UNIVERSIDADE ESTADUAL DE SANTA CRUZ
PRO-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM COMPUTACIONAL EM CIÊNCIA E TECNOLOGIA

VICTOR ROCHA NERES

IMPLEMENTAÇÃO PORTÁVEL DE ALTO DESEMPENHO PARA SIMULAÇÃO DE SISTEMAS DE PARTÍCULAS COM INTERAÇÃO DE LONGO ALCANCE.

PPGMC – UESC

ILHÉUS-BA
2018

VICTOR ROCHA NERES

IMPLEMENTAÇÃO PORTÁVEL DE ALTO DESEMPENHO PARA SIMULAÇÃO DE SISTEMAS DE PARTÍCULAS COM INTERAÇÃO DE LONGO ALCANCE.

PPGMC – UESC

Dissertação apresentada ao Programa de Pós-Graduação em Modelagem Computacional em Ciência e Tecnologia da Universidade Estadual de Santa Cruz, como parte das exigências para obtenção do título de Mestre em Modelagem Computacional em Ciência e Tecnologia.

Orientador: Prof. Dr. Esbel Tomás Valero Orellana

ILHÉUS-BA
2018

VICTOR ROCHA NERES

IMPLEMENTAÇÃO PORTÁVEL DE ALTO DESEMPENHO PARA SIMULAÇÃO DE SISTEMAS DE PARTÍCULAS COM INTERAÇÃO DE LONGO ALCANCE.

PPGMC – UESC

Ilhéus-BA, 06/07/2018

Comissão Examinadora

Prof. Dr. Esbel Tomás Valero Orellana
UESC
(Orientador)

Prof. Dr. Zolacir Trindade de Oliveira Junior
UESC

Prof. Dr. Jose Roberto Steiner de Moura
Orbistec

A minha esposa Christina, a minha filha Maria Fernanda, a minha irmã Rafaella, e aos amigos, pelo tempo que deixamos de estar juntos... Aos meus pais, Carlos e Sônia, e a minha avo Maria, a eles todos os créditos...

Agradecimentos

- A Universidade Estadual de Santa Cruz e ao NBCGIB por conceder toda infraestrutura que possibilitou a realização deste trabalho.
- Ao Departamento de Ciências Exatas e Tecnológicas e ao Colegiado do PPGMC da Universidade Estadual de Santa Cruz, pela oportunidade da realização do Curso.
- A minha equipe de trabalho, sua organização tornou meu desafio de lidar com o tempo possível.
- Ao Prof. Dr. Esbel Tomás Valero Orellana, pela dedicação nas correções, por toda paciência e orientações, pela amizade e pelo apoio neste período de aprendizado.
- Ao Prof. Dr. José Roberto Steiner, Prof. Dr. Marcelo Ossamu Honda, Prof. Dr. Susana Marrero Iglesias e Prof. Dr. Dany Sanchez Dominguez pela preciosa colaboração na minha formação.
- Aos meus colegas de mestrado que tornaram um período de longa dedicaç~ao em algo divertido.
- Aos amigos, em especial, Ciro Sobral, Gabriel Ganen, Matheu Anunciação Batista Dos Santos, Jhonnatan Soares, Anusio Menezes Correia, Ícaro Andrade, Rogério Sousa, Rogério Mattos Rocha, Pedro Paulo Libório, Fernando Quinto, Paulo Américo, Rui Barbosa e Daniel Nascimento pelo incentivo e pelo apoio constantes.
- A Eberty Alves, Prabhát K. Oliveira, Duca, Thiago Messias, Caio Argolo, Carlos Magno, Ian Moreira, do NBCGIB, pelo convívio, pela compreensão e pela amizade.
- Aos professores e funcionários do Departamento de Ciências Exatas e Tecnológicas, pelos ensinamentos e pela convivência durante este período.

“O sucesso é uma consequência e não um objetivo.”

A Sorte do Esforço (Gustave Flaubert)

Implementação portável de alto desempenho para simulação de sistemas de partículas com interação de longo alcance.

PPGMC – UESC

Resumo

O estudo de sistemas de partículas com iterações de longo alcance, é uma área de pesquisa de grande relevância. Simulações numéricas de tais sistemas, demandam um esforço computacional muito grande, devido ao fato de que precisa levar em conta que, cada partícula interage com todas as demais. Uma solução viável para este problema, é a utilização dos chamados toy models.Dentre estes modelos, um dos mais utilizados é o Hamiltonian Mean Field (HMF), que é formado por N partículas distribuídas ao longo de um anel de raio unitário. A realização de simulações que envolvam grande quantidade de partículas é um problema de grande porte, que não pode ser resolvido em um tempo razoável, com os recursos computacionais normalmente disponíveis em computadores pessoais. Por este motivo diversas técnicas de processamento paralelo tem sido utilizadas para implementar simulações para um número cada vez maior de partículas. Uma das mais eficazes tem oferecido excelentes resultados com uma abordagem baseada em SIMD para GPGPUs da Nvidia, utilizando a arquitetura CUDA. Esta solução, a pesar de oferecer resultados significativos, tem a limitação de estar restrita a um tipo de hardware / software específico, não podendo ser portada para outras arquiteturas. Neste projeto propomos a implementação de algoritmos de simulação de sistemas de partículas com iteração de longo alcance baseados em OpenCL. O padrão OpenCL oferece recursos para implementações paralelas, baseadas em modelos SIMD, altamente portáveis, que rodam com poucas ou nenhuma modificação em diversas arquiteturas. A implementação é comparada, enquanto aos resultados e a seu desempenho, com outras implementações tradicionais, para arquiteturas baseadas em CPUs, e com as soluções já existentes desenvolvidas para GPGPUs com a arquitetura CUDA.

Palavras-chave: OpenCL, Processamento Paralelo, SIMD, HMF.

Implementação portável de alto desempenho para simulação de sistemas de partículas com interação de longo alcance.

PPGMC - UESC

Abstract

Translation of the abstract into english, possibly adapting or slightly changing the text in order to adjust it to the grammar of english educated.

Keywords: latex. abntex. template.

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

UESC	Universidade Estadual de Santa Cruz
DCET	Departamento de Ciências Exatas e Tecnológicas
PPGMC	Programa de Pós-Graduação em Modelagem Computacional em Ciência e Tecnologia
HMF	*Hamiltonian Mean Field*
GPGPU	*General Purpose Graphics Processing Unit*
CPU	*Central Processing Unit*
SIMD	*Single Instruction Multiple Data*
CUDA	*Compute Unified Device Architecture*
OpenMP	*Open Multi-Processing*
MPI	*Message Passing Interface*
OpenCL	*Open Computing Language*
FPGA	*Field Programmable Gate Array*
UC	Unidade de Controle
ULA	Unidade Lógica e Aritmética
SISD	*Single Instruction, Single Data*
SIMD	*Single Instruction, Multiple Data*
MISD	*Multiple Instruction, Single Data*
MIMD	*Multiple Instruction, Multiple Data*
API	*Application Programming Interface*
ARB	*Architecture Review Board*

Lista de símbolos

	Letra grega Gama
λ	Comprimento de onda
2	Pertence

Sumário

1 Introdução

Sistemas físicos são constituídos por elementos que interagem entre si. Se a interação é de curto alcance, cada elemento é sensível apenas à sua vizinhança, diferente das interações de longo alcance, onde cada elemento interage com todos os outros componentes do sistema, isto é, o potencial de interação varia de acordo com a equação 1

$$r^{-\alpha} \qquad com \qquad (0 < \alpha < d) \tag{1}$$

onde r é a distância entre as partículas e d é a dimensão do espaço. Alguns exemplos deste tipo de sistemas são sistemas gravitacionais, plasmas carregados (sistemas Coulombianos), sistemas de vórtices bidimensionais e condensados de Bose-Einstein (SCIENCE, 2010).

A simulação de sistemas com interação de logo alcance é um processo muito custoso. Estudos prévios apontam que o tempo de processamento requerido para este tipo de simulação é muito alto, já que cada uma das N partículas interage com as outras em um determinado domínio. Desta forma, quanto maior o número N de partículas, mais custosa será a simulação. Para solucionar problemas deste tipo, devemos recorrer a modelos simplificados conhecidos como *toy models* que auxiliam na compressão de sistemas de iteração de longo alcance, retendo aspectos importantes de sistemas reais e permitindo uma descrição detalhada de seu comportamento dinâmico e estático. Os modelos de campo médio são exemplos de *toy model* e a aplicação destes modelos reduz drasticamente a complexidade dos sistemas viabilizando a simulação computacional dos mesmo. O modelo que foi utilizados neste trabalho é o *Hamiltonian Mean Field* (*HMF*) (CHAVANIS et al., 2005).

Recentemente, a evolução da tecnologia tornou possível obter resultados cada vez melhores em diversas áreas da ciência e das engenharias. Os estudos dos sistemas de interação de longo alcance avançaram bastante, pois as simulações agora podem ser feitas com um número muito maior de partículas. No entanto, vários problemas dessa natureza ainda continuam em aberto (DAUXOIS et al., 2002b).

Existem diversas abordagem para se resolver um problema de interação de longo alcance utilizando implementações paralelas, entretanto, a que tem apresentado melhores resultados são as implementações baseadas de Unidade de Processamento Gráfico de Propósito Geral ou *GPGPU* (*General Purpose Graphics Processing Unit*) via modelo baseado em controle de fluxo com instruções únicas e múltiplos dados ou *SIMD* (*Single Instruction Multiple Data*)](FLYNN, 1972).

Neste trabalho usamos como base para nossas implementações, um código

serial que realiza as simulações de sistemas de iteração com longo alcance baseada no modelo *HMF* e um código baseado na arquitetura *CUDA* para *GPGPUs* da *Nvidia*. Com o objetivo de conhecer e otimizar o problema, foi feito um estudo e analise de pontos criticos da versão serial do código computacional. Partindo desta análise inicial, realizamos a implementação de versões paralelas baseadas em arquiteturas de memória compartilhada, utilizando *OpenMP*, e de memoria distribuída utilizando *MPI*. Foi feita também uma versão híbrida que utiliza dessa duas abordagens em conjunto. O objetivo destas implementações foi avaliar o comportamento do programa quando paralelizamos suas regiões criticas e variamos os parâmetros de entrada, assim como a escalabilidade do problema quando alteramos a quantidade de recursos. Em seguida, foi feito um estudo e otimização da implementação paralela baseada na arquitetura *Nvidia CUDA*, que foi utilizada como base para a implementação de uma versão baseada no padrão *OpenCL*.

O código em *CUDA*, fornece resultados satisfatórios no que diz respeito a métricas como *speedup*. No entanto, essa implementação, apesar de ser eficiente, possui limitações à níveis de hardware por permitir apenas execução em *GPGPUs* da *Nvidia*. Desta forma a principal motivações que levaram a elaboração deste trabalho foi construir uma implementação portável, baseada em *OpenCL*, que posa ser executada em diversas arquiteturas de diferentes fornecedores de hardware, tanto baseadas em *CPUs* como *GPGPUs*.

1.1 Objetivos

1.1.1 Objetivo Geral

Desenvolver uma implementação portável, baseada em *OpenCL*, para simulação de sistemas de partículas com interação de forças de longo alcance.

1.1.2 Objetivos Específicos

Avaliar a escalabilidade do modelo *HMF* utilizando implementações tradicionais para arquiteturas baseadas em *CPUs*

Estudar e avaliar as implementações já existentes que utilizam a abordagem *SIMD*, baseadas em *CUDA*;

Portar os recursos já desenvolvidos de *CUDA* para *OpenCL* visando obter uma implementação portável;

Definir um conjunto de métricas que permitam avaliar a nova implementação;

Comparar a nova implementação, em termos de resultados e desempenho, com outras implementações.

1.2 Estrutura do texto

Esta dissertação esta organizada da seguinte maneira: no capítulo 2 fazemos uma revisão bibliográfica apresentando os sistemas de interação de longo alcance e suas características, são introduzidos os *toy models Hamiltonian Mean Field* e o *ring model*, assim como os integradores simpléticos. Apresentamos também as ferramentas de computação de alto desempenho e programação paralela, em *CPUs* e *GPGPUs* utilizadas para viabilizar a simulação de tais sistemas. No capítulo 3, oferecemos uma descrição dos materiais e métodos utilizados para analisar a versão sequencial e construir as versões paralelas do modelo *HMF*, principal objetivo deste trabalho, mostramos ainda o desenho dos experimentos realizados para validar as implementações. O capítulo 4 apresenta os resultados numéricos para os experimentos sequenciais e gráficos que ilustram os resultados e o desempenho computacional das versões paralelas implementadas neste trabalho. Por último, no capítulo 5 oferecemos as conclusões da pesquisa e sugerimos algumas linhas de trabalhos futuros.

2 Revisão Bibliográfica

Neste capítulo, abordamos os fundamentos dos Sistemas de Interação de Longo Alcance, apresentaremos o *toy model Hamiltonian Mean Field* (*HMF*), as definições e equações do modelo matemático. São apresentados também as ferramentas de computação de alto desempenho e programação paralela, em *CPUs* e *GPGPUs*, empregadas para a simulação usando este modelo quando aplicando-o para uma grande quantidade de partículas, em diferentes plataformas de *hardware*, buscando extrair maior eficiência de cada uma delas.

2.1 Sistemas de Interações de Longo Alcance

As propriedades de sistemas de partículas com interações de longo alcance são, em grande parte, pouco compreendidas, apesar de abranger um grande conjunto de problemas de física (DAUXOIS et al., 2002b). O crescente avanço da tecnologia, proporcionou novas ferramentas e metodologias que possibilitaram uma abordagem mais completa no estudo destes sistemas, o que revelou sua importância em uma perspectiva multidisciplinar abrangendo áreas como astrofísica (CHALONY et al., 2013), física nuclear (CHOMAZ; GULMINELLI, 2002), física de plasmas (ELSKENS; ESCANDE, 2002), condensados de Bose-Einstein, *clusters* atômicos (KIRKPATRICK, 1981) e hidrodinâmica (MILLER, 1990). A presença em muitas disciplinas da física justifica a necessidade de uma compreensão geral e interdisciplinar dos problemas físicos e matemáticos levantados por sistemas interativos de longo alcance. Estes sistemas físicos são estudados por uma variedade de métodos teóricos e numéricos, mas a sua descrição em termos da mecânica estatística e teoria cinética permanece um desafio aberto. Há recentemente, diversas atividades neste campo, uma vez que foi percebido que alguns modelos simplificados podem ser resolvidos exatamente em diferentes conjuntos (microcanônico, canônica, grande canônico, etc.).

Sistemas interação de longo alcance podem ser extensivos, no entanto, eles não são essencialmente aditivos, isto é, a soma das energias dos subsistemas macroscópicos não é igual à energia de todo o sistema. Outra característica importante destes sistemas é a falta de equivalência de *emsebles*, ou seja, se dividirmos o sistema em dois conjuntos esses podem ser desiguais (ou não equivalentes) (CHAVANIS, 2006; HERTEL; THIRRING, 1971; KIESSLING; LEBOWITZ, 1997; THIRRING, 1970). A desigualdade de *emsebles* é a causa de, provavelmente, uma das características mais marcantes dos sistemas de longo alcance que é a possibilidade de exibir calor específico negativo no conjunto microcanâmico (ANTONOV, 1962; Eddington, 1926; HERTEL; THIRRING,

1971; Lynden-Bell; Wood, 1968; THIRRING, 1970). O calor específico é sempre positivo no conjunto canônico, independentemente da natureza das interações, uma vez que é dado pelo valor esperado de uma quantidade positiva. Acontece que o equilíbrio microcanônico contém todas as informações sobre o equilíbrio canônico, enquanto o inverso está errado no caso de uma desigualdade de conjunto (ELLIS et al., 2004; GROSS; KENNEY, 2005).

Nas próximas subseções desta sessão vamos abordar algumas propriedades como extensividade e aditividade que serão uteis para diferenciar sistemas de interação de curto alcance dos de longo alcance, em seguida, será apresentada uma definição de interação de longo alcance.

2.1.1 Aditividade e Extensividade

De fato, é muito importante, dar as definições gerais de extensividade e aditividade e esclarecer a distinção entre esses dois conceitos. É conveniente considerar primeiro a situação que se encontra em sistemas de curto alcance. Podemos imaginar dividir um sistema em equilíbrio em duas partes ocupando volumes iguais. Algumas variáveis termodinâmicas de cada metade do sistema serão iguais às correspondentes do sistema total, outras serão reduzidas à metade. Temperatura e pressão são exemplos do primeiro tipo de variáveis termodinâmicas: elas não dependem do tamanho do sistema e são chamadas de variáveis intensivas. Energia, entropia e energia livre são variáveis do segundo tipo, seu valor é proporcional ao tamanho do sistema, ou seja, ao número de constituintes elementares e elas são chamadas de variáveis extensivas. A propriedade de que as variáveis termodinâmicas são proporcionais ao tamanho do sistema é chamada de extensividade, e os sistemas com essa propriedade são chamados extensivos (CAMPA et al., 2009). O valor específico de variáveis extensivas (por exemplo, a energia por unidade de partícula, ou por unidade de massa, ou por unidade de volume) dá origem a novas quantidades intensivas.

Considerando a energia de um sistema, vemos que ele também tem a propriedade de aditividade, que consiste no seguinte. Dividindo os sistemas em duas partes macroscópicas, a energia total E será igual a E_1 + E_2 + E_{int}, com E_i a energia da i-ésima parte e E_{int} a energia de interação entre as duas partes. No limite termodinâmico, a razão $E_{int}/(E_1 + E_2)$ tende a zero, portanto, neste limite $E \quad E_1 + E_2$. Essa propriedade é chamada de aditividade, sistemas com essa propriedade (para a energia, bem como para outras quantidades dependentes do tamanho) são chamados aditivos. É evidente que a extensividade e a aditividade estão relacionadas. De fato, na definição de extensividade apenas dada, não poderíamos ter concluído que a energia de cada parte do sistema é metade da energia total, se a interação energia E_{int} não fosse insignificante. A aditividade implica extensividade (portanto, não extensiva implica não aditividade),

mas não o inverso. Isto se aplica mais genericamente a todos os sistemas de longo alcance. As propriedades incomuns desses sistemas derivam da falta de aditividade (CAMPA et al., 2009).

2.1.2 Definição de Sistemas de Longo Alcance

Para definir a propriedade da interação que torna um sistema de curto ou de longo alcance, devemos considerar sistemas onde o potencial de interação é dado pela soma do potencial invariante translacional de pares de corpos que são constituintes elementares do sistema (CAMPA et al., 2009). Vamos considerar o potencial de interação entre dois corpos como:

$$V(r) = \frac{J}{r^{\alpha}} \tag{2}$$

onde r é o módulo da distâncias entre as partículas e J é a constante de acoplamento. Em distâncias r suficientemente grandes, o valor absoluto do potencial de dois corpos é limitado por $r^{-\alpha}$. Se a potência positiva α for maior que a dimensão d do espaço onde o sistema está embutido, $\alpha > d$, definimos o sistema como sendo de curto alcance. Caso contrário, se $\alpha \leq d$, o sistema é de longo alcance (CAMPA et al., 2009).

Outra maneira de caracterizar o alcance das interações é através do comportamento da densidade de energia ε. Se ela escala super linearmente com o volume com uma densidade constante, a interação é dita de longo alcance, pois isto implica que o potencial não decai suficientemente rápido, violando a extensividade (SCIENCE, 2010).

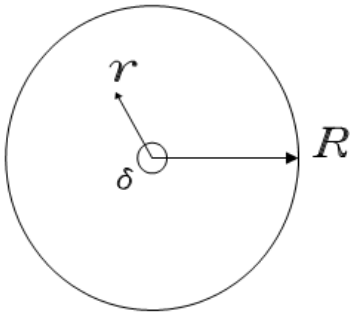
Podemos estimar a energia ε considerando uma dada partícula colocada no centro de uma esfera de raio R onde as outras partículas são distribuídas homogeneamente (SCIENCE, 2010). Vamos excluir a contribuição para ε vindo das partículas localizadas em uma pequena vizinhança de raio δ (veja a Figura 1). Isso é motivado pela necessidade de regularizar a divergência do potencial a pequenas distâncias, o que nada tem a ver com sua natureza de longo alcance. Se uma dada partícula interage com outras através de um potencial que decai a grandes distâncias como $\frac{1}{r}$, nós obtemos em d-dimensões:

$$\varepsilon = \int_{\delta}^{R} d^d r \rho \frac{J}{r^{\alpha}} = \rho J \ \Omega_d \int_{\delta}^{R} r^{d-1-\alpha} dr = \frac{\rho J \ \Omega_d}{d-\alpha} \left[R^{d-\alpha} - \delta^{d-\alpha} \right], \qquad se \ \alpha \neq d, \tag{3}$$

onde ρ é a densidade genérica, J é a constante de acoplamento e Ω_d é o volume angular na dimensão *d* (2π em $d = 2$, 4π em $d = 3$, etc).

Ao aumentar o raio R, a energia ε permanece finita se e somente se $\alpha > d$. Isto implica que a energia total E aumentará linearmente com o volume V, ou seja, o sistema é extenso. Tais interações são as de curto alcance usuais. Pelo contrário, se da

Figura 1 – Quadro esquemático do domínio considerado para a avaliação da energia ε de uma partícula. Isto é, uma casca esférica de raio externo R e raio interno δ.



energia ε cresce com o volume $V^{1-\alpha/d}$ (logaritmicamente, no caso marginal $\alpha = d$). Isto implica que a energia total E aumentará super linearmente, $E / V^{2-\alpha/d}$, com volume. Casos onde a energia cresce super linearmente definem a natureza de longo alcance da interação (SCIENCE, 2010). No entanto, como mostramos para sistemas de campo médio, o fato de a energia poder ser extensa não implica que o sistema seja aditivo.

2.2 Modelo Hamiltonian Mean Field (HMF)

O *Hamiltonian Mean Field* (HMF), é um modelo de campo médio cujo potencial mantém apenas o primeiro modo de expansão de *Fourier* do potencial de modelos unidimensionais gravitacionais e de folha carregada (ANTONI; RUFFO, 1995; CHAVANIS et al., 2005). O modelo HMF tem sido extensivamente estudado por mais de uma década. A interação simples de campo médio nos permite realizar cálculos analíticos, mas mantém várias características complexas de interações de longo alcance.

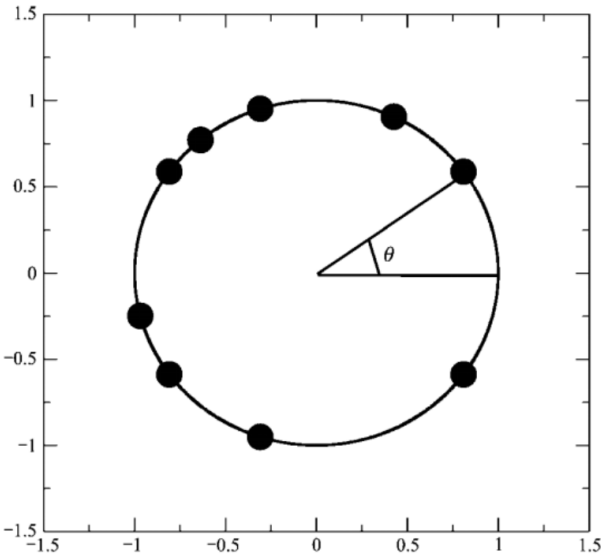
O modelo HMF é definido pelo seguinte Hamiltoniano (DAUXOIS et al., 2002a):

$$H_N = \sum_{i=1}^{N} \frac{\rho_i^2}{2} + \frac{J}{2N} \sum_{i,j} [1 - cos(\theta_i - \theta_j)] \tag{4}$$

onde $\theta_i \in [0, 2\pi]$ é a posição (ângulo) da i-ésima partícula de massa unitária num círculo e ρ_i o momento conjugado correspondente. Este sistema pode ser representando como um conjunto partículas movendo-se em um círculo unitário interagindo através de um potencial cosseno de alcance infinito (como mostra a figura 2) atrativo ($J > 0$) ou repulsivo ($J < 0$) ou, alternativamente, como rotores XY clássicos com alcance ferromagnética infinito ($J > 0$) ou acoplamentos antiferromagnéticos ($J < 0$).

O modelo surgiu ao longo dos anos como um modelo prototípico para estudar e elucidar as muitas características peculiares resultantes de interações de longo alcance. Para monitorar a evolução do sistema, é necessário calcular as equações (7, 6) onde θ é a posição da partícula e ρ é o momento.

Figura 2 – Ring Model.



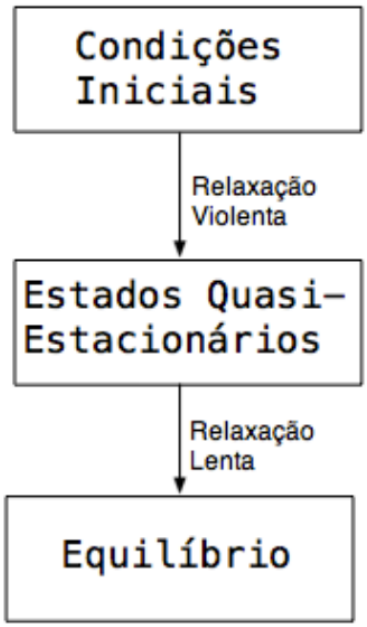
$$\dot{\theta}_i = \frac{\partial H}{\partial \rho_i} = \rho_i \tag{5}$$

$$\dot{\rho}_i = -\frac{\partial H}{\partial \theta_i} = -\frac{J}{N}\sum_{j=1}^{N} sin(\theta_i - \theta_j) \tag{6}$$

Quando fazemos uma simulação deste tipo de sistema podemos observar o seu comportamento com o passar do tempo, partimos das condições iniciais onde colocamos várias partículas distribuídas homogeneamente no sistema de anel, em seguida estas partículas vão começar a interagir, passando por um período de relaxação violenta, atingindo assim um estado conhecido com quase estacionário, depois elas passaram por uma relaxação lenta até que atinja o equilíbrio térmico, também conhecido como equilíbrio de Boltzman, a figura 3. O objetivo das simulações realizadas neste trabalho, é verificar a evolução temporal do sistema de partículas aplicando o *toy model* HMF e utilizando um integrador numérico para solucionar as equações de movimento.

