos termos do lado direito da equação (5-3) em cada subdomínio, o subdomínio 0 verifica a condição global de parada. Se a condição for satisfeita, o subdomínio informa aos demais subdomínios que o sistema de equações lineares foi resolvido.

7.3.1 Representação das matrizes esparsas

Para representação das matrizes esparsas do sistema de equações lineares local, bloco principal, e das interfaces entre os sistemas, blocos fora da diagonal, foi definida uma interface com uma método para inserção de uma entrada $A_{i,j}$, *Add()*, e um método para multiplicação da matriz por um vetor. Estas operações são

suficientes para a construção do sistema linear local e para as iterações de atualização do método de Jacobi por blocos.

A estrutura de dados que apresentou o melhor desempenho para representação dos blocos de fronteira, implementada na classe `CBasicSparseMatrix`, Código fonte 7-11, apresenta as seguintes características:

1. Cada entrada $A_{i,j}$ da matriz é representada pela estrutura `SBasicSparseMatrixElement` que armazena a posição e o valor da entrada.
2. As entradas da matriz são armazenadas em uma lista de elementos do tipo `SBasicSparseMatrixElement`.
3. Na construção de uma instancia da classe é alocada uma lista com tamanho especificado em `maxNonZeros`.
4. A inserção de uma entrada consiste de armazenar os índices da posição e valor da entrada na posição da lista armazenada no atributo `count` e na atualização deste atributo.
5. A multiplicação da matriz pelo vetor x , simplesmente percorre a lista de entradas da matriz e acumula os valores no vetor b , $b_i = \sum_j A_{i,j}x_j$, independente da ordem em que as entradas estão armazenadas.


```

struct SBasicSparseMatrixElement
{
(1) | long i;
    | long j;
    | double val;
};

class CBasicSparseMatrix
{
protected:
    int count;
    int maxNonZeros;
    SBasicSparseMatrixElement *Elements;

public:
    int n;

    CBasicSparseMatrix(long dimension, long maxNonZeros)
        : count(0),
          n(dimension),
          maxNonZeros(maxNonZeros),
(2) | Elements(new SBasicSparseMatrixElement[maxNonZeros]) {}

    inline bool Add(long i, long j, double val)
    {
        if (count == maxNonZeros)
            return false;
(3) | Elements[count].i = i;
        | Elements[count].j = j;
        | Elements[count].val = val;
        | count++;
        | return true;
    }

    inline void MutiplyVector(double *x, double *b)
    {
(4) | for (int k = 0; k < count; k++)
        | b[Elements[k].i] += Elements[k].val * x[Elements[k].j];
    }

    ~CBasicSparseMatrix() { delete [] Elements; }

```

Código fonte 7-11 Implementação da classe para matriz esparsa básica

A estrutura de dados para representação da matriz bloco principal foi implementada tendo em vista a utilização da biblioteca numérica Intel MKL [82] com as seguintes restrições:

- As entradas da matriz esparsa devem ser armazenadas no formato comprimido por linhas (*Compressed Row Storage* - CRS).
- A numeração dos índices de vetores segue o padrão FORTRAN, isto é, os valores devem ser contados a partir de 1 e não 0 como em C++.
- Para alocação e liberação dos vetores e devem ser usadas as rotinas específicas disponíveis na biblioteca Intel MKL.

As restrições acima foram consideradas no projeto da classe `CBasicSparseMatrix`, apresentada no Código fonte 7-11, cujas principais características são:

1. Declaração dos vetores `IA` e `JA` para armazenamento dos índices de linhas e colunas e do vetor `VA` para armazenamento dos valores das entradas.
2. Devido ao uso da representação FORTRAN, a numeração dos índices começa em 1, assim o vetor `IA` é alocado com uma posição a mais que a dimensão da matriz. Os demais vetores são alocados com base no número máximo de entradas da matriz.
3. A liberação dos vetores alocados é feita através de rotina própria da biblioteca Intel MKL.
4. Considerando-se que a construção da matriz de interação do método MPS é feita segundo a ordem crescente das linhas, optou-se por simplificar a implementação desta classe, impondo a restrição de que as adições de entradas à matriz nesta representação devam ser feitas respeitando a ordem crescente para os índices de linhas, enquanto que as colunas de uma linha adicionadas em qualquer ordem.
5. A inserção de uma entrada consiste de armazenar os índices da posição e valor da entrada na posição da lista armazenada no atributo `count` e na atualização deste atributo.
6. A multiplicação da matriz por um vetor x é feita utilizando-se uma rotina da biblioteca Intel MKL.

```

class CSparseMatrixCRS
{
protected:
    int count;
    int maxNonZeros;
    int curr_row;

public:
(1) | int n;
    | int *IA, *JA;
    | double *VA;

    CSparseMatrixCRS(long dimension, long maxNonZeros)
        : count(0), curr_row(0),
          n(dimension), maxNonZeros(maxNonZeros)
    {
        if (dimension > 0) {
(2) |   IA = (int*) mkl_malloc((n + 1) * sizeof(int), 128);
        |   JA = (int*) mkl_malloc(maxNonZeros * sizeof(int), 128);
        |   VA = (double*) mkl_malloc(maxNonZeros * sizeof(double), 128);
        }
    }

    ~CSparseMatrixCRS()
    {
(3) |   mkl_free(IA);
        |   mkl_free(JA);
        |   mkl_free(VA);
    }

    inline long Count() { return count; }

    inline void Add(long i, long j, double val)
    {
        if (curr_row < i) {
(4) |   for (int k = curr_row+1; k <= i; k++)
            |       IA[k] = count+1;
        }
(5) |   VA[count] = -val;
        |   JA[count] = j+1;    // FORTRAN Indexed array
        |   count++;
        |   IA[i+1]++;
        |   curr_row = i;
    }

    inline void MutiplyVector(double *x, double *b)
    {
(6) |   char transa = 'N';
        |   mkl_dcsrsmv(&transa, &curDim, VA, IA, JA, x, b);
    }
}

```

Código fonte 7-12 Implementação da classe para matriz esparsa comprimida por linhas (CRS)

7.3.2 Classe genérica de sincronização

A sincronização dos elementos de fronteira do sistema linear, assim como a sincronização das partículas das bordas do subdomínio, é realizada por especializações da classe genérica de sincronização, CSyncEngine, Figura 7-6. Esta classe é responsável pela coordenação das etapas de envio e recebimento de um buffer de dados entre subdomínios adjacentes.

Figura 7-6 Diagrama da classe genérica de sincronização

As principais etapas da sincronização, conforme destacado no Código fonte 7-13, são:

1. Realiza a sincronização entre à esquerda, isto é, cada subdomínio envia o buffer de dados ao seu vizinho à esquerda e recebe do seu vizinho à direita.
2. Realiza a sincronização entre à direita, isto é, cada subdomínio envia o buffer de dados ao seu vizinho à direita e recebe do seu vizinho à esquerda.
3. Na construção da instância da classe, são armazenados os números dos subdomínios vizinhos à esquerda e à direita. O primeiro e último subdomínios são considerados vizinhos, fechando o ciclo de comunicação.

```

class CSyncEngine
{
protected:
    int rank, nprocs, left, right, buffer_size;
    void SyncToLeft(void *sendbuf, void *recvbuf)
    {
(1) |     int tag = 1;
        MPI_Status status;
        MPI_Sendrecv(sendbuf, buffer_size, MPI_CHAR, left, tag, recvbuf,
                     buffer_size, MPI_CHAR, right, tag, MPI_COMM_WORLD, &status);
    }
    void SyncToRight(void *sendbuf, void *recvbuf)
    {
(2) |     int tag = 2;
        MPI_Status status;
        MPI_Sendrecv(sendbuf, buffer_size, MPI_CHAR, right, tag, recvbuf,
                     buffer_size, MPI_CHAR, left, tag, MPI_COMM_WORLD, &status);
    }
public:
    CSyncEngine(int rank, int nprocs, int buffer_size):
        rank(rank),
        nprocs(nprocs),
(3) |     left(rank > 0 ? rank - 1 : nprocs-1),
        right(rank+1 < nprocs ? rank + 1 : 0),
        buffer_size(buffer_size)
    {}
}

```

Código fonte 7-13 Implementação da classe genérica de sincronização

Nesta versão da classe de sincronização cada etapa de coordenação da comunicação entre os subdomínios é delegada à função `MPI_Sendrecv()` da biblioteca MPI. As classes que se utilizam desta podem realizar a sincronização de conjuntos específicos de dados sem se aterem às particularidades das operações de comunicação. Assim, é possível avaliar o desempenho de outras opções de comunicação disponíveis na biblioteca MPI, inclusive a comunicação assíncrona, fazendo-se alterações apenas nos métodos desta classe genérica.

7.3.3 Sincronização do solver distribuído

Para atualização do termo constante do método de Jacobi para matrizes em blocos é necessária, em cada estágio da iteração, trocar os elementos de fronteira do vetor solução entre os subdomínios vizinhos. A classe `CSyncSolver`, derivada da classe `CSyncEngine`, é responsável pela sincronização de segmentos contínuos do vetor solução, Figura 7-7.

Figura 7-7 Diagrama da classe de sincronização do solver

Os trechos numerados no Código fonte 7-14 Implementação classe para da sincronização do solver corresponde às seguintes tarefas:

1. Execução do construtor da classe base.
2. Identifica o início da borda interna para envio ao subdomínio à esquerda (primeira posição do interior do subdomínio, `Si`) e realiza a atualização no sentido da direita para a esquerda, os dados são recebidos no vetor `XR`.
3. Identifica o início da borda interna para envio ao subdomínio à direita (a partir do final do subdomínio, `Sf`, retrocede um número de posições correspondente ao tamanho da borda) e realiza a atualização no sentido da direita para a esquerda, os dados são recebidos no vetor `XL`.

```

class CSyncSolver: public CSyncEngine
{
protected:
    long Si, Sf;
    double *X, *XL, *XR;
    double *snd_border;
    int border_size;
public:
    CSyncSolver(int rank, int nprocs, long Si, long Sf, double *X, double *XL,
                double *XR, int border_size):
(1) |    CSyncEngine(rank, nprocs, border_size * sizeof(double)),
        Si(Si), Sf(Sf),
        X(X), XL(XL), XR(XR),
        border_size(border_size)
    {}

    void Synchronize()
    {
(3) |        snd_border = &X[Si];
        SyncToLeft((void*)snd_border, (void*)XR);
(4) |        snd_border = &X[Sf-border_size+1];
        SyncToRight((void*)snd_border, (void*)XL);
    }
}

```

Código fonte 7-14 Implementação classe para da sincronização do solver

7.3.4 Validação do solver distribuído

Para validação do *solver* paralelizado para memória distribuída foi utilizado o modelo dinâmico de *dam break* ilustrado na Figura 7-8. Este modelo, bastante simplificado, foi escolhido porque suas dimensões reduzidas permitiram a visualização detalhada da construção e solução do sistema de equações lineares em cada subdomínio, bem como a recomposição e solução do sistema global no MATLAB [119], para validação preliminar do *solver* distribuído.

Coluna d'água	0,17 x 0,43 m
Comprimento	0,7 m
Número de partículas	1512
Distância entre partículas	10^{-2} m
Intervalo de tempo	2×10^{-4} s

Figura 7-8 Modelo dinâmico de *dam break*

O modelo acima foi simulado usando a decomposição em dois subdomínios. Em cada time step da simulação, foram gravados, os dados dos sistemas lineares locais a cada um dos subdomínios: a submatriz de coeficientes de interação A_n , os blocos associados às regiões de interação entre subdomínios A_n^l e A_n^r , o termo constante B_n e a solução do sistema de equações lineares local X_n , para $n = 1, 2$.

Para estimar o desvio da solução global foi utilizada a função $inv(A)$ do MATLAB. Assim, o desvio pode ser calculado como $\|A^{-1}B - X\|$, onde A é a matriz global de coeficientes, B o termo constante global e X a solução calculada pelo *solver* distribuído. Os valores obtidos para os desvios foram da ordem de 10^{-8} e, portanto, compatíveis com a precisão necessária para simulação do modelo MPS.

7.4 IMPLEMENTAÇÃO DA RENUMERAÇÃO E SINCRONIZAÇÃO DAS PARTÍCULAS EM MEMÓRIA DISTRIBUÍDA

O MPS distribuído utiliza a mesma estrutura de dados utilizada pela versão serial para representação das propriedades das partículas do domínio. Nesta estrutura, listada no Código fonte 7-15, cada propriedade é representada por um vetor de elementos do tipo adequado.

```

struct particles
{
(1) | long *partID;
    intArray index;           // Specifies the material index
    intArray idMat;           // Specifies the material id
    intArray bc;              // Boundary conditions
    realArray p, pMin, pPrev; // Pressure and pressure variation
    realArray Temp;           // Temperature array
    realArray maskFree;       // Free Surface
    realArray pndS[2], pndL[2]; // PND
    vector r0, r, rn, dr;     // Positions: initial, current, prediction
                                // and correction
    vector u, un, du;         // Velocities: current, prediction
                                // and correction
    vector a;                 // Accelation
    vector normal;            // Normal vector
    vector FiST;              // Surface tension force
    bool *isFlow;
    bool *isFluid;
    bool *isSolid;
    bool *isDummy;
    bool *isGhost;
};

struct intArray { int n, nMax, *v; };
struct realArray { int n, nMax; double *v; };
struct vector { int n, nMax; double *x, *y, *z; };

```

Código fonte 7-15 Declaração da estrutura das partículas do domínio

Esta estrutura foi utilizada na implementação original do simulador MPS serial como forma de explorar recursos de processamento vetorial e de *cache* de memória para a execução do simulador. As rotinas de cálculo do MPS fazem uso intensivo desta estrutura, assim, foi necessário manter a mesma representação das partículas na versão distribuída.

Para sincronização das partículas de borá é necessário a troca de apenas um subconjunto das propriedades das partículas entre subdomínios vizinhos. Além disto, esta informação deve estar armazenada em uma área contínua de memória. Como as propriedades de uma única partícula representada na estrutura `particles`, estão armazenadas em áreas descontínuas da memória, foi necessária a definição de classes específicas para representação e sincronização de partículas de borda representadas. A implementação destas classes, conforme mostrado no diagrama Figura 7-9, é o temas das seções subsequentes.

Figura 7-9 Diagrama das classes para sincronização das bordas do subdomínios

7.4.1 Preparação das partículas de fronteira para sincronização

Devido forma como as partículas são representadas na estrutura `particles`, as propriedades de uma dada partícula estão armazenadas em áreas descontínuas da memória. Além disto, apenas algumas das propriedades das partículas de borda são utilizadas nos cálculos realizados para atualizar as propriedades das partículas do subdomínio. Assim, para sincronização das partículas de fronteira é necessário transferir apenas um subconjunto das propriedades das partículas para uma área contínua de memória.

Uma partícula de borda é representada na classe `CBorderParticle` apenas pelas propriedades que necessitam de sincronização, Código fonte 7-16.

```
Class CBorderParticle
{
    long partID;
    long index;           // Specifies the material index
    long idMat;           // Specifies the material id
    long bc;              // Doundary conditions
    double p;             // Pressure
    double Temp;          // Particle Temperature
    CPoint r0, rn;         // Position: initial and prediction
    CPoint un;            // Velocity prediction

    void Pack(particles *part, long i);
    void Unpack(particles *part, long i);
};

struct CPoint { double x, y, z; };
```

Código fonte 7-16 Declaração da classe para partículas de borda

Além disto, a classe `CBorderParticle` implementa dois métodos:

O método `Pack()` copia as propriedades da partícula armazenada na posição `i` para a instância corrente da partícula de borda. A execução deste método resulta no “empacotando” das propriedades da partícula em uma área contínua da memória antes da operação de envio, Código fonte 7-17.

O método `UnpackReceived()` realiza a operação inversa do método anterior: atualiza as propriedades da partícula armazenada na posição `i` com os valores da instância corrente da partícula de borda, Código fonte 7-18.

```

void CBorderParticle::Pack(particles *part, long i)
{
    partID = part->partID[i];
    idMat = part->idMat.v[i];
    p = part->p.v[i];
    Temp = part->Temp.v[i];
    index = part->index.v[i];
    bc = part->bc.v[i];
    rn.x = part->rn.x[i];
    rn.y = part->rn.y[i];
    rn.z = part->rn.z[i];
    un.x = part->un.x[i];
    un.y = part->un.y[i];
    un.z = part->un.z[i];
    r0.x = part->r0.x[i];
    r0.y = part->r0.y[i];
    r0.z = part->r0.z[i];
    isFlow = part->isFlow[i];
    isFluid = part->isFluid[i];
    isSolid = part->isSolid[i];
    isDummy = part->isDummy[i];
    isGhost = part->isGhost[i];
}

```

Código fonte 7-17 Implementação do “empacotamento” de partícula de borda

```

void CBorderParticle::Unpack(particles *part, long i)
{
    part->partID[i] = partID;
    part->idMat.v[i] = idMat;
    part->p.v[i] = p;
    part->Temp.v[i] = Temp;
    part->index.v[i] = index;
    part->bc.v[i] = bc;
    part->rn.x[i] = rn.x;
    part->rn.y[i] = rn.y;
    part->rn.z[i] = rn.z;
    part->un.x[i] = un.x;
    part->un.z[i] = un.z;
    part->un.y[i] = un.y;
    part->r0.x[i] = r0.x;
    part->r0.y[i] = r0.y;
    part->r0.z[i] = r0.z;
    part->isFlow[i] = isFlow;
    part->isFluid[i] = isFluid;
    part->isSolid[i] = isSolid;
    part->isDummy[i] = isDummy;
    part->isGhost[i] = isGhost;
}

```

Código fonte 7-18 Implementação do “desempacotamento” de partícula de borda

7.4.2 Sincronização das partículas de borda

De forma análoga à utilizada para sincronização do solver, 0, Para sincronização das partículas de borda uma classe para sincronização, `CSyncBorders`, foi derivada da classe `CSyncEngine`. Como as partículas de borda não são armazenadas em um área contínua de memória, para implementação desta classe foi necessário incluir as seguintes extensões na classe base, Código fonte 7-19:

1. Inclusão dos seguintes atributos: `*Isort` para referência ao vetor de renumeração; `*part` para referência à estrutura de partículas; `Si` e `Sf` para armazenar os índices da posição inicial e final do subdomínio, respectivamente.
2. Inclusão dos métodos para “empacotamento” e “desempacotamento” da lista e partículas de borda.
3. Cálculo do tamanho do buffer de comunicação considerando o tamanho da estrutura de representação de partículas de borda.
4. Alocação de vetores de partículas de borda para envio e recepção de dados.

```

class CSyncBorders: public CSyncEngine
{
protected:
    int rank, nprocs;
    long border_size;
(1) | long *Isort;
    particles *part;
    long Si, Sf;

    CBorderParticle *snd_border;
    CBorderParticle *rcv_border;

(3) | void PackToSend(long begin);
    void UnpackReceived(long begin);

public:
    CSyncBorders(int rank, int nprocs, long *Isort, particles *part,
        long Si, long Sf, int border_size):
(3) |     CSyncEngine(rank, nprocs, border_size * sizeof(CBorderParticle)),
        Si(Si), Sf(Sf), Isort(Isort), part(part),
        border_size(border_size),

(4) |     snd_border(new CBorderParticle [border_size]),
        rcv_border(new CBorderParticle [border_size])
    {}

    void Synchronize();
}

```

Código fonte 7-19 Declaração da classe de sincronização das partículas de borda

A implementação dos métodos da classe, Código fonte 7-20, apresenta as seguintes características:

1. O método `PackToSend()` aplica a cada partícula de uma borda, definida pelo parâmetro `begin`, o método `Pack()` para empacotamento da partícula.
2. De forma análoga, o método `UnpackReceived()` aplica o método `Unpack()` para desempacotamento das partículas da borda especificada.
3. Para sincronização à esquerda, empacota a borda interna à esquerda, `begin = Si`; sincroniza à esquerda e, exceto para o último subdomínio, desempacota os dados recebidos na borda externa à direita, `begin = Sf+1`.
4. Para sincronização à direita, empacota a borda interna à direita, `begin = Sf-border_size+1`; sincroniza à direita e, exceto para o primeiro subdomínio, desempacota os dados recebidos na borda externa à esquerda, `begin = Bi`.

```

void CSyncBorders::PackToSend(long begin)
{
(1)   for (long is = begin; is < begin+border_size; is++) {
        long i = Isort[is];
        snd_border[is - begin].Pack(part, i);
    }
}

void CSyncBorders::UnpackReceived(long begin)
{
(2)   for (long is = begin; is < begin+border_size; is++) {
        long i = Isort[is];
        rcv_border[is - begin].Unpack(part, i);
    }
}

void CSyncBorders::Synchronize()
{
(3)   PackToSend(Si);
        SyncToLeft((void*)snd_border, (void*)rcv_border);
        if (rank < nprocs-1) UnpackReceived(Sf+1);
(5)   PackToSend(Sf-border_size+1);
        SyncToRight((void*)snd_border, (void*)rcv_border);
        if (rank > 0) UnpackReceived(Bi);
}

```

Código fonte 7-20 Implementação classe de sincronização das partículas de borda

7.4.3 Reordenação das partículas em memória distribuída

Para reordenação das partículas é utilizado método `sort(Isort, v, first, last)` da classe `CSortEngine`, onde `Isort` é o vetor de renumeração, `v` é um vetor com os valores das coordenadas das partículas no eixo de ordenação e `first` e `last` delimitam o intervalo de ordenação das partículas.

As etapas de ordenação das partículas são realizadas sobre dados locais ao subdomínio havendo, portanto, grande liberdade para a escolha ao algoritmo de ordenação, o que permite otimizações e extensões da classe `CSortEngine` sem alterar sua interface externa. Estas etapas, conforme ilustra o Código fonte 7-21, são:

1. Utiliza o método `Sort()` da classe `CSortEngine` para ordenar o interior do subdomínio, partículas de `Si` a `Sf`.
2. Como não existe sobreposição entre os dados das bordas, estas podem ser ordenadas em paralelo, usando a diretiva `section` do OpenMP.
3. Ordena a borda à esquerda para se o subdomínio não for o primeiro.
4. Ordena a borda à direita para se o subdomínio não for o último.

```

void CSubdomain::RenumParticle()
{
(1) | CSortEngine::Sort(Isort, sort_axis, Si, Sf);
    | syncBorder->Sincronize();
(2) | #pragma omp parallel sections
    | {
        | #pragma omp section
        | {
(3) |     if (rank > 0) {
        |         long i_first = Si - border_size;
        |         long i_last = i_first + 2 * border_size - 1;
        |         CSortEngine::Sort(Isort, sort_axis, i_first, i_last);
        |     }
        | }
(2) | #pragma omp section
    | {
(4) |     if (rank < nprocs-1) {
        |         long i_first = Sf - border_size + 1;
        |         long i_last = i_first + 2 * border_size - 1;
        |         CSortEngine::Sort(Isort, sort_axis, i_first, i_last);
        |     }
    | }
}
}

```

Código fonte 7-21 Implementação do método de renumeração das partículas

Até o momento foram realizados testes com algoritmos seriais de ordenação, sendo que o melhor desempenho foi obtido com uma implementação não recursiva do QuickSort [88].

Uma vez que os dados das bordas estão armazenados localmente e que estão disponíveis 8 processadores em cada nó é 8, seria possível obter um uso mais intensivo dos recursos de processamento realizando sequencialmente duas etapas de ordenação paralelizada para memória compartilhada. No entanto, devido ao bom desempenho da solução proposta, Seção 0, e considerando a complexidade relativamente maior dos algoritmos para ordenação paralelizada em memória compartilhada, optou-se postergar a análise destes algoritmos para uma fase futura de otimização ‘fina’ do framework.

7.4.4 Validação da renumeração e sincronização das partículas

Nesta etapa de validação procuramos estudar o comportamento da solução com decomposição de domínio em uma situação em que ocorre uma grande variação na posição das partículas. Este teste foi fundamental para verificar a evolução da estrutura em banda da matriz de iteração e a preservação da distribuição de carga de processamento ao longo da simulação.

Com estes testes, utilizando o modelo de dam break com 1512 partículas, foi possível avaliar qualitativamente o comportamento da implementação completa da decomposição de domínio, isto é, renumeração das partículas, identificação das fronteiras, distribuição de carga de processamento e a solução do sistema de equações lineares, executando totalmente paralelizada em memória distribuída.

Optou-se por este modelo, com número reduzido de partículas, para que fosse possível o acompanhamento da posição das partículas, a evolução da forma matriz de iteração e a distribuição das abscissas das partículas, sendo as duas últimas visualizadas com e sem renumeração das partículas.

Alguns instantes mais significativos destas simulações são ilustrados nas na sequência formada pelas figuras de Figura 7-10 até Figura 7-12. Cada figura é formada por imagens justapostas: os gráficos à esquerda representam a estrutura da matriz dos coeficientes de interação calculada antes e após a renumeração das

partículas - canto superior e canto inferior, respectivamente; no gráfico do canto superior direito estão sobrepostas às abscissas das partículas antes da renumeração e após a renumeração – em azul e vermelho respectivamente; finalmente, o canto inferior direito mostra a configuração das partículas em cada subdomínio, no instante selecionado. Os resultados destes testes foram a primeira indicação da adequação de que a estratégia de decomposição de domínio implementada poderia ser utilizada para simulações de modelos dinâmicos.

Figura 7-10 Validação da renumeração e sincronização, passo da simulação = 1

Figura 7-11 Validação da renumeração e sincronização, passo da simulação = 1000

Figura 7-12 Validação da renumeração e sincronização, passo da simulação = 2000.

7.5 IMPLEMENTAÇÃO DA LISTA DE VIZINHANÇAS

7.5.1 Representação de vizinhanças na versão serial

Na versão serial, para simplificar o acesso direto, a um atributo que depende do raio efetivo, foram criados dois conjuntos de listas de atributos diferenciados pelo sufixo S (Small) ou L (Large). Esta estratégia resultou em duplicação de informação, uma vez que todo elemento de uma lista de sufixo S, também deve estar armazenado na respectiva lista de sufixo L. A forma de construir um laço para realizar o cálculo de uma propriedade das que depende das partículas vizinhas depende dos seguintes passos, Código fonte 7-22:

1. O contador do laço principal percorre lista de partículas: $0 \leq i \leq nPart$.
2. O contador do laço secundário percorre a lista de vizinhanças da partícula i , que tem tamanho $nNeighL.v[i]$.
3. Calcula a posição pa dos atributos nas respectivas listas.

Obtém o índice da partícula vizinha na lista $neighL$, j , e o valor da função peso, w , entre as partículas na lista $weightL$. De forma análoga são usadas as listas de atributos com sufixo S.

```
void calcViscosity()
{
(1) | for(int i = 0; i < sim->nPart; i++)
    {
(2) |     for(int k = 0; k < nNeighL.v[i]; k++)
        {
(3) |         pa = sim->neighMax * i + k;
(4) |         int j = neighL.v[pa];
            double w = weightL.v[pa];
            sumx += w * (part->un.x[j] - part->un.x[i]);
            ...
        }
        ...
    }
}
```

Código fonte 7-22 Exemplo de cálculo usando acesso direto à lista de vizinhanças

7.5.2 Representação de vizinhanças na versão distribuída

Para minimizar o uso de memória, eliminando a duplicação de dados, optou-se por separar a estrutura de armazenamento das listas de atributos do método de acesso aos mesmos. Com isto foi possível desenvolver uma estrutura de dados otimizada, sem dados redundantes, e, ao mesmo tempo, como um método de acesso mais simples aos atributos.

A duas classes desenvolvidas para representação de vizinhanças estão mostradas no diagrama da Figura 7-13. A primeira, CNeighborhood, encapsula a estrutura de dados e provê métodos para alocação de memória e inserção de elementos enquanto a segunda, CNeighborhoodWalker, prove uma forma simplificada de percorrer os elementos da vizinhança e acessar seus atributos de forma transparente (iterador), isto é, sem necessidade de conhecer a forma como a lista vizinhança está estruturada.

Figura 7-13 Diagrama das classes para representação de vizinhanças de partículas

Como cada instância da classe CNeighborhoodWalker administra um apontador para posição corrente durante a navegação da estrutura de vizinhanças. Isto permite a paralelização em memória compartilhada criando uma instanciação da classe CNeighborhoodWalker para cada processo.

Os principais elementos da classe CNeighborhood, Código fonte 7-23, são:

- Neighbors: estrutura de dados com os vizinhos de cada partícula do subdomínio.
- Distances: armazena as distâncias de cada partícula aos seus vizinhos.
- weightsLarge e weightsSmall: armazena o valor da função peso de cada partícula em relação aos seus vizinhos.
- AddL e AddS: insere um novo vizinho.

```
class CNeighborhood
{
    friend class CNeighborhoodWalker;

protected:
    long nPart;
    long maxNeighbors;
    long *Neighbors;
    double *Distances;           // holds ||r[i]-r[j]||
    int *countLarge, *countSmall;
    double reL, reS;
    double *weightsLarge, *weightsSmall;
    long Si;
public:
    CNeighborhood(long *Isort, long nPart, long neighmax, long Si, double reL,
                  double reS)
        : nPart(nPart),
          maxNeighbors(neighmax),
          Si(Si),
          reL(reL),
          reS(reS)
    {}

    void AddL(const long is, const long j, const double r);
    void AddS(const long is, const long j, const double r);

    inline long neighMax()      { return maxNeighbors; }
    inline long countS(long is) { return countSmall[is-Si]; }
    inline long countL(long is) { return (countLarge[is-Si] + countS(is)); }
}
```

Código fonte 7-23 Declaração da interface da estrutura de vizinhanças

Os principais métodos da classe CNeighborhood, Código fonte 7-24 são:

- `getFirstL(is)` e `getFirstS(is)`: define a vizinhança ativa, `is`, seleciona a primeira partícula ativa da vizinhança e inicia o apontador de partícula corrente, `curr`. Retorna o índice partícula corrente.
- `getNextL()` e `getNextS()`: seleciona a próxima partícula da vizinhança, atualiza o apontador de partícula corrente e retorna seu índice.
- `countL()` e `countS()`: número de partículas na vizinhança ativa.
- `r()`: distância da partícula de referência à partícula corrente.
- `wL()` e `wS()`: valor da função peso calculada para a partícula de referência e a partícula corrente.

```
class CNeighborhoodWalker
{
    protected:
(1) | CNeighborhood *Neigh;

    long first, last, curr;
(2) | bool isSmall;
    | long curr_is;
    long GetFirstNotS(const long is);
    long GetNextNotS();

public:
    CNeighborhoodWalker(CNeighborhood *Neigh): Neigh(Neigh) {}

    inline double r() { return Distances[curr]; }
    inline double wL() { return weighstLarge[curr]; }
    inline double wS() { return weightsSmall[curr]; }

    long GetFirstS(const long is);
    long GetNextS();
    long GetFirstL(const long is);
    long GetNextL();
}
```

Código fonte 7-24 Declaração interface do iterador de vizinhanças

Utilizando estas duas classes para representação de vizinhanças, a versão distribuída do exemplo anterior, Código fonte 7-22, pode ser construída como:

1. O contador do laço principal percorre lista de partículas interiores ao subdomínio: $S_i \leq is \leq S_f$.
2. Obtém o número i da partícula associada ao laço principal.
3. Cria uma instância do iterador para percorrer a vizinhança da partícula i .
4. Cada valor atribuído a j no laço secundário, enquanto o iterador `walker` percorre a vizinha, é o número de uma partícula na vizinhança de i .
5. Obtém o valor da função peso entre as partículas, `walker.wL()`. De forma análoga são acessados os atributos com sufixo S .

```
void CSubdomain::calcViscosity()
{
#pragma omp parallel for
(1) | for(long is = Si; is <= Sf; is++) {
(2) |     long i = Isort[is];
(3) |     CNeighborhoodWalker walker(Neigh);
(4) |     for (long j = walker.GetFirstL(is); j != -1; j = walker.GetNextL())
(5) |     {
        double w = walker.wL();
        sumx += w * (part->un.x[j] - part->un.x[i]);
        ...
    }
    ...
}
}
```

Código fonte 7-25 Exemplo de cálculo usando o iterador da lista de vizinhanças

7.5.3 Construção de vizinhanças na versão distribuída

Os métodos para a alocação e reinicialização da estrutura para armazenamento de atributos de vizinhanças estão mostrados no. As principais características destes métodos são:

1. Os vetores que armazenam o tamanho da vizinhança de cada partícula são alocados de acordo com o número máximo de partículas.
2. Para cada atributo de vizinhança é alocado apenas um vetor. Para a vizinhança cada partícula é reservada uma área contínua de memória corresponde ao tamanho máximo permitido para uma vizinhança.
3. Na reinicialização da estrutura de vizinhanças todos os vetores são zerados, com exceção da lista de vizinhos, que deve ser inicializada com -1.

```

inline void CNeighborhood::Alloc(long maxPart)
{
(1) | long size = maxPart;
    | countLarge = new int[size];
    | countSmall = new int[size];

(2) | size *= maxNeighbors;
    | Neighbors = new long[size];
    | Distances = new double[size];
    | weighstLarge = new double[size];
    | weightsSmall = new double[size];
    | this->Reset();
}

inline void CNeighborhood::Reset()
{
(3) | for (long i = 0; i < nPart; i++)
    |     countLarge[i] = countSmall[i] = 0;

    | long nSize = nPart*maxNeighbors;
    | for (long i = 0; i < nSize; i++) {
    |     Neighbors[i] = -1;
    |     Distances[i] = weighstLarge[i] = weightsSmall[i] = 0.0;
    | }
}

```

Código fonte 7-26 Alocação da estrutura de dados de vizinhanças

Uma vez que, na versão distribuída, a vizinhança é construída apenas as partículas do interior do subdomínio, Seção 5.4.1, a indexação das listas desta estrutura é baseada na posição ordinal das partículas, enquanto que para identificação de cada partícula de vizinhança é armazenado o número efetivo desta. Assim, adição da partícula j na vizinhança da partícula na posição ordinal is depende de distância

entre elas ser menor que o raio efetivo pequeno, reS , ou apenas inferior ao raio efetivo grande, reL .

A implementação da estratégia de armazenamento sem redundância de informação foi baseada nos seguintes fatos:

- O tamanho da área reservada deve ser suficiente para armazenar atributos dos pares de partículas com distância inferior ao raio efetivo grande (reL), o que inclui os pares com distância inferior reS .
- A área reservada para armazenar um atributo de vizinhança de uma partícula i_s corresponde às posições de $(i_s - S_i)$ à $(i_s - S_i + 1) * \text{maxNeighbors} - 1$ no respectivo vetor deste atributo.

