PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Programa de Pós-Graduação em Engenharia Elétrica

Cláudio Adão Nunes

AVALIAÇÃO DE DESEMPENHO DE POLÍTICAS DE ESCALONAMENTO DE TAREFAS EM APLICAÇÕES OPENMP

Belo Horizonte
2016

Cláudio Adão Nunes

AVALIAÇÃO DE DESEMPENHO DE POLÍTICAS DE ESCALONAMENTO DE TAREFAS EM APLICAÇÕES OPENMP

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Luís Fabrício Wanderley Góes

Belo Horizonte

2016

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Nunes, Cláudio Adão
N972a Avaliação de desempenho de políticas de escalonamento de tarefas em aplicações OpenMP / Cláudio Adão Nunes. Belo Horizonte, 2016.
60 f. : il.

Orientador: Luís Fabrício Wanderley Góes
Dissertação (Mestrado) – Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Engenharia Elétrica

1. Arquitetura de computador. 2. Programação paralela (Computação). 3. Algorítmos computacionais. 4. Sistemas de memória de computadores. 5. Tecnologia da informação. I. Góes, Luís Fabrício Wanderley. II. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

CDU: 681.3-11

Cláudio Adão Nunes

AVALIAÇÃO DE DESEMPENHO DE POLÍTICAS DE ESCALONAMENTO DE TAREFAS EM APLICAÇÕES OPENMP

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica.

Prof. Dr. Luís Fabrício Wanderley Góes (Orientador) - PUC Minas

---

Prof. Dr. Luiz Eduardo da Silva Ramos - Rutgers University (NJ)

Prof. Dr. Carlos Augusto Paiva da Silva Martins - PUC Minas

Prof. Dr. Rose Mary de Souza Batalha (Suplente) - PUC Minas

Belo Horizonte, 27 de Fevereiro 2016

Ao meu melhor amigo, em memória
por me guiar, compartilhar e amar

AGRADECIMENTOS

As pessoas que se dispuseram a ajudar para a concretização deste trabalho, especialmente:

A todo corpo docente e funcionários da PUC Minas pelos bons exemplos ao longo dessa jornada.

Ao apoio financeiro proporcionado pela FAPEMIG e família.

Em mais do que especial, ao meu orientador Prof. Dr. Luís Fabrício pela compreensão, dedicação, diálogos e principalmente seus ensinamentos.

À Dra. Simone Marcucci pelo apoio moral, psicológico e por nos acolher. Além de me ajudar a enxergar como as coisas realmente são e de me guiar à concretização desse trabalho.

A todos meus distantes amigos que mesmo sem poder prestigiá-los conforme merecem, me entenderam e torceram por mim mesmo de longe.

Todos da família Fonseca, Régio e Parreiras pelo suporte, aconchego e paciência. Até mesmo porque sem a ajuda de vocês seria praticamente impossível de se quer recomeçar.

Aos 4 membros da finada família Nunes pela base e instrução, no qual me tornei uma pessoa persistente e lutadora, onde aprendi o significado da palavra perseverança e a superar qualquer que seja o nível de dificuldade imposto pela vida.

A todas as pessoas que contribuíram de uma forma qualquer,

muito obrigado !

I don't wanna feel no more
It's easier to keep falling
Imitations are pale
Emptiness all tomorrows
haunted by your ghost

Fading out by design
Consciously avoiding changes
Curtains drawn, now it's done
Silencing all tomorrows
Forcing a goodbye

Lay down, black gives way to blue
Lay down, I'll remember you

(ALICE IN CHAINS, 2009)

RESUMO

Com o avanço das arquiteturas de processadores nos últimos anos, os processadores com múltiplos núcleos se tornaram comuns em supercomputadores, servidores, computadores de uso doméstico, empresariais e até em dispositivos móveis. Para melhor explorar o desempenho destes processadores, faz-se necessário o uso da programação paralela, que visa executar aplicações em mais de um núcleo de processamento simultaneamente. Porém, ao utilizar esse novo paradigma de programação, existem vários fatores que influenciam no desempenho de aplicações paralelas como, sincronização, concorrência, criação de *threads*, escalonamento de tarefas, entre outros. Especificamente, o escalonamento de tarefas é importante, pois, ele deve ser adaptado ao comportamento de cada aplicação para lidar com o balanceamento de carga, afinidade de memória, etc. Neste trabalho, é apresentada uma avaliação de desempenho de políticas de escalonamento de tarefas em aplicações paralelas em OpenMP (*Open Multi-Processing*). Nesta avaliação foram utilizadas aplicações contidas no CAP-Bench, um benchmark de aplicações paralelas em OpenMP. Através desta abordagem, os resultados da avaliação apresentaram ganhos de desempenho de até 85,3% melhor em relação a versão sequencial, alterando-se o tamanho das tarefas e as políticas de escalonamento.