O HMF, é um sistema para o qual o esforço computacional cresce com o número de partículas, N. Além disso, ao executar uma simulação os dados são coletados a cada instante de tempo, de acordo com o passo definido. Isto é, quanto menor o tamanho do passo de tempo, mais simulações serão realizadas o que implica em uma quantidade maior de iterações.

Figura 3 – Evolução de sistemas de interação de longo alcance.



2.3 Integrador Simplético

Em matemática, um integrador simplético é um esquema de integração numérica para sistemas hamiltonianos (TAO, 2016). Integradores simpléticos formam a subclasse de integradores geométricos que, por definição, são transformações canônicas. Eles são amplamente utilizados em dinâmica não linear, dinâmica molecular, métodos de elementos discretos, física de aceleradores, física de plasma, física quântica e mecânica celeste (RUTH, 1983; FOREST, 2006). Integradores simpléticos são projetados para a solução numérica das equações de Hamilton do tipo:

$$\dot{\rho} = \frac{\partial H}{\partial \theta} \qquad e \qquad \dot{\theta} = \frac{\partial H}{\partial \rho}, \tag{7}$$

onde θ denota as coordenadas da posição, ρ as coordenadas do momento, e H é o hamiltoniano. O conjunto de coordenadas de posição e momento (θ, ρ) são chamados de coordenadas canônicas (ABRAHAM; MARSDEN, 2008; VOGTMANN et al., 1997).

A evolução temporal das equações de Hamilton é um simplectomorfismo, o que significa que conserva a forma simplética de duas formas $(d\rho \wedge d\theta)$ (YOSHIDA, 1990). Um esquema numérico é um integrador simplético se também conservar essa forma dupla. Os integradores simplécticos possuem, como quantidade conservada, um hamiltoniano ligeiramente perturbado em relação ao original. Em virtude dessas vantagens, este esquema tem sido amplamente aplicado aos cálculos de evolução a longo prazo de sistemas Hamiltonianos caóticos, desde o problema de Kepler até as simulações clássicas e semi-clássicas na dinâmica molecular.

Os integradores simpléticos são ferramentas muito úteis para a modelagem aproximada de sistemas dinâmicos por intervalos de tempo muito longos (NOGUEIRA, 2009). Eles se comportam como simuladores que conseguem reproduzir muito bem as características dinâmicas globais de um dado sistema: pontos de equilíbrio, regiões de regularidade e caoticidade no espaço de fase, escalas de tempo de difusão e de instabilidade global, etc. (NOGUEIRA, 2009). Este tipo de integrador possui duas vantagens sobre outros integradores de N-corpos: eles não acumulam os erros do cálculo da energia e são muito mais rápidos do que os outros algoritmos convencionais. Nas aplicações utilizadas como base neste trabalho foi implementado o integrador simplético de Yoshida (YOSHIDA, 1990), para solucionar o modelo HFM. Integradores simpléticos são, por construção, as ferramentas mais apropriadas para integrar numericamente as equações de movimento de sistemas Hamiltonianos.

2.4 Processamento Paralelo

Os ambientes de computação de hoje estão cada vez mais diversificados, explorando a capacidade de uma série de micro processadores multi-core, unidades centrais de processamento (*CPUs*), processadores de sinal digital, hardware reconfigurável (*FPGAs*) e unidades de processamento gráfico (*GPUs*). Apresentado com muita heterogeneidade, o processo de desenvolvimento de software eficiente para uma grande conjunto de arquiteturas, o que traz uma série de desafios para a comunidade de programação (PACHECO, 2011).

As aplicações possuem diferentes comportamentos de acordo com sua carga de trabalho, que vão desde controle intensivo (pesquisa, ordenação e análise) até dados intensivos (processamento de imagem, simulação e modelagem, e mineração de dados). As aplicações também podem ser caracterizadas como de computação intensiva (métodos iterativos, métodos numéricos e modelagem financeira), onde o rendimento global da aplicação é fortemente dependente da eficiência do hardware computacional (PACHECO, 2011). Cada uma dessas classes de carga de trabalho é normalmente executada de forma mais eficiente em um estilo específico de arquitetura de hardware. Não existe um único arquitetura que é melhor para a execução de todas as classes de cargas de trabalho, e a maioria das aplicações possuem características de carga de trabalho mista. Por exemplo, aplicações de controle intensivo tendem a executar mais rápido em *CPUs* superescalares, enquanto que as aplicações de dados intensivos tendem a executar rápido em arquiteturas de vetor, onde a mesma operação é aplicada a múltiplos itens de dados simultaneamente (FLYNN, 1972).

Para escrever programas paralelos eficientes, precisamos de algum conhecimento do hardware subjacente e do software do sistema. Também é muito útil ter algum

conhecimento sobre diferentes tipos de software paralelo. *Hardware* e *software* paralelos cresceram a partir de hardware e software serial convencional os que executam um único trabalho de cada vez (TOBERGTE; CURTIS, 2013). Então, para entender melhor o estado atual dos sistemas paralelos, precisamos saber alguns aspectos dos sistemas seriais.

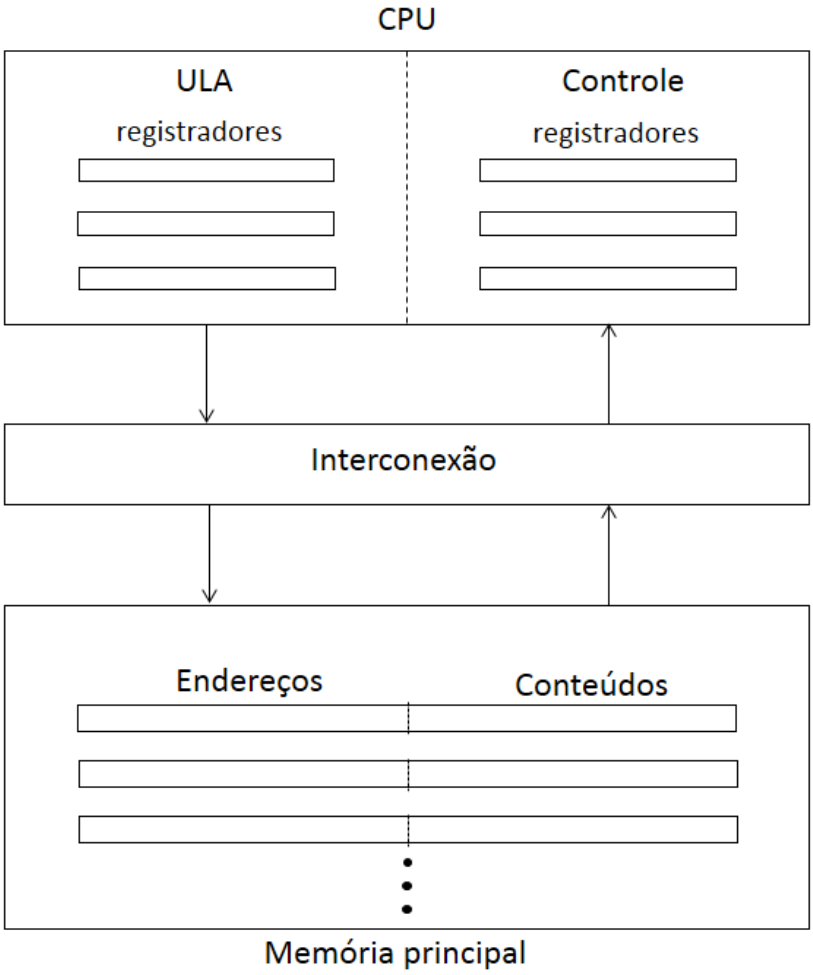
2.4.1 Arquitetura de Von Neumann

A arquitetura "clássica" Von Neumann é composta por memória principal, uma unidade central de processamento (*CPU*) ou processador ou núcleo, e uma interconexão entre a memória e a *CPU*. A memória principal contém uma coleção de locais, cada um capaz de armazenar instruções e dados. Cada local consiste em um endereço, que é usado para acessar este local e o seu conteúdo que são as instruções ou os dados armazenados no local.

A unidade central de processamento é dividida em uma unidade de controle (UC) e uma unidade lógica e aritmética (ULA). A unidade de controle é responsável por decidir quais instruções em um programa devem ser executadas, e a ULA é responsável pela execução das instruções reais. Os dados na *CPU* e as informações sobre o estado de um programa em execução são armazenados em um em um local especial, muito rápido, os chamados registradores. A unidade de controle tem um registro especial chamado contador de programa. Ele Armazena o endereço da próxima instrução a ser executada. Instruções e dados são transferidos entre a *CPU* e a memória através da interconexão. A interconexão, tem sido tradicionalmente um barramento que consiste de uma coleção de fios paralelos e algum *hardware* controlando o acesso aos fios. Uma máquina de Von Neumann executa uma única instrução por vez, e cada instrução opera apenas com alguns dados. A figura 4 ilustra esta arquitetura.

A separação de memória e *CPU* é frequentemente chamada de gargalo de Von Neumann, uma vez que o barramento determina a taxa na qual instruções e dados podem ser acessados. Os dados e instruções necessárias para executar um programa são efetivamente isolados da *CPU*. Atualmente as *CPUs* são capazes de executar instruções muito mais rápido do que podem buscar itens da memória principal (PACHECO, 2011). A fim de resolver o gargalo de Von Neumann e, mais genericamente, melhorar o desempenho da *CPU*, os engenheiros de computação e os cientistas da computação experimentaram muitas modificações na arquitetura básica de Von Neumann como: armazenamento em cache, memória virtual, paralelismo de nível de instrução como *pipelining* e *multiple issue* e hardware *multithread* (TOBERGTE; CURTIS, 2013).

Figura 4 – Arquitetura de Von Neumann.

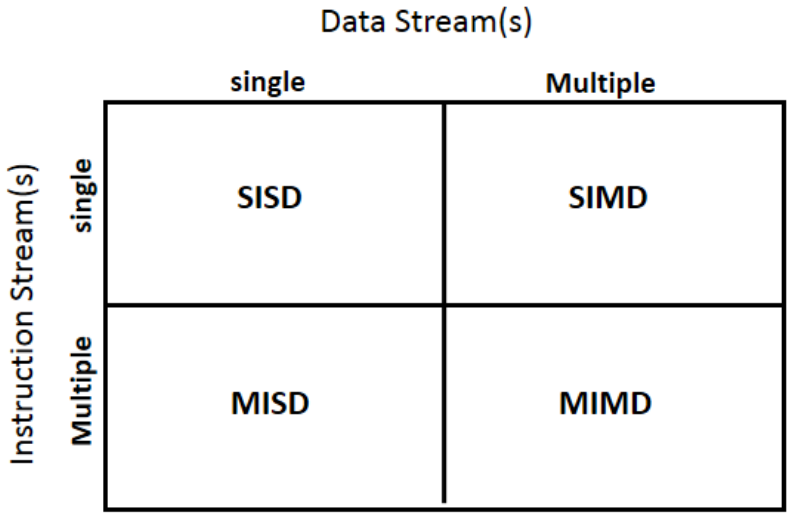


2.4.2 Taxonomia de Flynn

Na computação paralela, a taxonomia de Flynn (FLYNN, 1972) é frequentemente usada para classificar arquiteturas de computadores. Ele classifica um sistema de acordo com o número de fluxos de instruções e o número de fluxos de dados que ele pode gerenciar simultaneamente, veja a figura 5. Um sistema clássico de Von Neumann possui, um único fluxo de instrução e um único fluxo de dados, ele é considerado um sistema do inglês *Single Instruction Stream, Single Data Stream* ou (*SISD*), uma vez que executa uma única instrução de cada vez e pode buscar ou armazenar um item de dados de cada vez.

Os sistemas de instrução única e múltiplos dados do inglês *Single Instruction, Multiple Data* ou (*SIMD*), tipo de sistema utilizado neste trabalho, são sistemas paralelos. Como o nome sugere, os sistemas *SIMD* operam em múltiplos fluxos de dados, aplicando a mesma instrução a vários itens de dados, portanto, um sistema *SIMD* abstrato pode ser considerado como tendo uma única unidade de controle e várias ULAs. Uma instrução é transmitida da unidade de controle para as ULAs, e cada ULA aplica a instrução ao item de dados atual ou está inativa. Em um sistema *SIMD* "clássico",

Figura 5 – A classificação de Flynn Johnson para sistemas de computador.



as ULAs devem operar de maneira síncrona, ou seja, cada ULA deve esperar que a próxima instrução seja transmitida antes de prosseguir . Além disso, as ULAs não têm armazenamento de instruções, portanto, uma ULA não pode atrasar a execução de uma instrução, armazenando-a para execução posterior(PACHECO, 2011).

Os sistemas *SIMD* são ideais para paralelizar laços simples que operam em grandes matrizes de dados. O paralelismo que é obtido dividindo dados entre os processadores e fazendo com que os processadores apliquem (mais ou menos) as mesmas instruções aos seus subconjuntos de dados é chamado de paralelismo de dados(PACHECO, 2011). O paralelismo de *SIMD* pode ser muito eficiente em grandes problemas paralelos de dados, mas os sistemas *SIMD* geralmente não se dão bem em outros tipos de problemas paralelos. Recentemente, unidades de processamento gráfico de proposito geral, ou *GPGPUs* e *CPUs* estão fazendo uso de aspectos da computação *SIMD* (PACHECO, 2011).

Arquiteturas na categoria de *Multiple Instruction, Single Data* ou *MISD*, onde onde muitas unidades funcionais executam operações diferentes sobre os mesmos dados, não encontraram ampla aplicação, mas podem ser visualizadas como *pipelines*[1] generalizadas em que cada estágio executa uma operação relativamente complexa (em oposição a *pipelines* comuns encontrados em processadores modernos, onde cada estágio faz uma operação de nível de instrução muito simples)(QUINN, 2003; TOBERGTE; CURTIS, 2013).

Sistemas de múltiplas instruções, múltiplos dados doo inglês *Multiple Instruction Multiple Data* ou *MIMD*, suportam múltiplos fluxos de instruções simultâneas operando

[1] A segmentação de instruções do inglês *pipeline* é uma técnica hardware que permite que a *CPU* realize a busca de uma ou mais instruções além da próxima a ser executada. Estas instruções são colocadas em uma fila de memória dentro do processador (*CPU*) onde aguardam o momento de serem executadas.

em múltiplos fluxos de dados. Portanto, os sistemas *MIMD* normalmente consistem em uma coleção de unidades de processamento ou núcleos totalmente independentes, cada qual com sua própria unidade de controle e sua própria ULA. Além disso, ao contrário dos sistemas *SIMD*, os sistemas *MIMD* são geralmente assíncronos, isto é, os processadores podem operar em seu próprio ritmo. Em muitos sistemas *MIMD* não há *clock* global, e pode não haver relação entre os tempos do sistema em dois processadores diferentes. De fato, a menos que o programador imponha alguma sincronização, mesmo que os processadores estejam executando exatamente a mesma sequência de instruções, em qualquer instante eles podem estar executando instruções diferentes(PACHECO, 2011).

2.5 Computação de Alto Desempenho

A contínua busca por um maior poder de cálculo e processamento tem sido a força principal no desenvolvimento do hardware de computadores, pois a necessidade de maior poder de computo é sempre crescente. Uma das causas desta necessidade é a resolução de problemas complexos, que nos leva a níveis mais profundos de conhecimentos. Diversa áreas da vida moderna como, as ciências e as engenharias entre outras, impulsionam esta necessidade. Durante vários séculos, quando as ferramentas computacionais ainda não existiam, ou seja, período em que a tecnologia era bastante limitada, o paradigma da ciência consistia em observar, teorizar e comprovar a teoria através da experimentação. Já o paradigma da engenharia constituía-se em projetar, construir e testar protótipos e posteriormente construir o produto final. Com o crescente avanço da tecnologia estes paradigmas mudaram, e atualmente a computação é utilizada como forma de simplificar estas etapas, pois as simulações computacionais são mais baratas e confiáveis do que os métodos utilizados em épocas anteriores. Hoje os ambientes computacionais estão cada vez mais heterogêneos, dessa forma, o processo de desenvolvimento de software eficiente para uma muitas arquiteturas, traz uma série de desafios para a comunidade de programação. As próximas subseções discorrem sobre tecnologias de processamento paralelo em *CPUs* com ambientes de memoria compartilhada e distribuída e em *GPGPUs*.

2.5.1 Abordagem em CPUs com memória compartilhada usando OpenMP

Um programa em execução pode consistir em vários subprogramas que mantêm seu próprio fluxo de controle independente e que estão autorizados a executar concorrentemente. Estas sub-rotinas são definidas como *threads*. A comunicação entre *threads* é realizada através de atualizações e acesso à memória aparecendo no mesmo espaço de endereço. Cada *thread* tem seu próprio conjunto de locais de memória, ou seja, as variáveis. No entanto, todas as *threads* veem o mesmo conjunto de variáveis globais. Cada

thread é beneficiada com uma visão global da memória porque ela compartilha o mesmo espaço de endereçamento de memória do programa principal. *Threads* comunicam-se umas com as outras através da memória global. Isso pode exigir sincronização para garantir que mais de uma *thread* não está atualizando o mesmo endereço global (QUINN, 2003).

Uma característica chave do modelo de memória compartilhada é o fato de o programador não é responsável pela gestão de movimento de dados, embora, dependendo do modelo de consistência implementado no sistema de hardware ou de tempo de execução, um nível de consistência de memória pode ter que ser executada manualmente. Isso diminui os requisitos para especificar explicitamente a comunicação de dados entre as tarefas, e como resultado, o desenvolvimento de código paralelo muitas vezes pode ser simplificado (TOBERGTE; CURTIS, 2013).

Há um custo significativo na manutenção de um modelo de memória compartilhada totalmente consistente em hardware. Para sistemas com múltiplos processadores, a estrutura de hardware requerida para suportar este modelo torna-se um fator limitante. Barramentos compartilhados tornam-se gargalos no modelo. O hardware extra exigido normalmente cresce exponencialmente em termos de sua complexidade à medida que tenta adicionar processadores adicionais. Isso retardou a introdução de sistemas multicore e com múltiplos processadores e tem limitado o número de núcleos trabalhando juntos em um sistema de memória compartilhada consistente para números relativamente baixos, porque os barramentos compartilhados e protocolos de coerência de *overhead* se tornam gargalos (QUINN, 2003).

Um dos modelos de programação em memória compartilhada mais utilizados atualmente é o *Open Multi Processing* (*OpenMP*), que nasceu da cooperação de grandes fabricantes de hardware e software. O *OpenMP* é uma *API* para programação paralela em arquiteturas multiprocessadas, definida para ser utilizada em programas *C/C++* e *Fortran*. Desenvolvido e mantido pelo grupo *OpenMP Architecture Review Board* (ARB), permite a criação de programas paralelos com compartilhamento de memória, através da criação automática e otimizada de um conjunto de *threads* (QUINN, 2003).

O *OpenMP*, permite que o programador simplesmente declare que um bloco de código deve ser executado em paralelo, e a determinação precisa das tarefas e qual encadeamento deve executá-las é deixada para o compilador e o sistema de tempo de execução (PACHECO, 2011).

2.5.2 Abordagem em CPUs com memoria distribuída usando MPI

O modelo de comunicação por passagem de mensagens permite a intercomunicação explícita de um conjunto de tarefas concorrentes que podem utilizar a memória

durante a computação. Várias tarefas podem residir no mesmo dispositivo físico ou através de um número arbitrário de dispositivos. Tarefas trocam dados através de comunicações, enviando e recebendo mensagens explícitas. A transferência de dados geralmente requer operações de cooperação a serem executadas por cada processo. Por exemplo, uma operação de envio deve ter uma operação de recebimento correspondente (QUINN, 2003).

A partir de uma perspectiva de programação, as implementações de transmissão de mensagens comumente compreendem uma biblioteca de rotinas independentes de hardware para enviar e receber mensagens. O programador é responsável por gerenciar explicitamente a comunicação entre as tarefas (QUINN, 2003).

Message Passing Interface (MPI) é atualmente o middleware de troca de mensagens mais popular. O MPI é padrão para comunicação de dados em computação paralela. Neste tipo de programação a coordenação entre processos acontece através da troca explícita de informações, enviando e recebendo mensagens (QUINN, 2003).

2.6 Computação de alto desempenho utilizando GPGPUs

Durante a última década, houve um interesse crescente no uso de unidades de processamento gráfico (GPUs) para aplicações não-gráficas. O uso de GPUs amadureceu a um ponto onde há inúmeras aplicações industriais,a partir dos primeiros artigos acadêmicos, em torno do ano 2000, que provam conceitos teóricos a respeito do assunto. Juntamente com o crescente uso de GPUs, pôde-se observar também um enorme avanço nas linguagens de programação e ferramentas, desenvolvendo técnicas de programação em GPUs, algo que nunca foi tão fácil. Apesar de começar a programar para GPU ser algo simples, a capacidade de utilizar plenamente hardware da GPU é uma arte que pode levar meses ou anos para dominar (BRODTKORB et al., 2013).

Unidades de processamento gráfico (GPUs) ultimamente tem sido utilizada, para o cálculo de propósito geral, chamado de computação GPU ou GPGPU (OWENS et al., 2006). Na época em que os primeiros programas para GPU foram escritos, a GPU era utilizada como uma espécie de calculadora, com várias operações fixas que eram aplicadas para alcançar um resultado. A GPU era projetada para realizar fundamentalmente operações ligadas com gráficos e os primeiros programas GPU foram expressos como operações em primitivas gráficas tais como triângulos. Estes programas eram difíceis de desenvolver, debugar e otimizar. Porém, os programas desenvolvidos como prova de conceito demonstraram que o uso de GPUs poderia prover *speedups* muito maiores que unidades centrais de processamento (CPUs) para determinados algoritmos (OWENS et al., 2008; BRODTKORB et al., 2010) e as pesquisas sobre GPUs levaram ao desenvolvimento das linguagens que abstraem gráficos. Entretanto, essas linguagens

foram abandonadas quando os fornecedores de hardware lançaram linguagens não-gráficas dedicadas que permitiram o uso de GPUs para computação de propósito geral. O sucesso da computação GPU tem sido, em parte, o seu desempenho enorme quando comparado com o CPU. Esta diferença de desempenho tem suas raízes nas restrições físicas dos *cores* e nas diferenças de arquitetura entre os dois processadores. A CPU é em essência um processador serial de Von Neumann e é altamente otimizado para executar uma série de operações em ordem. Um dos principais fatores de desempenho de CPUs tem sido tradicionalmente a sua frequência cada vez maior. Uma vez que o desempenho é diretamente proporcional a frequência, houve uma tendência do crescimento exponencial da frequência na década de 2000. Todavia, este aumento sofreu uma parada brusca (ASANOVIC et al., 2006), pois o consumo de energia de uma CPU é correspondente ao cubo da frequência da mesma (BRODTKORB et al., 2010) e a densidade de potência estava se aproximando a de um reator nuclear (). A incapacidade de esfriar os chips suficientemente fez com que a tendência de crescimento exponencial da frequência parasse pouco abaixo de 4.0 GHz apenas. Juntamente com as restrições de memória e do limite do paralelismo a nível de instrução (ILP – *Instruction-level parallelism*), a computação serial atingiu seu ápice em desempenho (ASANOVIC et al., 2006), e CPUs começaram a aumentar o desempenho através de instruções multi-core e vetoriais.

Ao mesmo tempo, em que as CPUs bateram no teto do desempenho serial, as GPUs foram crescendo exponencialmente no desempenho devido ao enorme paralelismo. Um exemplo desse paralelismo massivo é calcular a cor de um pixel na tela, como esta tarefa pode ser executada independentemente de todos os outros pixels, o paralelismo é uma maneira natural de aumentar o desempenho deste tipo de atividade nas GPUs. O paralelismo parece ser uma forma sustentável de aumentar o desempenho, e há muitas aplicações que apresentam cargas de trabalho embaraçosamente paralelas que são perfeitamente adequados para GPUs (BRODTKORB et al., 2013).

No mercado de PCs existem três grandes fornecedores de GPUs, Intel, AMD e NVIDEA. A Intel é o maior, porém, ela só é dominante no mercado integrado de baixo desempenho. Já a AMD e a NVIDEA, são os únicos dois fornecedores de GPUs para alto desempenho e gráficos discretos.

2.6.1 Abordagem em hardware especifico utilizando CUDA da Nvidia

A Compute Unified Device Architecture (CUDA) é uma plataforma de computação paralela e um modelo de programação desenvolvido pela NVIDIA para computação de propósito geral em unidades de processamento gráfico (GPUs). Com o CUDA, é possível acelerar drasticamente as aplicações computacionais, aproveitando o poder das GPUs. Em programas acelerados pela GPU, a parte sequencial da carga de trabalho é executada na CPU, que é otimizada para desempenho *single-thread*, enquanto a parte

de cálculo intensivo do programa é executada em até milhares de núcleos de GPU em paralelo. O CUDA, pode ser utilizado com linguagens populares como C, C++, Fortran, Python e MATLAB e expressam paralelismo através de extensões na forma de algumas palavras-chave básicas.

As primeiras GPUs foram projetadas como aceleradores gráficas, tornando-se mais programáveis nos anos 90, culminando na primeira GPU da NVIDIA em 1999. Pesquisadores e cientistas rapidamente começaram a aplicar o excelente desempenho de ponto flutuante dessa GPU para computação de propósito geral. Em 2003, uma equipe de pesquisadores liderada por Ian Buck revelou o Brook, o primeiro modelo de programação amplamente adotado para estender C com construções de dados paralelos. Mais tarde, Ian Buck ingressou na NVIDIA e liderou o lançamento do CUDA em 2006, a primeira solução do mundo para computação geral em GPUs.