Assim, conforme ilustrado no Código fonte 7-27, os passos necessários para adicionar partícula j na vizinhança da partícula na posição ordinal i_s são:

1. No caso em que a distância entre o par de partículas é inferior reL , mas maior que reS , calcula a posição para adição para armazenamento do final para o início da área reservada, ***vizinhança Large***.
2. Armazena os atributos independentes do raio efetivo e aqueles dependentes do raio efetivo grande, reL .
3. Atualiza o contador da *vizinhança Large*.
4. No caso em que a distância entre o par de partículas é inferior reS , calcula a posição para adição para armazenamento do início para o final da área reservada, ***vizinhança Small***.
5. Armazena os atributos independentes do raio efetivo.
6. Armazena os atributos que dependem de raio efetivo, tanto para reS quanto para reL .
7. Atualiza apenas o contador da vizinhança *Small*.

```

inline void CNeighborhood::AddL(long is, long j, double r)
{
(1) | long k = is - Si;
    | long next = (k + 1) * maxNeighbors - countLarge[k] - 1;
(2) | Neighbors[next] = j;
    | Distances[next] = r;
    | weighstLarge[next] = weight(r, reL);
(3) | countLarge[k]++;
    }

inline void CNeighborhood::AddS(long is, long j, double r)
{
(4) | long k = is - Si;
    | long next = k * maxNeighbors + countSmall[k];
(5) | Neighbors[next] = j;
    | Distances[next] = r;
(6) | weightsSmall[next] = weight(r, reS);
    | weighstLarge[next] = weight(r, reL);
(7) | countSmall[k]++;
    }

```

Código fonte 7-27 Adição de itens na estrutura de vizinhanças

7.5.4 Acesso à vizinhança das partículas na versão distribuída

Para percorrer uma vizinhança associada ao raio efetivo pequeno, re_s , devem ser utilizados os métodos de iteração com prefixo S , mostrados no Código fonte 7-28. Neste caso, a vizinhança é percorrida a partir da posição inicial de vizinhança, segundo as seguintes etapas:

1. Sinaliza iteração pela *vizinhança Small* e armazena o identificador ordinal da vizinhança ativa.
2. Calcula o início e o final da vizinhança ativa.
3. Inicializa a posição corrente na vizinhança ativa com um valor anterior à primeira posição da vizinhança e retorna o valor calculado por `GetNextS()`.
4. Se a posição corrente da célula ativa é menor que a última posição, incrementa a posição corrente e retorna o índice da partícula vizinha, senão retorna -1.

```

inline long CNeighborhoodWalker::GetFirstS(const long is)
{
(1) |   isSmall = true;
    |   curr_is = is;
(2) |   long k = is - Neigh->Si;
    |   first = k * Neigh->maxNeighbors;
    |   last = first + Neigh->countSmall[k]-1;
(3) |   curr = first-1;
    |   return GetNextS();
    | }

inline long CNeighborhoodWalker::GetNextS()
{
(4) |   if (curr < last) {
    |       curr++;
    |       return Neigh->Neighbors[curr];
    |   }
    |   else
    |       return -1;
    | }

```

Código fonte 7-28 Iteração de vizinhanças *Small*

Para percorrer uma *vizinhança Large*, devem ser utilizados os métodos de iteração com prefixo L, apresentados no Código fonte 7-29. Neste caso, o algoritmo percorre, inicialmente, a *vizinhança Small* e, ao final desta, percorre a *vizinhança Large* de acordo com os seguintes passos:

1. Utiliza os mesmos passos do método `GetFirstS()` para Iniciar a iteração pela *vizinhança Small*.
2. Se a *vizinhança Small* está vazia, inicia a iteração pela *vizinhança Large*, como será mostrado no próximo parágrafo.
3. Se a posição corrente da célula ativa é menor que a última posição, incrementa a posição corrente e retorna o índice da partícula vizinha;
4. Se a vizinhança ativa era *Small*, inicia a iteração da *vizinhança Large*, senão retorna -1.

```

inline long CNeighborhoodWalker::GetFirstL(const long is)
{
(1) | isSmall = true;
    | curr_is = is;
    | long k = is-Neigh->Kf;
    | rc;
    | first = k * Neigh->maxNeighbors;
    | last = first + Neigh->countSmall[k]-1;
    | curr = first-1;
(2) | long rc = GetNextS();
    |
    | if (rc < 0)
    |     return GetFirstNotS(is);
    | else
    |     return rc;
}

inline long CNeighborhoodWalker::GetNextL()
{
    if (curr < last)
    {
(3) |     curr++;
    |     return Neigh->Neighbors[curr];
    | }
    else if (isSmall)
(4) |     return GetFirstNotS(curr_is);
    | else
    |     return -1;
}

```

Código fonte 7-29 Iteração de *vizinhança Large*

Os dois métodos apresentados no Código fonte 7-1, de uso exclusivo da classe `CNeighborhoodWalker`, permitem a que troca da iteração da *vizinhança Small* pela iteração na *vizinhança Large* seja feita de forma transparente ao usuário. Os principais passos da estratégia de troca são:

1. Sinaliza o final da iteração pela na *vizinhança Small*.
2. Calcula o início e o término da *vizinhança Large*.
3. Inicializa a posição corrente na vizinhança ativa com um valor anterior à primeira posição da *vizinhança Large* e retorna o valor calculado por `GetNextNotS()`.
4. Se a posição corrente da célula ativa é menor que a última posição, incrementa a posição corrente e retorna o índice da partícula vizinha, senão retorna -1.

```

inline long CNeighborhoodWalker::GetFirstNotS(const long is)
{
(1) | isSmall = false;
    | curr_is = is;
    | long k = is - Neigh->Si;
(2) | last = (k + 1) * Neigh->maxNeighbors - 1;;
    | first = last - Neigh->countLarge[k] + 1;
    | curr = first - 1;
(3) | return GetNextNotS();
    | }

inline long CNeighborhoodWalker::GetNextNotS()
{
(4) | if curr < last) {
    |     curr++;
    |     return Neigh->Neighbors[curr];
    | }
    | else
    |     return -1;
    | }

```

Código fonte 7-30 Implementação da troca ‘transparente’ de vizinhanças

7.6 IMPLEMENTAÇÃO DA PARALELIZAÇÃO DAS OPERAÇÕES DE ENTRADA E SAÍDA

7.6.1 Paralelização da leitura dos dados do modelo

Para leitura do arquivo de configuração do modelo foi criada uma função de serialização. Esta função implementa um semáforo simples para garantir que apenas um subdomínio de cada vez tenha acesso a um recurso. O recurso em questão é determinado pela que função passada como parâmetro à rotina de serialização. Deste modo a sub-rotina de leitura da configuração do modelo pôde ser reaproveitada da versão serial sem qualquer alteração

Para leitura em paralelo do arquivo de partículas, foi necessário estabelecer um layout de arquivo com todos os registros de mesmo tamanho, como ilustrado na Figura 7-14. As linhas têm tamanho fixo (texto ou binário) e devem estar ordenadas por um dos eixos de coordenadas. Na primeira linha o número 1 deve ser colocado nas colunas referentes ao selecionado para ordenação; na segunda linha devem ser informados o número de partículas e os limites do domínio; cada uma das demais linhas contém as propriedades iniciais de uma partícula:

- Posição inicial: (r_x , r_y , r_z).
- Velocidade inicial: (u_x , u_y , u_z).
- Valor inicial da pressão: p .
- Densidade do número de partículas de referência: $pnd0$
- Valor inicial da temperatura: $Temp$

ord-x	ord-y	ord-z							
1	0	0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
n.part.	xmin	xmax	ymin	ymax	zmin	zmax			
1512	-0.0300	0.7200	-0.4600	0.0300	0.0000	0.0000	0.0000	0.0000	0.0000
idmat	rx	ry	rz	ux	uy	uz	p	pnd0	Temp
3	-0.0300	-0.4500	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
3	-0.0300	-0.4400	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
3	-0.0300	-0.4300	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
3	-0.0300	-0.4000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
3	-0.0300	-0.3900	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
...									

Figura 7-14 Layout do arquivo de partículas para leitura em paralelo

Para implementação da leitura paralelizada foi utilizada a biblioteca MPI-IP. O uso desta biblioteca é análogo ao MPI: `write|send` e `read|receive` mensagens e todas as funcionalidades disponíveis através de chamadas de função: `MPI_File_{open, close, read, write, seek}`, como mostrado no Código fonte 7-31:

```

void CSubdomain::ParallelReadGridFile()
{
(1) | size_t view_size = sim->nPart * gridLineSize;
    | size_t view_offset = (Si + 2) * file_lineSize;

(2) | char *buffer = (char*) malloc(view_size * sizeof(char));

    MPI_File file;
    MPI_Status status;
    char datarep[] = "native";

(3) | MPI_File_open(MPI_COMM_WORLD, grid_file, MPI_MODE_RDONLY, MPI_INFO_NULL, &file);
    | MPI_File_set_view(file, view_offset, MPI_CHAR, MPI_CHAR, datarep, MPI_INFO_NULL);
    | MPI_File_read(file, buffer, view_size, MPI_CHAR, &status);

    std::istringstream input(buffer, std::istringstream::in);
    for (size_t i = 0; i < sim->nPart; i++)
    {
(4) |     input >> part->idMat.v[i]
        |         >> part->r.x[i] >> part->r.y[i] >> part->r.z[i]
        |         >> part->u.x[i] >> part->u.y[i] >> part->u.z[i]
        |         >> part->p.v[i] >> part->pndS[0].v[i] >> part->Temp.v[i];
    }
(5) | MPI_File_close(&file);
    | free(buffer);
}

```

Código fonte 7-31 Leitura paralelizada das condições iniciais das partículas

1. Calcula o tamanho `view_size` e o início `view_offset` do segmento do arquivo de partículas correspondente ao subdomínio corrente. No cálculo do início do

segmento é necessário inclui as duas linhas as duas linhas iniciais de cabeçalho do arquivo de partículas.

2. Aloca o buffer de memória temporário.
3. Utiliza os parâmetros calculados para configurar a biblioteca MPI-IO e realiza a leitura do segmento definido no buffer de memória.
4. Transfere cada linha do buffer para uma partícula, utilizando um `std::istringstream` para conversão de texto para o tipo específico dos parâmetros da partícula.
5. Fecha os arquivos e libera a memória utilizada pelo buffer.

7.6.2 Paralelização da gravação dos dados da simulação

A biblioteca VTK disponibiliza classes para gravação dos dados, que encapsulam totalmente as tarefas de comunicação e sincronização necessárias à paralelização. Deste modo, a implementação do método da classe subdomínio, utilizada para geração dos dados de pós-processamento, ficou bastante simplificada, como é mostrados nos fragmentos de códigos fonte Código fonte 7-32 e Código fonte 7-33. Para maior concisão da apresentação deste método, foram selecionados entre os parâmetros gravados no arquivo de instantâneos da simulação, apenas um exemplo de cada tipo de dado.

```
(1) | #include "vtkXMLPUnstructuredGridWriter.h"
    | #include "vtkUnstructuredGrid.h"
    | #include "vtkPointData.h"
    | #include "vtkFloatArray.h"
    | #include "vtkIntArray.h"
    |
    | void CSubdomain::vtkWrite()
    | {
(2) |     std::string fname = this->filename + num2str(int(sim->t/sim->dt))+".pvtu";
(3) |     vtkPoints *points = vtkPoints::New();
    |     points->Allocate(nPart);
(4) |     vtkIntArray *material=vtkIntArray::New();
    |     material->SetNumberOfComponents(1);
    |     material->SetName("Material");
(5) |     vtkFloatArray *pressure = vtkFloatArray::New();
    |     pressure->SetNumberOfComponents(1);
    |     pressure->SetName("Pressure");
(6) |     vtkFloatArray *velocity = vtkFloatArray::New();
    |     velocity->SetNumberOfComponents(3);
    |     velocity->SetName("Velocity");
(7) |     for(int is = Si; is <= Sf; is++) {
    |         long i = Isort[is];
    |
    |         points->InsertNextPoint(part->rn.x[i], part->rn.y[i], part->rn.z[i]);
    |         material->InsertNextValue(part->idMat.v[i]);
    |         pressure->InsertNextValue(part->p.v[i]);
    |         velocity->InsertNextTuple3(part->u.x[i], part->u.y[i], part->u.z[i]);
    |     }
    |
    |     ...
```

Código fonte 7-32 Preparação dos dados para gravação paralelizada usando a biblioteca VTK

O Código fonte 7-32 implementa a preparação dos dados para gravação paralelizada, as principais etapas desta tarefa são:

1. Inclui os arquivos de declaração das classes da biblioteca VTK necessárias à gravação paralelizada dos dados.

2. Inclui o número do passo da simulação no sufixo do nome do arquivo
3. Declara e aloca a estrutura para armazenar as posições das partículas.
4. Declara uma matriz unidimensional de números inteiros para o tipo de material.
5. Declara uma matriz unidimensional de números reais para a pressão.
6. Declara uma matriz tridimensional de números reais para o vetor velocidade.
7. Para cada partícula do subdomínio, insere o valor do parâmetro na matriz correspondente.

O Código fonte 7-33 implementa a gravação paralelizada dos dados da simulação, as principais etapas desta tarefa são:

1. Declara a estrutura de grid usada pela biblioteca para consolidação dos dados antes da gravação.
2. Insere na grid um apontador para a estrutura de pontos previamente preenchida com a posição das partículas.
3. Define uma ordenação para das propriedades de partículas e insere na grid os respectivos apontadores.
4. Configura o arquivo de saída, com base no número de processos, `nproc`, e no rank do processo corrente. O número do processo é incluído no sufixo do nome do arquivo automaticamente. Neste caso foi selecionada a opção de geração do arquivo no modo binário.
5. Grava um arquivo para cada processo e simultaneamente, insere a descrição do arquivo no respectivo índice.
6. Libera a memória utilizada neste processo.

```

    ...
(1) | vtkUnstructuredGrid *grid=vtkUnstructuredGrid::New();
(2) | grid->SetPoints(points);
(3) | grid->GetPointData()->SetActiveScalars("0");
    | grid->GetPointData()->SetScalars(material);
    | grid->GetPointData()->SetActiveScalars("1");
    | grid->GetPointData()->SetScalars(pressure);
    | grid->GetPointData()->SetActiveScalars("2");
    | grid->GetPointData()->SetScalars(velocity);
    | vtkXMLPUnstructuredGridWriter *writer = vtkXMLPUnstructuredGridWriter::New();

(4) | writer->SetFileName(fname.c_str());
    | writer->SetNumberOfPieces(nprocs);
    | writer->SetStartPiece(rank);
    | writer->SetEndPiece(rank);
    | writer->SetInput(grid);
    | writer->SetDataModeToBinary();
(5) | writer->Write();

(6) | points->Delete();
    | material->Delete();
    | pressure->Delete();
    | velocity->Delete();
    | grid->Delete();
    | writer->Delete();
}

```

Código fonte 7-33 Gravação paralelizada dos dados da simulação usando a biblioteca VTK

7.6.3 Validação

A primeira etapa para validação da leitura dos dados das partículas foi verificar que os dados estavam sendo lidos corretamente, comparando os dados da leitura paralelizada e os da leitura serial utilizando-se, para isto, os recursos de trace disponíveis. A segunda tarefa de validação visava verificar o desempenho da leitura paralelizada através da biblioteca MPI-IO. Os resultados destes testes estão sumarizados no gráfico da Figura 7-15.

Figura 7-15 Tempo de leitura paralelizada de arquivo com 22 milhões de partículas

Estes resultados mostram que estratégia adotada tem boa escalabilidade e, além disto, a comparação dos tempos de leitura paralelizada com o tempo de leitura serial para um milhão de partículas (1ª coluna nos gráfico) sugere que estratégia adotada deve apresentar desempenho superior ao da versão serial para o mesmo número de partículas.

A adequação do desempenho da gravação paralelizada foi contatada comparando-se o tempo gasto nesta tarefa em simulações do modelo de 22 milhões de partículas utilizando subdomínios. O tempo médio para gravação dos dados, nestas simulações, foi em geral inferior a 2,0% do tempo de iteração e inferior a 0,05% do tempo total da simulação. Considerando que este último valor depende do intervalo especificado para gravação dos dados, que usualmente fica entre 50 e 200 vezes o passo de iteração, considerou-se que a opção paralelizada na biblioteca VTK totalmente satisfatória.

8 RESULTADOS

8.1 AVALIAÇÃO DO USO DE MEMÓRIA PELAS ESTRUTURAS DE DADOS DISTRIBUÍDAS

Foram feitas estimativas da memória alocada pelas estruturas de dados distribuídas para partículas, *particles*, e para vizinhanças *Neighbors* em simulações com 5 milhões e 22 milhões de partículas com diversas configurações de divisão do domínio. Os resultados estão sumarizados nos gráficos abaixo, Figura 8-1 e Figura 8-2 respectivamente. O item memória total inclui, além das estruturas acima, a memória utilizada pelo *solver*, pela lista de materiais e algumas estruturas menores utilizadas pelo framework.

A abscissa inicial de cada gráfico, 2 no primeiro e 8 no segundo, corresponde ao menor número de subdomínios para o qual a memória necessária pode ser alocada em um nó.

Figura 8-1 Utilização de memória em função do número de subdomínios para modelo com 5 milhões de partículas.

Figura 8-2 Utilização de memória em função do número de subdomínios para modelo com 22 milhões de partículas.

O gráfico da Figura 8-1 mostra que consumo total de memória, para o modelo de 5 milhões de partículas, decresce quase linearmente com o aumento do número de subdomínios. No entanto, para o modelo de 22 milhões de partículas, o consumo total de memória se afasta significativamente da taxa linear de decaimento. Uma análise mais atenta da Figura 8-2 revela que este comportamento se deve, principalmente, à taxa de decréscimo do uso de memória a estrutura de células.

A estrutura de células ainda usa a mesma representação da versão serial e, portanto, não se beneficia integralmente da divisão de domínio. Assim, o decréscimo observado no uso de memória se deve ao fato de que, embora o número de células alocadas seja fixo, com o aumento do número de subdomínios aumenta o percentual de células vazias. Este efeito fica mais evidente nos gráfico da Figura 8-3 que, à esquerda mostra a variação na taxa do uso de memória, desconsiderando a memória usada pela estrutura vizinhanças e, à direita, o gráfico análogo, quando não é considerada a memória utilizada pela estrutura de células.

Figura 8-3 Comparação da influência uso de memória nas estruturas de células e vizinhanças sobre a taxa de variação do uso utilização de memória total em função do número de subdomínios, para modelo com 22 milhões de partículas.

Embora esta limitação da estrutura de células não tenha impacto significativo para as simulações com modelos de dezenas de milhões de partículas, estima-se que poderá ser um fator limitante para modelos acima dos 100 milhões de partículas. Tendo em vistas necessidades futura de simulações com modelos desta ordem de grandeza, já temos previsto o desenvolvimento de uma versão totalmente distribuída para a representação de células.

8.2 VALIDAÇÃO COM MODELO DINÂMICO 2D

Nas etapas iniciais do desenvolvimento foi utilizado para validação da versão paralelizada o modelo de dam break com 1512 partículas, apresentado na Seção 0, pois o seu tamanho reduzido era ideal para utilização das ferramentas de *trace* desenvolvidas para os testes. Estas ferramentas permitiam a geração de arquivos para qualquer dos vetores de dados utilizados pelo programa, que podiam, então, ser visualizados em forma gráfica utilizando o MATLAB.

Para validação da simulação a posição das partículas em um dado instante obtida com a versão paralelizada do simulador era sobreposta à posição das partículas obtida, nas mesmas condições, pelo simulador serial. A Figura 8-4 ilustra um exemplo deste procedimento. Nesta figura, os círculos em vermelho representam a posição de uma partícula calculada pela versão serial e os pontos em azul representam a posição da mesma partícula calculada pela versão distribuída.

Figura 8-4 Sobreposição das partículas de simulações paralela e serial

Com esta técnica foi possível reduzir significativamente o tempo para validar o framework nas etapas iniciais do desenvolvimento, inclusive com a detecção de problemas que só apareciam em simulações de longa duração.

Para as etapas finais do desenvolvimento era necessário, contudo, avaliar o correto funcionamento da implementação em condições mais abrangentes. Deste modo, procurou-se estudar modelos dinâmicos, com tamanhos que permitissem variar amplamente o número de subdomínios. Estes testes foram fundamentais para verificar a evolução das regiões de fronteira e a preservação da distribuição de carga de processamento ao longo da simulação.