Palavras-Chave: Avaliação de desempenho. Políticas de Escalonamento de Tarefas. OpenMP. Processadores com Múltiplos Núcleos.

ABSTRACT

Recent improvements in computer architectures let multicores became commonly available in supercomputers, servers, personal computers, and business to mobile. In order to fully exploit its performance, parallel programming is needed to run applications in several cores simultaneously. However when using this new programming paradigm there are many factors that can influence in parallel applications performance such as synchronization, concurrency, creation of threads, task scheduling and others. In particular, task scheduling is important because it must be adapted to the behavior of application to handle load balancing, memory affinity, etc. This work presents an analysis of task scheduling policies on parallel applications. In this analysis, OpenMP (Open Multi-Processing) parallel applications available in CAP-Bench (benchmarks suite) was used. Through this approach the evaluation results showed 85% of speedups better compared to sequential version by changing the tasks size and task scheduling polices.

Keywords**:** Performance Evaluation. Task Scheduling Policies. OpenMP. Multi-Processors.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ARB	Architecture Review Board
CAP-Bench	Cart nAnosim gPpd Benchmark suite
CMP	Chip Multiprocessor
CUDA	Compute Unified Device Architecture
DLP	Data Level Parallelism
FAST	Features from Accelerated Segment Test
FN	Friendly Numbers
GF	Gaussian Filter
GPU	Graphic Processing Unit
ILP	Instruction Level Parallelism
IS	Integer Sort
KM	K-means
LU	Lower to Upper
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MPI	Message Passing Interface
OpenACC	Open Accelerators
OpenMP	Open Multi-Processing
OpenCL	Open Computing Language
RAM	Random Access Memory
RT	Ray Trace
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMT	Simultaneous Multithreading
TBB	Intel Threading Building Blocks
TLP	Thread level Parallelism

SUMÁRIO

1 INTRODUÇÃO

Historicamente, os processadores sequenciais evoluíram em termos de ganho de desempenho, com o aumento da frequência de *clock*, que ocasionava em alto consumo de energia (AJI, 2015; CHAPMAN; JOST; PAS 2007; DIENER, 2015; DE ROSE; NAVAUX, 2008; HENNESSY; PATTERSON, 2014; RAUBER; RÜNGER, 2013).

Cientistas e engenheiros da computação conseguiram ainda otimizar esses processadores por meio do paralelismo no nível de instruções nos processadores escalares e superescalares (arquiteturas *pipeline*), onde várias instruções podem ser executadas simultaneamente, mas mesmo assim, a exploração deste tipo de paralelismo atingiu o seu limite (CHAPMAN; JOST; PAS 2007; HENNESSY; PATTERSON, 2014).

A abordagem contemporânea visa trabalhar com uma frequência mais baixa do processador e uso de mais de um núcleo de processamento no mesmo *chip*, criando assim, uma arquitetura paralela composta de múltiplos núcleos, para a execução de programas computacionais, também chamados de processadores com múltiplos núcleos (*multicore*) (HENNESSY; PATTERSON, 2014).

Esta abordagem, porém, encadeou uma série de problemas para os desenvolvedores. Estes problemas estão relacionados com a abordagem clássica de desenvolvimento de programas da forma estrutural e sequencial, ou seja, os programas não estavam paralelizados e nem os compiladores seriam capazes de lidar com o paralelismo de forma automática, além do mais, nem mesmo a maioria dos programadores sabiam desenvolver aplicações paralelas.