2.6.2 Abordagem em hardware genérico utilizando OpenCL

O Open Computing Language (OpenCL) é um framework de programação heterogéneo que é gerido pelo consórcio de tecnologia sem fins lucrativos Khronos Group. OpenCL é um framework para desenvolvimento de aplicações que executam em toda em vários tipos de dispositivos feitos por diferentes fornecedores. Ele suporta muitos níveis de paralelismo e mapeia de forma eficiente para sistemas homogêneos ou heterogêneos, simples ou multi-dispositivos que consistem em CPUs, GPUs e outros tipos de dispositivos, limitados apenas pela imaginação dos fornecedores. A especificação OpenCL oferece tanto uma linguagem do lado do dispositivo e uma camada de gerenciamento de *host* para os dispositivos em um sistema (GASTER et al., 2011).

OpenCL proporciona computação paralela utilizando paralelismo com base em tarefas e dados. Ele atualmente suporta CPUs que incluem x86, ARM, e PowerPC, e tem sido adotada em *drivers* da placa gráfica da AMD, Apple, Intel e NVIDIA. O suporte para OpenCL está se expandindo rapidamente, pois vários fornecedores de plataformas estão adotando o OpenCL ou o plano para apoiá-lo em suas plataformas de hardware. Estes vendedores operam dentro de uma ampla gama de segmentos de mercado, desde os fornecedores incorporados (ARM e Imagination Technologies) até os fornecedores de HPC (AMD, Intel, NVIDIA, e IBM). As arquiteturas suportadas incluem CPUs multi-core, taxa de transferência e processadores vetoriais, como GPUs e dispositivos paralelos refinadas, tais como FPGAs (GASTER et al., 2011).

O mais importante do OpenCL multi-plataforma é o amplo suporte da indústria, o que faz com que seja um excelente modelo de programação para que os desenvolvedores aprendam e usem, com a confiança de que ele vai continuar a estar amplamente disponível para os próximos anos, cada vez com maior alcance e aplicabilidade (GASTER et al., 2011).

2.6.3 Desempenho em Programas Paralelos

O objetivo principal ao escrever programas paralelos é geralmente o aumento do desempenho. Para avaliar o desempenho de programas seriais é geralmente utilizada a notação assintótica, que é um método que descrever o comportamento de limites de algoritmos. No entanto, a análises assintóticas não são apropriadas para caracterizar o desempenho de programas paralelos. Analise de desempenho de programas paralelos geralmente é feita utilizando métricas como:

Granularidade,

Fator de balanço de carga,

Speedup,

Eficiência.

Vamos definir p como úmero de processos (processadores) ou *threads* e n como tamanho do problema, para auxiliar na compreensão dos conceitos a seguir. A **granularidade** $G(n)$ expressa a relação entre a quantidade de computação e a quantidade de comunicação em um algoritmo paralelo

$$G(n) = \frac{T_{cp}(n)}{T_{cm}(n)} \tag{8}$$

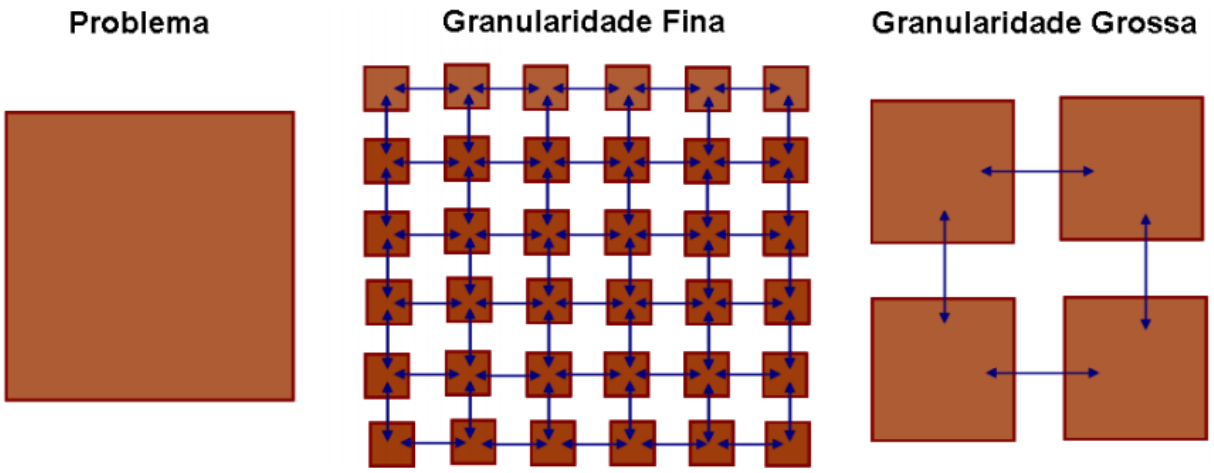
onde T_{cp} é o tempo de computação e T_{cm} o tempo de comunicação. A granularidade de um problema é inversamente proporcional ao volume de paralelização do problema. O paralelismo de granularidade fina, facilita o balanceamento de carga, porém o processamento exigido para a comunicação entre tarefas pode ser maior que o processamento para executá-las, o inverso ocorre no paralelismo de granularidade grossa. O tipo de granularidade a ser utilizada, depende do algoritmo e do ambiente em que o programa será executado. A tabela 1 e a figura 6 ilustram esta relação.

Tabela 1 – Granularidade e volume de paralelização

Nro subproblemas	Computação	Comunicação	Granularidade	Paralelismo
Grande	Pouca	Alta	Fina	Alto
Baixo	Grande	Baixa	Grossa	Baixo

O Balanceamento de carga consiste na divisão da tarefa entre os processadores de forma proporcional ao desempenho e disponibilidade dos recursos que compõem o ambiente de execução, de maneira a utilizar-lo em sua totalidade. O **Fator de balanço**

Figura 6 – Granularidade



de carga ou razão de sincronização F_{LB} é um indicativo que descreve a situação do balanceamento de em uma aplicação.

$$F_{LB}(n) = \frac{T_{max}(n) \quad T_{min}(n)}{T_{max}(n)} \tag{9}$$

onde T_{max} tempo do último processo a terminar, T_{min} tempo do primeiro processo a terminar. Quando F_{LB} tende a 1, a situação é indesejável pois o T_{max} neste caso, é muito maior que o T_{min}, isto significa que o balanceamento de carga não esta adequado e que alguns processadores estão trabalhando mais que outros. Já quando F_{LB} tende a 0, a situação é ótima, T_{max} esta muito próximo de T_{min}, o que caracteriza um balanceamento de carga apropriado.

Normalmente, o melhor que podemos esperar é dividir igualmente o trabalho entre os núcleos e, ao mesmo tempo, não introduzir nenhum trabalho adicional para os núcleos. Se conseguirmos fazer isso, e executarmos nosso programa com p núcleos, uma *thread* ou processo em cada núcleo, nosso programa paralelo será executado p vezes mais rápido que o programa serial. Se chamarmos o T_{serial} de tempo de execução serial e nosso $T_{paralelo}$ de tempo de execução paralelo, o melhor que podemos esperar é $T_{paralelo} = \frac{T_{serial}}{p}$. Quando isso acontece, dizemos que nosso programa paralelo tem *speedup* linear.

Na prática, é improvável que obtenhamos *speedup* linear porque o uso de vários processos / *threads* quase sempre apresenta alguma sobrecarga. Por exemplo, os programas de memória compartilhada quase sempre terão seções críticas, o que exigirá o uso de algum mecanismo de exclusão mútua, como um *mutex*. As chamadas para as funções *mutex* são indiretas que não estão presentes no programa serial, e o uso do *mutex* força o programa paralelo a serializar a execução da seção crítica. Os programas de memória distribuída quase sempre precisarão transmitir dados pela rede, o que

geralmente é muito mais lento que o acesso à memória local. Os programas seriais, por outro lado, não têm essas despesas gerais. Assim, é muito incomum programas paralelos com *speedup* linear. Além disso, é provável que as despesas gerais aumentem à medida que aumentamos o número de processos ou *threads*, ou seja, mais *threads* provavelmente significarão que mais *threads* precisam acessar uma seção crítica. Mais processos provavelmente significarão que mais dados precisam ser transmitidos pela rede. Podemos definir o *speedup* de um programa paralelo como:

$$S(n,p) = \frac{T_{serial}(n)}{T_{paralelo}(n,p)} \tag{10}$$

então o *speedup* linear tem $S = p$, o que é algo incomum pois a maioria das soluções paralelas introduzem alguma sobrecarga produto da distribuição de carga e a comunicação entre processos. Além disso, à medida que p aumenta, esperamos que S se torne uma fração menor e menor do ideal que é a velocidade linear p. Se a sobrecarga da paralelização for muito alta, teremos que $T_{serial}(n) < T_{paralelo}(n,p)\ e\ S(n,p) < 1$ esta situação indesejável é chamada de *slowdown*. Em resumo os *speedups* podem ser:

$S(n,p) < 1$, *slowdown*, situação indesejável;

$1 < S(n,p) < p$, sublinear, comportamento geral;

$S(n,p) = p$, linear, ideal não existe sobrecarga;

$S(n,p) > p$, *supralinear*, situação possível;

Outra maneira de avaliar o desempenho é a relação $\frac{S}{p}$, esse valor é chamado de **eficiência** e consiste na medida de utilização dos processos em um programa paralelo em relação ao programa serial.

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{serial}(n)}{pT_{paralelo}(n,p)} \tag{11}$$

O $T_{paralelo}$, S e E dependem de p, o número de processos ou *threads*. É importante notar também que $T_{paralelo}$, S, E e T_{serial} dependem do tamanho do problema.

3 Metodologia

Para a consecução dos resultados esperados, foi definida uma sequência de passos que permitiram o correto delineamento dos objetivos desejados. Neste capítulo serão abordados todos os materiais, métodos e o desenho dos experimentos usados neste trabalho.

3.1 Materiais

Todos os equipamentos de *hardware* usados para realizar os testes neste trabalho encontram-se fisicamente instalados nas dependências do Núcleo de Biologia Computacional e Gestão de Informações Biotecnológicas (NBCGIB) da Universidade Estadual de Santa Cruz (UESC), dispostos em um único *rack* e unidade de distribuição de energia (*Power Distribution Unit* – PDU). A arquitetura do conjunto de equipamento *BULL-HPC* compõem-se de 23 nós de cálculo ou processamento e 4 servidores:

20 (vinte) nós de processamento, cada um com 02 (dois) processadores *Intel Xeon QuadCore* E5430 – 2.66GHz 13.333 MHz – 12 Mbytes L2 *Cache* por processador – 32 Gbytes RAM (2Gbytes / core), 1 HDD 160 Gbytes (SATA2). Totalizando 160 (cento e sessenta) núcleos (cores) e desempenho estimado de 1.7 foi seguida ponto flutuante.

03 (três) nós de processamento *bullx* B515, cada um com 02 (dois) processadores *Intel* R *Xeon* R E5-2400 – 2.40 GHz 1.333MHz – 15Mbytes L3 *Cache* por processador – 48 Gbytes RAM (4Gbytes / processador), 1 HDD 500 Gbytes. Cada nó de processamento possui 2 (duas) placas gráficas (*Graphics Processing Unit - GPUs*) para processamento de alto desempenho *Nvidia* Tesla K20, cada uma com 5Gbytes RAM e 2.496 processadores *CUDA*. Totalizando 36 núcleos (*cores*) nas *CPUs* e 7.488 núcleos (*cores*) nas *GPUs* e desempenho estimado de 7.3 Teraflops em ponto flutuante. Este módulo de processamento de alto desempenho utilizando placas gráficas foi instalado em março de 2014.

01 servidor de gerenciamento (*Management*) com 02 processadores *Intel Xeon QuadCore* 5.405 8Gbytes RAM – 2HD 160 Gbytes (SATA2)

01 servidor de sistema de arquivos (*File system*) com 02 processadores *Intel Xeon-QuadCore* 5405 12Gbytes RAM – 2HD 160 Gbytes (SATA2)

01 servidor de correio eletrônico (*Mail Server*) com 02 processadores *Intel Xeon-QuadCore* 5405 12Gbytes RAM – 2HD 146 Gbytes (SATA2)

01 servidor de banco de dados (*Database Server*) com 02 processadores *Intel Xeon-QuadCore* 5405 12Gbytes RAM – 2HD 146 Gbytes (SATA2)

01 servidor *Firewall* com 01 Processador *Intel Xeon QuadCore* X3220 – 02 Gbytes RAM – 2HDDs 250 Gbytes (SATA)

01 *Storage EMC Clariion* AX4 – 20 HDD de 400 Gbytes (*SAS*) totalizando 8 Terabytes de espaço em disco:

conectado aos servidores de gerenciamento e de sistema de arquivos através de conexões *FiberChannel* 4 GB / por segundo.

01 *Switch Foundry* FLS624 – 24 portas

01 *Switch INFINIBAND – Voltarie* ISR 9024D-M, 24 portas 4x DDR

Tabela 2 – Configuração da placa *Nvidia* R Tesla R K20

Especificações	Tesla K20
Chip	GK110
Dimensões da GPU	45 mm 45 mm 2397-pin S-FCBGA
Clock do processador	706 MHz
Clock de memória	2.6 GHz
Tamanho da memória	5 GB
Memória I/O	320-bit GDDR5
Configuração de memória	20 x 64M x16 GDDR5 SDRAM
Conectores de exibição	None
Conectores de alimentação	Conectores PCI Express de 8 pinos Conectores PCI Express de 6 pinos
Potência placa	225 W
Potência inativa	25 W
Solução de refrigeração térmica	Heat Sink Passivo

Os softwares utilizado para realizar a analise, otimização e implementação das diferentes versões da aplicação foram os seguintes:

Sistema Operacional - Red Hat Enterprise Linux Server release 6.3

Gprof - GNU gprof version 2.20.51.0.2-5.34.el6 20100205, Baseado em BSD gprof, Copyright (C) 1983 Regents of the University of California.

GCC - 5.4.0, Copyright (C) 2015 *Free Software Foundation*, Inc.

OpenMP - OpenMP Application Program Interface v2.5

MPI - Bullxmpi 1.2.4, Baseado em *Open MPI* 1.6.2.

CUDA - NVCC: *Nvidia* (R) *Cuda compiler driver*, Copyright (C) 2005-2016 *Nvidia Corporation*, *Built on Sun Sep* 4 22:14:01 CDT 2016, *Cuda compilation tools*, *release* 8.0, V8.0.44

OpenCL - 1.2

3.2 Métodos

3.2.1 Analise da implementação Sequencial

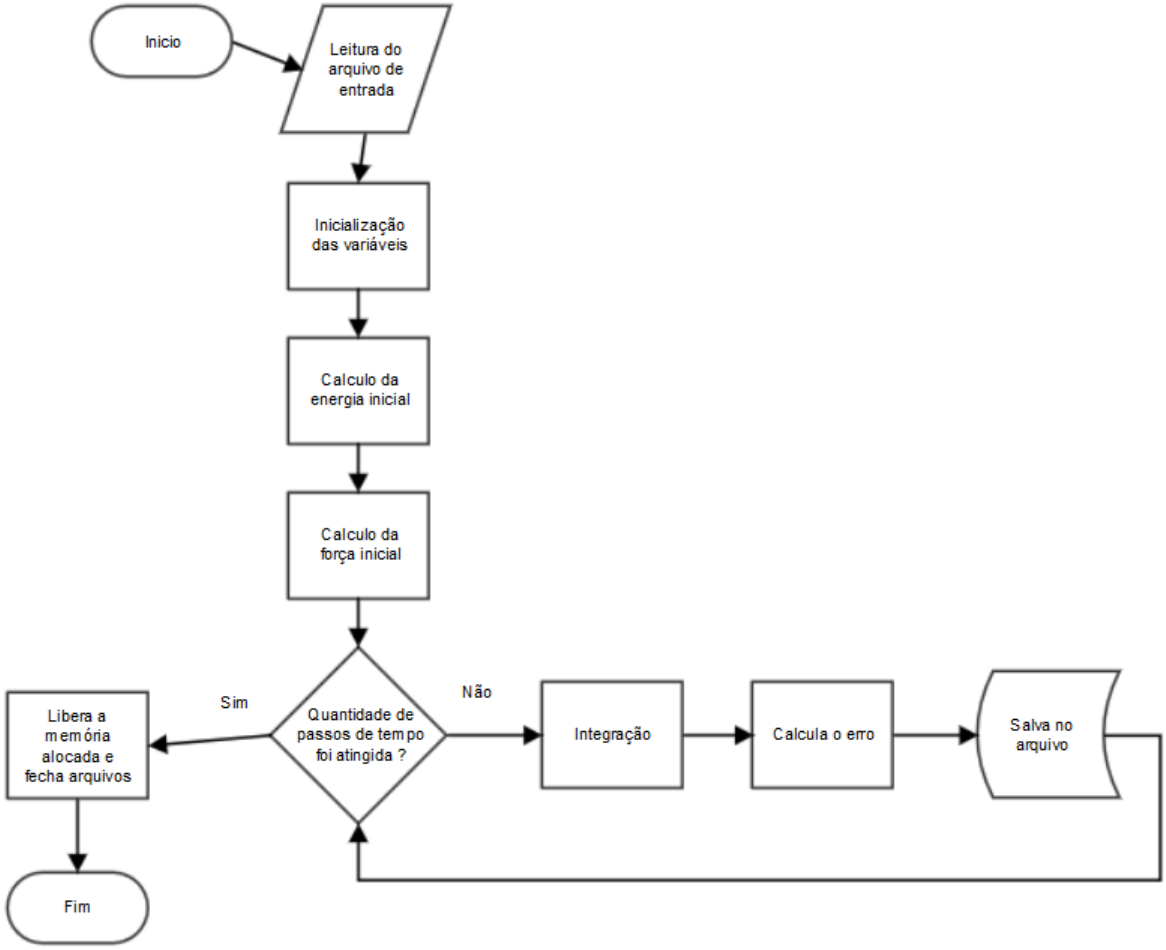
Um dos objetivos deste trabalho é analisar e otimizar um código serial escrito em linguagem *C* já existente, que realiza simulação de sistemas de partículas que interagem com força de longo alcance e paralelizar o mesmo com o proposito de realizar as simulações em um tempo hábil. Foram utilizadas, algumas ferramentas que auxiliaram na otimização e posteriormente na paralelização do código, como descrito nas seguintes subseções.

O programa sequencial esta devidamente divido em módulos responsáveis por tarefas especifica. Inicialmente é feita leitura do arquivo de entrada, alocação de memoria e a inicialização das variáveis que definem as condições iniciais para a simulação. Em seguida, é feito o calculo das energias cinética e potencial iniciais do sistema, a soma destas duas resulta na energia total inicial. Posteriormente o programa calcula a força inicial.

Após estabelecidas as todas as condições iniciais, o programa incia o resolução de um problema de valor inicial utilizando um método de integração explícito. Desta forma executa-se um laço que vai desde o tempo inicial até um tempo final previamente estabelecido. As iterações ocorrem de acordo com um certo passo de tempo t, todas essas informações são passadas no arquivo de entrada. A cada iteração do laço, são executadas a função responsável pela integração numérica. A função para calcular a energia cinética, a função para calcular a energia potencial e o calculo do erro relativo são executados a cada determinada quantidade de iterações a depender da frequência de escrita em disco para que os resultados parciais sejam salvos em arquivo. Quando a condição de parada do laço é atendida, ou seja, o tempo final de simulação foi alcançado, o programa libera a memoria alocada, salva no arquivo o estado final e fecha os arquivos. Exibimos um fluxograma que ilustra o programa serial na figura 7.

Inicialmente, fizemos um estudo do código para obter o máximo de informações referentes ao seu funcionamento. Em seguida, foi utilizada a ferramenta *Gprof*, para análise dinâmica da execução da aplicação. O propósito usual desse tipo de análise é determinar o quanto de recurso computacional é consumido por cada parte do código, com o objetivo de otimizar o tempo de execução e diminuir quando possível

Figura 7 – Fluxograma da versão serial da aplicação.



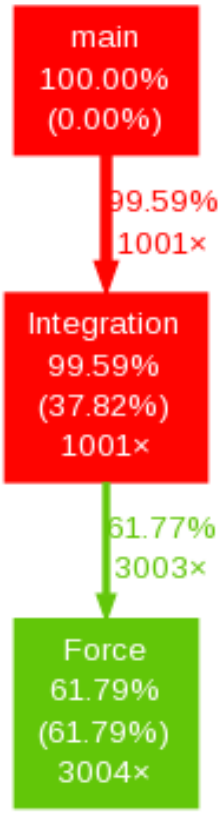
o consumo de memória. Essa ferramenta foi usada em conjunto com o compilador *GCC*. Basicamente, o *Gprof* analisa cada uma das funções da aplicação e insere código na cabeça e cauda de cada uma para coletar informações de tempo. Então, quando o programa é executado normalmente, é criado um arquivo *"gmon.out"* contendo dados brutos que o programa *Gprof* transforma em estatísticas de *profiling*. A partir do resultados obtidos utilizando a ferramenta *Gprof*, foi possível identificar as funções mais custosas, otimizar o código reestruturando algumas funções, remover as partes inutilizadas do código e refatorar algumas operações.

Para conseguirmos visualizar de forma gráfica o resultado do *profiler* utilizamos um *script* em *Python* chamado *gprof2dot.py*. Esse *script* transforma a saída do *profiler* em um *dot graph*. E para finalizar, foi utilizado o comando *dot* do pacote *Graphviz* para converter o *dot graph* em uma imagem. A figura 8 exibe a imagem obtida como resultado do *profiling*, onde cada retângulo representa uma função do programa, e tem o seu respectivo nome escrito na primeira linha de cada elemento. Na segunda linha temos a porcentagem do tempo que a função levou para ser executada por completa em relação ao tempo que o programa demorou pra ser executado por inteiro. A cor dos

retângulos é definida justamente por esse valor, onde as funções que gastam o maior tempo possuem cores mais próximas do vermelho, e quanto menos tempo mais perto do azul escuro. Na terceira linha, entre parenteses, temos a percentagem do tempo gasto pela função propriamente dita (sem contar o tempo gasto nas chamadas de funções) em relação ao tempo que o programa demorou pra ser executado por inteiro. Na última linha temos o número de vezes que a determinada função foi chamada por qualquer outra função. Nas setas temos a quantidade de vezes que uma função chamou a outra, além da porcentagem do tempo total de execução do programa que foi gasto nessas chamadas.

Analisando a figura 8, notamos que a maior carga de trabalho esta na função *Integration*, que é a responsável pela integração numérica no programa. Ela levou foi responsável por 99.59% do tempo de execução do programa. A função *Force* foi encarregada de 61.79% do tempo de execução do programa.

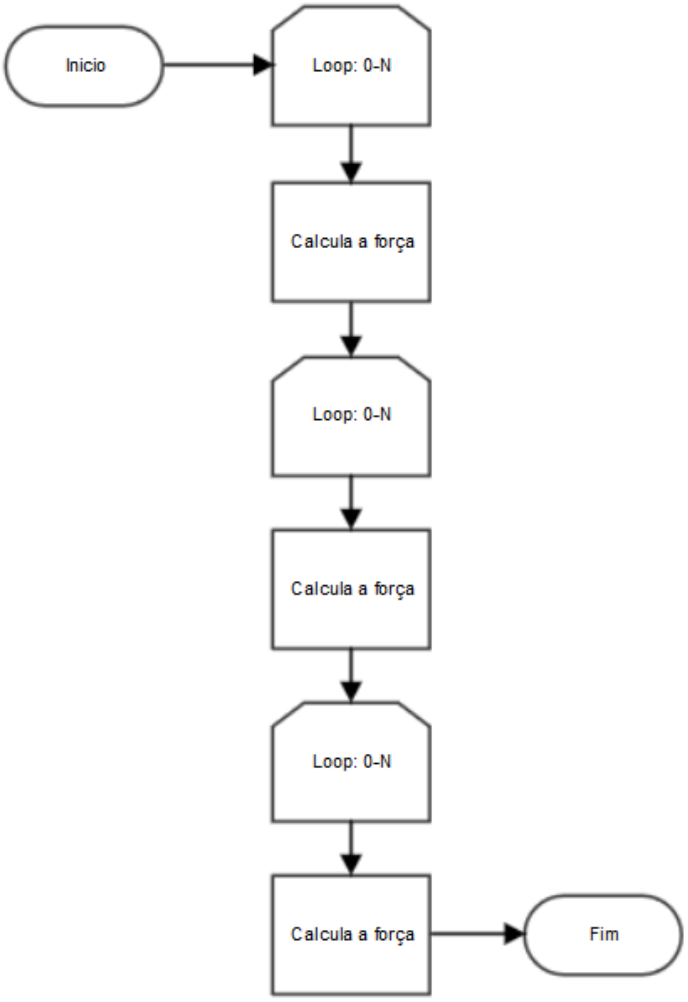
Figura 8 – Resultado do profiling usando o Gprof.



Devido a grande parte do tempo de execução do programa se passar dentro da função *Integration*, fizemos uma analise detalhada desta função. Ela consiste basicamente de 3 laços *for* que iteram de 0 até o número de partículas N intercalados com chamadas da função *Force*. O fluxograma exposto na figura 9 descreve de maneira visual a função *Integration*.

Neste ponto faz-se necessário realizar um análise de como varia o uso de recur-

Figura 9 – Fluxograma da função Integração.



sos computacionais da implementação, com o objetivo de analisar a maneira como a aplicação se comporta quando variamos alguns parâmetros. Por se tratar de um método explícito de integração a complexidade d o problema depende fundamentalmente de dois fatores:

A quantidade de iterações, ou seja a quantidade de passos no tempo;

O tamanho do problema em cada iteração, ou seja a quantidade de partículas na simulação.