A qualidade dos resultados obtidos pelo modelo MPS deveria ser preservada, uma vez que o framework não interfere na formulação matemática do modelo, deste modo, como base de comparação para validação da versão paralelizada os resultados obtidos com esta versão foram comparados com os da versão serial nas mesmas condições. Nas simulações com modelos 2D foi possível construir a sobreposição dos resultados, de forma análoga ao mostrado na Figura 8-4, utilizando o aplicativo de visualização Paraview [120]. Para os modelos 3D, no entanto, esta técnica não se mostrou viável e os resultados foram comparados por justaposição das imagens obtidas em ambas as versões.

Embora possa parecer que a estratégia a comparação visual distribuição dos valores parâmetros significativos da simulação (pressão, velocidade, etc.) tenha alcance limitado, devido às restrições de resolução, para representação espacial dos pontos, e escala de cores, para diferenciação dos valores escalares, a importância desta estratégia pode ser mais bem avaliada recuperando-se algumas imagens obtidas nas etapas iniciais da validação do framework, quando havia ainda diferenças significativas entre os resultados das simulações em paralelo.

A Figura 8-5 mostra o instante inicial de simulações do modelo de dam break com 1512 partículas, utilizando ambas as versões do simulador. As partículas da simulação distribuídas foram representadas por discos coloridas segundo o número do subdomínio a que pertencem, (0 = azul e 1 = vermelho) e sobrepostas às partículas da simulação serial, representadas por discos brandos com raio ligeiramente maior. Esta configuração permitia detectar visualmente desvio entre os o movimento das partículas em cada simulador.

Figura 8-5 Posição das partículas, calculadas em simulações serial e paralela do modelo de dam break 2D, sobrepostas da para inspeção visual

A Figura 8-6 mostra dois fragmentos de instantâneos de instantes posteriores da simulação. Com poucos passos de simulação já era visível que as partículas na simulação distribuída estão se movendo mais rápido que na versão serial, região circundada em verde na imagem à esquerda. Com o avanço da simulação começavam a surgir indícios de que havia, também, problemas na região de fronteira, região em destaque na imagem à direita.

Figura 8-6 Exemplo de diferenças nas velocidades das partículas entre simulações serial e paralela, com modelo de dam break 2D, identificada por inspeção visual

Com esta mesma técnica, pode-se confirmar que havia problemas na região de simulando um modelo hidrostático, tanque totalmente preenchido. Na imagem à esquerda na Figura 8-7, era perceptível, em poucos passos de simulação, a formação de faixas verticais na distribuição da pressão nas partículas nas regiões de interface entre os subdomínios, já nos instantes iniciais da simulação. Na imagem à direita, que mostra um instante posterior da simulação, estas faixas são claramente visíveis.

Figura 8-7 Exemplo de problema na fronteira de subdomínios em simulação paralela com quatro subdomínios, com modelo de tanque hidrostático, identificado por inspeção visual

As seções a seguir apresentam os resultados dos testes de validação com modelos 2D e 3D de variados tamanhos. Para modelos cujo tamanho permitia a simulação em modo serial são apresentadas as imagens obtidas para parâmetros significativos da simulação (pressão, velocidade, etc.) para ambas as versões, correspondentes a determinados instante da simulação. Modelos acima da capacidade de simulação da versão serial foram simulados apenas na versão paralela.

8.2.1 Modelo de dam break 2D com 65K partículas

Na validação do modelo de dam break 2D, Figura 8-8, foi possível comparar os resultados pela sobreposição da posição das partículas, primeira sequência de imagens, e também pela justaposição das imagens de parâmetros significativos da simulação: pressão, PND e velocidade, ilustrados na segunda sequência de imagens.

Número de partículas	65.418
Distância entre partículas	10^{-2} m
Intervalo de tempo	5×10^{-4} s
Tempo de simulação	7 s
Coluna d'água (x,y)	3,0 x 2,0 m
Comprimento do tanque	10 m
Número de subdomínios	4

Figura 8-8 Modelo dinâmico com 65K partículas, dam break 2D

8.2.2 Validação da posição das partículas

Cada imagem da sequência abaixo, Figura 8-9 até Figura 8-12, foi formada pela justaposição de três imagens: a imagem menor à esquerda, obtida com a versão paralelizada do simulador, mostra a posição das partículas coloridas segundo o subdomínio a que pertencem, enquanto a imagem menor à direita, obtida com versão serial do simulador, mostra a posição das partículas coloridas segundo o tipo de material. A parte superior mostra a sobreposição destas duas imagens. A coincidência entre os valores de posição se manteve dentro da precisão esperada, apenas nos instantes finais da simulação, quando há um forte espalhamento das partículas, os valores se distanciam de forma significativa.

Figura 8-9 MPS paralelizado (num. subdomínio) x MPS serial (material), $t = 0,4$ s.

Figura 8-10 MPS paralelizado (num. subdomínio) x MPS serial (material), $t = 0,8$ s.

Figura 8-11 MPS paralelizado (num. subdomínio) x MPS serial (material), $t = 1,6$ s.

Figura 8-12 MPS paralelizado (num. subdomínio) x MPS serial (material), $t = 2,80$ s.

8.2.3 Validação da pressão

A imagem da sequencia abaixo, Figura 8-13, mostra a posição das partículas coloridas segundo o respectivo valor de pressão calculado pela versão serial simulador (imagem superior) e pela versão paralelizada do simulador (imagem inferior). A semelhança entre padrões de cores, que representam a pressão calculada em ambos os casos, indicam que os valores se mantiveram dentro da precisão esperada.

Figura 8-13 Velocidade: MPS serial (superior) x MPS paralelizado (inferior)

8.2.4 Validação da velocidade

Cada imagem da sequencia abaixo, Figura 8-14, mostra a posição das partículas coloridas segundo o respectivo valor de velocidade calculado pela versão serial do simulador (imagem superior) e pela versão paralelizada do simulador (imagem inferior). A semelhança entre padrões de cores, que representam a velocidade calculada em ambos os casos, indicam que os valores se mantiveram dentro da precisão esperada.

Figura 8-14 Velocidade: MPS serial (superior) x MPS paralelizado (inferior)

8.3 VALIDAÇÃO COM MODELO DINÂMICO 3D

Para validação do simulador MPS paralelizado com modelos 3D foi utilizado o modelo de *dam break*, Figura 8-15 e Figura 8-16, seguindo a especificação apresentada por Issa e Violeau [26] para validação de códigos de Dinâmica de Fluidos Computacional.

Este modelo foi escolhido por apresentar um comportamento altamente dinâmico, importante para verificar o comportamento da evolução da geometria dos subdomínios.

Figura 8-15 Dam break 3D para validação, imagem extraída de Issa e Violeau [26]

Figura 8-16 Especificação dos Sensores de pressão e de nível do fluido, imagem extraída de por Issa e Violeau [26]

As resoluções de discretização deste modelo, utilizadas nos estudos de convergência e desempenho estão resumidas nas Tabela 1 e Tabela 2 abaixo. Para maior concisão do texto deste trabalho, cada uma destas discretizações será referenciada pelos identificadores do tamanho do modelo, conforme definidos na primeira linha da Tabela 2.

Coluna d'água (x,y,z)	1,228 x 1,00 x 0,55 m
Comprimento do tanque	3,22 m
Tempo de simulação	6 s

Tabela 1 Geometria do modelo de Dam break 3D

Identificação do modelo	200K	1M	7M	32M
Número de partículas	199.845	1.135.005	7.221.306	29.997.789
Distância entre partículas	20 mm	10 mm	5 mm	3 mm
Time step (s)	0,002 s	0,001 s	0.0005	0.0003

Tabela 2 Discretizações do modelo de dam break 3D.

8.3.1 Dam break 3D com um milhão de partículas

Para validação do simulador paralelizado com modelos 3D, foi utilizado inicialmente o modelo de *dam break* discretizado com um milhão de partículas (1M). Esta discretização foi escolhida por permitir a simulação tanto na versão serial quanto na distribuída. No entanto, devido à natureza 3D dos resultados a comparação não pode ser feita por sobreposição, como no caso dos modelos 2D. Assim, foi utilizado um recurso do Paraview que permite justapor e sincronizar exibição das imagens, correspondentes a uma dada propriedade das partículas, de instantes de correspondentes da simulação.

Nesta etapa de validação foi realizada a comparação visual do comportamento dinâmico da divisão de domínios, para a versão paralelizada, e na e dos valores de pressão e velocidade calculado pelas duas versões.

8.3.1.1 Validação da estratégia de divisão de domínio

Para simulação do modelo 1M, foi utilizada divisão do domínio em 8 subdomínios. A Figura 8-17 mostra a configuração dos subdomínios no instante inicial da simulação. Os subdomínios são numerados de 0 a 7 da esquerda para a direita e coloridos conforme a escala no canto superior direito desta figura.

Figura 8-17 Configuração inicial da divisão do domínio no instante inicial da simulação

Como não existem diferenças no formato da caixa nem no formato da coluna d'água no segmento do domínio que contém os subdomínios de 2 a 6, estes subdomínios tem exatamente a mesma largura na direção do eixo x, eixo de ordenação. O subdomínio 7 é ligeiramente mais estreito por conter as partículas da parede direita do tanque. O subdomínio 0 por não conter partículas da coluna d'água, e

subdomínio 1, por conter uma parte insignificante destas, são visivelmente mais largos que os demais. Finalmente, a largura menor do subdomínio 0, em relação ao subdomínio 1, se deve às partículas que representam o obstáculo e às relativas à parede esquerda do tanque.

Como o avanço das partículas em direção aos subdomínios vazios, Figura 8-18, a largura dos subdomínios à esquerda diminui devido ao ingresso destas partículas, enquanto aumenta para os subdomínios à direita, pela razão oposta.

Figura 8-18 Evolução da configuração espacial dos subdomínios, $t = 0,4$ s

Devido à variação da largura dos subdomínios na direção do eixo de ordenação, os subdomínios podem ter, eventualmente, larguras equivalentes, Figura 8-19.

Figura 8-19 Evolução da configuração espacial dos subdomínios, $t = 3,0$ s

Esta variação na geometria dos subdomínios, definidos pelos limites ordinais fixos, S_i^n , S_f^n , Seção 5.1, confirma a adequação da estratégia proposta para também modelos dinâmicos 3D.

8.3.1.2 Verificação da distribuição de pressão e velocidade

Para comparação visual da pressão nas partículas, calculada pelo simulador distribuído e pela versão serial, a distribuição da pressão registrada em instantes selecionados em cada simulação foram justapostos, como ilustra a Figura 8-20.

Figura 8-20 Instantâneos da distribuição de pressão calculada pelo simulador serial (1.a e 1.b) e pelo simulador paralelizado (2.a e 2.b).

Nas imagens da Figura 8-21, a imagem 1.a mostra a pressão calculada pelo simulador MPS serial no instante 0,4s, enquanto a imagem 1.b mostra a pressão calculada pelo MPS paralelizado no mesmo instante. Analogamente, imagens 2.a e 2.b mostram os valores de pressão para o instante 1.0 s.

O mesmo procedimento foi utilizado para comparação visual da distribuição de velocidades, como ilustra a Figura 8-21.

Figura 8-21 Instantâneos da distribuição de velocidades calculada pelo simulador serial (1.a e 1.b) e pelo simulador paralelizado (2.a e 2.b).

A conclusão destes testes é que não há diferença significativa entre as duas versões do simulador, embora, um exame mais minucioso da distribuição da pressão junto á parede esquerda do tanque, Figura 8-20 Instantâneos da distribuição de pressão calculada pelo simulador serial (1.a e 1.b) e pelo simulador paralelizado (2.a e 2.b). , sugere a existência de uma ligeira diferença entre distribuição da pressão nas partículas calculada por cada uma das versões. Neste caso dever ser lembrado que, embora a precisão exigida na solução do sistema linear seja a mesma para as duas versões do simulador, nem uma hipótese é feita sobre os desvios dos valores individuais da solução. Assim, enquanto o desvio máximo em cada solução é limitado pelo mesmo parâmetro, a distribuição dos desvios individuais é característica de cada algoritmo de solução. Como a solução do sistema linear é utilizada para correção da velocidade, a cada passo de simulação são introduzidas pequenas diferenças que, após um grande número de iterações podem resultar nas sutis em diferenças perceptíveis na Figura 8-20 Instantâneos da distribuição de pressão calculada pelo simulador serial (1.a e 1.b) e pelo simulador paralelizado (2.a e 2.b). .

Em resumo, estes resultados confirmaram que a duas versões do simulador são equivalentes, e que a divisão do domínio, necessária à paralelização, não acarreta em qualquer efeito significativo nas simulações. Com isto foi concluída a etapa de comparação visual entre as duas versões do simulador. Para finalizar a validação o simulador distribuído, foi realizada a simulação de um modelo 3D de grande porte relatada na próxima seção.

8.3.2 Simulação de plataforma em tanque de provas com 43 milhões de partículas

Este modelo numérico de plataforma em tanque de provas foi parcialmente baseado no modelo experimental desenvolvido por Lee et al. [121] para estudo do efeito de *green water*. O desenvolvimento completo e a validação do modelo numérico é parte do trabalho de mestrado em Engenharia Naval e Oceânica pela Escola Politécnica da USP, de Cezar Augusto Bellezi. Neste estudo utilizamos a discretização com 43 milhões de partículas (43M) de 10 mm, apresentada nas figuras Figura 8-22 e Figura 8-23. Com mais de uma ordem de grandeza acima da capacidade de simulação da versão serial do MPS, este modelo contribui para validação da capacidade de processamento do framework, além de permitir testar a adequação dos procedimentos de pré e pós-processamento com arquivos da ordem de 3 gigabytes.

Número de partículas	43.143.374
Distância entre partículas	10 mm
Time step	1×10^{-4} s
Tempo de simulação	6 s
Dimensões do tanque	11.38 x 4.30 x 1.67 m
Período do batedor	1,55 s
Nível d'água	1 m
Movimento de translação do batedor no eixo x	Amplitude 0,1 m, período 1,55 s
Número de subdomínios	50
Duração do experimento	~ 12 dias

Figura 8-22 Corte do modelo do tanque de provas com plataforma discretizado com 43M partículas (o interior da água foi suprimido da imagem).

Dimensões da plataforma	1.50 x 0.60 x 0.30 m
Dimensões do anteparo	0.02 x 0.60 x 0.29 m

Figura 8-23 Detalhe da plataforma para o modelo do tanque de provas com plataforma discretizado com 43M partículas.

O aumento significativo no tamanho do modelo não causou nenhuma dificuldade adicional no uso do MPS paralelizado, no entanto, os procedimentos atualmente utilizados para construção de modelos e para visualização de resultados se mostraram pouco adequados para modelos desta ordem de grandeza, indicando que, com a consolidação desta solução de paralelização, deverão ser planejados estudos para o aperfeiçoamento destes procedimentos.

A sequência de imagens abaixo, Figura 8-24 até Figura 8-30, apresenta a saída gráfica correspondente à distribuição de velocidades nos instantes de amostragem selecionados. Para diminuir o tamanho dos arquivos de saída e possibilitar a visualização dos resultados no Paraview [120], apenas as partículas do sólido e da superfície livre do fluido foram registradas nos arquivos. Uma partícula i é considerada de superfície livre se $n_i^* < \beta n_i^0$, onde $0 < \beta n_i^0 < 1$ é deve ser especificado na configuração do modelo [6].

Figura 8-24 Plataforma com 43M partículas: distribuição de velocidades 1/7.

Figura 8-25 Plataforma com 43M partículas: distribuição de velocidades 2/7.

Figura 8-26 Plataforma com 43M partículas: distribuição de velocidades 3/7.

Figura 8-27 Plataforma com 43M partículas: distribuição de velocidades 4/7.

Figura 8-28 Plataforma com 43M partículas: distribuição de velocidades 5/7.

Figura 8-29 Plataforma com 43M partículas: distribuição de velocidades 6/7.

Figura 8-30 Plataforma com 43M partículas: distribuição de velocidades 7/7.

8.4 ESTUDO DE CASO COM MODELO DE DAM BREAK 3D

Análise de convergência do simulador MPS paralelizado com modelos 3D foi realizada com o modelo de *dam break* especificado por Issa e Violeau[122] e os resultados foram comparados dos os resultados experimentais obtidos por Issa e Violeau [123].

A evolução do nível de água e da pressão foram medidos executando o MPS paralelizado com partículas de 20 mm, 10 mm e 5 mm de raio, correspondentes aos modelos de dam break com 200K, 1M e 7M partículas respectivamente, Em todas as simulações o domínio foi dividido em 10 subdomínios.

8.4.1 Medidas no sensor de altura do nível da água

As medidas do nível de água nos sensores H1, H2 e H3, posicionados conforme ilustra a Figura 8-35, apresentaram a qualidade esperada, com base em estudos anteriores realizado com a versão serial. Além disto, o aumento da resolução do modelo melhora o ajuste dos valores obtidos por simulação aos valores experimentais, como pode ser observado nas figuras de Figura 8-32 até Figura 8-34.

Figura 8-31 Posição dos sensores de nível da água, imagem extraída de Issa e Violeau [26]

Figura 8-32 Evolução do nível da água em H1

Figura 8-33 Evolução do nível da água em H2

Figura 8-34 Evolução do nível da água em H4

8.4.2 Medidas no sensor de pressão

As medidas realizadas nos sensores de pressão P1, P2 e P3, posicionados conforme ilustra a Figura 8-35, mostraram, como no caso dos sensores de pressão, a qualidade esperada com base em estudos anteriores. No entanto, o aumento da resolução do modelo não resultou em uma clara melhora no ajuste das medidas de pressão nos sensores aos valores experimentais, como pode ser observado nas figuras de Figura 8-36 até Figura 8-37.

Figura 8-35 Posição dos sensores de pressão, imagem extraída de Issa e Violeau [26]

Figura 8-36 Evolução da pressão em P1

Figura 8-37 Evolução da pressão em P2

Figura 8-38 Evolução da pressão em P3

Estudar as causas desta diferença do ajuste das medidas dos sensores em relação aos dados experimentais, em função da resolução de discretização do modelo, deverá contribuir para o aperfeiçoamento tanto do modelo MPS quanto da implementação numérica deste. No entanto, como é praticamente nula a possibilidade de explicar este comportamento como tendo sido causado pela paralelização do simulador, o aprofundamento deste estudo está além do âmbito do presente trabalho.