Diante deste cenário, foi necessário adaptar ferramentas que eram utilizadas apenas em supercomputação, do qual o paralelismo de *threads* já era usado em arquiteturas multiprocessadas para serem executados em aplicações no cotidiano. Sendo assim, a partir dessas mudanças, o problema de paralelizar passa a ser do programador. Diante deste cenário, foi necessário trazer ferramentas que eram utilizadas em supercomputação (como centrais de processamentos, servidores, etc), onde já era usado o paralelismo com múltiplas *threads* , passa ser agora utilizado em aplicações do cotidiano. Sendo assim, a partir dessas mudanças, o problema de paralelizar passa a ser do programador.

A programação paralela introduz vários problemas não existentes na programação sequencial, como, localidade de dados, condição de disputa, sobrecargas de criação de *threads*, balanceamento de carga entre outros (CHANDRA, 2001; CHAPMAN; JOST; PAS 2007; DETONI, 2010; WANG, 2016; MUDDUKRISHNA, 2016). Todos esses problemas influenciam no desempenho da aplicação. Um problema fundamental desse paradigma é o escalonamento de tarefas. Referenciado também como balanceamento de carga, ele é responsável por alocar tarefas aos elementos de processamento de uma arquitetura paralela (SILVA, 2015).

A biblioteca de programação paralela mais utilizada nos processadores com múltiplos núcleos é o OpenMP (DIAZ; MUÑOZ-CARO; NIÑO, 2012; SIVANANDAN; KUMAR; MEHER, 2015). Ele possibilita realizar o gerenciamento, criação e sincronismo de *threads*, permite também, selecionar e configurar o escalonamento de tarefas. Portanto, o **objetivo** deste trabalho é realizar uma avaliação de desempenho de políticas de escalonamento de tarefas em aplicações OpenMP.

Este texto está estruturado em 5 capítulos. O capítulo 2 apresenta a fundamentação teórica e conceitos relacionados com o trabalho, sendo dividida em computação paralela, modelos de arquiteturas paralelas, modelos de programação paralela, desafios na programação paralela, escalonamento de tarefas, OpenMP e trabalhos relacionados. O capítulo 3 descreve a natureza da pesquisa e suas respectivas etapas, bem como o ambiente experimental. O capítulo 4 apresenta os resultados experimentais das políticas de escalonamento nas aplicações OpenMP do CAP-Bench. Por último, o capítulo 5 conclui o trabalho e apresenta os trabalhos futuros.

2 REFERENCIAL TEÓRICO

Neste capítulo são apresentados os conceitos, terminologias básicas, trabalhos relacionados e técnicas utilizadas nesse trabalho.

2.1 Arquiteturas Paralelas

As arquiteturas paralelas possibilitam que programas sejam executados em paralelo. Recentemente, elas têm recebido maior destaque devido aos limites físicos no aumento de frequência dos processadores. Nas décadas de 50 e 60, a tecnologia presente nos processadores disponíveis, possuía capacidade suficiente, no qual, atendia os requisitos de processamento das aplicações naquela época (PATTERSON; HENNESSY, 2014; RAUBER; RÜNGER, 2013; RAYNEI, 2010;).

Porém, com o avanço tecnológico, processadores fabricados e embasados na arquitetura sequencial de Von Neumman ou arquiteturas monoprocessadas, deixaram de suprir a demanda por processamento, além de não proporcionar desempenho suficiente ou satisfatório sobre as novas aplicações emergentes (DE ROSE; NAVAUX, 2008; STALLINGS, 2010).

Por outro lado, o paralelismo de instruções presente nos processadores escalares e superescalares permitem executar várias instruções por ciclo de *clock* sobre diversas etapas de modo sobreposto. Mas, o principal agravante é que estas arquiteturas exigem altas frequências de *clock* e consomem muita energia, ao contrário de processadores paralelos compostos de núcleos simplificados e frequências mais baixas (DE ROSE; NAVAUX, 2008; FARKAS et al., 2003; HENNESSY; PATTERSON, 2008).

Tais processadores com múltiplos núcleos, ou processadores *multicore*, estão presentes nos mais diferentes computadores fornecidos pela indústria, desde computadores pessoais e portáteis no uso cotidiano até grandes servidores empresariais no processamento massivo de dados. Estas arquiteturas paralelas permitem o uso de técnicas de paralelismo ao executar aplicações (CHANDRA et al., 2001; SENA; COSTA, 2008).