A quantidade de iterações depende do tempo total de simulação e do passo de tempo utilizado para a integração no tempo. Neste trabalho optamos, sem perda de generalidade, variar apenas o passo de tempo. Tem ainda um terceiro fator que pode influenciar no desempenho da aplicação que é o frequência de escrita em disco. Desta forma os parâmetros utilizados para estudar o modelo foram o número de partículas, o passo de tempo e a frequência de escrita em disco. Os experimentos computacionais para avaliar

o comportamento da aplicação serial nestas condições aparecem descritos no final deste capítulo.

3.2.2 OpenMP

Tomando como base a versão sequencial do programa que realiza simulação de sistemas de partículas que interagem com força de longo alcance, foi implementada uma versão para ambientes de memória compartilhada utilizado o padrão *OpenMP*. O principal objetivo desta implementação, foi avaliar a maneira com que a aplicação se comporta quando paralelizamos as regiões definidas como criticas do programa. A abordagem em *OpenMP* permite implementar de maneira simples uma versão paralela do código serial, apenas utilizando diretivas de preprocessamento. Esta abordagem possibilita também, aplicar uma estrategia de paralelismo incremental nas regiões criticas.

Na implementação utilizando *OpenMP*, incluímos a biblioteca *omp.h*, na função responsável por inicializar os vetores contem um laço *for* e nele utilizamos um construtor de região paralela combinado com o construtor de divisão de trabalho *for*.

```
1 #pragma omp parallel for
```

A função para calcular a energia cinética do sistema *KineticEnergy*, também possui um laço que foi paralelizado usando um construtor de região paralela combinado com o construtor de divisão de trabalho *for*, desta vez em conjunto com a clausula *reduction*, que realiza a operação de redução de maneira eficiente. *for*.

```
1 #pragma omp parallel for reduction(+:soma)
```

A função *Force*, utilizada para calcular a força, possui dois laços *for*, neste casa para reduzir o tempo de gerenciamento criamos apenas uma região paralela que envolve os dois laços e para cada laço criamos um construtor de divisão de trabalho *for*. No entanto, entre esses laços exite código que deve ser executado apenas por uma *thread*, neste local utilizamos o construtor *single* para estabelecer que aquele bloco sintático será executado por uma única *thread*.

```
1  #pragma omp parallel private(var1, var2)
2  {
3      #pragma omp for reduction(+:somaX, somaY)
4
5      #pragma omp single
6      {
7          //regiao executada por uma unica thread
8      }
9
10     #pragma omp for
11 }
```

A função *Integration* como descrita na figura 9, possui três laços for intercalados entre chamadas da função *Force*. Neste três laços utilizamos o construtor de região paralela combinado com o construtor de divisão de trabalho *for*. A quantidade de *thread* a ser utilizada foi definida via variável de ambiente *OMP_NUM_THREADS*.

Neste ponto se faz necessário realizar um análise de como escala a implementação paralela com o objetivo de analisar a maneira como a aplicação se comporta quando variamos a complexidade do modelo. Esta análise deve levar em consideração a utilização dos recursos disponíveis, *CPUs* e memória, Os experimentos computacionais para avaliar o comportamento da aplicação serial nestas condições aparecem descritos no final deste capítulo.

3.2.3 MPI

O *OpenMP* é uma maneira de programar em dispositivos de memória compartilhada, ou seja, todas as *threads* paralelas têm acesso aos dados do programa. O paralelismo pode acontecer durante a execução de um laço *for* específico ao dividir o laço entre as diferentes *threads*. Já o *MPI* é uma maneira de programar em dispositivos de memória distribuída, isto é, cada processo paralelo está trabalhando em seu próprio espaço de memória isoladamente dos demais. Todo código é executado independentemente por cada processo. O paralelismo neste caso acontece quando indicamos a cada processo exatamente em qual parte do problema global eles devem trabalhar com base inteiramente em seu ID de processo. A maneira de escreve um programa em *OpenMP* é muito diferente do *MPI*.

Neste trabalho implementamos uma versão em *MPI* do programa que realiza simulação de sistemas de partículas que interagem com força de longo alcance para ambientes de memória distribuída. O objetivo principal desta implementação, é analisar como paralelismo o escala neste ambiente. As funções do *MPI* são utilizadas para distribuir processos e promover a comunicação e sincronização dos mesmos via troca de mensagens. Assim, é possível distribuir as tarefa para diversos nós de computo, aumentando a utilização dos disponíveis.

Utilizamos as seguinte funções na implementação *MPI*:

MPI_Init - inicializa o ambiente de execução do *MPI*;

MPI_Comm_rank - determina o *rank* do processo;

MPI_Comm_size - determina a quantidade de processos;

MPI_Barrier - sincroniza todos os processos bloqueando-os até que todos eles tenham atingido essa rotina;

MPI_Bcast - informa a todos os processos os valores inicializados nas variáveis, transmitindo uma mensagem do processo com *id* 0 para todos os outros.

MPI_Scatter - dividir os dados dos vetores do processo raiz para os demais processos;

MPI_Gather - reúne no processo raiz os dados dos vetores de um grupo de processos;

MPI_Reduce - realiza a operação de redução dos valores em todos os processos para um único valor;

MPI_Allreduce - realiza a operação de redução combinando os valores de todos os processos e distribui o resultado de volta para todos os processos;

MPI_Reduce - encerra o ambiente de execução de *MPI*.

Neste ponto se faz necessário realizar um análise de como escala a implementação paralela com o objetivo de analisar a maneira como a aplicação se comporta quando variamos a complexidade do modelo. Esta análise deve levar em consideração a utilização dos recursos disponíveis, *CPUs* e memória, Os experimentos computacionais para avaliar o comportamento da aplicação serial nestas condições aparecem descritos no final deste capítulo.

3.2.4 Hibrido OpenMP / MPI

Conforme descrito na sessão 3.1 o CACAU, supercomputador utilizado para realização dos testes, é composto por 23 nós multiprocessados de memoria compartilhada. Com o objetivo de otimizar a utilização dos recurso disponíveis no CACAU, implementamos uma versão híbrida que utiliza *MPI* e *OpenMP*. Exploramos as melhores características de ambos os modelos de programação, mesclando a paralelização explícita de grandes tarefas com o *MPI* com a paralelização de tarefas simples com o *OpenMP*. O programa está estruturado com uma série de tarefas *MPI* que são quebradas em processos e dividas para os nós de computo, no código sequencial destas tarefas está inserido diretivas *OpenMP* para paralelizar usando *multithreading* e aproveitar as características da presença de memória compartilhada e multiprocessadores dos nós.

Neste ponto se faz necessário realizar um análise de como escala a implementação paralela com o objetivo de analisar a maneira como a aplicação se comporta quando variamos a complexidade do modelo. Esta análise deve levar em consideração a utilização dos recursos disponíveis, *CPUs* e memória, Os experimentos computacionais para avaliar o comportamento da aplicação serial nestas condições aparecem descritos no final deste capítulo.

3.2.5 Cuda

O *CUDA* é uma *API* destinada a computação paralela em *GPGPU* criada pela *Nvidia*. Neste trabalho, utilizamos uma versão *CUDA* já existente do programa que realiza simulação de sistemas de partículas que interagem com força de longo alcance. Esta versão foi implementada por professores e pesquisadores na linguagem *C* utilizando a *API Nvidia CUDA* para paralelizar o programa com os recursos das *GPGPUs*. De posse do código fonte deste programa, realizamos uma analise, que possibilitou determinar uma estrategia para implementar um algoritmo paralelo do tipo *SIMD* utilizando a arquitetura *CUDA*. Neste sentido, o aspecto critico da implementação esta relacionado a operações de redução amplamente utilizada na função de integração. Vale lembrar que o modelo de programação *CUDA*, assume um sistema composto de um *host* (CPU) e um *device* (GPGPU), cada um com sua própria memória separada. Os *kernels* operam na memória do *device*, portanto, o *CUDA* fornece funções para alocar, desalocar e copiar dados da memória do *device*, bem como transferir dados entre a memória do *host* e a memória do *device*. A memória linear é normalmente alocada usando a função *cudaMalloc()* e liberada usando *cudaFree()* e a transferência de dados entre a memória do *host* e a memória do dispositivo é feita normalmente usando *cudaMemcpy()*.

Com o objetivo de demonstrar a maior complexidade de implementar a operação de redução em *GPGPUs*, vamos descrever como foi implementada a operação de redução nas versões paralelas baseadas em *CPU* e comparar com a implementação em *CUDA*. Na versão sequencial do programa, as operações de redução são feitas de maneira simples utilizando um laço *for*, para exemplificar usaremos um trecho da função *KineticEnergy*, nela existe um laço que faz uma operação de redução:

```
1 for (long i = 0; i < n; i++){
2     energKin += p[i]   p[i];
3 }
```

neste caso, a variável *energKin* reúne de maneira incremental os valores do quadrado de cada elemento do vetor *p*. No final do laço que realiza N iterações, obtemos como resultado na variável *energKin* a soma dos quadrados de todos os itens do vetor *p*.

Implementar operações de redução em *OpenMP* é algo relativamente simples, necessitando apenas inserir a clausula *reduction* na diretiva de pré-processamento que contém o construtor de divisão de trabalho *for*, partido do exemplo anterior temos:

```
1 #pragma omp parallel for reduction(+:energKin)
2     for (i = 0; i < n; i++){
3         energKin += p[i]   p[i];
4     }
```

a diretiva de preprocessamento *#pragma omp parallel for* transforma o laço *for* em um laço paralelo. O *OpenMP* divide o laço pelo número de *threads* definidas e cada uma

delas executa um pedaço laço. A clausula *reduction(+:energKin)* declara que estamos realizando uma operação de redução na variável *energKin*. Cada *thread* tem uma cópia privada da variável *energKin* na qual é somando o quadrado de cada um dos itens elementos do vetor *p*. Depois que os laços parciais que cada *thread* executa são concluídos, os resultados parciais de *energKin* são somados numa cópia da variável que é compartilhada entre todas as *threads*.

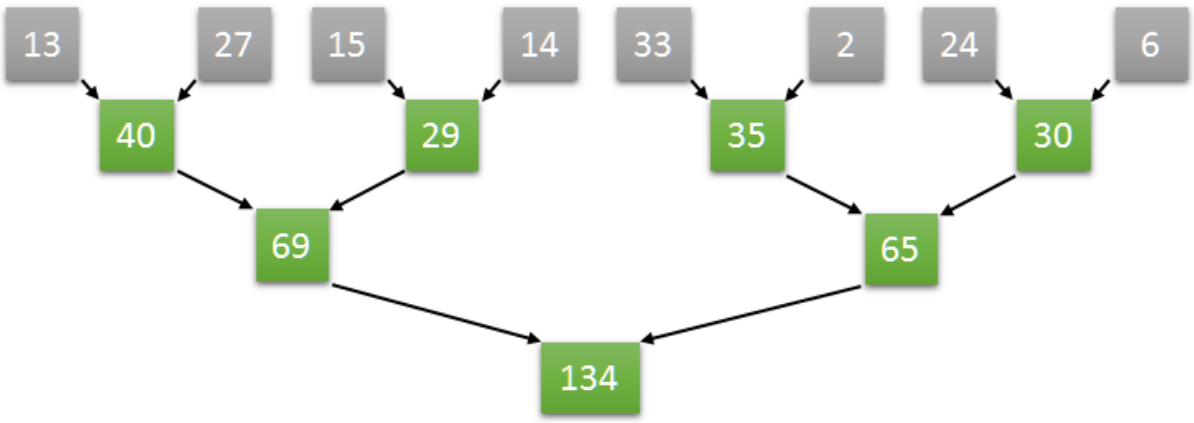
Para realizar a operação de redução em *MPI*, utilizamos a função *MPI_Reduce*:

```
for (long i = 0; i < nLoc; i++){
  energLocal += p[i]   p[i];
}
MPI_Reduce(&energLocal, energKin, 1, MPI_DOUBLE,
            MPI_SUM, 0, MPI_COMM_WORLD);
```

cada processo em *MPI* executa de forma individual o programa e se comunica com os demais processos via troca de mensagem. Note que o laço não itera N vezes e que a variável que armazena a soma também é diferente dos casos anteriores. A razão disto é que cada processo executa a fração do laço que lhe foi destinada e esta divisão é feita de forma explicita, neste caso *nloc* é igual a N dividido pelo número de processos e *energLocal* corresponde a soma local de cada processo. Quando chamamos a função *MPI_Reduce*, ela soma os valores de *energLocal* em todos os processos para um único valor *energKin*.

Finalmente na implementação em *CUDA* utilizamos uma estrategia baseada em árvore dentro de cada bloco de *threads*. Assim cada bloco de *threads* reduz uma parte do vetor. Como a adição de valores é uma operação associativa, então o esquema exposto na figura 10 é viável.

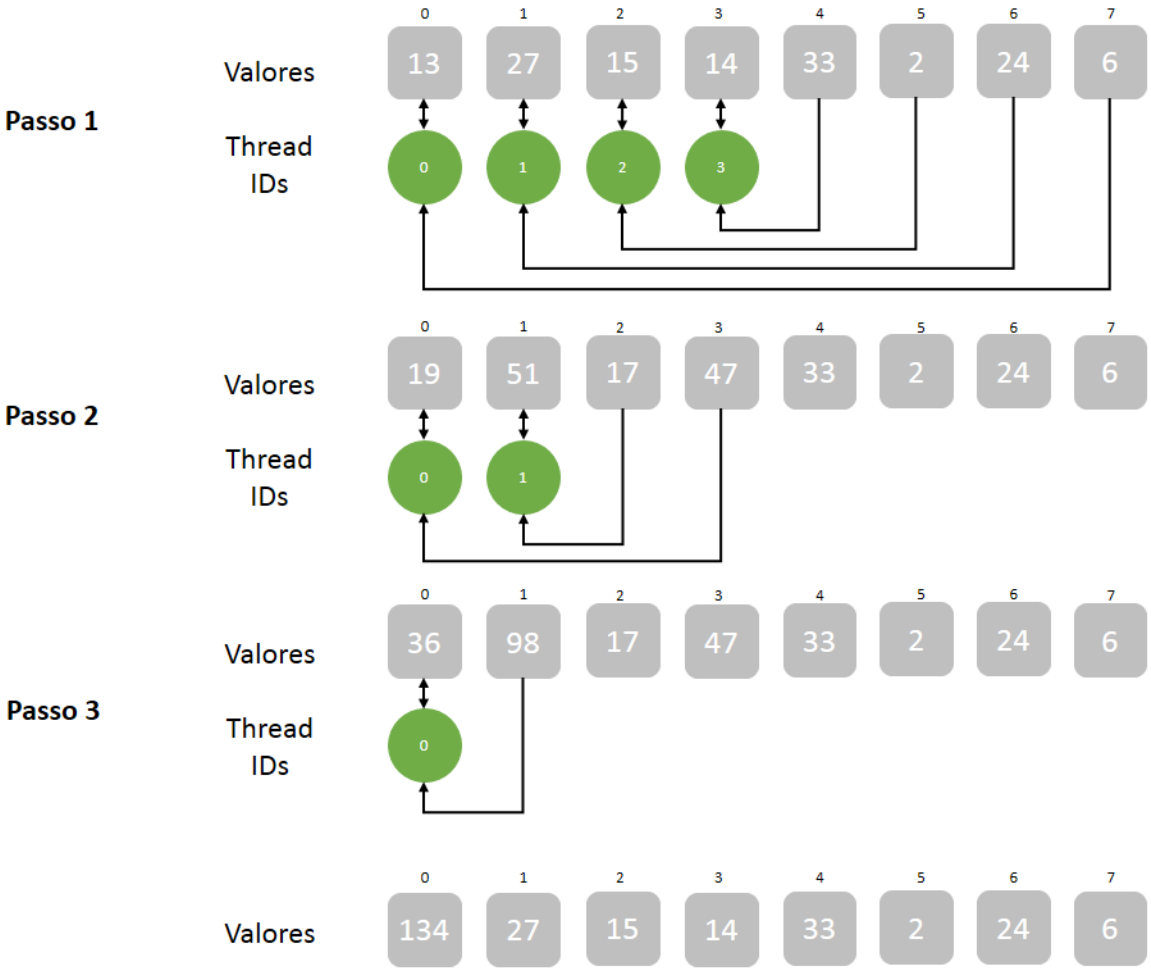
Figura 10 – Soma paralela



A ideia principal consiste em dado N, o número dos elementos em um vetor, iniciamos $\frac{N}{2}$ *threads*, uma *thread* para cada dois elementos. Cada *thread* calcula a soma

dos dois elementos correspondentes, armazenando o resultado na posição do primeiro. Iterativamente a cada passo o número de *threads* é dividido pela metade (por exemplo, começando com 4, depois 2, depois 1). Reduzimos pela metade o tamanho do passo entre os dois elementos correspondentes da mesma forma (começando com 4, depois 2, depois 1) após algumas iterações, o resultado da redução será armazenado no primeiro elemento da vetor. A figura 11 ilustra estes passos.

Figura 11 – Redução paralela



Nos experimentos realizados neste trabalho, o valor de N que determina o tamanho dos vetores costuma ser grande, isto nos leva a utilizar vários blocos de *threads* simultâneos na execução do programa. Em virtude disto, cada bloco de *threads* faz o calculo de um pedaço do vetor e a operação de redução é feita inicialmente em cada bloco, depois de obtido o resultado da soma de todos os elementos em cada bloco, teremos o resultado parcial da redução. Então somamos esses resultados para alcançar o valor final, que neste caso corresponde a soma do quadrado de todos os elementos do vetor.

O fragmento do código *CUDA* que realiza o calculo da energia cinética na função *KineticEnergy*, é um pouco mais complexo que nos exemplos anteriores. Definimos *tid* como sendo o *id global* de uma *thread* e *cind* o *id* da *thread* no bloco em que ela se encontra. O vetor *energKinBlock* foi definido na memoria *shared*, isto é, cada bloco de *threads* possui um vetor *energKinBlock* distinto de acesso restrito as *threads* do bloco. O vetor *energKinBlock* tem o tamanho igual a número de *threads* por bloco. O vetor *p* foi alocado na memoria global, ou seja, pode ser acessado por todas as *threads* de qualquer bloco. Quando o *kernel* é executado, o vetor *p* é dividido entre os blocos e as *threads* de cada bloco executam as instruções destinadas a elas. Primeiro é calculado o quadrado do item do vetor *p* e este valor é somado a uma posição do vetor *energKinBlock*. Note que no indice do vetor *p* é utilizado o *id* global *tid*, já no vetor *energKinBlock* é utilizado o *id* local *cind*. Deste modo, cada bloco de *thread* realiza o calculo e obtém como resultado um vetor com estes valores. Por fim, fazemos uma chamada da função *__syncthreads()* para sincronizar as *threads*.

```
while(tid<n){
    energKinBlock[cind]+=p[tid] p[tid];
    tid+=blockDim.x gridDim.x;
};
__syncthreads();
```

O processo de redução inicia no próximo passo, onde definimos *i* igual a metade da quantidade de blocos e executamos um laço que realiza a operação ilustrada na figura 11.

```
i=blockDim.x/2;
while(i!=0){
    if(cind<i){
        energKinBlock[cind]+=energKinBlock[cind+i];
    };
    __syncthreads();
    i/=2;
};
```

Quando o laço é finalizado o resultado obtido é a soma de todos os elementos do vetor *energKinBlock*, o valor estará alocado na primeira posição deste vetor. Em seguida, copiamos o valor da primeira posição do vetor *energKinBlock* de cada bloco para um vetor auxiliar alocado na memoria global *kinp*. Agora temos o valor em cada posição do vetor *kinp* a soma parcial efetuada em cada bloco.

```
if(cind==0){
    kinp[blockIdx.x]=kk[0];
};
```

Finalmente para concluir o processo de redução, utilizamos um *kernel* auxiliar que realiza a novamente a operação ilustrada pela figura 11. Este processo é feito com a

soma parcial calculada em cada bloco, para em fim obter a soma total e assim concluir o processo de redução.

```
i=blockDim.x/2;
while(i!=0){
    if(cind<i){
        kinp[cind]+=kinp[cind+i];
    };
    __syncthreads();
    i/=2;
};
```

3.2.6 OpenCL

OpenCL é uma arquitetura para escrever programas que funcionam em plataformas heterogêneas (*CPUs*, *GPGPUs* e outros processadores). Neste trabalho, utilizamos a biblioteca *Heterogeneous Programming Library* (HPL) que fornece um ambiente de programação para *C++*, cujo objetivo é maximizar a capacidade de programação de sistemas heterogêneos ao mesmo tempo em que permite controle e desempenho de baixo nível, semelhantes aos das abordagens de nível inferior. A biblioteca é construída em dois conceitos principais:

Arrays: tipos de dados especiais que podem ser usados tanto no programa principal, que é executado em uma *CPU* regular (*host*), quanto no código que é executado nos dispositivos heterogêneos (*devices*).

Kernels: funções que podem ser executadas em qualquer dispositivo. Eles podem ser escritos em uma linguagem incorporada em *C++* fornecida pela HPL. A biblioteca traduz esses *kernels* para *OpenCL*, permitindo a execução em uma ampla gama de dispositivos. Esta linguagem é extremamente semelhante ao *C++* regular.

A implementação em *OpenCL* com o auxilio do *HPL* seguiu a mesma ideia da implementação em *CUDA*, porem com as diferenças básicas entre as arquiteturas. Começamos nossa implementação, incluindo o arquivo de cabeçalho para *HPL* e declarando o uso de seu *namespace*.

```
#include "HPL.h"

using namespace HPL;
```

Implementamos os *kernels* para cada função que será paralelizada. Utilizamos variáveis e vetores do tipo *Array*, o que torna transparente a transferência de dados entre o *host* e o *device*.

```
Array<int, 0> n_gpu = n;
Array<double, 1, Local> energKinBlock(nthread);
```

Para declarar uma variável escalar que será utilizada num *kernel HPL*, usamos o tipo *Array* de dimensão 0. Neste caso, para fazer a declaração da variável *n_gpu*, chamamos o tipo *Array* e passamos como parâmetro o tipo de dados que a variável declarada receberá, e a quantidade de dimensões. Quando declaramos o vetor unidimensional *energKinBlock*, utilizamos o tipo *Array* de dimensão 1, passando como parâmetros o tipo *double* e a quantidade de dimensões 1. Note que na segunda declaração, existe um terceiro parâmetro *Local*, isto especifica que o vetor utilizará a memoria compartilhada, ou seja, para cada bloco de *threads* existe um vetor *energKinBlock* distinto e com acesso restrito as *threads* do bloco. Esta declaração é semelhante ao *__shared__* em *CUDA*.

A código em *HPL* foi construído com base no código *CUDA*. Assim como *CUDA*, o ponto critico da implementação *HPL* também foram as operações de redução. Para exemplificar usaremos um trecho da função *KineticEnergy* assim como na sessão 3.2.5.

O fragmento do *kernel HPL* que realiza o calculo da energia cinética é muito semelhante a implementação em *CUDA*, com algumas diferenças básicas de sintaxe. Os *kernels HPL* usam variáveis pré-definidas para identificar cada *thread* paralela que executa um *kernel*, o número de *threads* em cada dimensão e outras propriedades importantes. Por exemplo, *idx* é uma variável predefinida que identifica a *thread* que está executando a instância atual do *kernel* na primeira dimensão do domínio do problema, *idx* equivale ao identificado global da *thread tid* que definimos em *CUDA*. O *lidx* é o identificador local de uma *thread*, ele indica o id de uma *thread* dentro de um bloco, equivale ao *cind* definido em *CUDA*. A variável *ngroupsx*, fornece o numero de grupos na primeira dimensão. Já o *lszx* informa o tamanho do bloco, ou seja, a quantidade de *threds* por bloco. A variável *gidx*, informa o id do bloco. O vetor *energKinBlock* foi definido como *LOCAL*, isto é, cada bloco de *threads* possui um vetor *energKinBlock* distinto de acesso restrito as *threads* do bloco, equivalente as declarações na memoria *shared* em *CUDA*. O vetor *energKinBlock* tem o tamanho igual a número de *threads* por bloco. No momento da alocação do vetor *p*, não foi informado o tipo de memoria, então por padrão ste vetor é definido na memoria global, ou seja, pode ser acessado por todas as *threads* de qualquer bloco. Quando o *kernel* é executado, o vetor *p* é dividido entre os blocos e as *threads* de cada bloco executam as instruções destinadas a elas. Primeiro é calculado o quadrado do item do vetor *p* e este valor é somado a uma posição do vetor *energKinBlock*. Por fim, fazemos uma chamada da função *barrier(LOCAL)* para sincronizar as *threads* em um bloco.

```
while_(tid < n){
    energKinBlock[cind] += p[idx] p[idx] ;
    idx += ngroupsx lszx;
}
```

```
barrier(LOCAL);
```

O processo de redução segue a mesma estrategia que utilizamos em *CUDA*, onde definimos *i* igual a metade da quantidade de blocos e executamos um laço que realiza a operação ilustrada na figura 11.

```
i = lszx/2;
while_(i!=0){
    if_(lidx<i){
        energKinBlock[lidx] += energKinBlock[lidx+i];
    }
    barrier(LOCAL);
    i = i/2;
}
```

Assim como em *CUDA*, quando o laço é finalizado o resultado obtido é a soma de todos os elementos do vetor *energKinBlock*, o valor estará alocado na primeira posição deste vetor. Em seguida, copiamos o valor da primeira posição do vetor *energKinBlock* de cada bloco para um vetor auxiliar alocado na memoria global *kinp*. Agora temos o valor em cada posição do vetor *kinp* a soma parcial efetuada em cada bloco.

```
if_(!lidx){
    kinp[gidx] = energKinBlock[0];
}
```

Finalmente concluímos o processo de redução de maneira similar a implementação *CUDA*, utilizamos um *kernel* auxiliar que realiza a novamente a operação ilustrada pela figura 11. Este processo é feito com a soma parcial calculada em cada bloco, para em fim obter a soma total e assim concluir o processo de redução.

```
i = lszx/2;
while_(i!=0){
    if_(lidx<i){
        kinp[lidx] += kinp[lidx+i];
    }
    barrier(LOCAL);
    i = i/2;
}
```

A implementação utilizando *HPL* tem um nível de complexidade análogo a *CUDA*, com a vantagem de ter transparente, do ponto de vista do programador, as operação de transferência e copia de dados entre o *host* e o *device*.