8.5 AVALIAÇÃO DE ESCALABILIDADE E DESEMPENHO

Para avaliação de escalabilidade e desempenho foram medidos os tempos de processamento de diversas etapas que compõe uma iteração da simulação. Para efeito de concisão dos rótulos e legendas dos gráficos apresentados nesta seção, foram atribuídos os seguintes identificadores às tarefas:

- **Solver:** construção da matriz de coeficientes e solução do sistema linear.
- **Domínio:** renumeração das partículas, sincronização.
- **Vizinhança:** construção da lista de vizinhança.
- **Cálculo explícito:** funções associadas ao cálculo explícito e à atualização do movimento das partículas.
- **Outras tarefas:** diferença entre o tempo total e a soma dos tempos das tarefas consideradas nos itens acima. Corresponde, basicamente, ao tempo das operações de entrada e saída.
- **Linear(*tarefa*):** identifica a linha de tendência linear associada às medidas de tempo de uma *tarefa*.

Da mesma forma que para as tarefas, foram atribuídos os seguintes identificadores para cada uma das configurações do simulador:

- **MPS Serial:** versão original do simulador MPS executando em modo serial
- **MPS SMP:** simulador paralelizado MPS executando com número de subdomínios igual a 1. Neste caso não divisão de domínio, mas as partículas são mantidas ordenadas e é utilizada apenas a paralelização em memória compartilhada (SMP).
- **MPS paralelização híbrida:** simulador paralelizado MPS executando com número de subdomínios maior que 1. Neste caso, são combinadas as duas formas de paralelização: para memória compartilhada e para memória distribuída.

Para realização dos testes de speedup é necessária a execução do simulador com modelos de tamanhos diversos usando diferentes configurações de divisão do domínio. A comparação de desempenho, utilizando simulações completas com período de 6 segundos de dam break, como utilizado no estudo de caso, iria consumir um tempo demasiadamente longo. Assim, foi necessário estabelecer um

período de simulação que permitisse a realização de todas as simulações necessárias em tempo hábil e, ao mesmo tempo, resultassem em medidas do desempenho do simulador distribuído com grau aceitável de precisão.

8.5.1 Definição do intervalo de simulação para medidas de speedup

Para determinação do intervalo de simulações, procurou-se estimar a variação do tempo necessário para execução de cada iteração da simulação. Para isto foram utilizadas simulações do modelo de dam break com um milhão de partículas (1M), usando a versão serial e a versão distribuída. Na Figura 8-39 os tempos por time step destas simulações estão representados por linhas pontilhadas, enquanto as respectivas linhas de tendência linear estão representadas por linhas sólidas na mesma cor.

Figura 8-39 Evolução do tempo por iteração em simulação longa para o modelo com 1M partículas de raio 10 mm, em simulações serial e distribuída.

Estes testes mostraram que, enquanto o tempo de iteração tanto no modo serial cresce de forma consistente ao longo da simulação, no modo distribuído, o tempo de iteração oscila em torno de um valor médio, com inclinação da linha de tendência muito próxima a zero ($\sim 10^{-4}$).

A princípio pensou-se que a causa deste crescimento no tempo de processamento por iteração fosse devido ao aumento no tempo de solução do sistema linear, devido ao aumento da dispersão das entradas da matriz de coeficientes de interação, fenômeno que não ocorre na versão distribuída. No entanto, um exame mais detalhado do tempo de processamento por tarefa na versão serial, como mostra a Figura 8-40, revelou que o aumento do tempo de processamento ocorre, principalmente, na construção da lista de vizinhanças. Embora, entender a causa deste comportamento do algoritmo serial de construção de vizinhanças esteja fora do escopo deste trabalho, a constatação deste fato revelou um benefício não previsto da renumeração de partículas em conjunto com estrutura de vizinhanças, que foi reformulada para o processamento distribuído, conforme apresentada na Seção 5.4.1.

Figura 8-40 Evolução do tempo por iteração por tarefa na versão serial do simulador, para simulação do modelo com 1M partículas de raio 10 mm

Como os tempos de processamento por iteração não apresentaram tendência linear significativa, procuramos determinar um intervalo mínimo de simulação, o qual permitisse realizar as medidas necessárias à avaliação de escalabilidade e desempenho da aplicação paralelizada. Por outro lado, para avaliação da versão serial, cujo tempo de iteração apresenta uma forte tendência crescente, foi necessário utilizar as simulações completas.

Nos parágrafos seguintes são detalhados os resultados da análise das variações do tempo de iteração em simulações do modelo de 1M, que justificaram considerar as estimativas de tempo de processamento, feitas a partir de simulações de apenas 0,1 segundo, representativas dos valores esperados para simulações de longa duração.

Na Figura 8-41 estão sumarizados a razão entre a duração por passo de tempo em relação a duração média por iteração, em simulações de 0,1s utilizando a divisão do domínio entre 5 a 30 subdomínios, linhas pontilhadas.

Figura 8-41 Evolução do tempo por iteração no intervalo de amostragem para o modelo com 1M de partículas de raio 10 mm, em simulações com 5 a 30 subdomínios.

A linha sólida em vermelho representa a média destas variações, enquanto a linha sólida preta representa tendência linear da média neste intervalo de tempo. Todas as variações seguem um padrão semelhante, independente do número de subdomínios utilizado e a linha de tendência linear da média tem inclinação próxima de zero ($\sim 10^{-3}$). A comparação do desvio da média obtida para o intervalo de 0,1s com a média da simulação completa foi realizada para simulações com 5, 10 e 15 subdomínios, obtendo-se desvios de no máximo 11% em todos os casos.

Também nas simulações dos modelos com 7M e 30M, para o intervalo de 0,1s os tempos de iteração apresentam características às análogas às simulações do modelo de 1M. A Figura 8-42 ilustra o tempo de iteração destes modelos utilizando 20 e 30 subdomínios, linhas pontilhadas, e as respectivas linhas de tendência, linha

sólida na respectiva cor.

Figura 8-42 Evolução do tempo por iteração no intervalo de amostragem para os modelos com 7M e 30M de partículas, raios de 5 mm e 3 mm respectivamente, em simulações com 20 e 30 subdomínios.

Os desvios máximos inferiores a 9%, nestes casos, confirmam o intervalo de 0,1s é adequado para as medidas de desempenho do simulador paralelizado.

8.5.2 Avaliação do desempenho da paralelização em memória compartilhada (SMP) por tarefa

Enquanto a paralelização distribuída é necessária devido à limitação de memória disponível no nó, a paralelização em memória compartilhada foi implementada para aproveitamento da capacidade de processamento dos oito processadores disponíveis em cada nó.

Um benefício adicional desta paralelização é que, para modelos menores que 3M partículas, o framework executado no modo SMP acelera de forma significativa a execução de todas as tarefas, permitindo diminuir em até 85% o tempo total da simulação, quando comparado ao modo serial, como mostrado no gráfico da Figura 8-43.

Figura 8-43 Tempo de execução por tarefa com paralelização SMP em relação ao modo serial, para simulação do modelo com 1M de partículas de raio 10 mm.

8.5.3 Avaliação do desempenho da paralelização em memória distribuída (DM) por tarefa

Para tarefas que não dependam da solução do sistema de equações lineares, a paralelização distribuída é responsável pela diminuição da quantidade de dados que deve ser manipulado em cada tarefa. Assim, o tempo de execução destas tarefas deve decrescer significativamente com o aumento do número de subdomínios.

Com o modelo com 1M partículas, que pôde ser executado tanto no modo SMP quanto em memória distribuída, foi possível comparar o desempenho com a paralelização distribuída em relação ao desempenho com paralelização SMP. O gráfico da Figura 8-44 mostra que o ganho de desempenho com o aumento do número de subdomínios é próximo ao valor ótimo, como o previsto no pré grafo acima.

Figura 8-44 Tempo de execução por tarefa com paralelização distribuída em relação ao modo serial, para simulação do modelo com 1M de partículas de raio 10 mm.

Por outro lado, a solução do sistema de equações lineares em modo distribuído é uma tarefa executada em dois níveis. Globalmente, é realizada uma sequência de iterações para obtenção da solução do sistema de equações lineares, enquanto que, a cada iteração global é necessário resolver localmente um sistema de equações lineares de dimensão reduzida. Assim, se por um lado, o tempo total da solução do sistema aumenta com número de iterações necessárias para a convergência da

solução do sistema global, por outro lado, este tempo decresce com a dimensão do sistema local.

O gráfico da Figura 8-45 mostra o tempo de execução do solver com paralelização híbrida, em relação ao tempo de execução utilizando apenas a paralelização SMP (número de subdomínios igual a 1) para a simulação do modelo com apenas um milhão de partículas. Neste caso, como a cada iteração do método de Jacobi é necessário resolver o sistema linear de local, o resultado é um aumento significativo do tempo necessário para solução do sistema linear global. Ainda assim, quando a escala do sistema linear local é suficientemente reduzida, neste caso para 20 subdomínios ou mais, é possível compensar o custo das iterações globais e obter uma redução no tempo total de solução do sistema.

Figura 8-45 Tempo de execução do Solver com paralelização distribuída em relação ao modo serial, para simulação do modelo com 1M de partículas de raio 10 mm.

8.5.4 Avaliação do desempenho da paralelização híbrida (DM+SMP) por tarefa

Embora a paralelização distribuída tenha sido desenvolvida como recurso para simulação de modelos com dezenas de milhões de partícula, a combinação desta paralelização com a paralelização SMP pode ser utilizada como recurso para acelerar ainda mais as simulações de modelos relativamente pequenos.

O gráfico da Figura 8-46 mostra que, para o modelo de 1M partículas, a paralelização híbrida, com divisão em 25 subdomínios, permite reduzir o tempo total da simulação para 9% do tempo da simulação em modo serial.

Figura 8-46 Tempo de execução por tarefa com paralelização híbrida (25 subdomínios) em relação ao modo serial. Simulação do modelo com 1M de partículas de raio 10 mm.

Para avaliar o ganho de desempenho por tarefa com o modelo de 30M partículas foi usado como referência a simulação com 15 subdomínios, correspondente ao menor número de subdomínios necessário para execução do simulador. Os resultados, sumarizados no gráfico da Figura 8-47, mostram que a maioria das tarefas, com exceção do controle da decomposição de domínio (Domínio), apresenta ganho de desempenho próximo do ideal (speedup linear).

Figura 8-47 Tempo de execução por tarefa com paralelização híbrida em relação à simulação com 15 subdomínio, usando o modelo com 30M de partículas de raio 3 mm.

8.5.5 Avaliação do speedup da paralelização híbrida (DM+SMP) em função do número de subdomínios

Diversos indicadores têm sido utilizados para análise do desempenho de aplicações paralelizadas [87]. Em geral procura-se medir o ganho de desempenho alcançado pela paralelização de uma dada aplicação em relação a uma implementação serial. Dois dos indicadores mais comumente usado são o Speedup e a Eficiência da Paralelização (*Parallel Efficiency*). O speedup de uma aplicação paralelizada é a razão do tempo de execução do algoritmo serial $T(n)$ pelo tempo de execução do algoritmo paralelo:

$$S(n, p) = \frac{T(n)}{T(n, p)} \quad (6-1)$$

onde $T(n)$ é o tempo de execução do algoritmo em modo serial, $T(n, p)$ é o tempo de execução do algoritmo paralelo utilizando p elementos de processamento e n é a dimensão do problema. Em geral, $S(n, p) < p$, e $S(n, p) = p$ é denominado speedup ideal, ou speedup linear.

A Eficiência da Paralelização é definida pelo coeficiente $E(n) = S(n, p)/p$, é uma estimativa da eficiência com que os processadores adicionais são utilizados. Em geral, $E(n) < 1$ e $E(n) = 1$, é denominada eficiência ideal, ou eficiência linear.

Para o modelo de 1M partículas, que pode ser executado em modo serial, o tempo de referência, $T(n)$ foi calculado baseado no tempo de execução serial, enquanto que para as simulações dos modelos de larga escala, foi necessário considera uma definição mais abrangente de speedup.

Embora definição speedup conforme a equação tem sido utilizada com sucesso quando a dimensão do problema permite a avaliação do tempo de execução do algoritmo serial em estudo. Contudo, devido às dimensões dos modelos estudados e ao uso de processamento paralelizado híbrido, SMP e distribuído simultaneamente, foi necessário rever o objetivo da medida de escalabilidade segundo, por exemplo, a definição de Williams e Smith []:

“Scalability is a measure of an application system’s ability to - without modification - cost-effectively provide increased throughput, reduced response time and/or support more users when hardware resources are added.”

Esta definição enfatiza que o foco da medida de escalabilidade é medir o ganho obtido com o aumento dos recursos computacionais utilizados, sem a necessidade de tomar uma versão serial otimizada da aplicação como referência. Estratégia análoga foi seguida por [] para a formulação de uma formulação estatística de speedup, que compara o desempenho de duas aplicações quaisquer que resolvem um dado problema. Assim, para os modelos maiores, cada nó do *cluster*, processando um subdomínio, foi tomando como elemento de processamento e como referência foi utilizado o tempo $T(n, p_{min})$, onde p_{min} é o número mínimo de subdomínios necessários para simulação do modelo de tamanho n .

Como pode ser observado na Figura 8-48, há um aumento significativo do speedup, quando não há decomposição do domínio, isto é, número de subdomínio igual a 1. Neste caso, o ganho de desempenho é resultado da paralelização em memória compartilhada utilizando os oito processadores do nó.

Figura 8-48 Speedup simulando modelo com 1M partículas de 10 mm.

Nas simulações com paralelização híbrida do modelo de 1M partículas, número de subdomínio maior que 1, há uma queda expressiva a partir de 5 subdomínios e um crescimento posterior muito abaixo do speedup linear. Este comportamento é explicado pelas limitações da utilização de paralelização distribuída em modelos

pequenos, conforme apresentado na Seção 8.5.3. A curva de eficiência da paralelização para este modelo, Figura 8-49, ilustra claramente este comportamento: a partir de 5 subdomínios a eficiência conseguida é de aproximadamente 0,45, ou 45% da eficiência ideal.

Figura 8-49 Eficiência da paralelização, simulando modelo com 1M partículas de 10 mm.

Para avaliação do speedup com modelos grandes, que é o principal foco deste trabalho, foram utilizados dois modelos: um com 7M de partículas e outro com 30M de partículas. No primeiro caso, Figura 8-50, foi usado como tempo de referência $T(7M, 5)$, isto é, o tempo de simulação do modelo de 7M partículas divididas em 5 subdomínios. De modo análogo, no segundo caso a referência utilizada foi $T(30M, 15)$, Figura 8-51.

Figura 8-50 Speedup simulando o modelo de 7M partículas com raio de 5 mm.

Figura 8-51 Speedup simulando modelo de 30M partículas com raio de 3 mm.

Em ambos os casos a eficiência média do MPS paralelizado ficou muito próximo da eficiência ideal: 0,99 para o modelo com 7M de partículas e 0,93 para o modelo com 30M de partículas, Figura 8-52.

Figura 8-52 Eficiência da paralelização, simulando modelos com 7M e 30M partículas.

8.5.6 Comparação dos tempos de execução serial x paralelo

O aumento da resolução de um modelo MPS, isto é, a diminuição do raio das partículas por fator f , além de aumentar o tamanho do modelo por um fator proporcional a f^3 , ainda implica na redução do time step por um fator da ordem de f para preservar a estabilidade do modelo. Assim, além do desempenho da solução de paralelização, revelado pelo cálculo do speedup, o tempo simulação em uma dada configuração experimental, depende também da calibração dos parâmetros do modelo MPS.

No gráfico da Figura 8-53, estão consolidadas as estimativas de tempo para simulação de 6 segundos de dam break para os modelos de 1M, 7M e 30M partículas, considerando o valor de time step utilizado em cada configuração.

Figura 8-53 Tempo estimado para simulação de 6 segundos de dam break.

Nesta figura as barras representam o tempo por time step (em vermelho) e o tempo total de simulação (em azul) em ao tempo de simulação serial do modelo de 1M partículas. Em cada caso foi considerada a melhor configuração de paralelização para cada modelo. Os passos de tempo para as simulações mais refinadas, partículas com raios de 5mm e 3mm foram calculados diretamente aplicando-se o

mesmo fator de redução do raios da partículas em relação raio de 10mm, aos respectivos passos de simulação.

Estes resultados mostram que uma calibração mais cuidadosa do modelo, que permita a definição de um time step ligeiramente superior aos utilizados pode ter um efeito significativo no tempo total da simulação.

Outro fato importante, que deve ser destacado destes resultados, é que o framework de paralelização, além de permitir a simulação de modelos com mais de 3M partículas, ainda é capaz de simular o modelo com 7M partículas praticamente no mesmo tempo necessário para a simulação do modelo de 1M partículas, a despeito do aumento no número de iterações necessários à simulação de um mesmo intervalo de tempo.

8.5.7 Comparação dos tempos relativos de execução por tarefa

Nos gráficos da Figura 8-54 foram consolidados os consumos por CPU das principais tarefas na simulação do modelo MPS. Estes gráficos revelam claramente o efeito da divisão do problema sobre o tempo gasto nas etapas de cálculo explícito e cálculo de vizinhanças.

Figura 8-54 Consumo de CPU por tarefa, simulação com 1M partículas distribuídas em 20 subdomínios

O custo computacional da realização destas tarefas diminuiu significativamente com o aumento do número de subdomínio. Nos modelos de dam break este custo é reduzido de um 43%, no modo serial, e 19% no modo SMP, para cerca de 4%, na versão distribuída. Por outro lado, o custo computacional relativo da solução do sistema de equações lineares com paralelização híbrida, teve um aumento significativo, passando a ser responsável por até 95% do tempo de simulação. Este resultado indica claramente que o *solver* deverá ser o foco principal dos aperfeiçoamentos deste framework.

8.5.8 Balanceamento de carga

O balanceamento de carga foi avaliado utilizando um modelo de dam break, com 1M de partículas de 10 milímetros de raio dividido em 8 subdomínios. Como esperado, a configuração espacial dos subdomínios segue o deslocamento do fluido ao longo da simulação e do número de partículas mantém-se constante. Esta evolução é mostrada pelos instantâneos da simulação selecionados que estão agrupados na Figura 8-55, onde os números na imagem do canto superior esquerdo identificam cada subdomínio.

Figura 8-55 Evolução da distribuição das partículas em 8 subdomínios.

Os resultados do monitoramento do consumo de processamento ao longo de 24 horas de simulação, estão consolidados na Figura 8-56. Como mostrado nesta figura, o subdomínio 0 tem inicialmente apenas partículas fixas e é separado da coluna de líquido pelo Subdomínio 1. Por esta razão, o seu consumo de processamento é de 20% mais baixo do que a média dos subdomínios 3-7 durante as primeiras 750 iterações, menos de 8% do tempo total de simulação.