De modo geral, para explorar o paralelismo nas diferentes arquiteturas paralelas, é necessário contextualizar os três diferentes níveis sendo, instrução, tarefas e dados:

a) Paralelismo de instruções ou *Instruction Level Parallelism* (ILP), trata-se do paralelismo no nível de *hardware*, onde instruções independentes são executadas em paralelo diretamente por ele. Porém, nos outros dois níveis, é necessária a intervenção do programador ou compilador para que o paralelismo possa ser explorado através de mecanismos implementados em *software,* as APIs (*Application Programming Interface*) ou bibliotecas de programação (CHAPMAN; JOST; PAS, 2007; STALLINGS, 2010).

b) O paralelismo no nível de tarefa ou *Thread level Parallelism* (TLP) é explorado por máquinas paralelas, tais como, multiprocessadores, as máquinas SMT (*Simultaneous Multithreading*) e máquinas CMP (*Chip Multiprocessor*). Essas arquiteturas são capazes de executar várias *threads* (processos) ao mesmo tempo de forma independente. Para que esse modelo possa ser explorado, a intervenção do programador é necessária através da programação paralela e suas respectivas técnicas de desenvolvimento (HENNESSY; PATTERSON, 2014).

c) Outra maneira de explorar o paralelismo em software é conhecida como paralelismo no nível de dados ou *Data Level Parallelism* (DLP), Essa técnica, aplica a mesma operação em diferentes dados simultaneamente (HENNESSY; PATTERSON, 2014).

2.2 Modelos de Arquiteturas Paralelas

Cada tipo de exploração de paralelismo é mais adequado à um modelo de arquitetura paralela. De acordo com a taxonomia de Flynn (1966), um modelo de arquitetura de um computador executa uma série de instruções sobre uma série de dados, ou seja, fluxo de instruções e fluxo de dados. Assim, obtemos quatro divisões distintas (HENNESSY; PATTERSON, 2014; STALLINGS, 2010):

a) SISD (*Single Instruction stream, Single Data stream*): apenas um fluxo de instruções age sobre um único fluxo de dados. É nessa classificação que se enquadram as máquinas de Von Neumann convencionais, porém, é nessa classificação que se explora o paralelismo em nível de instrução, através de arquiteturas contendo *pipeline*.

b) MISD (*Multiple Instruction stream, Single Data stream*): múltiplos fluxos de instruções agem sobre um único fluxo de dados. Essa classe é ainda tecnicamente impraticável, não existindo nenhuma máquina que implemente esse modelo.

c) SIMD (*Single Instruction stream, Multiple Data stream*): o único fluxo de instrução é executado sobre vários dados ao mesmo tempo. São características da classe SIMD, uma única unidade de controle, possuir vários elementos de processamento envolvidos entre si e que executam suas instruções em paralelo sobre vários fluxos de dados como, processadores vetoriais (por exemplo, Cray Y-MP 816 e Fujitsu VPP 500), GPUs (Gforce GTX980, Radeon R9 380x, etc) entre outros (AMD, 2016; DE ROSE; NAVAUX, 2008; NVIDIA, 2016).

d) MIMD (*Multiple Instruction stream, Multiple Data stream*): ao contrário da classificação SIMD, esta categoria permite executar vários fluxos de instruções sobre as unidades de controle e elementos de processamento em vários fluxos de dados, como por exemplo, os multiprocessadores. Esta classificação permite explorar o paralelismo de tarefas. É importante também ressaltar que máquinas MIMD também podem explorar paralelismo de dados, porém, enfrenta mais problemas com sobrecarga de comunicação e sincronização entre as *threads*, do que em máquinas SIMD. Para explorar de forma mais eficiente o paralelismo de dados em arquiteturas MIMD, a aplicação deve conter características de granularidade grossa (termo abordado na seção 2.4.4 Sobrecarga) para evitar esta sobrecarga (HENNESSY; PATTERSON, 2014).

Atualmente, processadores com múltiplos núcleos são em sua grande maioria híbridos, possuindo os mecanismos dos três modelos SISD, SIMD e MIMD. Entretanto, as máquinas paralelas ainda concentradas nas classes SIMD e MIMD, provêm de tipos distintos de acesso à memória, que pode ser compartilhada ou não compartilhada (HENNESSY; PATTERSON, 2014). As máquinas MIMD ainda podem ser divididas em multicomputadores onde a comunicação é feita através de passagem de mensagens, tais como, *grids* e *clusters*, onde cada máquina possui sua própria memória privada (memória local para cada processador). Já os multiprocessadores são máquinas com memória compartilhada e trabalham com variáveis compartilhadas (HENNESSY; PATTERSON, 2008).