3.3 Desenho experimental

O Centro de Armazenamento de dados e Computação Avançada da UESC (CACAU), em sua configuração original possuía 20 nós de computo R422-E1 da fabricante

Bull novascale, cada um destes conta com 2 processadores QuadCore Intel ® Xeon ® E5430 @ 2.66GHz, totalizando 8 núcleos em cada nó. O CACAU Recebeu um *upgrade* e hoje conta com mais 3 nós de processamento *Bullx* B515, cada um com 2 processadores HexaCore Intel ® Xeon ® E5-2400 @ 2.40 totalizando 12 núcleos em cada nó, no entanto, cada um destes novos nós contam também com 2 placas gráficas *GPUs* para processamento de alto desempenho Nvidia Tesla K20. Por conversão vamos nos referir aos 20 primeiros nós como nós da fila *LONG* e aos 3 últimos como nós da fila *GPU*, pois é assim que eles são identificado pelo *Slurm*, o gerenciador de *clusters* utilizado no CACAU.

Neste trabalho, foram executados 9 diferentes cenários de teste, para cada um dos métodos descritos anteriormente na sessão 3.2, em diferentes quantidades de nós. Os cenários dos casos de teste variam em relação a três parâmetros, o número de partículas, o tamanho do passo de tempo e a fremência de escrita em disco. Nos três primeiros experimentos variamos N o número de partículas e mantemos fixos os outros dois parâmetros, para N foram escolhidos os valores múltiplos de 1.024 mais próximos de 40.000, 400.000 e 4.000.00. Nos três seguintes casos, variamos T_s o tamanho do passo de tempo e mantemos fixo os outros dois parâmetros e por fim, nos três últimos variamos T_e a fremência de escrita em disco mantendo fixos os dois parâmetros anteriores. Os casos de teste são descritos com mais detalhes a seguir:

Quando variamos N, mantemos fixo $T_s = 0,1$ e $T_e = 1$:

- N_1 = 40.960,
- N_2 = 400.384,
- N_3 = 4.000.768.

Quando variamos T_s, mantemos fixo N = 400.384 e $T_e = 1$:

- $T_s = 0,01$,
- $T_s = 0,1$,
- $T_s = 1$.

Quando variamos T_e, mantemos fixo N = 400.384 e $T_s = 0,1$:

- $T_e = 0,01$,
- $T_e = 0,1$
- $T_e = 1$.

3.3.1 Experimentos realizados na fila LONG

Os 9 cenários descritos na sessão 3.3 foram executados em 1, 2 e 4 nós da fila LONG, primeiro de maneira sequencial, os resultados obtidos dessas execuções servem como base para comparar o desempenho das versões paralelas. Realizamos os experimentos com OpenMP para apenas 1 nó de computo, pois não faz sentido executar a versão *OpenMP* para uma quantidade maior de nós, já que é uma implementação para ambientes de memoria compartilhada. Como cada nó de computo na fila LONG possui 8 núcleos, nas execuções em *MPI* utilizamos 1 processo para cada núcleo no maior caso.

A tabela 3 exibe os recursos usados para executar os experimentos em apenas 1 nó de computo da fila LONG. Fizemos uma execução utilizando *OpenMP* com 1 processo e 8 *threads*, uma *thread* para cada núcleo do nó de computo. Fizemos três execuções com *MPI* usando 2, 4 e 8 processos, cada uma destas execuções com apenas uma *thread*. Executamos também a versão hibrida em duas diferentes configurações, primeiro usando 2 processos e 4 *threads* e depois usando 4 processos e 2 *threads*.

Tabela 3 – Experimento utilizando 1 nó da fila LONG

Processos	Threads	Tecnologia
8	1	MPI
4	1	MPI
2	1	MPI
2	4	MPI, Hibrido
4	2	MPI, Hibrido
1	8	OpenMP

A tabela 4 se refere aos experimento realizados utilizando 2 nós de computo da fila LONG. Executamos a versão *MPI* utilizando 2, 4, 8 e 16 processos com apena 1 *thread* em cada. Executamos também a versão hibrida, desta vez com 8 processos e 2 *threads*, 4 processos e 4 *threads* e 2 processos e 8 *threads*.

A tabela 5 mostra os experimentos realizados usando 4 nós de computo da fila LONG. Executamos a versão *MPI* para 32, 16, 8 e 4 processos. Executamos também a versão hibrida para 16 processos e 2 *threads*, 8 processos e 4 *threads* e 4 processos e 8 *threads*.

3.3.2 Experimentos realizados na fila GPU

Assim como na fila LONG, os 9 cenários descritos na sessão 3.3 também foram executados na fila GPU usando 1, 2 e 3 nós de computo. Executamos primeiro de maneira sequencial para obter os resultados que são a base de comparação com as

Tabela 4 – Experimento utilizando 2 nós da fila LONG

Processos	Threads	Tecnologia
16	1	MPI
8	1	MPI
4	1	MPI
2	1	MPI
8	2	MPI, Hibrido
4	4	MPI, Hibrido
2	8	MPI, Hibrido

Tabela 5 – Experimento utilizando 4 nós da fila LONG

Processos	Threads	Tecnologia
32	1	MPI
16	1	MPI
8	1	MPI
4	1	MPI
16	2	MPI, Hibrido
8	4	MPI, Hibrido
4	8	MPI, Hibrido

versões paralelas. Realizamos os experimentos com *OpenMP* para apenas 1 nó de computo, por ser uma implementação para ambientes de memoria compartilhada. Cada nó de computo na fila GPU possui 12 núcleos, seguimos a mesma ideia das execuções na fila LONG, onde utilizamos 1 processo para cada núcleo no maior caso em execuções *MPI*.

A tabela 6 exibe os recursos usados para executar os experimentos em apenas 1 nó de computo da fila GPU. Fizemos uma execução utilizando *OpenMP* com 1 processo e 12 *threads*, uma *thread* para cada núcleo do nó de computo. Executamos a versão *MPI* utilizando 2, 3, 4, 6 e 12 processos com apenas 1 *thread*. Executamos também a versão hibrida usando 2 processos e 6 *threads*, 4 processos e 3 *threads* e 6 processos e 2 *threads*.

A tabela 7 exibe os experimentos realizados utilizando 2 nós de computo da fila *GPU*. As execuções em *MPI* foram feitas com 2, 4, 6, 12 e 24 processos, usando apenas 1 *thread*. Com a versão hibrida foram feitas execuções com 2 processos e 12 *threads*, 4 processos e 6 *threads*, 6 processos e 4 *threads* e 12 processos e 2 *threads*.

A tabela 7 refere-se aos experimentos realizados usando 3 nós da fila GPU. Para

Tabela 6 – Experimento utilizando 1 nó da fila GPU

Processos	Threads	Tecnologia
12	1	MPI
6	1	MPI
4	1	MPI
3	1	MPI
2	1	MPI
6	2	MPI, Hibrido
4	3	MPI, Hibrido
2	6	MPI, Hibrido
1	12	OpenMP

Tabela 7 – Experimento utilizando 2 nós da fila GPU

Processos	Threads	Tecnologia
24	1	MPI
12	1	MPI
6	1	MPI
4	1	MPI
2	1	MPI
12	2	MPI, Hibrido
6	4	MPI, Hibrido
4	6	MPI, Hibrido
2	12	MPI, Hibrido

MPI executamos com 3, 6, 12 e 36 processos, usando apenas 1 *thread*. Já para versão hibrida executamos usando 3 processos e 12 threads, 6 processos e 6 threads e 12 processos e 3 threads.

Tabela 8 – Experimento utilizando 3 nós da fila GPU

Processos	**Threads**	**Tecnologia**
36	1	MPI
12	1	MPI
6	1	MPI
3	1	MPI
12	3	MPI, Hibrido
6	6	MPI, Hibrido
3	12	MPI, Hibrido

4 Resultados e Discussões

Dedicamos este capítulo à apresentação dos resultados obtidos por meio do desenvolvimento e implementação das versões paralelas propostas neste trabalho. A fim de mensurar e comparar o desempenho computacional e a eficiência das implementações, utilizamos duas métricas bastante conhecidas no âmbito da computação paralela, o *speedup* e a **eficiência** e realizamos simulações computacionais para as implementações paralelas do HMF tanto em CPU nos ambientes de memória distribuída e memória compartilhada, bem como em GPGPU. Os testes foram realizados no ambiente de hardware e seguiram os desenhos experimentais descrito capítulo (3), apresentamos através de gráficos e tabelas comparativas os resultados numéricos gerados a partir destes experimentos. A implementação serial foi utilizada como base para avaliar o desempenho computacional das implementações em CPU, por outro lado, a implementação em CUDA serviu como base para avaliar o desempenho em GPGPU.

Na seção 4.1, apresentamos e discutimos os resultados numéricos dos experimentos computacionais que envolvem OpenMP, MPI, Hibrido (MPI-OpenMP), cujo objetivo é verificar como as diferentes implementações escalam e se é viável a utilização de mais recursos para obter melhores resultados, isto serve também como uma etapa intermediária para validar a utilização da GPGPUS massivamente paralelas. Na seção 4.2, apresentamos e discutimos os resultados numéricos obtidos para simulações computacionais utilizando CUDA e OpenCL (HPL) em GPGPU, cujo objetivo é obter um speedup ainda maior do que nos modelos anteriores.

4.1 Soluções para CPU

4.1.1 Serial

Os testes foram realizados seguindo os desenhos experimentais descritos no capitulo (3). Executamos o código Serial em 1 nó da fila LONG, variando primeiro o número de partículas N e mantendo fixos os tamanhos do passo de tempo T_s em 0.01 e a frequência de escrita em disco T_e em 1. Em seguida, executamos novamente mantendo valores fixos agora para N = 400.384 e o T_e em 1 e variando o tamanho do passo de tempo T_s. Logo após, fizemos novas execuções, desta vez com uma terceira configuração, mantendo fixos o N = 400.384 e o T_s em 0.01 e variando agora a frequência da escrita em disco T_e. Após realizados os teste executando 3 vezes a aplicação para cada um dos casos tanto na fila LONG quanto na fila GPU, obtivemos os tempos médios dados em segundos descritos na tabela (9).

Tabela 9 – Resultados numéricos dos tempos médios de execução do código serial em 1 nó do CACAU.

Parâmetro	Tempo fila LONG (s)	Tempo fila GPU (s)
N - Número de Part.	-	-
N_1 = 40960, Ts = 1, Te = 0,1	7,48667	8,08333
N_2 = 400.384, Ts = 1, Te = 0,1	87,54000	80,00333
N_3 = 4.000.768, Ts = 1, Te = 0,1	891,82667	818,15667
Ts - Passo de Tempo	-	-
N = 400.384, Ts_1 = 0,01, Te = 1	870,75667	794,68333
N = 400.384, Ts_2 = 0,1, Te = 1	87,65667	80,04333
N = 400.384, Ts_3 = 1, Te = 1	9,17667	8,43000
Te - Freq. Escrita	-	-
N = 400.384, Ts = 0.01, Te_1 = 0.01	877,27000	795,32000
N = 400.384, Ts = 0.01, Te_2 = 0.1	871,62333	800,25000
N = 400.384, Ts = 0.01, Te_3 = 1	868,71333	794,29000

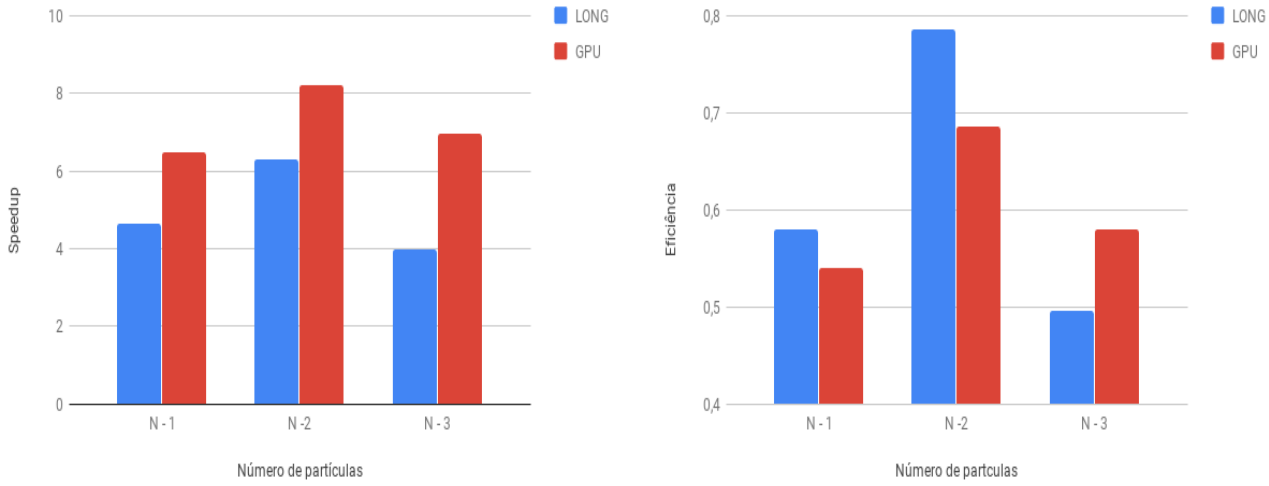
O tempo de execução do código serial é usado como base para mensurar o *speedup* e a *eficiência* das versões paralelas, e auxilia na avaliação de escalabilidade do problema.

4.1.2 OpenMP

Analisando a tabela (9), podemos perceber que a execução das tarefas em paralelo com OpenMP leva uma significante vantagem sobre a execução sequencial. Quando comparamos as execuções utilizando o OpenMP na fila LONG e na fila GPU variando o número de partículas N, observamos que a fila GPU possui maior *speedup* em relação a fila LONG, no entanto, quando comparamos as mesmas levando em conta a métrica de eficiência, a fila LONG é superior para os dois primeiros valores de N de tamanho pequeno e médio, já quando o valor de N cresce a GPU passa a ser mais eficiente. Nos testes realizados, o *speedup* e eficiência alcançam valores maiores no caso de teste N_2, de tamanho médio onde N = 400.384. A figura (12) exibe os resultados obtidos na execução dos testes com OpenMP variando o número de partículas.

As execuções utilizando OpenMP nas duas filas variando o T_s tamanho do

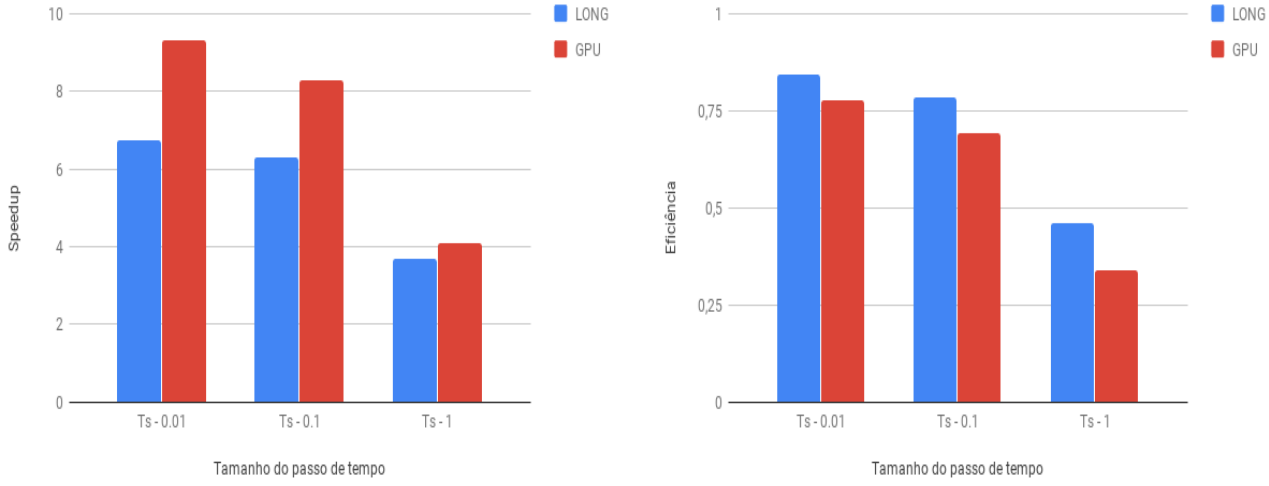
Figura 12 – *Speedup* e eficiência da versão *OpenMP* executada nas filas LONG com 8 *threads* e GPU com 12 *threads* variando o número de partículas.



(a) Gráfico de *Speedup* (b) Gráfico de Eficiência

passo de tempo, apresentaram resultados semelhantes as execuções variando o número de partículas. Toda via, o *speedup* obtido comparando cada caso de teste variando o numero de partículas e variando o tamanho do passo de tempo foi maior tanto na fila LONG quanto na fila GPU, mantendo a superioridade em *speedup* da fila GPU quando comparamos as duas filas. O pico de *speedup* e eficiência ocorreu no caso de teste N_1, de tamanho pequeno onde $N = 40.960$. Podemos observar também na figura (13), que em todos os tamanhos de passo T_s testados, a eficiência da fila LONG foi maior que a da fila GPU.

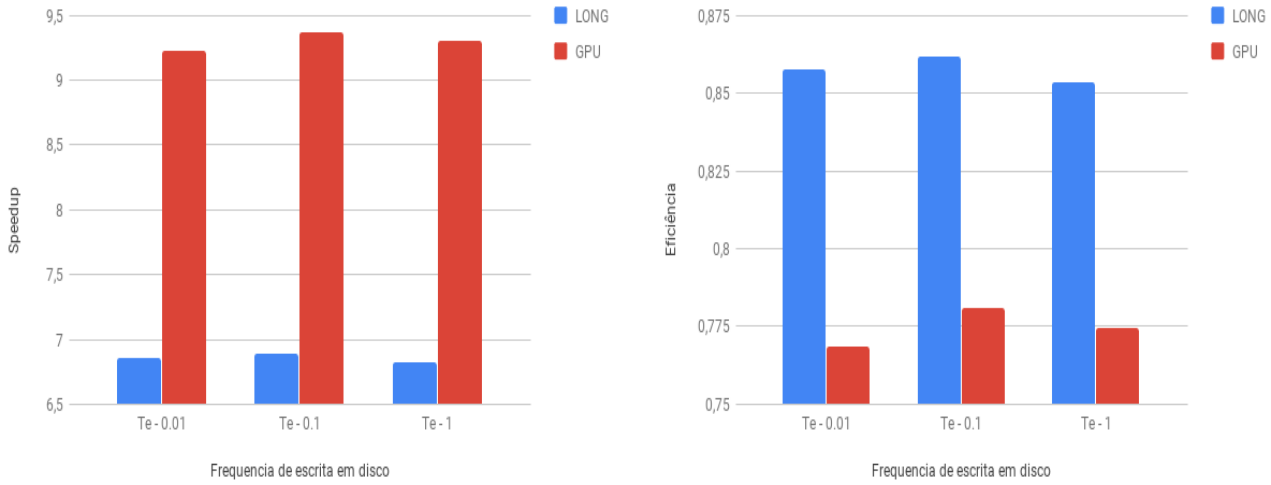
Figura 13 – *Speedup* e eficiência da versão *OpenMP* executada nas filas LONG com 8 *threads* e GPU com 12 *threads* variando o tamanho do passo de tempo.



(a) Gráfico de *Speedup* (b) Gráfico de Eficiência

Ao analisar as execuções em OpenMP variando a frequência de escrita em disco, percebemos que o *speedup* se mantém praticamente constante em todos os casos de teste. Observamos também que a vantagem de *speedup* da fila GPU em relação a fila LONG se mantém, bem como a superioridade na eficiência da fila LONG. Na figura (14) são exibidas essas informações.

Figura 14 – *Speedup* e eficiência da versão *OpenMP* executada nas filas LONG com 8 *threads* e GPU com 12 *threads* variando a frequência de escrita em disco.



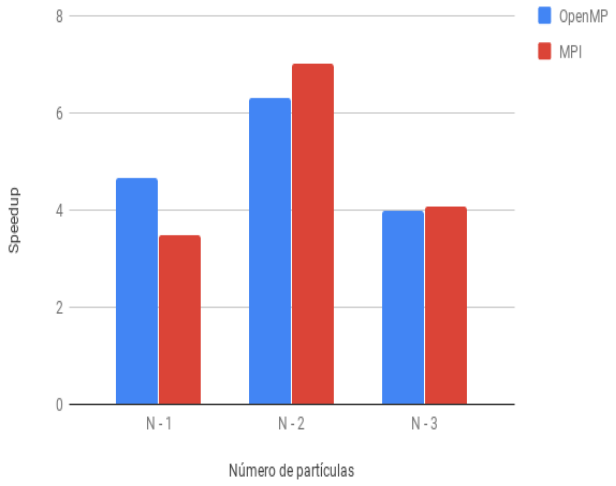
(a) Gráfico de *Speedup*

(b) Gráfico de Eficiência

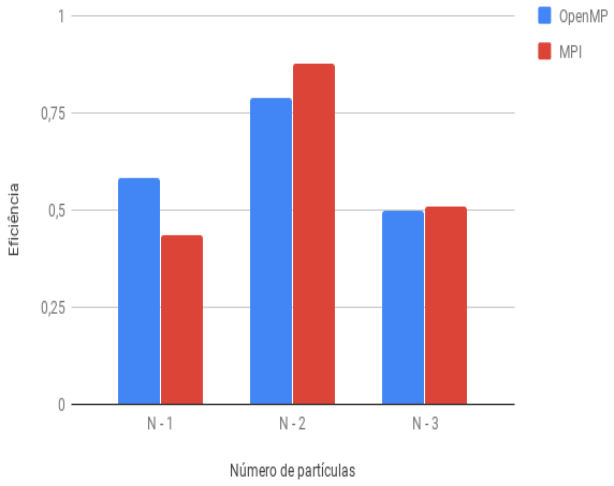
A fila GPU dispõe de um nó de computo com 12 núcleos e foram criadas 12 *threads* para a execução em OpenMP, já o processador da fila LONG possui 8 núcleo e foram criadas 8 *threads* para a execução dos teste. Essa diferença na quantidade de núcleos e *threads*, explica o maior *speedup* obtido na fila GPU. Entretanto, os núcleos da fila LONG operam numa frequência maior que os da fila GPU, o que explica a maior eficiência na fila LONG.

Com a finalidade de comparar os resultados obtidos de OpenMP com o MPI, executamos os mesmos casos de teste na versão MPI utilizando apenas um nó de computo do CACAU. Usamos a mesma quantidade de processos que o OpenMP usou de *threads* nas execuções em MPI, ou seja, na fila LONG foram utilizados 8 processos e na fila GPU 12 processos, e variamos N, T_s e T_e. Analisando os resultados variando o número de partículas exibidos na figura (15), podemos notar valores bem próximos de *speedup* e eficiência na fila LONG, com o OpenMP sendo um pouco melhor no caso de teste N_1 para uma pequena quantidade de partículas e com o MPI sendo levemente superior nos demais. Já na fila GPU observamos uma boa vantagem do MPI em relação ao OpenMP tanto em *speedup* quanto em eficiência em todos os casos de teste.

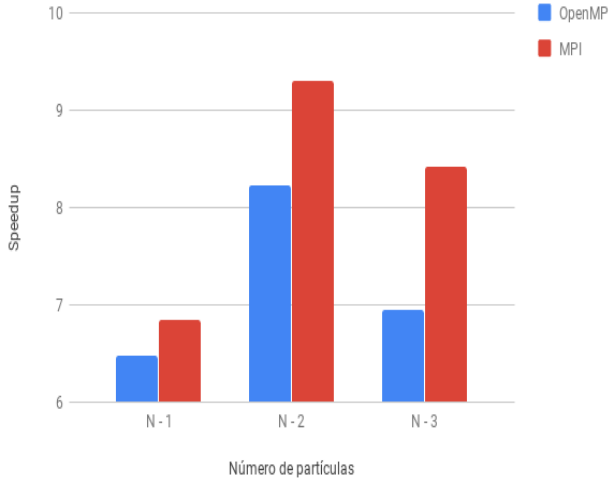
Figura 15 – Comparativo das versões OpenMP vs MPI em *Speedup* e eficiência executada nas filas LONG e GPU variando o número de partículas.