O subdomínio 1 tem inicialmente poucas partículas de fluido, mas é adjacente ao Subdomínio 2 que contém fluido. O consumo de processamento deste subdomínio cresce mais rapidamente porque a proporção partículas de fluido no seu interior aumenta com o fluxo de partículas de fluido provenientes do subdomínio 2.

Para os subdomínios 2-7, o consumo de CPU é praticamente constante ao longo do período controlado. Após o período inicial em que o desequilíbrio de carga devido a

O subdomínio é de cerca de 20%, a carga média dos processadores por subdomínios é praticamente o mesmo.

Configuração inicial dos subdomínios

Figura 8-56 Uso de CPU na simulação do dam break 1M com 8 subdomínios

Estes resultados indicam que a estratégia adotada para a decomposição de domínio assegura um equilíbrio de carga homogêneo durante a maior parte da simulação.

A análise do balanceamento de carga foi complementada com uma avaliação da qualidade do paralelismo alcançado pelo framework na utilização dos processadores de cada nó. Para isto, foi utilizada a ferramenta Intel® Concurrency Checker (ICC) [124], que permite uma análise de concorrência da aplicação. O ICC mede a execução da aplicação durante um intervalo de tempo e calcula o “Computed

Scaling Factor”. Este fator é uma estimativa do aumento de desempenho quando o aplicativo é executado em um sistema com múltiplos processadores, comparado a um sistema com um processador. Devido a limitações do ICC, foi utilizado um *cluster* com 4 processadores por nó.

A visualização do resultado da análise ICC deve ser feita em uma página personalizada, que fica disponível no site da Intel após o *upload* do arquivo gerado pelo ICC. Nesta página os resultados são comparados com os valores mínimos sugeridos pela Intel e, além disso, é mostrada a posição da aplicação no ranking das aplicações que submeteram resultados para avaliação. O MPS distribuído alcançou um “Computed Scaling Factor” de 3,98 (valor máximo = 4,0). A Figura 8-57 mostra o relatório da análise de concorrência do ICC disponibilizado no site da Intel.

Figura 8-57 Relatório de análise de concorrência pelo Intel® Concurrency Checker

9 CONCLUSÕES

9.1.1 Estratégia de decomposição

A formulação da estratégia de decomposição utilizada na elaboração deste framework foi direcionada pelos seguintes princípios:

- Tirar o máximo proveito dos recursos computacionais disponíveis.
- Evitar a elaboração de algoritmos complexos para otimização da solução de problemas específicos o que, em geral, acarreta no acoplamento de elementos da solução global e, portanto, limitam a escalabilidade.
- Resolver problemas globais combinando soluções simplificadas sobre dados localizados e, com isto, aumentar o desacoplamento dos elementos da solução.

Seguindo estes princípios, foi possível a elaboração de uma estratégia de divisão de domínio, conceitualmente simples, que utiliza apenas quatro números ordinais fixos para limites de cada subdomínio e, mesmo assim capaz de combinar as melhores características das estratégias de divisão de domínio estáticas e dinâmicas mais utilizadas. A estratégia proposta não só preserva a simplicidade da divisão estática de domínio como, também, garante a homogeneidade do balanceamento de carga, que é o principal obstáculo nas formulações dinâmicas de divisão de domínio.

Mesmo considerando que a implementação do framework ainda precise aperfeiçoamentos para atingir todo seu potencial, os resultados mostram que a estratégia de decomposição de domínio proposta é uma opção viável para estender a aplicação do modelo MPS para problemas com dezenas de milhões de partículas.

9.1.2 Metodologia orientada a objetos e C++

A utilização de modelagem orientada a objetos mostrou-se uma estratégia essencial na estruturação do framework. O encapsulamento dos algoritmos não numéricos, através da definição criteriosa das responsabilidades dos objetos, permitiu a separação quase integral das tarefas de gerenciamento da paralelização, de responsabilidade do framework, dos algoritmos numéricos do simulador. Além disto, tarefas como sincronização, ordenação e controle de estruturas de dado, foram

delegadas a objetos com baixo acoplamento, de forma a permitir que a extensão das funcionalidades do simulador e o aumento do desempenho dos algoritmos do framework possam ser desenvolvidos de forma independente. Nos itens abaixo, exemplificamos a utilização de cada um destes no desenvolvimento do framework:

- Encapsulamento: como exemplo mais significativo da vantagem deste recurso, podemos citar a implementação da estrutura de dados distribuída para vizinhanças de partículas. Embora tenha sido necessário o projeto de uma representação mais complexa que a utilizada na versão serial, o resultado final foi uma redução significativa dos requisitos de memória, além de uma forma de acesso relativamente mais simples.
- Hierarquia de classes: a paralelização para memória distribuída é reconhecidamente uma tarefa muito mais complexa que a paralelização para memória compartilhada, devido, principalmente, à necessidade de sincronizar as operações de comunicação entre processadores. O uso de especialização na formulação da estratégia de sincronização permitiu definir uma hierarquia de classes que encapsula as funções da biblioteca MPI. Isto simplificará significativamente o projeto de novas funcionalidades do simulador, que poderão ser desenvolvidas com um conhecimento mínimo de programação MPI.
- Polimorfismo: os recursos da linguagem C++, que poderiam comprometer o desempenho da implementação, como sobrecarga de operadores (overload), polimorfismo dinâmico, etc., foram utilizados de forma apenas marginal. Isto porque as operações matemáticas, cuja simplificação é, usualmente, a principal justificativa para uso destes recursos estava concentradas no código reutilizado da versão serial.

9.1.3 Resultados

9.1.3.1 Pré e pós-processamento

A realização dos testes de validação com modelos de dezenas milhões de partículas mostrou que tanto a leitura e como a geração de dados para pós-processamento foram resolvidas de forma satisfatória. A primeira através de uso funções para

processamento distribuído da biblioteca MPI-IO. Enquanto a segunda foi resolvida utilizando-se a versão da biblioteca VTK para processamento distribuído.

Embora a geração dos arquivos de dados para pós-processamento não tenha impacto significativo no tempo global da simulação, a visualização destes arquivos no Paraview, quando gerados em simulações com modelos desta dimensão, pode ser uma restrição ao tamanho máximo dos modelos. Além disto, a própria geração do arquivo de partículas para estas representa um desafio computacional que deverá ser focado em trabalhos futuros.

9.1.3.2 **Memória**

Com a distribuição das estruturas de dados foi possível simular modelos com dezenas de milhões de partículas, que era o principal objetivo deste trabalho. No entanto, a utilização da versão original, não distribuída, da estrutura de células, utilizada para acelerar a construção das vizinhanças de partículas, representa um limitação da implementação, que poderá ter impacto em futuras simulações como modelos com mais de uma centena de milhões de partículas. Para enfrentar este desafio estão sendo estudadas estratégias de distribuição da estrutura de células, bem como de representação de conjuntos esparsas de células destas por uma matriz esparsa de células.

9.1.3.3 **Balanceamento de carga**

Como resultado da decomposição de domínio proposta, os subdomínios têm, praticamente, o mesmo tamanho e o balanceamento de carga é quase homogêneo. O balanceamento de carga é afetado apenas pela diferença de cálculos realizados para cada tipo de partícula: subdomínios com maior número de partículas fixas num dado instante requerem menos processamento para resolver o sistema de equações lineares local. Para casos como quebra da barragem, isto resulta em um ligeiro desequilíbrio nas fases iniciais da simulação.

9.1.3.4 **Desempenho**

O desempenho e escalabilidade da solução mostram que o tempo necessário para a simulação de modelos muito grandes pode ser mantido dentro de limites razoáveis se houver recursos de processamento suficientes. Aumento de quase 100% no

desempenho, em simulação de modelos da ordem de poucos milhões de partículas, conseguido com a paralelização distribuída, foi um benefício não esperado. Embora este resultado dependa da alocação de recursos computacionais desproporcionais ao tamanho do problema, ainda assim é uma opção útil se houver extrema urgência na obtenção dos resultados.

A diminuição significativa do tempo de processamento da etapa de cálculo explícito deverá trazer ganhos expressivos na simulação de modelos de fluídos não newtonianos ou de sólidos deformáveis. Nestes modelos, em desenvolvimento pelo grupo MPS do TPN, grande parte dos recursos de processamento e memória é consumido pela etapa de cálculo explícito.

Os testes de desempenho realizados mostraram que a construção e solução do sistema de equações lineares são responsáveis, na versão paralelizada, por até 95% do tempo de processamento. A causa principal deste aumento no tempo de solução do sistema se deve, claramente, à introdução do laço de iteração do método de Jacobi, que implica em repetidas soluções do sistema de equações lineares local. A redução no tempo de solução do sistema de equações lineares local, pelo método do Gradiente Conjugado, não é suficiente para compensar o número de repetições introduzidas pela iteração de Jacobi. No entanto, conforme alguns autores já haviam observado o número de iterações do método de Jacobi para matrizes em blocos é inesperadamente pequeno, quando comparado à versão clássica do método. Contudo, não foi encontrada na literatura uma análise que relacione o aumento da taxa de convergência do método de Jacobi com propriedades dos blocos da matriz original. Embora já tenha sido verificado, também, que técnicas de sobreposição de subdomínios podem melhorar a taxa de convergência do método Jacobi por blocos. Um estudo mais aprofundado desta relação poderá resultar em estratégias de aceleração do método de Jacobi como empregado neste trabalho.

Além disto, como o efeito de acelerar a solução do sistema linear de equações local tem o mesmo impacto no tempo total de solução do sistema global, deverão ser avaliadas outras opções de solução pelos métodos do GC e GC Pré-condicionado, tanto os disponíveis em bibliotecas numéricas e como desenvolvidos especificamente para o framework.

Como consideração final, é importante ressaltar que, por estender a aplicabilidade do MPS a problemas de dezenas de milhões de partículas, este estudo contribui para a consolidação do método MPS como ferramenta prática para a investigação de complexos problemas de engenharia, os quais são difíceis analisar usando abordagens analíticas ou técnicas baseadas em malha.

9.1.4 Desenvolvimentos Futuros

Além das atividades anteriormente relacionadas, como otimização do solver, distribuição da estrutura de dados para células, planejamos investir na extensão do framework para suporte aos requisitos de novas funcionalidades do Modelo MPS, como modelagem de sólidos deformáveis e fluidos não newtonianos.

10 BIBLIOGRAFIA

1. PEIRO, J.; SHERWIN, S. Finite Difference, Finite Element and Finite Volume Methods for Partial Differential Equations. In: YIP, S. **Handbook of Materials Modeling. Volume I: Methods and Models**. [S.l.]: Springer, 2005. p. 1-32.
2. BUDD, C. J.; HUANG, W.; RUSSELL, R. D. Adaptivity with moving grids. **Acta Numerica**, v. 18, p. 111-241, 2009. ISSN DOI: 10.1017/S0962492906400015. Disponivel em: <<http://opus.bath.ac.uk/26694/>>.
3. LI, S.; LIU, W. K. **Meshfree Particle Methods**. [S.l.]: Springer, 2007.
4. GINGOLD, R. A.; MONAGHAN, J. J. Smoothed particle hydrodynamics - Theory and application to non-spherical stars. **Monthly Notices of the Royal Astronomical Society**, v. 181, p. 375-389, nov. 1977.
5. LUCY, L. B. A numerical approach to the testing of the fission hypothesis. **The Astronomical Journal**, v. 82, p. 1013-1024, dez. 1977. ISSN DOI: 10.1086/112164.
6. KOSHIZUKA, S.; NOBE, A.; OKA, Y. A particle method for incompressible viscous flow with fluid fragmentation. **CFD Journal**, v. 4, n. 1, p. 29-46, 1995.
7. KOSHIZUKA, S.; NOBE, A.; OKA, Y. Moving-particle semi-implicit method for fragmentation of incompressible fluids. **Nucl Sci Eng**, v. 123, n. 3, p. 421-434, 1996.
8. KOSHIZUKA, S.; NOBE, A.; OKA, Y. Numerical analysis of breaking waves using the moving particle semi-implicit method. **Internat. J. Numer. Methods Fluids**, v. 26, p. 751-769, 1998.
9. PREMOŽE, S. . T. T. B. J.; LEFOHN, A.; WHITAKER, T. **Particle-Based Simulation of Fluids**. Computer Graphics Forum - EUROGRAPHICS. [S.l.]: [s.n.]. 2003. p. 401-410.
10. TOVAT, J. H.; NISHIMOTO, K. **Método de partículas para a modelagem de fluidos incompressíveis**. Congresso Nacional de Matemática Aplicada e Computacional. [S.l.]: [s.n.]. 2005.
11. TSUKAMOTO, M. M. **Desenvolvimento do método de partículas na**

representação de corpos flutuantes em ondas altamente não-lineares.

Escola Politécnica, Universidade de São Paulo. São Paulo, São Paulo, BR. 2006.

12. DIAS, B. Z.; ALVES JR., N. **Evolução do Padrão Ethernet**. Centro Brasileiro de Pesquisas Físicas. Rio de Janeiro - Brasil. 2002. (CBPF-NT-002/02).
13. GRUN, P. **Introduction to InfiniBand for End Users**. InfiniBand Trade Association. [S.l.]. 2012.
14. MPI. MPI: A Message-Passing Interface Standard. **Message Passing Interface Forum**, 2011. Disponível em: <<http://www.mpi-forum.org>>. Acesso em: 2011.
15. OPENMP. **The OpenMP® API specification for parallel programming**, 2011. Disponível em: <<http://www.openmp.org>>. Acesso em: 2011.
16. TOP500.ORG. Development over Time. **Top 500 Supercomputes**, 12 novembro 2012. Disponível em: <<http://www.top500.org/statistics/overtime/>>. Acesso em: 26 march 2013.
17. ALDINUCCI, M.; DANELUTTO, M. Skeleton-based parallel programming: Functional and Parallel Semantics in a Single Shot. **Computer Languages, Systems and Structures**, 33, October 2007. 179-192.
18. FOSTER, I. **Designing and Building Parallel Programs**. [S.l.]: Addison-Wesley, Reading, Massachusetts, 1995.
19. CHANDY, K. M.; TAYLOR, S. **An Introduction to Parallel Programming**. Boston, Massachusetts: Jones and Bartlett Publishers, 1992.
20. SMITH, L. A. Mixed Mode MPI + OpenMP Programming. **Technology Watches, Edinburgh Parallel Computing Centre**, 10 nov. 2011. Disponível em: <<http://www.ukhec.ac.uk/publications/tw/mixed.pdf>>.
21. IKEDA, H.; KOSHIZUKA, S.; OKA, Y. Numerical analysis of jet injection behavior for fuel-coolant interaction using particle method. **Journal of Nuclear Science and Technology**, v. 38, n. 3, p. 174–182, 2001.
22. TANAKA, M.; MASUNAGA, T. Stabilization and smoothing of pressure in MPS method. **Journal of Computational Physics**, v. 229, p. 4279–4290, 2010.

23. YAO, Z.; WANG, J.-S.; CHENG, M. Improved $O(N)$ neighbor list method using domain decomposition and data sorting. **Computer Physics Communications**, 161, n. 1-2, 1 August 2004. 27-35.
24. FORMAGGIA, L.; SALA, M.; SALERI, F. **Domain decomposition techniques**. Sandia National Laboratories. [S.I.]. 2005.
25. GIRAUD, L.; TUMINARO, R. S. **Algebraic Domain Decomposition Preconditioners**. ENSEEIHT- Institut de Recherche en Informatique de Toulouse. Toulouse, France. 2006. (RT/APO/06/07).
26. IWANO, K. et al. **A Parallel Processing Method in Finite Element Analysis using Domain Division**. Faculty of Engineering, Hiroshima University, Kagamiyama 1-4. Higashi, Hiroshima, 724 Japan.
27. IRIBE, T.; FUJISAWA, T.; KOSHIZUKA, S. Reduction of Communication In Parallel Computing of Particle Method for Flow Simulation of Seaside Areas. **Coastal Engineering Journal (CEJ)**, 52 (4), December 2010. 261-285.
28. RAPAPORT, D. C. **The Art Of Molecular Dynamics Simulation**. Second Edition. ed. [S.I.]: Cambridge University Press, 2004. 565 p.
29. TSAO, A.; TURNBULL, T. **A Comparison Algorithms for Banded Matrix Multiplication**. Sumpercomputer Research Center. Bowie, Maryland, USA. 2003. (Relatório técnico SRC-TR-093-092).
30. HAHAD, M.; ERHEL, J.; PRIOL, T. **A New Approach to Parallel Sparse Cholesky Factorization on Distributed Memory Parallel Computers**. INRIA. RENNES CEDEX-France, p. 17. 1993. (Relatório técnico 2801).
31. FLEISSNER, F.; EBERHARD, P. Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection. **Int. J. Numer. Meth. Engng**, v. 74, p. 531–553, 2008.
32. IKARI, H.; GOTOH, H. **Parallelization of MPS Method for 3-D Wave Analysis**. Proceedings of 8th ICHE - Parallelization of MPS Method for 3-D Wave Analysis, Advances in Hydro-Science and Engineerin. Nagoya, Japan: [s.n.]. 2008.