Existem máquinas de memória compartilhada e distribuída. Uma máquina de memória compartilhada é caracterizada pelos acessos provenientes de vários processadores aos endereços desta memória. Ou seja, a memória compartilhada é acessada por vários processadores. Em uma máquina de memória distribuída, para cada processador é distribuído fisicamente uma memória local, no qual seu acesso é realizado apenas pelo seu respectivo processador e onde a sincronização requer comunicação entre os mesmos através de troca de mensagens (STALLINGS, 2010).

2.3 Modelos de Programação Paralela

Ao contrário da programação sequencial, a programação paralela é mais complexa, devido à preocupação com a dependência de dados além da sincronização de tarefas, uma vez que vários elementos de processamento são responsáveis pela cooperação e execução dessas tarefas (DE ROSE; NAVAUX, 2008).

Um dos modelos de programação paralela existentes é chamado de passagem de mensagens. Uma das bibliotecas existentes que implementam este modelo é o MPI (*Message Passing Interface*), no qual um processo envia uma mensagem e o processo receptor recebe a mensagem do processo emissor, ou seja, a base da MPI é a comunicação entre processos (STALLINGS, 2010).

Outro modelo de programação paralela é por meio de variáveis compartilhadas em máquinas paralelas de memória compartilhada. O endereçamento nesse modelo, é realizado em um único espaço existente, através

de operações de leitura e escrita (*load* e *store*), como implementado nas bibliotecas *Open Multi-Processing* ou OpenMP e *Intel Threading Building Blocks* (TBB) (DE ROSE; NAVAUX, 2008; DIAZ; MUÑOS-CARO; NIÑO, 2012).

Ainda no escopo de máquinas MIMD, é possível através de técnicas de paralelismo, realizar uma implementação híbrida utilizando o modelo de passagem de mensagens com o MPI e OpenMP usando memória compartilhada em cada nodo de processamento em *clusters* SMP (*Symmetric Multi-Processors*), ambos sendo executados simultaneamente (JOST et al., 2003).

Com o avanço das tecnologias nos modelos MIMD e SIMD (como os processadores gráficos - GPUs (*Graphic Processing Unit*)), a literatura vem sendo atualizada e dispõe de vários trabalhos com combinações entre arquiteturas paralelas e modelos de programação paralela, por exemplo, em supercomputadores construídos por diversos processadores e co-processadores, processando dados através de diversas bibliotecas como OpenCL, MPI, OpenMP, CUDA, OpenACC, etc (AJI, 2015; COUDER-CASTAÑEDA et al., 2015; KHRONOS, 2016; SIVANANDAN; KUMAR; MEHER, 2015).

2.4 Desafios na Programação Paralela

A programação paralela permite que aplicações sejam modificadas com o principal intuito de otimizar o desempenho das mesmas, além de permitir extrair o paralelismo em arquiteturas paralelas, ou seja, utilizar melhor os recursos de hardware implementados nos processadores com múltiplos núcleos.

Ao contrário da programação sequencial, o paradigma de programação paralela apresenta uma gama de complexidades ao paralelizar uma aplicação sobre as arquiteturas atuais. Isto ocorre pelo fato de que, o modelo clássico de desenvolver programas não há preocupações em gerenciar *threads* ou a forma com que elas irão se comportar durante a execução da aplicação (CHANDRA et al., 2001).

Os principais desafios quando se trata em paralelizar aplicações são: condição de disputa, sincronização, concorrência, exclusão mútua, sobrecargas, escalonamento de tarefas, entre outros (TREW; WILSON, 2012). Esses problemas serão mais detalhados nas próximas subseções.

2.4.1 Condição de disputa

Em um ambiente onde várias tarefas são executadas simultaneamente e de forma concorrente pode ocorrer um problema chamado de condição de disputa, que, trata-se de um evento onde várias tarefas concorrentes tentam acessar a mesma posição de memória, ou seja, se um ou mais processos realizarem leituras em uma determinada posição na memória, enquanto outro processo precisa executar uma escrita na mesma posição de memória. Dessa forma, as operações anteriores ao processo de escrita podem coletar valores desatualizados.