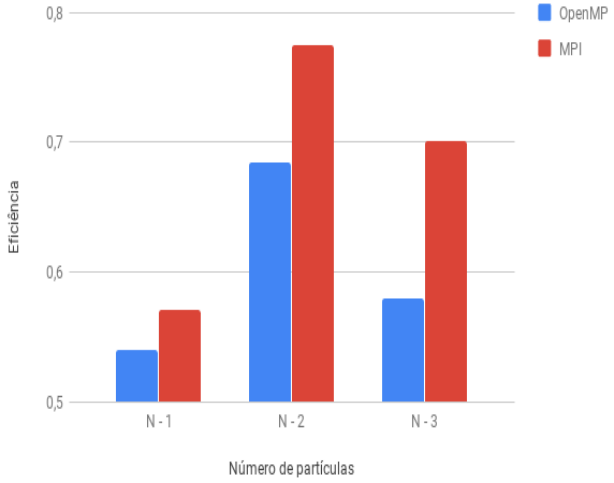
(a) Gráfico de *Speedup* na fila LONG



(b) Gráfico de Eficiência na fila LONG



(c) Gráfico de *Speedup* na fila GPU

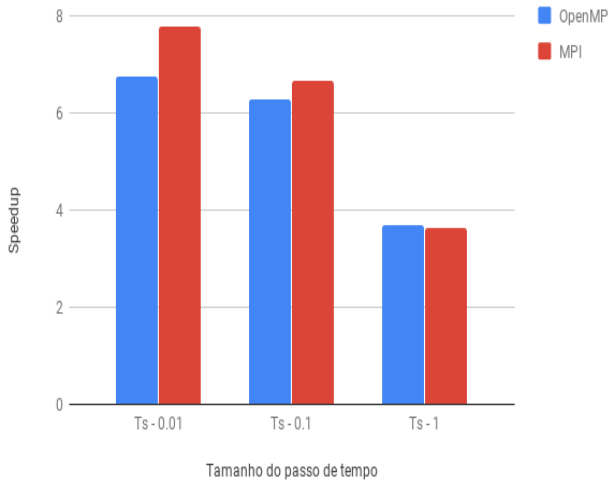


(d) Gráfico de Eficiência na fila GPU

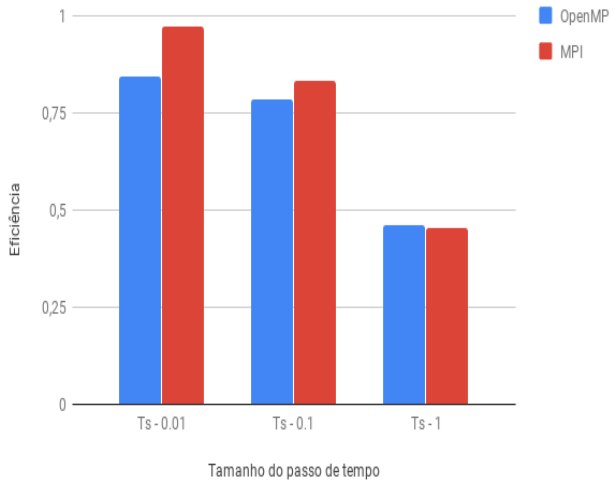
Quando comparamos os resultados variando o tamanho do passo de tempo T_s, verificamos na figura (16) que o MPI supera o OpeMP na maioria dos casos de teste na fila LONG tendo apenas uma pequena desvantagem quando o passo de tempo $T_s = 1$, ou seja, quando a aplicação executa menos simulações. Na fila GPU, o MPI é superior em todos os casos de teste, tanto em *speedup* quanto em eficiência. O pico de *speedup* e eficiência nas duas filas, são obtidos quando $T_s = 0.01$, na maior quantidade de simulações.

Os resultados da comparação quando variamos a frequência de escrita em disco se mantém praticamente constantes como exibido na figura (17), com o MPI superior ao OpenMP em todos os casos tanto em *speedup* quanto em eficiência duas nas filas.

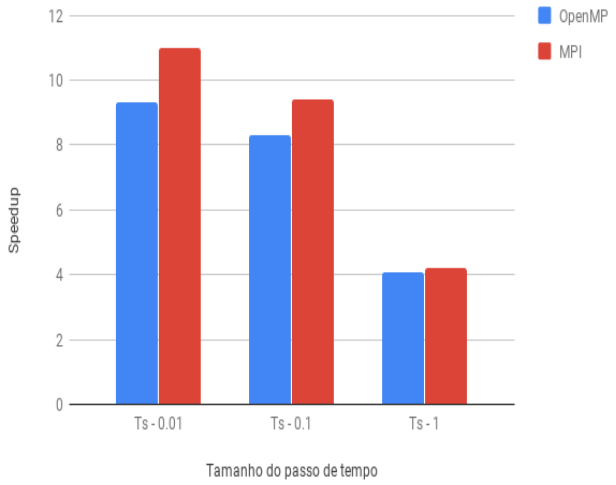
Figura 16 – Comparativo das versões OpenMP vs MPI em *Speedup* e eficiência executada nas filas LONG com 8 *threads* e GPU com 12 *threads* variando o tamanho do passo de tempo.



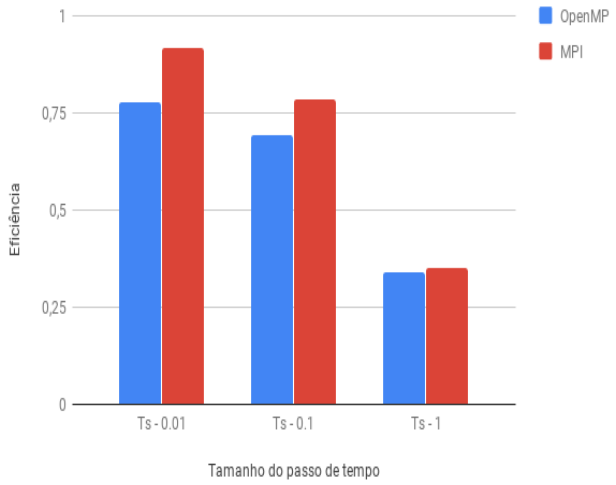
(a) Gráfico de *Speedup* na fila LONG



(b) Gráfico de Eficiência na fila LONG



(c) Gráfico de *Speedup* na fila GPU

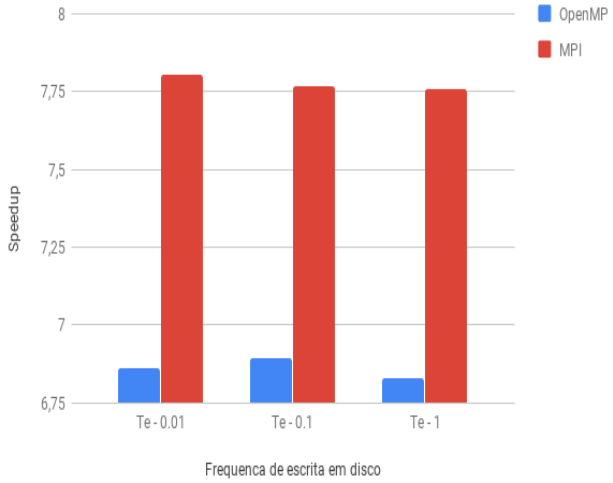


(d) Gráfico de Eficiência na fila GPU

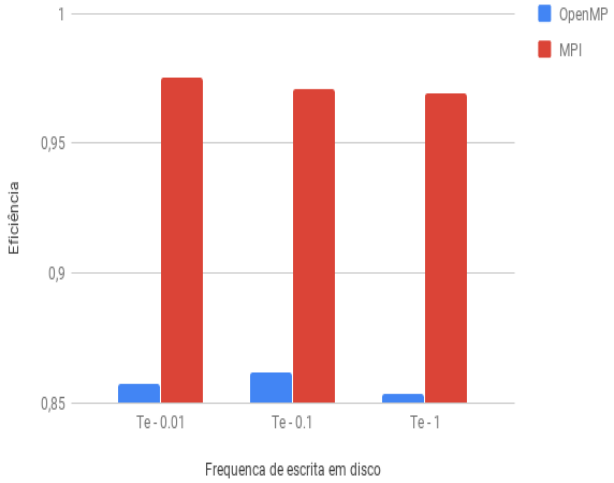
4.1.3 MPI

Os testes em *MPI* foram realizados seguindo o mesmo esquema dos teste em *OpenMP*, primeiro variamos o número de partículas, depois o tamanho do passo de tempo e em seguida a frequência de escrita em disco. No entanto, como *MPI* permite paralelização em ambientes de memoria distribuída através da passagem de mensagens, utilizamos dos recursos disponíveis para avaliar a escalabilidade da aplicação utilizando mais nós do CACAU na execução dos testes. Realizamos os testes em 1, 2 e 4 nós da fila LONG e em 1, 2 e 3 nós da fila *GPU*. Na fila LONG foram utilizados 8 processos nas execuções em 1 nó, 16 processos nas execuções em 2 nós e 32 processos nas execuções

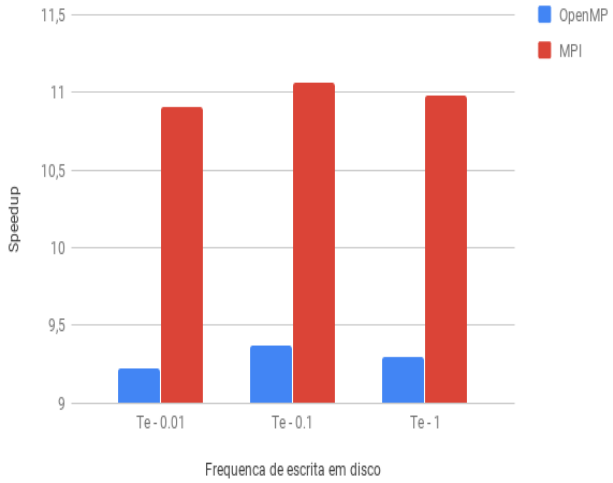
Figura 17 – Comparativo das versões OpenMP vs MPI em *Speedup* e eficiência executada nas filas LONG com 8 *threads* e GPU com 12 *threads* variando a frequência de escrita em disco.



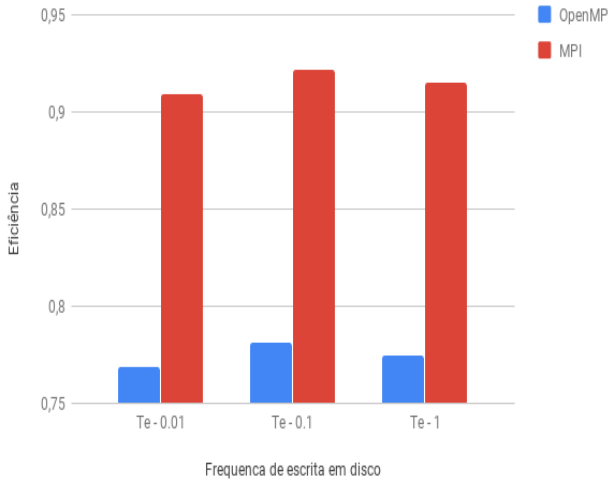
(a) Gráfico de *Speedup* na fila LONG



(b) Gráfico de Eficiência na fila LONG



(c) Gráfico de *Speedup* na fila GPU



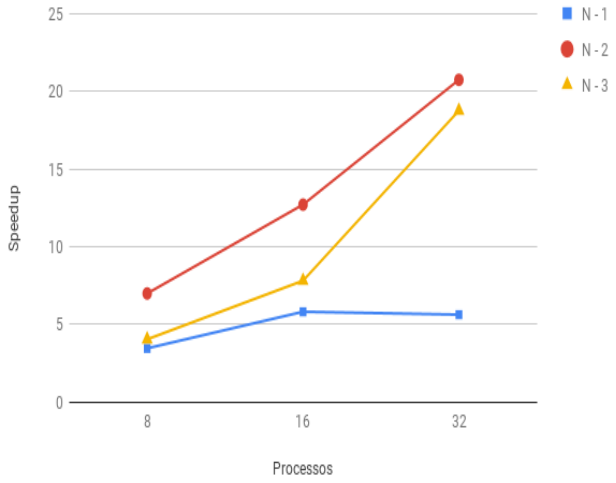
(d) Gráfico de Eficiência na fila GPU

em 4 nós. Já na fila *GPU* utilizamos 12 processos em 1 nó, 24 processos em 2 nós e 36 processos em 3 nós.

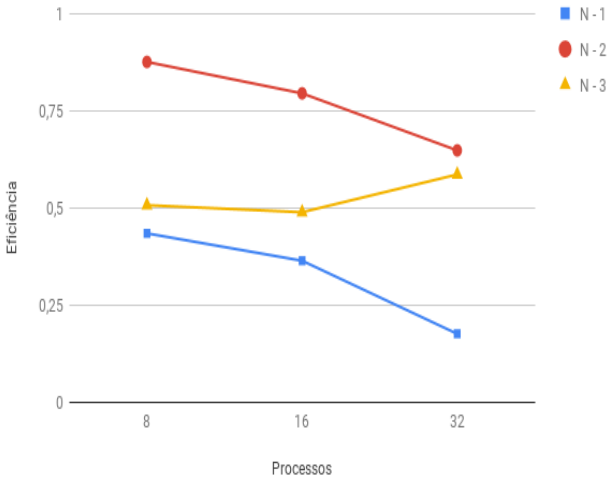
A partir da analise dos resultados exibidos na figura (18) variando o número de partículas, podemos verificar na fila LONG que o *speedup* é maior para o caso de teste N_2, com todas as diferentes quantidades de processos, atingindo o maior *speedup* quando utilizado 32 processo nos 4 nós. A eficiência também é maior no caso N_2 e atinge o auge utilizando 8 processos em 1 nó LONG. Na fila GPU, observamos um *speedup* maior também para N_2, porém o *speedup* para N_3 esta bem próximo, o que não acontece nos nós LONG. O topo do *speedup* é atingido quando utilizado os 36 processos nos 3 nós da fila GPU. A curva de eficiência da fila GPU é também muito semelhante a da fila

LONG, exceto no caso N_2 onde a eficiência é bem semelhante ao do caso N_3. O máximo de eficiência é atingido utilizando 12 processos em apenas 1 nó GPU no caso N_2.

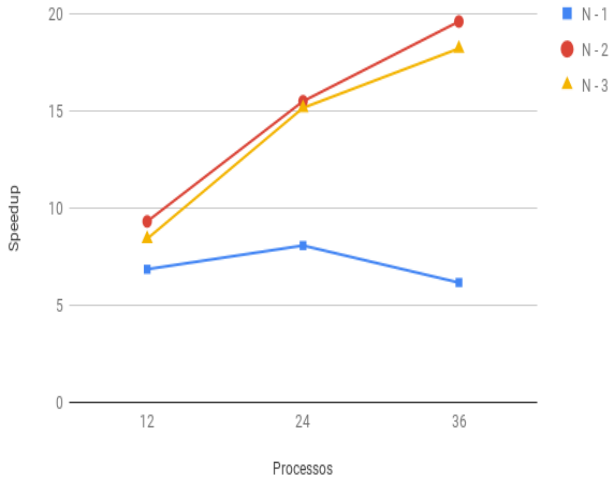
Figura 18 – Escalabilidade MPI em *Speedup* e eficiência executada nas filas LONG com e GPU com variando o número de processos e a quantidade de partículas.



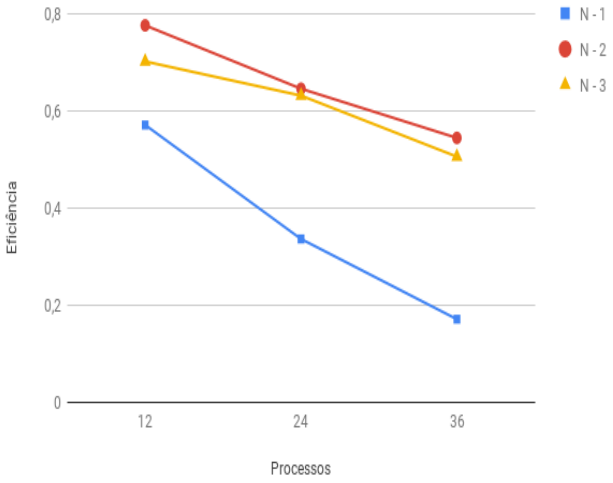
(a) Gráfico de *Speedup* na fila LONG



(b) Gráfico de Eficiência na fila LONG



(c) Gráfico de *Speedup* na fila GPU

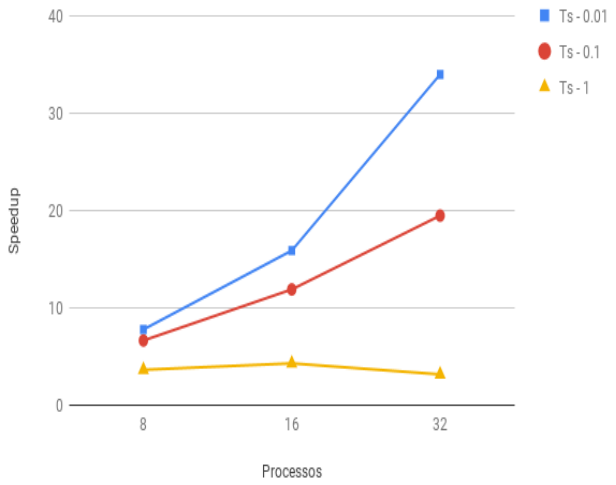


(d) Gráfico de Eficiência na fila GPU

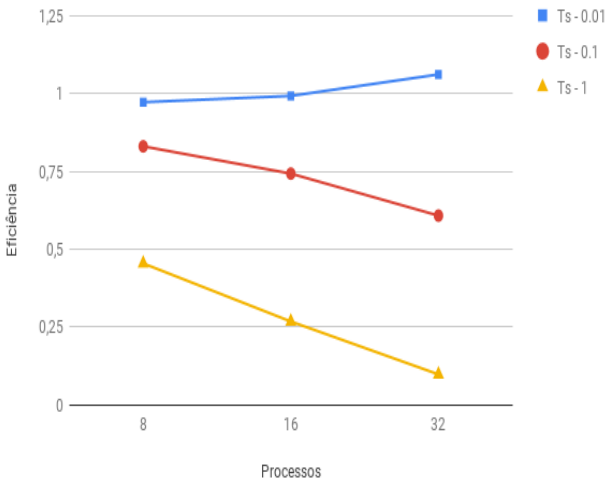
Quando avaliamos a escalabilidade variando o tamanho do passo de tempo exibido na figura (19), podemos verificar um *speedup* muito alto no caso de teste $T_s = 0,01$ utilizando 32 processos em 4 nós da fila LONG. O *speedup* na fila LONG aumenta a medida em que diminuímos o tamanho do passo de tempo e aumentamos a quantidade de processos. Quanto menor o tamanho do passo de tempo mais simulações a aplicação realiza. A eficiência máxima obtida nos nós da fila LONG também foi no caso de teste $T_s = 0,01$ e ela decai a medida em que aumenta o tamanho do passo de tempo e a quantidade de processos. Na fila GPU observamos um comportamento muito

semelhante ao da fila LONG, em que o auge do *speedup* se encontra no caso $T_s = 0,01$ utilizando 36 processos nos 3 nós da GPU. Já a eficiência máxima notada na fila GPU foi também no $T_s = 0,01$, porém na utilização de 12 processos em apenas 1 nó GPU.

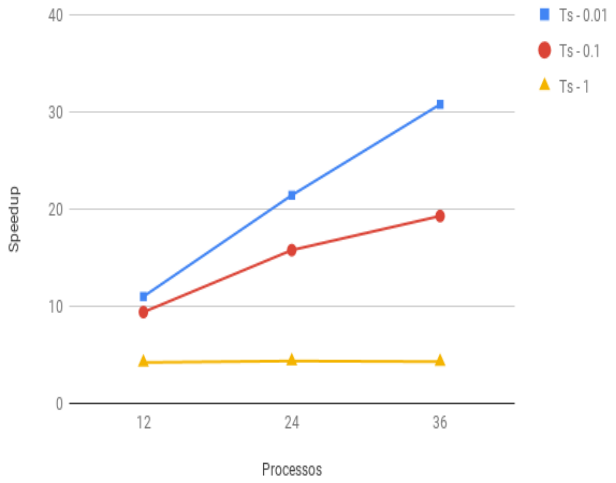
Figura 19 – Escalabilidade MPI em *Speedup* e eficiência executada nas filas LONG com e GPU com variando o número de processos e o tamanho do passo de tempo.



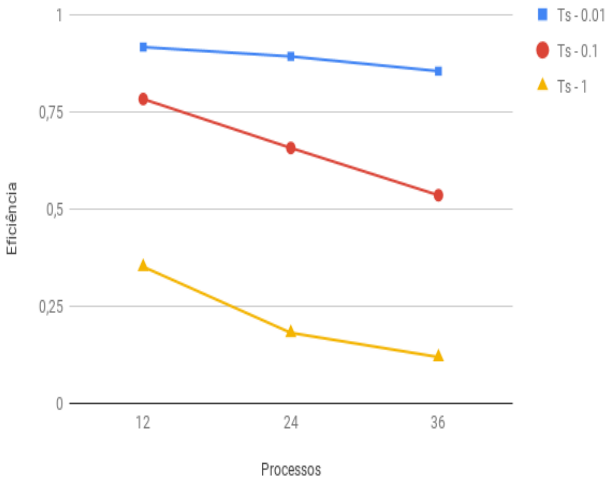
(a) Gráfico de *Speedup* na fila LONG



(b) Gráfico de Eficiência na fila LONG



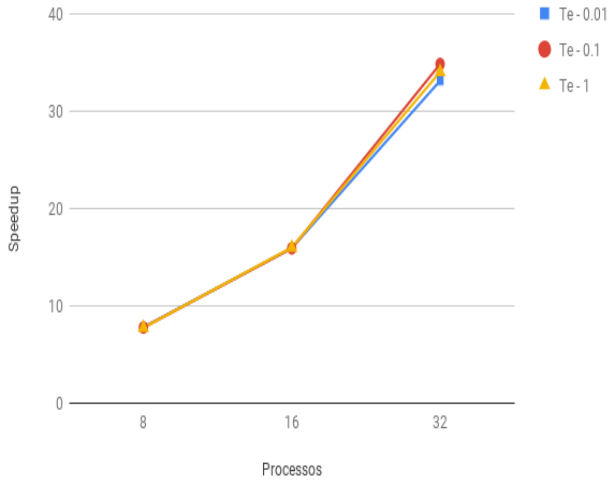
(c) Gráfico de *Speedup* na fila GPU



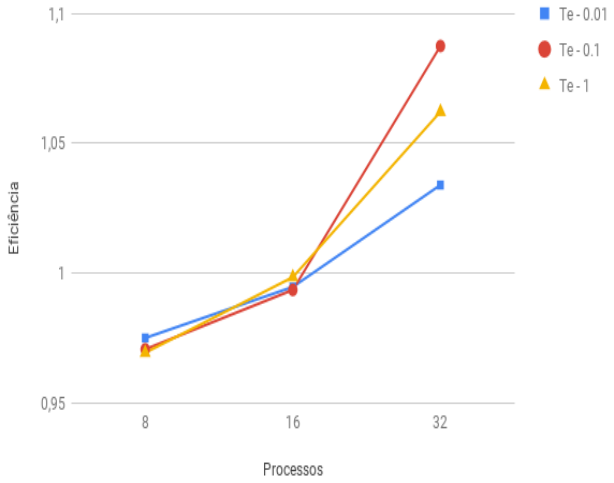
(d) Gráfico de Eficiência na fila GPU

Avaliando a escalabilidade variando desta vez a frequência da escrita em disco, como mostra a figura (20), percebemos que para todos os valores de T_e o *speedup* obtido é praticamente o mesmo, ele aumenta conforme aumentamos o número de processos tanto nos nós LONG quanto GPU. O maior *speedup*, é obtido quando utilizamos 32 processos nos 4 nós na fila LONG e 36 processos nos 3 nós na fila GPU. A eficiência é levemente diferente na fila LONG quando observamos as execuções com 32 processos e na fila GPU quando observamos as execuções com 24 e 36 processos.

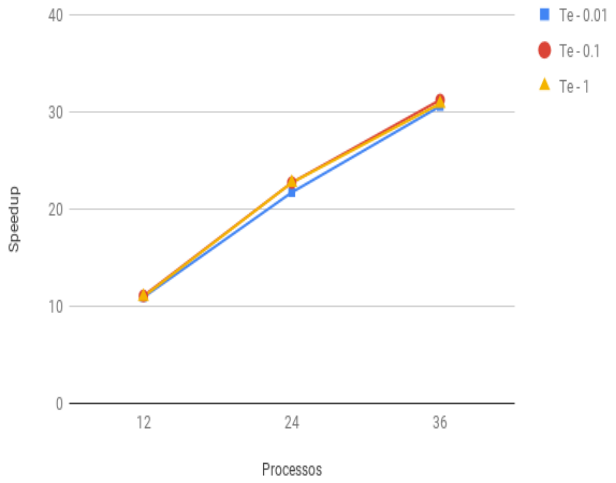
Figura 20 – Escalabilidade MPI em *Speedup* e eficiência executada nas filas LONG com e GPU com variando o número de processos e a frequência de escrita em disco.