33. GOTOH, S. **3D-CMPS Method For Improvement of Water Surface Tracking In Breaking Waves, Simulation of SPHERIC Benchmark Test 2, '3D schematic dam break and evolution of the free surface', by an improved parallelized particle method and SPysics**. Proceedings of 4th SPHERIC Workshop. Nantes, France: [s.n.]. 2009. p. 265-272.
34. IWASHITA, T.; SHIMASAKI, M. Parallel Processing of 3-D Eddy Current Analysis with Moving Conductor Using Parallelized ICCG Solver with Renumbering Process. **IEEE Transactions On Magnetism**, 36(4), 2000. 1504-1509.
35. KOSHIZUKA, S.; OKA, Y. Moving Particle Semi-Implicit Method for Fragmentation of Incompressible Fluid. **Nuclear Science and Engineering**, 123, 1996. 421-434.
36. KOSHIZUKA, S.; NOBE, A.; OKA, Y. Numerical Analysis of Breaking Waves Using the Moving Particle Semi-implicit Method. **Int. J. Numer. Meth. Fluid**, 26, 1998. 751–769.
37. VERTANEN, K. D. **A Parallel Implementation of a Fluid Flow Simulation using Smoothed Particle Hydrodynamics Domain Decomposition and Inter-processor Communication**. Oregon State University. Corvallis, Oregon, USA. 1999.
38. SHIBATA, K.; KOSHIZUKA, S. Numerical analysis of shipping water impact on a deck using a particle method. **Ocean Engineering**, 34, 2007. 585–593.
39. WATANABE, M.; SUZUKI, T. Object-Oriented Fluid Flow Simulation System. **Computers & Fluids**, 27 (97), 1998. 581-597.
40. BADUEL, L. et al. **A Parallel Object-Oriented Application for 3D Electromagnetism**. Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS). [S.I.]: [s.n.]. 2004.
41. KOPYSOV, S. P. et al. Parallel Distributed Object-Oriented Framework for Domain Decomposition. In: _____ **Domain Decomposition Methods in Science and Engineering**. [S.I.]: Springer, v. 40, 2005. p. 605-614.
42. SAAD, Y. A. V. D. V. H. A. Iterative solution of linear systems in the 20th century. **J. Comput. Appl. Math.**, Amsterdam, The Netherlands, The

- Netherlands, v. 123, n. 1-2, p. 1--33, November 2000.
43. HEATH, M. Parallel Direct Methods for Sparse Linear Systems. In: KEYES, D.; SAMEH, A.; VENKATAKRISHNAN, V. **Parallel Numerical Algorithms**. [S.l.]: Springer Netherlands, v. 4, 1997. p. 55-90. ISBN ISBN: 978-94-010-6277-0 DOI: 10.1007/978-94-011-5412-3_3. Disponivel em: <http://dx.doi.org/10.1007/978-94-011-5412-3_3>.
 44. DUFF, I. S.; VAN DER VORST, H. A. Developments and trends in the parallel solution of linear systems. **Parallel Comput.**, Amsterdam, The Netherlands, The Netherlands, v. 25, n. 13-14, p. 1931-1970, 1999. ISSN ISSN: 0167-8191 DOI: 10.1016/S0167-8191(99)00077-0. Disponivel em: <[http://dx.doi.org/10.1016/S0167-8191\(99\)00077-0](http://dx.doi.org/10.1016/S0167-8191(99)00077-0)>.
 45. SAAD, Y. **Iterative Methods for Sparse Linear Systems**. 2nd. ed. [S.l.]: Society for Industrial and Applied Mathematics Philadelphia, 2003.
 46. DUFF, I. S. **The influence of vector and parallel processors on numerical analysis**. INRIA. [S.l.]. 1987.
 47. MATHEW, T. P. A. Decomposition Frameworks. In: _____ **Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations**. [S.l.]: Springer-Verlag Berlin Heidelberg, 2008b. p. 763. Lecture Notes in Computational Science and Engineering ISSN 1439-7358.
 48. KELLEY, C. T. **Iterative Methods for Linear and Nonlinear Equations**. [S.l.]: SIAM - Society for Industrial and Applied Mathematics, v. 16, 1995. pp. 166 p. Disponivel em: <<http://link.aip.org/link/doi/10.1137/1.9781611970944>>.
 49. HAASE, G.; LANGER, U. Multigrid methods: from geometrical to algebraic versions. In: BOURLIOUX, A.; GANDER, M.; SABIDUSSI, G. **Modern Methods in Scientific Computing and Applications**. [S.l.]: Springer Netherlands, v. 75, 2002. p. 103-153. ISBN ISBN: 978-1-4020-0782-8 DOI: 10.1007/978-94-010-0510-4_4. Disponivel em: <http://dx.doi.org/10.1007/978-94-010-0510-4_4>.
 50. FONSECA, B. M. D. C. **Projeto de Ensino Multigrid**. UFMG - Universidade Federal de Minas Gerais. [S.l.]. 2009.
http://www.mat.ufmg.br/~rodney/notas_de_aula/multigrid.pdf.

51. BAKER, A. H. et al. **Challenges of Scaling Algebraic Multigrid Across Modern Multicore Architectures**. Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium. Washington, DC, USA: IEEE Computer Society. 2011. p. 275-286.
52. CHOW, E. et al. A Survey of Parallelization Techniques for Multigrid Solvers. In: M.A. HEROUX, P. R. A. H. D. S. **Parallel Processing for Scientific Computing -SIAM Series on Software, Environments, and Tools**. [S.l.]: SIAM, 2006. Cap. 10.
53. PEREIRA, F. H. **O método multigrid algébrico na resolução de sistemas lineares oriundos do método dos elementos finitos**. Escola Politécnica, Universidade de São Paulo. São Paulo. 2007. Tese de Doutorado em Sistemas de Potência.
54. GEORGE, T.; GUPTA, A.; SARIN, V. **An Empirical Analysis of Iterative Solver Performance for SPD Systems**. IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598. [S.l.]. 2009.
55. TRILINOS. Trilinos Project -Sandia National Laboratories, 2011. Disponível em: <<http://trilinos.sandia.gov/>>. Acesso em: 11 out. 2011.
56. TSITSIKLIS, J. N. A comparison of Jacobi and Gauss-Seidel parallel iterations. Applied. **IEEE Trans. Aut. Control**, v. 2, p. 325-332, 1989.
57. VAN DER VORST, H. A. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. **Parallel Computing**, v. 5, n. 1-2, p. 45-54, 1987. Disponível em: <<http://dblp.uni-trier.de/db/journals/pc/pc5.html#Vorst87>>.
58. TRAUB, J. F. **Iterative Solution of Tridiagonal Systems on Parallel Or Vector Computers**. [S.l.]: Carnegie-Mellon University., 1973. Disponível em: <<http://books.google.com.br/books?id=kwuUGwAACAAJ>>.
59. GANDER, W.; GOLUB, G. H. **Cyclic Reduction - History and Applications**. Stanford University. [S.l.]. 1997.
60. SMART, D.; WHITE, J. **Reducing the Parallel Solution Time of Sparse Circuit Matrices Using Reordered Gaussian Elimination and Relaxation**. In Proc. International Symposium on Circuits and Systems. [S.l.]: [s.n.]. 1988. p. 627-630.

61. BERTSEKAS, D. P. et al. **Asynchronous Iterative Algorithms, Convergence Rate and Termination of**. [S.l.]: Center for Intelligent Control Systems, 1989. ISBN ISBN: 9780897913096.
62. KOESTER, D. P.; RANKA, S.; FOX, G. C. **Parallel block-diagonal-bordered sparse linear solvers for electrical power system applications**. Proceedings of the Scalable Parallel Libraries Conference, 1993. [S.l.]: IEEE. october 1993. p. 195-203.
63. YAMAKAWA, E.; FUKUSHIMA, M. A Block-Parallel Conjugate Gradient Method for Separable Quadratic Programming Problems. **J Oper Res Soc**, v. 39, n. 3, p. 407-427, September 1996.
64. BOUDEWIJN, A. B.; BELL, E.; HAVERKORT, B. R. **Serial And Parallel Out-Of-Core Solution of Linear Systems arising from Generalised Stochastic Petri Nets**. Proceedings High Performance Computing 2001. [S.l.]: [s.n.]. 2001. p. 22-26.
65. MARTINOTTO, A. L. **Resolução de Sistemas de Equações Lineares através de Métodos de Decomposição de Domínio**. "Instituto De Informática da Universidade Federal do Rio Grande do Sul". [S.l.]. 2007.
66. MEHMOOD, R.; CROWCROFT, J. **Parallel iterative solution method for large sparse linear equation systems**. University of Cambridge, Computer Laboratory. [S.l.]. 2005.
67. RAIM, A. M.; GOBBERT, M. K. **Parallel Performance Studies for an Elliptic Test Problem on the Cluster tara**. UMBC High Performance Computing Facility, University of Maryland, Baltimore County. [S.l.]. 2010. (HPCF machines used: tara.).
68. TERESCO, J. D.; DEVINE, K. D.; FLAHERTY, J. E. Numerical Solution of Partial Differential Equations on Parallel Computers. [S.l.]: Springer-Verlag, 2005. Cap. Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations.
69. BYLINA, J.; BYLINA, J. **Merging Jacobi and Gauss-Seidel Methods for Solving Markov Chains on Computer Clusters**. Proceedings of the International Multiconference on Computer Science and Information

- Technology. [S.l.]: [s.n.]. October 2008. p. 263–268.
70. WANG, G.; GOBBERT, M. K. **Performance Comparison between Blocking and Non-Blocking Communications for a Three-Dimensional Poisson Problem**. Department of Mathematics and Statistics, University of Maryland, Baltimore County. [S.l.]. 2009.
 71. TOTONI, E. et al. **Comparing the power and performance of Intel's SCC to state-of-the-art CPUs and GPUs**. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). [S.l.]: [s.n.]. April 2012. p. 78-87.
 72. WATKINS, D. S. **Fundamentals of Matrix Computations**. [S.l.]: John Wiley & Sons, Inc., 1991.
 73. W., S. S.; S., N. W. A direct comparison of three algorithms for reducing profile and wavefront. **Computers & Structures**, v. 33, n. 2, p. 411-419, 1989.
 74. MELONI, S.; ROSATI, M.; COLOMBO, L. Efficient particle labeling in atomistic simulations. **The Journal of Chemical Physics**, v. 126, 2007.
 75. GRAMA, A. et al. **Introduction to Parallel Computing**. Second Edition. ed. [S.l.]: Addison Wesley, 2003a.
 76. BARRETT, R. et al. **Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods**, 2nd Edition. Philadelphia, PA: SIAM, 1994.
 77. GOLUB, G. H.; VAN LOAN, C. F. **Matrix Computations**. [S.l.]: Johns Hopkins University Press, 1996. ISBN ISBN: 9780801854149 LCCN: 96014291. Disponível em: <<http://books.google.com.br/books?id=mIOa7wPX6OYC>>.
 78. CIEGIS, R. Analysis of Parallel Preconditioned Conjugate Gradient Algorithms. **Informatica**, Amsterdam, The Netherlands, The Netherlands, v. 16, n. 3, p. 317-332, August 2005. ISSN ISSN: 0868-4952. Disponível em: <<http://dl.acm.org/citation.cfm?id=1413792.1413793>>.
 79. VEENSTRA, A. H.; LIN, H. X.; VOLLEBREGT, E. A. H. A comparison of scalability of different parallel iterative methods for shallow water equations. **Contemporary Mathematics**, v. 218, p. 357-364, 1998.
 80. SARITA. **Parallel Numerical Algorithms For Symmetric Positive Definite**

- Linear Systems**. Indian Institute of Technology Delhi. [S.I.]. 2007.
81. GOTOH, H. et al. **3D-CMPS Method for Improvement of Water Surface Tracking In Breaking Waves**. Proceedings of 4th SPHERIC Workshop. Nantes, France,; [s.n.]. 2009. p. 265-272.
 82. INTEL. Intel(R) Math Kernel Library (Intel(R) MKL) Documentation, 2012. Disponivel em:
<http://www.nacad.ufrj.br/online/intel/Documentation/en_US/mkl/mkl_documentation.htm>.
 83. OBERHUBER, M. **Elimination of Nondeterminacy for Testing and Debugging Parallel Programs**. Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging. Saint Malo, France: [s.n.]. 1995.
 84. BISWAS, S.; MALL, R. Regression Test Selection Techniques: A Survey. **Informatica**, v. 35, p. 289–321, 2011.
 85. BLELLOCH, G. Is Parallel Programming Hard? **Intel® Developer Zone**, 2009. Disponivel em: <<http://software.intel.com/en-us/articles/is-parallel-programming-hard>>. Acesso em: 08 jun. 2013.
 86. EDELMAN, A. Lectures Mathematics 18.337, Computer Science 6.338, SMA 5505, 2004.
 87. GRAMA, A. et al. **Introduction to Parallel Computing**. Second Edition. ed. [S.I.]: Addison Wesley, 2003b. ISBN ISBN: 0-201-64865-2.
 88. BIGGAR, P.; GREGG, D. **Sorting in the Presence of Branch Prediction and Caches Fast Sorting on Modern Computers**. University of Dublin, Trinity College. Dublin 2, Ireland. 2005.
 89. KHAMITKAR, S. et al. The Folklore of Sorting Algorithms. **International Journal of Computer Science Issues**, 4, 2009.
 90. JOHNSON, S. L. **Computer Science 6365, Lecture 28: Sparse matrix computations**. [S.I.]: [s.n.]. 2008.
 91. POTHEN, A. Predicting the structure of sparse orthogonal factors. **Linear Algebra and its Applications**, 194, November 1993. Pages 183-203.

92. POTHEN, A.; FAN, C.-J. Computing the Block Triangular Form of a Sparse Matrix. **ACM Transactions on Mathematical Software**, 16 (4), 2003. 303-324.
93. REIF, J. H. Efficient parallel factorization and solution of structured and unstructured linear systems. **Journal of Computer and System Sciences**, 71, 2005. 86–143.
94. CHEN, K. **Matrix Preconditioning Techniques and Applications - Cambridge Monographs On Applied And Computational Mathematics**. [S.I.]: Cambridge University Press, v. 19, 2005.
95. CHAPMAN, B.; JOST, G.; PA, R. V. D. **Using OpenMP: Portable Shared Memory Parallel Programming**. Cambridge, MassachusettsLondon, England: The MIT Press, 2007.
96. JOST, G.; HAOQIANG, J.; DIETER AN MEY, H. F. F. **Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster**. [S.I.]. 2003.
97. ARMSTRONG, B.; WOOK KIM, S.; EIGENMANN, R. Quantifying Differences between OpenMP and MPI Using a Large-Scale Application Suite, 2000.
98. KRAWEZIK, G.; CAPPELLO, F. **Performance Comparison of MPI and Three OpenMP Programming Styles on Shared Memory Multiprocessors**. Proceedings of ACM SPAA 2003. San Diego, USA: [s.n.]. 2003.
99. KESSLER, M. **To OO or Not to OO**. Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'03). [S.I.]: [s.n.]. 2003.
100. DOBRIAN, F.; KUMFERT, G.; POTHEN, A. The Design of Sparse Direct Solvers using Object-Oriented Techniques. In: BRUASET, A. M.; E, L.; QUAK, H. P. **Modern Software Tools in Scientific Computing**. [S.I.]: Springer-Verlag, 2000. p. 89-131.
101. GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.I.]: Addison-Wesley, Reading, Massachusetts, 1999.
102. BALAY, S. et al. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries, Modern Software Tools in Scientific Computing.

- [S.I.]: Birkhauser Press, 1997. p. 163-202.
103. MEYER, B. **Object Oriented Software Construction**. 2nd edition. ed. [S.I.]: Prentice-Hall, 1997.
 104. BARTON, J. J.; NACKMAN, L. R. **Scientific and Engineering C++ An Introduction with Advanced Techniques and Exemples**. [S.I.]: Addison Wesley Longnan, 1994.
 105. STROUSTRUP, B. **The C++ Programming Language**. Reading, Massachusetts, USA: Addison-Wesley, 2000.
 106. TALBOT, D.; ZHOU, S.; HIGGINS, G. **Software Engineering Support of the Third Round of Scientific Grand Challenge Investigations**. Nothrop - Grumman Information Technology / TASC - NASA. Hanover - USA. 2001. (NASA/TM - 2001-209993).
 107. VELDHUIZEN, T. L. **Techniques for Scientific Computing C++**. Indiana University Computer Science. [S.I.]. 2000.
 108. NORRIS, B. et al. **Parallel Components for PDEs and Optimization: Some Issues and Experiences**. Argonne National Laboratory. [S.I.].
 109. COMMUNITY, L. Lustre Community Portal, 2012. Disponivel em: <<http://lustre.opensfs.org/>>. Acesso em: 15 fev. 2103.
 110. MITCHELL, B. NFS (Network File System), 2012. Disponivel em: <<http://compnetworking.about.com/od/networkstorage/g/nfs-file-system.htm>>.
 111. INTEL. Intel® Xeon® Processor X5560, 2012. Disponivel em: <<http://ark.intel.com/Product.aspx?id=37109>>. Acesso em: 19 jan. 2013.
 112. COMMUNITY, C. CentOS - Community ENTERprise Operating System, 2012. Disponivel em: <<http://centos.org>>.
 113. INTEL. Intel(R) C++ Compiler Documentation, 2012. Disponivel em: <http://www.nacad.ufrj.br/online/intel/Documentation/en_US/compiler_c/compiler_documentation_c.htm>.
 114. MPICH. MPICH - High-Performance Portable MPI, 2012. Disponivel em: <<http://www.mcs.anl.gov/research/projects/mpich2/>>. Acesso em: 11 out. 2012.
 115. VTK. Visualization Toolkit. **Visualization Toolkit**, 2013. Disponivel em:

- <<http://www.vtk.org/>>. Acesso em: 28 mar. 2013.
116. LLNL, L. L. N. L. Slurm - Simple Linux Utility for Resource Management, 2012. Disponivel em: <<https://computing.llnl.gov/linux/slurm/documentation.html>>. Acesso em: 17 jun. 2012.
 117. SCONS. SCons: A software construction tool, 10 dez. 2012. Disponivel em: <<http://www.scons.org/>>. Acesso em: 10 dez. 2012.
 118. APACHE, X. Xerces-C++ XML parser, 2012. Disponivel em: <<http://xerces.apache.org/xerces-c/>>. Acesso em: 10 dez. 2012.
 119. MATHWORKS INC. MATLAB - The Language of Technical Computing, jan. 2013. Disponivel em: <<http://www.mathworks.com/products/matlab/>>. Acesso em: 03 April 2013.
 120. PARAVIEW. **ParaView - Open Source Scientific Visualization**, out. 2011. Disponivel em: <<http://www.paraview.org/>>. Acesso em: nov. 2011.
 121. HYUN-HO LEE, H.-J. L. S. H. R. Experimental investigation of green water on deck for a CFD validation database. **Ocean Engineering**, v. 42, p. 47–60, 2012.
 122. ISSA, R.; VIOLEAU, D. **Test Case 2: 3D-Dambreaking**. [S.l.]. 2006.
 123. ISSA, R.; VIOLEAU, D. Test2, 2006b. Disponivel em: <<https://wiki.manchester.ac.uk/spheric/index.php/Test2>>. Acesso em: November 2012.
 124. INTEL. Intel® Concurrency Checker 4.0 - Multi-Core Software Assessment User's Guide, December 2010. Disponivel em: <http://software.intel.com/sites/default/files/Multi-Core_Software_Assessment_Guide.pdf>. Acesso em: 14 jan. 2013.
 125. DONGARRA, J. et al. **Sourcebook of parallel computing**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN ISBN: 1-55860-871-0.
 126. ASENJO, R. et al. **HPF-2 Support for Dynamic Sparse Computations**. Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing. London, UK, UK: Springer-Verlag. 1999. p. 230-246.
 127. BARRETT, R. et al. **Templates for the Solution of Linear Systems: Building**

Blocks for Iterative Methods. Disponivel em:

<http://www.netlib.org/linalg/html_templates/Templates.html>. Acesso em: may 2011.