Na condição de disputa não se pode garantir a ordem de execução, pois o controle referente à ordenação é feito pelo próprio *hardware* ou ainda pelo sistema operacional. Se não tratada essa condição de disputa pelo recurso, ela pode gerar inconsistência de dados (SENA; COSTA, 2008).

Sendo assim, para tratar o problema de disputa entre os processos, se faz necessário o uso de sincronização em relação ao acesso à memória. Isto é possível através da exclusão mútua.

2.4.2 Exclusão Mútua

A exclusão mútua é um mecanismo utilizado para evitar que várias tarefas acessem um mesmo dado em paralelo. Várias linguagens de programação permitem implementar este mecanismos através de técnicas de programação como, variáveis *mutex*, *locks*, semáforos, etc.

Essas técnicas fazem com que apenas uma tarefa de cada vez atualize um determinado dado na memória, antes, disputado de forma simultânea e trazendo a inconsistência. O maior problema em utilizar tais técnicas é devido a perda de desempenho que pode ocorrer, pois, essas técnicas fazem com que naquele trecho da aplicação, sua execução se torne serializada (CHANDRA et al., 2001).

Como a paralelização de uma aplicação sequencial é de responsabilidade do programador, cabe a ele promover tais mecanismos ao identificar esse problema sem comprometer o desempenho da aplicação. Por este motivo deve-se evitar o uso frequente de exclusão mútua (DETONI, 2010).

2.4.3 Sincronização

O problema de sincronização de tarefas acontece através de outros desafios presentes na programação paralela, como condições de disputa e exclusão mútua entre outros. O sincronismo entre as tarefas são as trocas de seus estados e informações de cada uma, através da comunicação entre elas em relação aos seus resultados obtidos ao acessar um recurso compartilhado.

Além da sincronização consumir processamento devido a comunicação das tarefas, ela pode fazer com que processadores fiquem ociosos esperando o término da execução das tarefas, enquanto apenas um, ou outros processadores, ainda permanecem em execução (CHANDRA et al., 2001).

2.4.4 Sobrecarga

A sobrecarga ou *overhead* acontece devido ao tempo que se gasta no processo de criação das tarefas. Este processo pode ser descrito pelo momento de identificação, busca de um processador para execução, carregamento da tarefa e de seus dados para a execução no processador e inicialização da tarefa (CHANDRA et al., 2001).

Após realizar todo esse processo de inicialização da tarefa, a sobrecarga ainda está presente no tempo para finalizar uma tarefa (DIAZ; MUÑOS-CARO; NIÑO, 2012).

Em relação a exploração do paralelismo em modelos de arquiteturas paralelas, as sobrecargas também podem ocorrer com mais frequência ao explorar paralelismo de dados sobre máquinas MIMD do que em máquinas SIMD, uma vez que esse modelo de arquitetura é adaptada melhor a operações de granularidade grossa (HENNESSY; PATTERSON, 2014). A granularidade (tamanho do grão) é uma definição dada para a quantidade de computação atribuída à tarefa (tamanho da tarefa) em relação à quantidade de núcleos em processadores de múltiplos núcleos (CHANDRA et al., 2001).

Entretanto, o tamanho do grão pode ser fino ou grosso. Granularidade fina é quando serão executadas tarefas pequenas (exemplo, 1 e 32), neste caso cada

thread executa tarefas pequenas. A granularidade grossa ocorre quando o tamanho das tarefas são maiores e as *threads* processam uma quantidade maior de trabalho por tarefa.

Dessa forma, a granularidade causa impacto no desempenho da aplicação devido ao compromisso entre sincronização e computação de tarefas (SILVA, 2011).

2.5 Escalonamento de Tarefas

O escalonamento de tarefas ou balanceamento de carga são dois termos muito relacionados trazendo a ideia de como distribuir tarefas entre as várias *threads* para o processamento em paralelo. Na computação paralela, esta é uma área amplamente estudada, pois, as tarefas das aplicações podem ser balanceadas ou desbalanceadas, ou seja, o tamanho da tarefa pode variar. Essas propriedades podem ocasionar a perda de desempenho e eficiência em relação à forma com que o escalonamento é determinado para executar as tarefas (CHAPMAN, 2014; QAWASMEH; MALIK; SILVA, 2015).