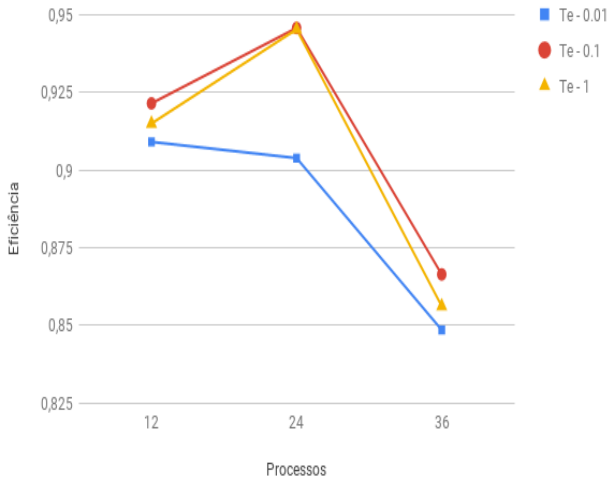
(a) Gráfico de *Speedup* na fila LONG



(b) Gráfico de Eficiência na fila LONG



(c) Gráfico de *Speedup* na fila GPU



(d) Gráfico de Eficiência na fila GPU

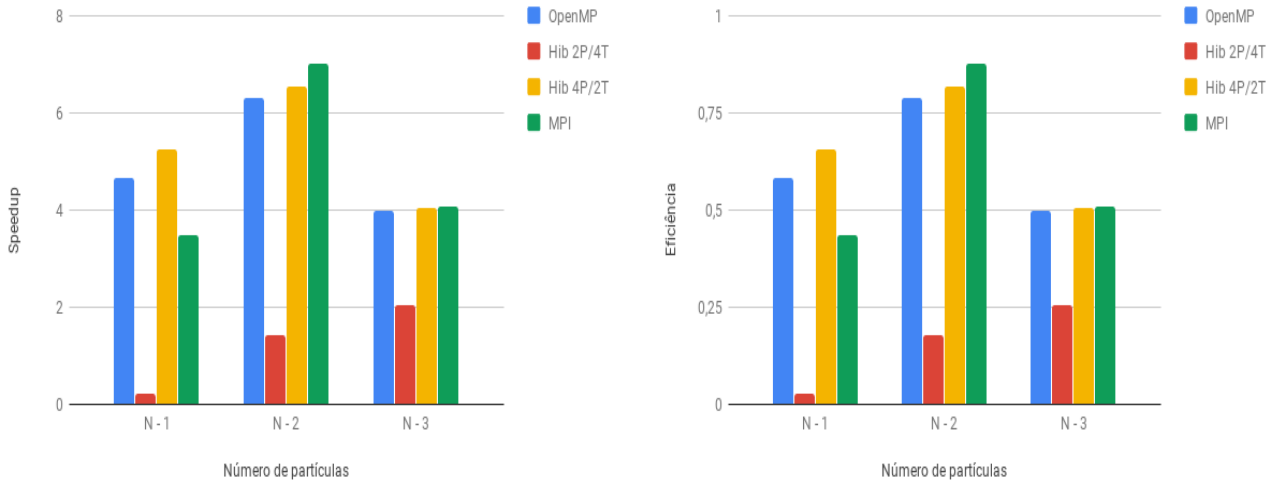
4.1.4 Hibrido (MPI / OpenMP)

Os testes da versão hibrida da aplicação foram realizados utilizando 1, 2 e 4 nós da fila LONG e 1, 2 e 3 nós da fila GPU. Quando avaliamos a aplicação em 1 único nó, comparamos a versão hibrida às versões em OpenMP e MPI, nos demais casos a versão hibrida foi comparada apenas ao MPI. A versão hibrida foi testada em diferentes configurações de acordo com a quantidade de nós utilizados.

A versão hibrida foi executada de duas formas diferentes em 1 nó da fila LONG, a primeira foi utilizando 2 processos e 4 *threads* e a segunda utilizando 4 processos e 2 *threads*. Os resultados obtidos da versão hibrida são exibidos na figura (21) e comparando

com os resultados do OpenMP e MPI, notamos que para N_1 a versão hibrida com 4 processos e 2 *threads* leva uma vantagem em relação as demais. Para N_2 a versão em MPI é superior as outras, já para N_3 o *speedup* é bem parecido tanto na verão OpenMP, MPI e hibrido com 4 processos e 2 *threads*. Em todos os casos a versão hibrida com 2 processos e 4 *threads* é inferior a todas as outras em relação ao *speedup*. Quanto a eficiência, como foram utilizada as mesmas quantidades de recursos para todos os testes, ela segue a mesma linha do *speedup* neste caso, tendo seu auge no caso de teste N_2 utilizando MPI.

Figura 21 – Comparação da versão híbrida OpenMP / MPI em um nó LOGN variando N.



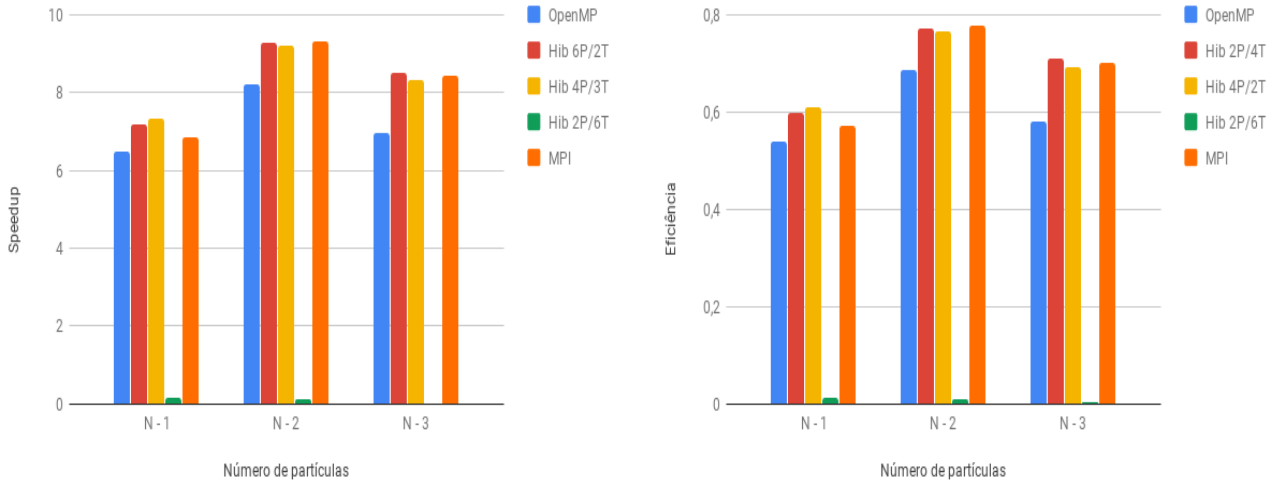
(a) Gráfico de *Speedup* na fila LONG

(b) Gráfico de Eficiência na fila LONG

A versão hibrida executada em 1 nó da fila GPU, foi dividida em 3 configurações diferentes, a primeira foi utilizando 6 processos e 2 *threads*, a segunda 4 processos e 3 *threads* e a teceria 2 processos e 6 *threads* seus resultados são exibidos na figura (22). O *speedup* das versões hibridas utilizando 6 e 4 processos foi bem parecida com o da versão MPI, levemente superior ao da versão em OpenMP. A versão hibrida utilizando 2 processos não obteve ganhos na sua execução. O maior *speedup* das versões hibridas foi obtido na execução do caso de teste N_2. A eficiência também é muito parecida dentre as versões com 6 e 4 processos, que são muito próximas da versão em MPI e assim como no *speedup*, levemente superior ao OpenMP.

Definimos 3 configurações distintas para executar a versão hibrida usando 2 nós da fila LONG, a primeira com 8 processos e 2 *threads*, a segunda com 4 processos e 4 *threads* e a terceira com 2 processos e 8 *threads*. Podemos observar os resultados exibidos na figura (23), que a versão hibrida com 8 processos tem um maior *speedup* no caso de teste N_1, já no segundo caso de teste N_2 a versão MPI é superior, chegando a um certo equilíbrio de desempenho quando observarmos o caso N_3. A versão hibrida com 2 processos não apresentou um bom *speedup* em nenhum dos casos de teste. O maior

Figura 22 – Comparação da versão híbrida OpenMP / MPI em um nó GPU variando N.

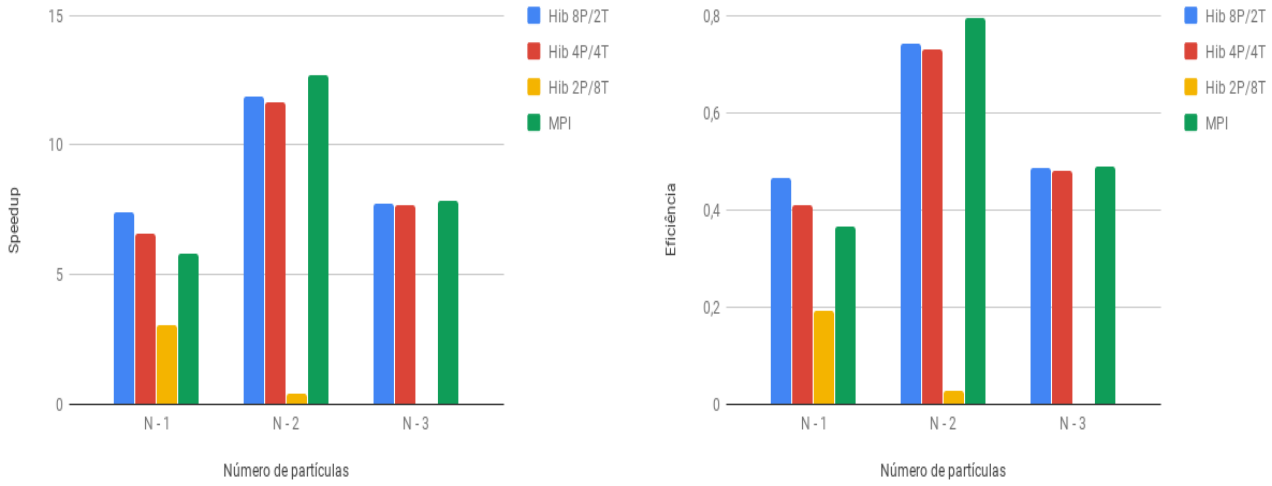


(a) Gráfico de *Speedup* na fila GPU

(b) Gráfico de Eficiência na fila GPU

speedup foi alcançado no caso N_2 com a versão em MPI. A eficiência segue a mesma linha do *speedup* nestes casos, sendo maior na versão hibrida com 8 processos no caso de teste N_1 e no caso N_2 onde é alcançado o pico de eficiência, o MPI é superior.

Figura 23 – Comparação da versão híbrida OpenMP / MPI em 2 nós LONG variando N.



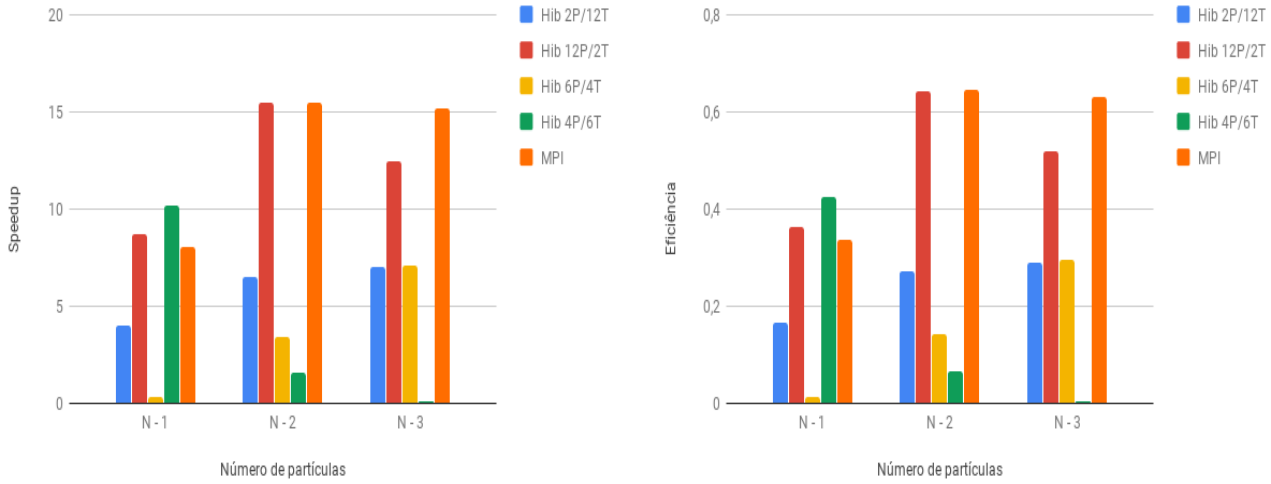
(a) Gráfico de *Speedup* na fila LONG

(b) Gráfico de Eficiência na fila LONG

Para executar os testes da versão hibrida em 2 nós da fila GPU exibidos na figura (24), foram definidas 4 configurações, primeiro 2 processos e 12 *threads*, segundo 12 processos e 2 *threads*, terceiro 6 processos e 4 *threads* e quarto 4 processos e 6 *threads*. Na execução do casos de teste N_1 notamos um maior *speedup* na versão hibrida com 4 processos, já para N_2 o maior *speedup* foi atingido pela versão hibrida com 12 processos, seguido de perto pela versão MPI. No caso N_3 o MPI supera os demais em *speedup*.

A versões hibridas com 2 e 6 processos não obteveram um *speedup* significativo em nenhum dos casos, a versão hibrida com 4 processos só obteve um grande *speedup* no primeiro caso de teste, nos demais não obteve aceleração. A versão hibrida com 12 processos manteve-se consistente. Quanto a eficiência, tivemos o melhor resultado no caso N_1 para a versão hibrida com 4 processos, no caso N_2 as versões hibrida com 12 processos e MPI ficam praticamente empatadas, vale ressaltar que foi neste caso onde a excussão obteve sua maior eficiência, no caso N_3 o MPI supera todos os outros em eficiência.

Figura 24 – Comparação da versão híbrida OpenMP / MPI em 2 nós GPU variando N.



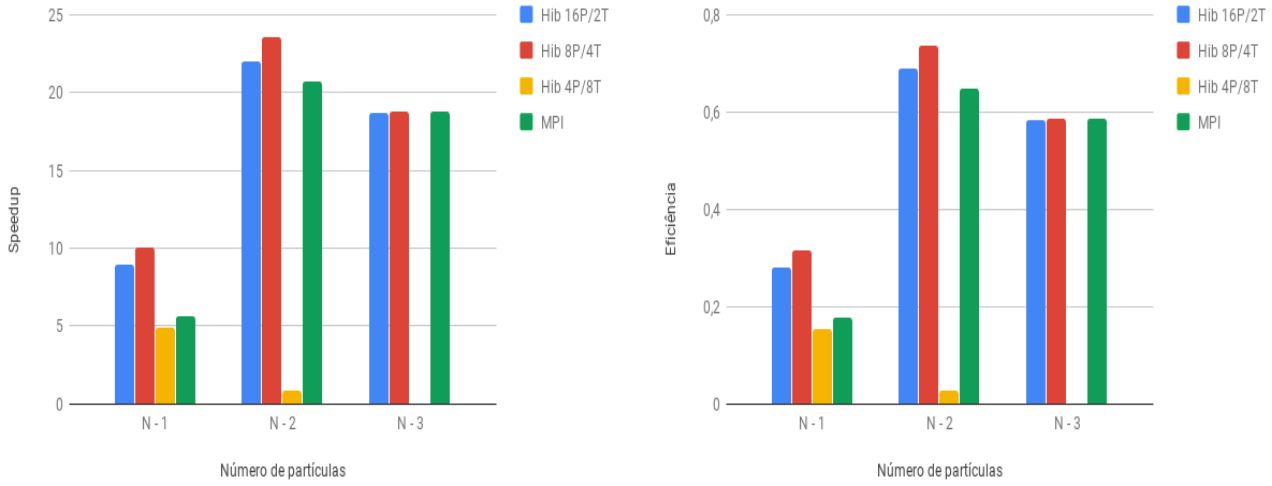
(a) Gráfico de *Speedup* na fila GPU

(b) Gráfico de Eficiência na fila GPU

Para executar a versão hibrida utilizando 4 nós da fila LONG cujos resultados são exibidos na figura (25), definimos 3 configurações, primeira 16 processos e 2 *threads*, segundo 8 processos e 4 *threads* e terceiro 4 processos e 8 *threads*. O maior speedup foi alcançado com a versão hibrida utilizando 8 processos no caso de teste N_2, as versões hibrida utilizando 16 processos e MPI apresentaram um *speedup* próximo da versão hibrida com 8 processos porém levemente inferior. A versão hibrida com com 4 processos ão apresentou *speedup* significativo. A maior eficiência também ficou com a versão hibrida com 8 processos seguida de perto pelas versões hibrida com 12 processos e MPI, exceto no caso N_1 onde a versão hibrida com 8 processos é bem superior aos demais.

Os testes executados com a versão hibrida utilizando 3 nós da fila GPU, seguiram 3 configurações distintas, a primeira com 12 processos e 3 *threads*, a segunda com 6 processos e 6 *threads* e a terceira com 3 processos e 12 *threads*. Conforme verificamos na figura 26, a versão hibrida com 6 processos obteve o maior *speedup* para o caso de teste N_1 superando todos os demais de forma significante. No caso de teste N_2 a versão hibrida com 6 processos obteve o maior *speedup*, sendo este o máximo de *speedup* obtido

Figura 25 – Comparação da versão híbrida OpenMP / MPI em 4 nós LONG variando N.

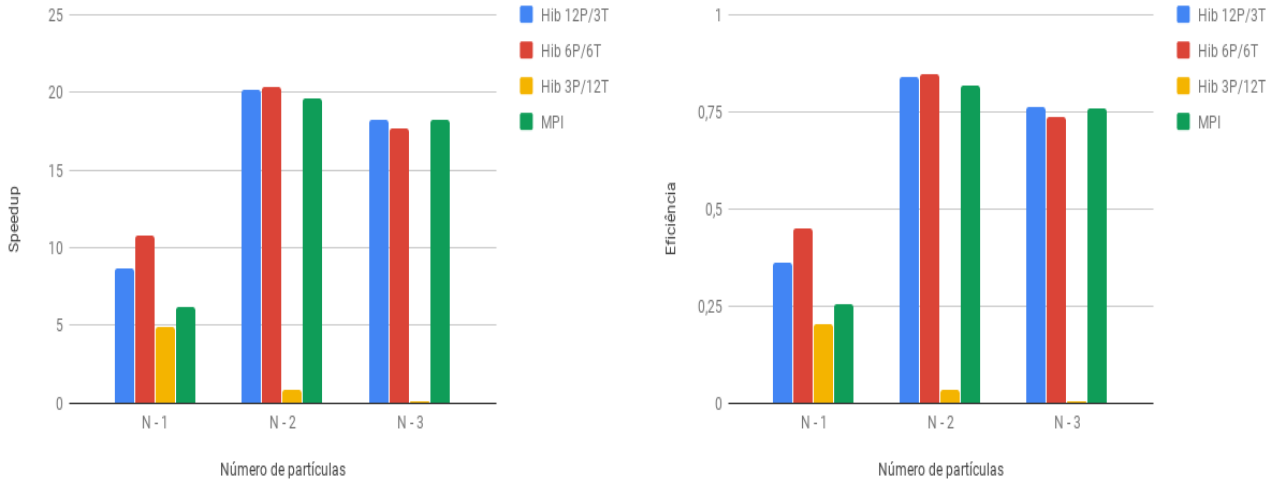


(a) Gráfico de *Speedup* na fila LONG

(b) Gráfico de Eficiência na fila LONG

dentre estes casos de teste, no entanto a versão hibrida com 12 processos e a versão em MPI apresentaram resultados bem próximos. No caso de teste N_3 o MPI supera levemente os demais. A versão hibrida com 3 processos não obteve *speedup* significativo. Quanto a eficiência, tivemos um pico com a versão hibrida com 6 processos no caso de teste N_2, no caso de teste N_1 a versão com 6 processos supera as demais, já no caso N_3 o MPI é seguido da versão hibrida com 12 processos tem uma eficiencia ligeiramente maior.

Figura 26 – Comparação da versão híbrida OpenMP / MPI em 3 nós GPU variando N.



(a) Gráfico de *Speedup* na fila GPU

(b) Gráfico de Eficiência na fila GPU

4.2 Soluções para GPGPU

Os testes realizados com as soluções para *GPGPUs* seguiram os desenhos experimentais descritos no capitulo (3). Executamos os códigos *CUDA* e *OpenCL/HPL* em 1 nó da fila *GPU*, variando primeiro o número de partículas N e mantendo fixos os tamanhos do passo de tempo T_s em 0.01 e a frequência de escrita em disco T_e em 1. Em seguida, executamos novamente mantendo valores fixos agora para N = 400.384 e o T_e em 1 e variando o tamanho do passo de tempo T_s. Logo após, fizemos novas execuções, desta vez com uma terceira configuração, mantendo fixos o N = 400.384 e o T_s em 0.01 e variando agora a frequência da escrita em disco T_e. Os resultados obtidos são mostrados em gráficos nas próximas subseções.

4.2.1 CUDA vs OpenCL

Os testes executados com a versão CUDA utilizando 1 nó da fila *GPU* foram comparados com a versão *OpenCL/HPL* executado na mesma *GPU* e com as melhores execuções em *CPU* utilizando *MPI* com 32 processos em 4 nós da fila LONG, as versões hibridas com 16 processos e 2 *threads*, e 8 processos e 4 *threads*.

Quando variamos o N, número de párticas e mantemos fixo o T_s e o T_e, tivemos um desempenho maior da versão hibrida com 8 processos para N_1. Para N_2 a versão CUDA leva uma pequena vantagem seguido de perto pelas versões hibridas e pelo MPI. Finalmente em N_3, a vantagem do CUDA cresce muito em relação a todas as outras implementações chegando a atingir uma *speedup* de $42,6$ contra $18,7$ do segundo colocado MPI. Estes resultado estão ilustrados na figura 27.

Quando variamos o T_s, tamanho do passo de tempo e mantemos fixo o N e o T_e, tivemos um desempenho levemente maior da versão hibrida com 16 processos para T_s = $0,01$. Para T_s = 0.1 a versão CUDA leva uma pequena vantagem seguido de perto pelas duas versões hibridas. Finalmente em T_s = $1,0$, a vantagem do CUDA cresce muito em relação a todas as outras implementações chegando a atingir uma *speedup* anida maior que do experimento anterior totalizando $62,8$ contra $34,9$ do segundo colocado, a versão hibrida com 8 processos. Estes resultado estão ilustrados na figura 28.

A figura 29 ilustra os resultados quando variamos a frequência de escrita em disco T_e e mantemos fixo N e T_s. Em todos os casos do experimento, *CUDA* teve um desempenho muito superior aos demais.

Após a comparação com as execuções utilizando 4 nós da fila LONG, comparamos a versão *CUDA* utilizando 1 nó da fila *GPU* com a versão *OpenCL/HPL* executado na mesma *GPU* e com as melhores execuções em *CPU* utilizando *MPI* com 36 processos em 3 nós da fila GPU, as versões hibridas com 12 processos e 3 *threads*, e 6 processos e 6

Figura 27 – Comparação das versões *CUDA*, híbrida, *MPI* e *OpenCL* em nós LONG variando o número de partículas.

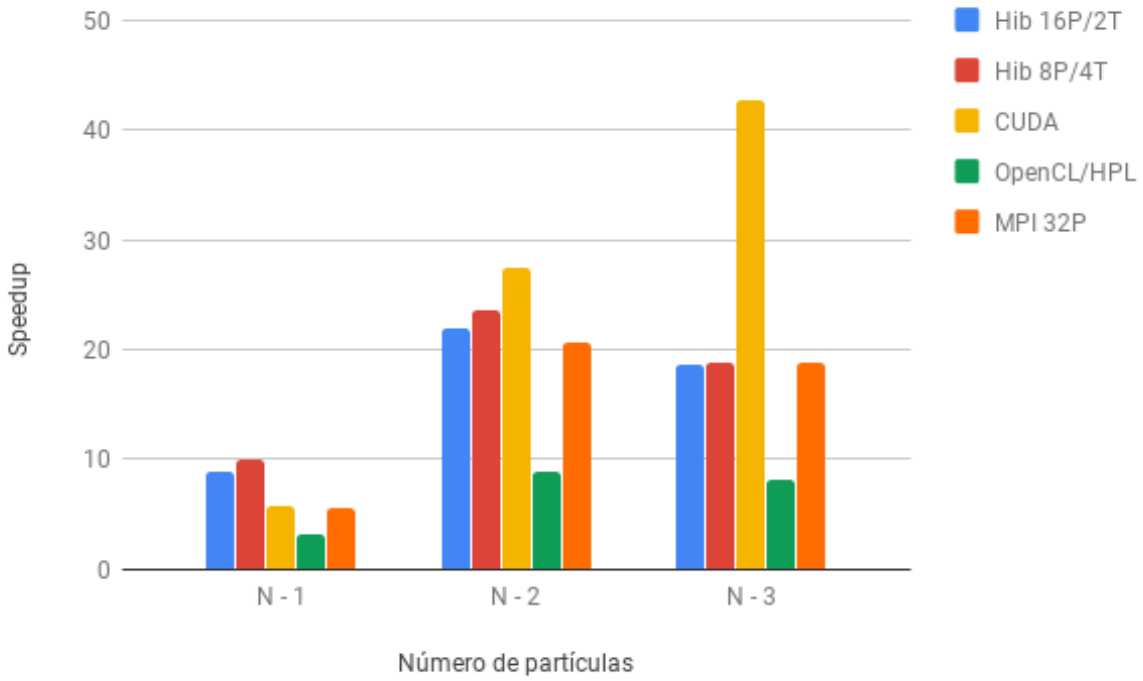


Figura 28 – Comparação das versões *CUDA*, híbrida, *MPI* e *OpenCL* em nós LONG variando o tamanho do passo de tempo.