128. LIVNE, O. E. **Diagonal dominance of SPD matrices**. Stanford University. [S.l.]. 2004.
129. HO, C.; OF HONG KONG, U. **Iterative Methods for the Robbins Problem**. [S.l.]: University of Hong Kong, 2000. Disponivel em: <<http://books.google.com.br/books?id=7SsWHQAACAAJ>>.
130. K. FÜRLINGER, G. M. **Analyzing Overheads and Scalability Characteristics of OpenMP Applications**. Proceedings of the Seventh International Meeting on High Performance Computing for Computational Science (VECPAR'06). [S.l.]: [s.n.]. 2006.
131. KARNIADAKIS, G. E.; KIRBY II, R. M. **Parallel Scientific Computing in C++ and MPI -A Seamless Approach to Parallel Algorithms and Their Implementation**. [S.l.]: Cambridge University Press, 2003.
132. OBERKAMPF, W. L.; TRUCANO, T. G. **Verification and Validation in Computational Fluid Dynamics**. SANDIA. [S.l.]. 2002.
133. OLIKER, L. et al. Effects of ordering strategies and programming paradigms on sparse matrix computations. **Siam Review**, 44 (3), 2002. 372-393.
134. PEREZ, F.; RIVERA, W. An Object Oriented Framework for Computational Fluid Dynamics Simulations.
135. SHAKIBAEINIA, A.; YEE-CHUNG, J. A weakly compressible MPS method for modeling of open-boundary free-surface flow. **International Journal for Numerical Methods in Fluids**, v. 63, n. 10, p. 1208–1232, 2010. Disponivel em: <<http://onlinelibrary.wiley.com/doi/10.1002/fld.2132/pdf>>.
136. KWIATKOWSKA, M.; MEHMOOD, R. Out-of-Core Solution of Large Linear Systems of Equations Arising from Stochastic Modelling. In: HERMANNNS, H.; SEGALA, R. **Process Algebra and Probabilistic Methods: Performance Modeling and Verification**. [S.l.]: Springer Berlin Heidelberg, v. 2399, 2002. p. 135-151. ISBN ISBN: 978-3-540-43913-4 DOI: 10.1007/3-540-45605-8_9. Disponivel em: <http://dx.doi.org/10.1007/3-540-45605-8_9>.

137. GOTOH, H.; SAKAI, T. Key issues in the particle method for computation of wave breaking. **Coastal Engineering**, 53, February 2006. 171-179.
138. BERTSEKAS, D. P.; TSITSIKLIS, J. N. Some Aspects of Parallel and Distributed Iterative Algorithms -- A Survey. **Automatica**, v. 27, p. 3-21, 1991.
139. FALGOUT, R. D.; YANG, U. M. **hypre**: A Library of High Performance Preconditioners. Proceedings of the International Conference on Computational Science-Part III. London, UK, UK: Springer-Verlag. 2002. p. 632-641.
140. LI, S.; LIU, W. K. Meshfree and particle methods and their applications. **Applied Mechanics Reviews**, v. 55, n. 1, p. 1-34, 2002. ISSN DOI: 10.1115/1.1431547. Disponível em: <<http://dx.doi.org/10.1115/1.1431547>>.
141. DEVINE, K. D. et al. New challenges in dynamic load balancing. **Appl. Numer. Math.**, v. 52, n. 2-3, p. 133-152, 2005.
142. KHAYYER, A. . G. H. I. H.; HORI, C. **Development of 3D Parallelized CMPS-HS with a Dynamic Domain Decomposition Approach**. Proc. International Sessions in Annual Japanese Coastal Engineering Conference, JSCE. Tokyo, Japan: [s.n.]. November 2010. p. 6-10.
143. MATHEW, T. P. A. **Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations**. [S.l.]: Springer-Verlag Berlin Heidelberg, 2008. 763 p. Lecture Notes in Computational Science and Engineering ISSN 1439-7358.
144. LAWRENCE LIVERMORE NATIONAL LABORATORY. Message Passing Interface (MPI), 13 jul. 2012. Disponível em: <https://computing.llnl.gov/tutorials/mpi/#Collective_Communication_Routines>. Acesso em: 2013 jan. 9.
145. BALAY, S. et al. PETSc, 2012. Disponível em: <<http://www.mcs.anl.gov/petsc>>. Acesso em: 11 maio 2012.
146. PANTALÉ, O.; CAPERAA, S.; RAKOTOMALALA, R. Development of an object-oriented finite element program: application to metal-forming and impact simulations. **Journal of Computational and Applied Mathematics**, v. 168, p. 341-351, 2004. ISSN ISSN: 0377-0427 DOI: 10.1016/j.cam.2003.04.009. Disponível em:

<<http://www.sciencedirect.com/science/article/pii/S0377042703009981>>.

Selected Papers from the Second International Conference on Advanced Computational Methods in Engineering (ACOMEN 2002).

11 GLOSSÁRIO

G.1. CLUSTER COMPUTING

Conjunto de computadores, denominados nós, interconectados de maneira a formarem uma rede, de qualquer topologia. Cada nó tem sua própria memória e um espaço de endereçamento distinto. Os nós se comunicam através de um canal de rede especializado de alta velocidade e baixa latência. O modelo mais comum de paralelização é o “Single Program Multiple Data” (SMPD): múltiplas cópias de um mesmo programa executam em diferentes processadores utilizando dados diferentes. A transferência de dados entre os processadores é programada de forma explícita, em geral utilizando rotinas MPI.

G.2. ENTRADAS DE UMA MATRIZ

Uma matriz quadrada $A_{n \times n}$ é definida como um conjunto de elementos numéricos dispostos em n linhas n colunas:

$$A = [a_{ij}] = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

Os elementos a_{ij} não nulos da matriz A são denominados **entradas** de A .

Uma matriz $A_{n \times n} = [a_{ij}]$ é simétrica se $a_{ij} = a_{ji}$ para $1 \leq i, j \leq n$.

G.3. FILL IN

Redução da esparsidade de uma matriz. Ocorre, por exemplo, no método de eliminação de Gauss, quando muitos elementos nulos na matriz original tornam-se não nulo.

G.4. **FRAMEWORK**

É um conjunto de classes e interfaces que cooperam para resolver um tipo de problema de software. As funcionalidades disponibilizadas por um framework podem ser customizadas por configuração durante a programação de uma aplicação. Ao contrário das bibliotecas, é o framework quem dita o fluxo de controle da aplicação.

G.5. **GRID COMPUTING (COMPUTAÇÃO EM GRID)**

Refere-se a uma técnica computacional que possibilita o acesso “transparente” a recursos de diferentes computadores com o intuito de resolver problemas de grande complexidade e/ou volume. Em uma grade o processamento paralelo distribuído é realizado através do compartilhamento, seleção e agregação dinâmica, em tempo de execução, de recursos computacionais autônomos, possivelmente heterogêneos e geograficamente distribuídos. A seleção é feita com base na sua disponibilidade, capacidade, desempenho, custo e qualidade de serviços requeridos pelo usuário.

G.6. **LARGURA DE BANDA DE COMUNICAÇÃO (*BANDWIDTH*)**

Taxa de transferência de dados entre os nós, tipicamente GB/s.

G.7. **LATÊNCIA (*LATENCY*)**

Tempo necessário para iniciar uma transferência de dados entre nós, o mesmo que tempo para enviar uma mensagem de zero byte. Tipicamente Normalmente alguns micros segundos.

G.8. **LINEAR SPEEDUP**

Valor teórico ideal (ou ótimo) de speedup ^[G.22], $S(n, p) = p$, isto é, para um dado problema de dimensão n o ganho de desempenho é equivalente ao número de processadores utilizados p .

G.9. MATRIZ Densa E MATRIZ ESPARSA

Uma matriz é denominada **esparsa**, quando o número de entradas é ‘*muito*’ inferior ao número total de elementos na matriz, isto é n^2 , caso contrário a matriz é denominada **densa**. Embora a diferenciação entre matrizes densas e esparsas seja algebricamente irrelevante, ela é essencial no desenvolvimento de algoritmos computacionais. Na prática, uma matriz é denominada esparsa se o consumo de memória pode ser reduzido e o desempenho aumentado usando uma representação especializada armazenar apenas as entradas da matriz. Não existe, portanto, uma fronteira clara entre as duas classes: uma matriz considerada esparsa em um contexto pode ser considerada densa em um contexto diferente. A Figura G-1 ilustra a forma usual de representação gráfica da estrutura de uma matriz. Os índices i e j são distribuídos segundo os eixos horizontal e vertical respectivamente, e as entradas são representadas por pontos na superfície do gráfico.

Figura G-1 Estrutura de uma matriz esparsa 1024x1024

O armazenamento computacional de uma matriz esparsa requer a utilização de algoritmos e estruturas de dados especializados que tirem vantagem da estrutura da matriz. Dados esparsos são facilmente comprimidos, o que resulta em menor requisito de memória para armazenamento da matriz. De fato, algumas matrizes muito grandes esparsas são inviáveis para manipular usando algoritmos padrão para matrizes densas. Por outro lado, a economia de memória é penalizada por um aumento da complexidade de programação e pela diminuição da eficiência, mesmo de operações simples como o produto matriz x vetor [125]. Além disto, a aplicação de operações como a inversão ou a fatoração sobre uma matriz esparsa apresentam um nível adicional de complexidade, pois o resultado é uma matriz com menor esparsidade, ou eventualmente densa, aumentando consideravelmente os requisitos de armazenamento.

Quanto à forma de representação computacional as matrizes esparsas podem ser classificadas como **fixas** ou **dinâmicas**. Para matrizes esparsas com estrutura fixa, existem representações computacionais estáticas simples e de grande eficiência. No entanto, quando as operações sobre a matriz esparsa envolvem pivotamento, ou a inserção de novas entradas, é necessária a utilização de representações dinâmicas, com de estruturas dados flexíveis, que são muito mais complexas e aumentam significativamente o custo computacional dos algoritmos [126].

G.10. MATRIZ EM BANDA

Se as entradas de uma matriz esparsa $A_{n \times n} = [a_{ij}]$ estão confinadas em torno da diagonal principal, isto é, com $a_{ij} = 0$ para $|i - j| > nband$, com $nband \ll n$, então A é denominada **matriz esparsa em banda**, ou simplesmente **matriz em banda**,

com $\text{banda} = 2 * \text{nband} - 1$ [77]. Da mesma forma que a esparsidade, a caracterização de uma matriz em banda depende da natureza computacional do problema.

Em muitos casos, uma matriz esparsa pode ser transformada em uma matriz em banda pela reordenação das linhas e/ou colunas, como no exemplo da Figura G-2, que mostra a estrutura da matriz da esparsa apresentada na Figura A-1, após a reordenação das linhas.

Figura G.2 Estrutura da matriz em banda resultante da reordenação de linhas e colunas da matriz da Figura G.1

G.11. MATRIZ BLOCO TRIDIAGONAL OU MATRIZ TRIDIAGONAL DE BLOCOS

Uma matriz é denominada **matriz tridiagonal** se todos os elementos não nulos estiverem concentrados em três diagonais: inferior, principal e superior. Esta definição pode ser estendida para matrizes esparsas em banda, agrupando-se as entradas da matriz original em banda em três diagonais de blocos quadrados e representando os blocos restantes como matrizes nulas.

Dada uma matriz esparsa em banda $A_{n \times n}$, é possível construir uma matriz tridiagonal de blocos $\tilde{A}_{p \times p}$ agrupando-se as entradas

de A em blocos $A_1 \dots A_{1p}$, $B_1 \dots B_{p-1}$ e $C_2 \dots C_p$,

$$\check{A}_{p \times p} = \begin{pmatrix} A_1 & B_1 & & & \\ C_2 & A_2 & B_2 & & \\ & C_3 & A_3 & & \\ \vdots & & & A_{p-1} & B_{p-1} \\ & & & C_p & A_p \end{pmatrix}$$

Onde $A_{1m_1 \times n_1} = [a_{ij}]$ para $1 \leq i \leq m_1$ e $1 \leq j \leq n_1$ e, de forma análoga, são definidos os demais blocos.

O resultado da aplicação deste processo à matriz em banda da Figura G-2 está ilustrado na Figura G-3.

Figura G-3 Representação da matriz em banda da figura G.2 na forma de Bloco Tridiagonal

As operações algébricas com matrizes tridiagonal de blocos são essencialmente as mesmas que uma matriz tridiagonal, considerando-se submatrizes no lugar de coeficientes escalares. A representação de uma matriz em banda como matriz tridiagonal simplifica a expressão de certos algoritmos numéricos.

G.12. MATRIZ DIAGONAL ESTRITAMENTE DOMINANTE (DD)

Uma matriz A é diagonal dominante se cada elemento a_{ii} da diagonal é maior em módulo que a soma dos módulos de todos os outros elementos da linha i , isto é:

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}| \text{ para } i = 1, 2, \dots, n \quad (\text{A-1})$$

G.13. MATRIZ SIMÉTRICA POSITIVA DEFINIDA (SPD)

Uma matriz $A^{n \times n} \in R^{n \times n}$ satisfazendo $A = A^T$ e $x^T A x > 0$ para $x \in R^n$ e $x \neq 0$ é denominada simétrica positiva definida. Esta condição garante a convergência dos principais algoritmos iterativos [127]. Embora determinar se uma matriz é SPD, com base apenas na definição acima seja computacionalmente inviável, é possível demonstrar que uma matriz simétrica diagonal dominante com diagonal real positiva é SPD [128].

G.14. MÉTODOS DIRETOS PARA SOLUÇÃO DE SISTEMAS DE EQUAÇÕES LINEARES

Método numérico que, na ausência de erros de arredondamento, determina a solução exata do sistema em um número finito de passos previamente conhecido. Os métodos diretos, em sua maioria, transformam o sistema original num sistema triangular equivalente.

G.15. MÉTODOS ITERATIVOS PARA SOLUÇÃO DE SISTEMAS DE EQUAÇÕES LINEARES

O termo método iterativo refere-se a uma vasta gama de técnicas que usam aproximações sucessivas para obter soluções mais precisas para um sistema de equações lineares em cada etapa. Métodos iterativos que podem ser expressos na forma simples:

$$x^{(k+1)} = Cx^{(k)} + d \quad (\text{A-2})$$

quando nem C nem d dependem do passo de iteração k , o método é denominado estacionário [127].

Nos métodos iterativos, ao contrário dos métodos diretos, o tempo para solução dos sistemas lineares depende das propriedades

numéricas da matriz de coeficientes. Isto os torna potencialmente mais rápido que os métodos diretos, mas também potencialmente inadequados para certos sistemas[129].

G.16. Nó (*NODE*)

Coleção (física) de CPU's, memória e interfaces com outros nós e dispositivos. O processamento utilizando apenas a memória do nó é denominado "processamento em memória compartilhada" - Shared Memory Processing (SMP). É usual, também, denominar o próprio nó de SMP. A figura G.4 ilustra uma configuração para processamento distribuído com dois nós SMP.

Figura G-4 Esquema simplificado de *cluster* contendo dois nós SMP com acesso uniforme à memória

Em alguns contextos SMP pode significar Multi-Processamento Simétrico (Symmetric Multi-Processing). Para nós, como ilustrados na figura G-4, o acesso à memória é simétrico, isto é, igual para todas as CPU's do nó, neste caso as duas definições são equivalentes. No entanto, para nós com um grande número de CPU's, é mais comum que o acesso à memória compartilhada do nó não seja uniforme, como ilustrado na Figura G-5.

Figura G-5 Esquema de nó com com acesso não uniforme à memória – Non Uniforme Memory Access (NUMA)

Nesta configuração, a CPU1 e a CPU2 tem acesso direto à memória1, mas necessitam de um canal de comunicação adicional para acesso à memória. Analogamente, o acesso à memória do nó pelos outros dois nós da configuração também não é simétrica. O espaço de endereçamento corresponde à união das duas memórias, mas o tempo de acesso é diferente para cada uma delas. Esta assimetria influencia a escalabilidade quando todas as CPU's do nó são utilizadas no processamento em memória compartilhada.

G.17. NÚCLEO (CORE)

Unidade de hardware que realiza operações aritméticas, uma CPU moderna tem usualmente muitos núcleos (multicore).

G.18. NÚMERO DE CONDICIONAMENTO DE UMA MATRIZ (*CONDITION NUMBER*)

O número de condicionamento é definido como:

$$\kappa(A) = \|A\|/\|A^{-1}\| \quad (\text{A-3})$$

Se $\|\cdot\| = \|\cdot\|_2$, então $\kappa(A)$ pode ser também definido como:

$$\kappa(A) = |\lambda_{\max}(A)/\lambda_{\min}(A)| \quad (\text{A-4})$$

onde $\lambda_{\max}(A)$ e $\lambda_{\min}(A)$ são o maior e menor autovalores A de respectivamente.

G.19. **PROCESSAMENTO PARALELO MASSIVO (MASSIVELY PARALLEL PROCESSING – MPP)**

Tipo de computação em que o processamento é realizado por muitas CPU's, usualmente milhares, trabalhando em paralelo para executar um único programa. As CPU's realizam simultaneamente uma combinação de várias instâncias de instruções e dados de forma que o resultado pode ser obtido de forma muito mais eficiente e rápida. A arquitetura do hardware e do software para gerenciamento do paralelismo é específico de cada fabricante.

G.20. **SMP (SHARED MEMORY PROCESSING OU SYMMETRIC MULTIPROCESSING)**

Arquitetura de hardware consistindo de uma coleção de processadores que compartilham um banco (pool) de memória, com acesso aproximadamente homogêneo (ou simétrico). usando um espaço único de endereçamento.

G.21. **SOLVER**

Programa de computador que implementa um algoritmo para solução numérica de sistemas de equações lineares.

G.22. **SPEEDUP**

$S(n, p)$, é a razão do tempo de execução do algoritmo serial $T(n)$ pelo tempo de execução do algoritmo paralelo:

$$S(n, p) = T(n)/T(n, p) \quad (\text{A-5})$$

onde n é a dimensão do problema e p o número de processadores.

Em geral $0 < S(n, p) \leq p$. Denomina-se *speedup* linear ou ideal,

quando $S(n, p) = p$. O coeficiente $E(n) = S(n, p)/p$, denominado Eficiência da Paralelização (*Parallel Efficiency*), é comumente utilizado para medir com que eficiência os processadores adicionais são utilizados. Programas paralelos às vezes podem atingir uma aceleração superlinear devido ao uso mais eficiente de *cache* de memória pelo sistema operacional de cada nó, quando operando com quantidades menores de dados.