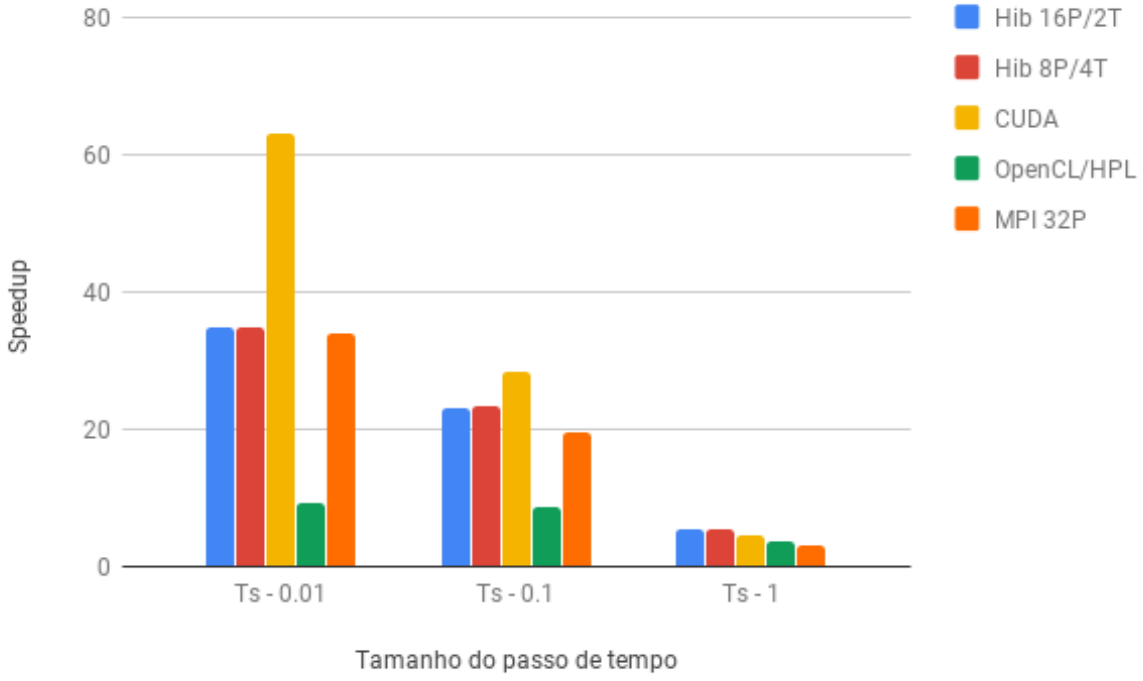
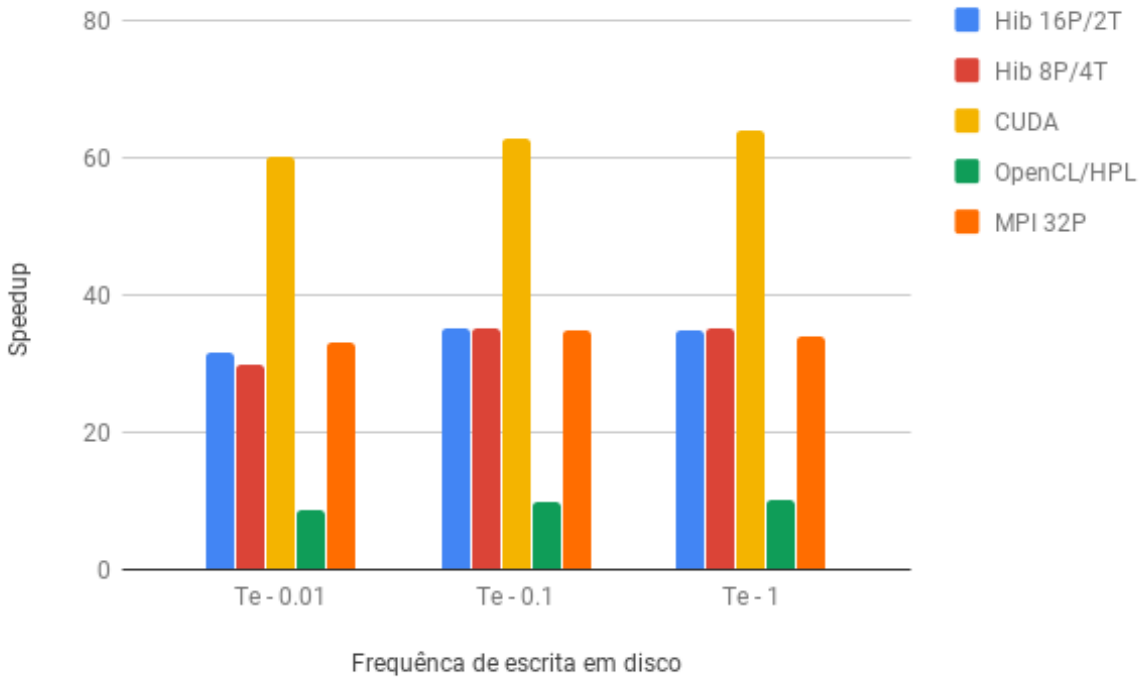


Figura 29 – Comparação das versões *CUDA*, híbrida, *MPI* e *OpenCL* em nós LONG variando a frequência de escrita em disco.



threads.

Quando variamos o N, número de párticas e mantemos fixo o T_s e o T_e, tivemos um desempenho maior da versão hibrida com 6 processos para N_1. Para N_2 a versão CUDA leva vantagem seguida pelas versões hibridas e pelo MPI. Finalmente em N_3, a vantagem do CUDA cresce muito em relação a todas as outras implementações chegando a atingir uma *speedup* de $39,1$ contra $18,2$ do segundo colocado hibrido com 12 processos. Estes resultado estão ilustrados na figura 30.

Quando variamos o T_s, tamanho do passo de tempo e mantemos fixo o N e o T_e, tivemos um desempenho levemente maior da versão hibrida com 6 processos para $T_s = 0,01$. Para $T_s = 0.1$ a versão CUDA leva uma pequena vantagem seguido pelas duas versões hibridas e pelo *MPI*. Finalmente em $T_s = 1,0$, a vantagem do CUDA cresce muito em relação a todas as outras implementações chegando a atingir uma *speedup* anida maior que do experimento anterior totalizando $57,3$ contra $30,7$ do segundo colocado, a versão MPI. Estes resultado estão ilustrados na figura 31.

A figura 32 ilustra os resultados quando variamos a frequência de escrita em disco T_e e mantemos fixo N e T_s. Em todos os casos do experimento, *CUDA* teve um desempenho muito superior aos demais.

Figura 30 – Comparação das versões *CUDA*, híbrida, *MPI* e *OpenCL* em nós GPU variando o número de partículas.

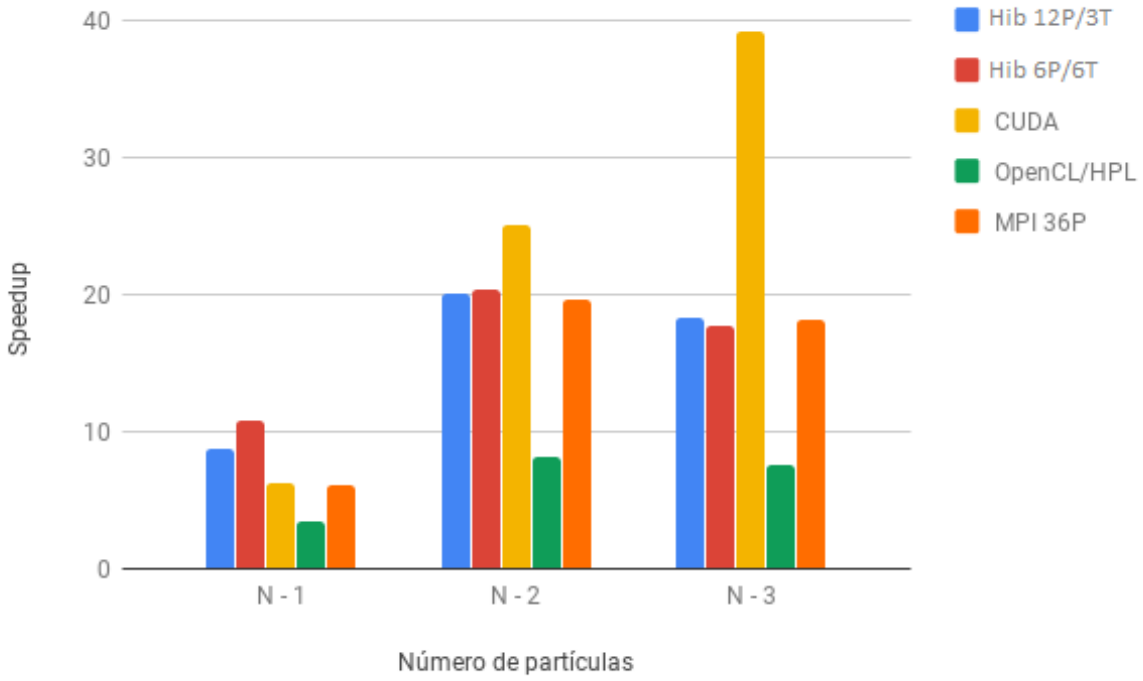


Figura 31 – Comparação das versões *CUDA*, híbrida, *MPI* e *OpenCL* em nós GPU variando o tamanho do passo de tempo.

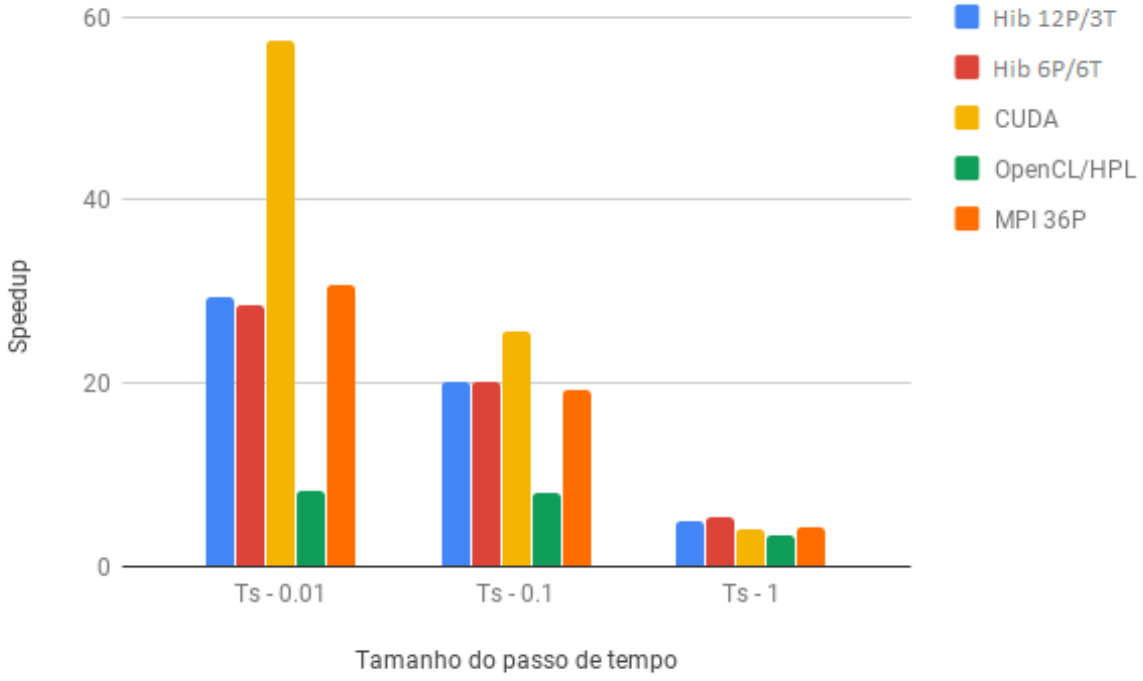
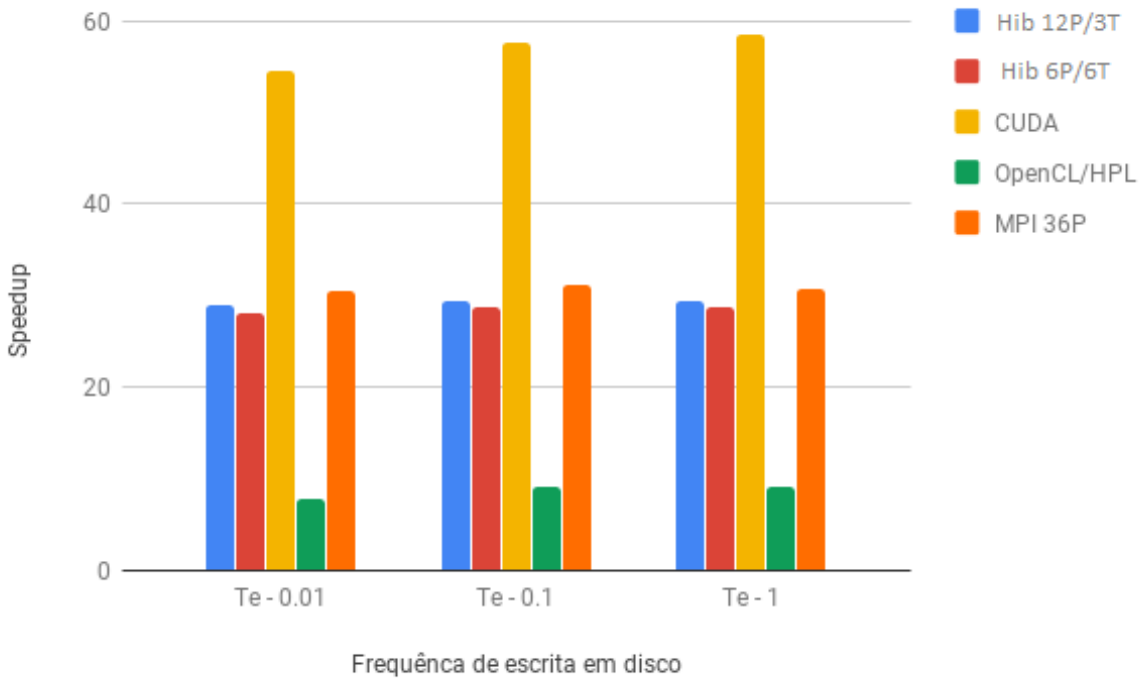


Figura 32 – Comparação das versões *CUDA*, híbrida, *MPI* e *OpenCL* em nós GPU variando a frequência de escrita em disco.



5 Conclusão

Neste trabalho foi realizado um estudo de escalabilidade do modelo HMF utilizando algumas técnicas tradicionais de paralelização de código em arquiteturas baseadas em COUs. Foram construídas então versões paralelas para arquiteturas baseadas em CPUs, usando OpenMP para ambientes de memoria compartilhada, MPI para ambientes de memoria distribuída e uma versão hibrida que mistura estas duas abordagens. Para avaliar o desempenho destas implemantações foram definidas duas métricas: *speedup* e eficiência. As mesmas mostram-se bem apropriada para medir tanto grau de desempenho entre o tempo de execução serial e o tempo de execução paralelo, quanto o grau de aproveitamento dos recursos computacionais.

Os experimentos realizados com as implementações paralelas foram satisfatórios e exibiram ótimos resultado. Quando variamos o número de partículas e mantemos fixo o tamanho do passo de tempo e a frequência de escrita em disco, conseguimos um *speedup* de 21,75 usando MPI, ou seja, nossa aplicação executou quase 22 vezes mais rápido do que o sequencial. Quando variamos o tamanho do passo de tempo e matemos fixo o número de partículas e a frequência da escrita em disco, nossos resultados foram ainda melhores com o *speedup* chegando a 34,8, isto é, quase 35 vezes mais rápido do que o sequencial. O resultado é semelhante quando variamos a frequência de escrita em disco e mantemos os outros parâmetros fixos.

Os resultados obtidos com as implementações para GPGPUS mostraram que a implementação baseada na arquitetura CUDA tem um desempenho significativamente superior às outras implementações. Com *speedup* variando entre 40 e 60 podem ser obtidos resultados significativamente mais rápido que com as implementações baseadas em CPU. Deve-se levar em consideração que nestes casos foi utilizado um único nó de computo, ao contrário dos melhores resultados obtidos com CPU que envolveram a utilização de vários nós de computo.

O desempenho da versão com OpenCL foi pode ser considerado fraco em comparação com a versão CUDA. Os resultados são apenas comparáveis com os das verões MPI e Híbrida. Não entanto estes resultados ainda podem ser considerados significativos levando em consideração que apenas um nó de computo foi utilizado. Testes de desempenho ainda estão sendo realizados e deverão ficar prontos até a data da apresentação dos resultados ante a banca, comparando a versão OpenCL rodando em CPU.

5.1 Trabalhos Futuros

Apesar dos resultados apresentarem melhorias significativas, há diversas abordagens e cenários que precisam ser explorados. Há um vasto campo de estudo, de técnicas que podem ser agregadas a solução atual. A seguir são apresentados possíveis trabalhos futuros.

Implementar um novo integrador mais eficiente;

Utilizar todas as *GPUs* disponíveis e um nó de computo simultaneamente.

Escalar o paralelismo utilizando na arquitetura *GPU* usando *MPI*, para operar em vários nós simultaneamente e assim aproveitar melhor os recursos disponíveis.

Avaliar o desempenho e a eficiência do paralelismo em múltiplas *GPUs*, com o objetivo de constatar a viabilidade da utilização de uma grande quantidade de recursos.

Realizar experimentos utilizando *OpenCL* em *GPUs* da fabricante *AMD*.

Referências

ABRAHAM, R.; MARSDEN, J. **Foundations of Mechanics**. AMS Chelsea Pub./American Mathematical Society, 2008. (AMS Chelsea publishing). ISBN 9780821844380. Disponível em: <https://books.google.com.br/books?id=4Y-ownk6ilsC>.

ANTONI, M.; RUFFO, S. Clustering and relaxation in hamiltonian long-range dynamics. **Phys. Rev. E**, American Physical Society, v. 52, p. 2361–2374, Sep 1995. Disponível em: <https://link.aps.org/doi/10.1103/PhysRevE.52.2361>.

ANTONOV, V. A. Most probable phase distribution in spherical star systems and conditions for its existence. **Vest. Leningrad Univ.**, v. 7, p. 135, 1962. Disponível em: <https://ci.nii.ac.jp/naid/10021088873/en/>.

ASANOVIC, K.; CATANZARO, B. C.; PATTERSON, D.; YELICK, K. The Landscape of Parallel Computing Research : A View from Berkeley. **EECS Department University of California Berkeley Tech Rep UCBEECS2006183**, v. 18, p. 19, 2006. ISSN 00010782. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.8705{&}rep=rep1{&}ty>.

BRODTKORB, A. R.; DYKEN, C.; HAGEN, T. R.; HJELMERVIK, J. M.; STORAASLI, O. O. State-of-the-art in heterogeneous computing. **Scientific Programming**, v. 18, n. 1, p. 1–33, 2010. ISSN 10589244.

BRODTKORB, A. R.; HAGEN, T. R.; SÆTRA, M. L. Graphics processing unit (GPU) programming strategies and trends in GPU computing. **Journal of Parallel and Distributed Computing**, v. 73, n. 1, p. 4–13, 2013. ISSN 07437315.

CAMPA, A.; DAUXOIS, T.; RUFFO, S. Statistical mechanics and dynamics of solvable models with long-range interactions. **Physics Reports**, Elsevier B.V., v. 480, n. 3-6, p. 57–159, 2009. ISSN 03701573. Disponível em: <http://dx.doi.org/10.1016/j.physrep.2009.07.001>.

CHALONY, M.; BARRÉ, J.; MARCOS, B.; OLIVETTI, A.; WILKOWSKI, D. Long-range one-dimensional gravitational-like interaction in a neutral atomic cold gas. **Physical Review A - Atomic, Molecular, and Optical Physics**, v. 87, n. 1, p. 1–7, 2013. ISSN 10502947.

CHAVANIS, P. H. Phase transitions in self-gravitating systems. **International Journal of Modern Physics B**, v. 20, n. 22, p. 3113–3198, 2006. Disponível em: <https://www.worldscientific.com/doi/abs/10.1142/S0217979206035400>.

CHAVANIS, P. H.; VATTEVILLE, J.; BOUCHET, F. Dynamics and thermodynamics of a simple model similar to self-gravitating systems: the hmf model. **The European Physical Journal B - Condensed Matter and Complex Systems**, v. 46, n. 1, p. 61–99, Jul 2005. ISSN 1434-6036. Disponível em: <https://doi.org/10.1140/epjb/e2005-00234-0>.

CHOMAZ, P.; GULMINELLI, F. Phase transitions in finite systems. In: ____. **Dynamics and Thermodynamics of Systems with Long-Range Interactions**. Berlin, Heidelberg:

Springer Berlin Heidelberg, 2002. p. 68–129. ISBN 978-3-540-45835-7. Disponível em: <https://doi.org/10.1007/3-540-45835-2_4>.

DAUXOIS, T.; LATORA, V.; RAPISARDA, A. The Hamiltonian mean field model: from dynamics to statistical mechanics and back. **Dynamics and Thermodynamics of Systems with Long Range Interactions**, p. 458–487, 2002. Disponível em: <http://link.springer.com/chapter/10.1007/3-540-45835-2{_}.>

DAUXOIS, T.; RUFFO, S.; ARIMONDO, E.; WILKENS, M. Dynamics and Thermodynamics of Systems with Long Range Interactions: an Introduction. p. 1–19, 2002. Disponível em: <http://arxiv.org/abs/cond-mat/0208455{\%}0Ahttp://dx.doi.org/10.1007/3-540-45835->.

Eddington, A. S. **The Internal Constitution of the Stars**. [S.l.: s.n.], 1926.

ELLIS, R. S.; TOUCHETTE, H.; TURKINGTON, B. Thermodynamic versus statistical nonequivalence of ensembles for the mean-field Blume-Emery-Griffiths model. **Physica A: Statistical Mechanics and its Applications**, v. 335, n. 3-4, p. 518–538, 2004. ISSN 03784371.

ELSKENS, Y.; ESCANDE, D. **Microscopic Dynamics of Plasmas and Chaos**. Taylor & Francis, 2002. (Series in Plasma Physics and Fluid Dynamics). ISBN 9780750306126. Disponível em: <https://books.google.com.br/books?id=3vjtngEACAAJ>.

FLYNN, M. J. Some computer organization and their effectiveness. **IEEE Transaction on Computers**, C-21, n. 9, p. 948–960, 1972. ISSN 00189340.

FOREST Étienne. Geometric integration for particle accelerators. **Journal of Physics A: Mathematical and General**, v. 39, n. 19, p. 5321, 2006. Disponível em: <http://stacks.iop.org/0305-4470/39/i=19/a=S03>.

GASTER, B.; HOWES, L.; KAELI, D. R.; MISTRY, P.; SCHAA, D. **Heterogeneous computing with OpenCL, 2nd Edition**. [S.l.: s.n.], 2011. ISBN 9780124058941.

GROSS, D. H. E.; KENNEY, J. F. The microcanonical thermodynamics of finite systems: The microscopic origin of condensation and phase separations, and the conditions for heat flow from lower to higher temperatures. **Journal of Chemical Physics**, v. 122, n. 22, p. 1–23, 2005. ISSN 00219606.

HERTEL, P.; THIRRING, W. A soluble model for a system with negative specific heat. **Annals of Physics**, v. 63, n. 2, p. 520–533, 1971. ISSN 1096035X.

KIESSLING, M. K.-H.; LEBOWITZ, J. L. The micro-canonical point vortex ensemble: Beyond equivalence. **Letters in Mathematical Physics**, v. 42, n. 1, p. 43–56, Oct 1997. ISSN 1573-0530. Disponível em: <https://doi.org/10.1023/A:1007370621385>.

KIRKPATRICK, S. **Lecture Notes in Physics** . [S.l.: s.n.], 1981. v. 149. 3 p. ISSN 00280836. ISBN 3540414754.

Lynden-Bell, D.; Wood, R. The gravo-thermal catastrophe in isothermal spheres and the onset of red-giant structure for stellar systems. , v. 138, p. 495, 1968.

MILLER, J. Statistical mechanics of Euler equations in two dimensions. **Physical Review Letters**, v. 65, n. 17, p. 2137–2140, 1990. ISSN 00319007.

NOGUEIRA, E. O uso de integradores numéricos no estudo de encontros próximos. v. 2, p. 21, 09 2009.

OWENS, J. D.; HOUSTON, M.; LUEBKE, D.; GREEN, S.; STONE, J. E.; PHILLIPS, J. C. Gpu computing. **Proceedings of the IEEE**, v. 96, n. 5, p. 879–899, May 2008. ISSN 0018-9219.

OWENS, J. D.; LUEBKE, D.; GOVINDRAJU, N.; HARRIS, M.; KRUGER, J.; LEFOHN, A. E.; PURCELL, T. J. A Survey of General Purpose Computation on Graphics Hardware. **Computer Graphics Forum**, v. 26, n. 1, p. 80–113, 2006. ISSN 1467-8659. Disponível em: <http://www.cs.virginia.edu/papers/ASurveyofGeneralPurposeComputationonGraphicsHardware.pdf>.

PACHECO, P. S. **Introduction to Parallel Programming**. [s.n.], 2011. 391 p. ISSN 01928651. ISBN 9780123814722. Disponível em: <http://aims.me.cycu.edu.tw/courses/101-2/IPMC/lecture{_}material/IntroductiontoParallelProgramming-PeterPacheco(2010).>

QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. [s.n.], 2003. 1–521 p. ISSN 0272-4987. ISBN 0072822562. Disponível em: <https://engineering.purdue.edu/{~}smidkiff/ece563/files/FirstSet563.pdf{\%}5Cnhttp://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:Parallel+Programming+in+C+with+MPI+a>.

RUTH, R. D. A canonical integration technique. **IEEE Trans. Nucl. Sci**, p. 2669–2671, 1983.

SCIENCE, N. P. **Long-range Interactions, Stochasticity and Fractional Dynamics**. [s.n.], 2010. ISBN 978-3-642-12342-9. Disponível em: <http://link.springer.com/10.1007/978-3-642-12343-6>.

TAO, M. Explicit symplectic approximation of nonseparable hamiltonians: Algorithm and long time performance. v. 94, 09 2016.

THIRRING, W. Systems with negative specific heat. **Zeitschrift für Physik A Hadrons and nuclei**, v. 235, n. 4, p. 339–352, Aug 1970. ISSN 0939-7922. Disponível em: <https://doi.org/10.1007/BF01403177>.

TOBERGTE, D. R.; CURTIS, S. **Introduction to Parallel Processing Algorithms and Architectures**. [S.l.: s.n.], 2013. v. 53. 1689–1699 p. ISSN 1098-6596. ISBN 9788578110796.

VOGTMANN, K.; WEINSTEIN, A.; ARNOL'D, V. **Mathematical Methods of Classical Mechanics**. Springer New York, 1997. (Graduate Texts in Mathematics). ISBN 9780387968902. Disponível em: <https://books.google.com.br/books?id=Pd8-s6rOt_cC>.

YOSHIDA, H. Construction of higher order symplectic integrators. **Physics Letters A**, v. 150, n. 5, p. 262 – 268, 1990. ISSN 0375-9601. Disponível em: <http://www.sciencedirect.com/science/article/pii/0375960190900923>.