No intuito de evitar o desbalanceamento, uma estratégia a ser utilizada é realizar o escalonamento de tarefas de forma que os processadores não fiquem ociosos, ou ao menos, reduzir a ociosidade dos processadores.

Porém, ao atribuir as tarefas para serem executadas de forma eficiente é necessário considerar seus atributos. Tais atributos estão relacionados ao custo de cada tarefa. A dependência entre tarefas também deve ser avaliada, no caso, se a ordem importa, ou seja, se uma tarefa pode ser executada antes de outra determinada tarefa sem interferir no resultado final do processamento (SILVA, 2015).

Uma importante característica a ser observada é em relação à afinidade de memória. A afinidade de memória está ligada à localidade onde tarefas estão sendo alocadas para processamento, pois, quanto mais próximo do elemento de processamento, ou núcleo, menor a latência de acessos à memória principal e consumo de energia entre outros. (CHANDRA et al., 2001; DIENER, 2015; GOES et al., 2014; NIKOLOPOULOS et al. 2001).

Existem dois tipos de escalonamento: estático e dinâmico. O escalonamento estático distribui tarefas por tamanhos iguais definidos o que proporciona pouca sobrecarga. Por outro lado, devido aos diferentes tamanhos das tarefas a serem

executadas, isso pode gerar desbalanceamento de carga no que acarreta em perda de desempenho, pois, pode haver processadores ociosos (CHANDRA et al., 2001). Neste tipo de escalonamento, as tarefas são alocadas às *threads* em tempo de compilação, ou seja, antes do tempo de execução da aplicação (GOES, 2004).

Ao contrário do escalonamento estático, o escalonamento dinâmico atribui tarefas de acordo com a demanda de processamento. Só é atribuída uma nova tarefa à uma *thread* quando a tarefa atual for finalizada. Neste caso, as decisões são tomadas em tempo de execução pelo escalonador (GOES, 2004).

Este tipo de escalonamento possui a característica de minimizar o desbalanceamento de carga, mas pode gerar muita sobrecarga para verificar onde uma tarefa inicia e termina, procurando processadores ociosos pra execução da tarefa, etc. (CHANDRA et al., 2001; CHAPMAN; JOST; PAS, 2007).

2.6 Open Multi Processing - OpenMP

O OpenMP – *Open Multi Processing* é uma API desenvolvida pela corporação OpenMP ARB ou ARB, que é responsável por manter o OpenMP além de fomentar eventos, workshops, conferências, etc. A corporação é mantida por várias empresas membros, tais como, AMD, HP, Cray, Intel, Red Hat entre outras (OPENMP, 2016).

Esta API trata-se de um conjunto de implementações fazendo interface entre o compilador e as linguagem de programação C, C++, FORTRAN, para paralelizar aplicações de alto nível em máquinas de memória compartilhada. Ele começou suas atividades no final dos anos 90. Em 1997, o OpenMP foi oficialmente lançado para FORTRAN na sua versão 1.0, mas foi apenas em 1998 através da versão 2.0 que os compiladores para linguagem C aderiram a API. A versão 3.0 foi lançada dez anos depois com novas implementações, porém, hoje já se encontra em desenvolvimento a versão 4.5 (OPENMP, 2016; SILVA, 2011).

2.6.1 Escalonamento com OpenMP

A programação com o OpenMP é composta de diretivas de compilação, funções de interface, variáveis de ambiente e cláusulas. As diretivas são descritas pelas sintaxe *#pragma* (nas linguagens C/C++), isso significa que, onde houver essa

diretiva, o compilador compatível com o OpenMP executa uma funcionalidade da API. O construtor *parallel*, cria um grupo de *threads* sendo possível compartilhar tarefas dentro deste mesmo grupo ou as separando em regiões distintas de execução (CHAPMAN; JOST; PAS, 2007; GONÇALVES et al, 2016; SENA; COSTA, 2008).

Para trabalhar em regiões paralelas é necessário utilizar algum dos tipos de construtores disponíveis para tarefas compartilhadas entre *threads* ou para dividir as tarefas em diferentes *threads*. No primeiro caso, o construtor *for* faz com que as *threads* do mesmo grupo executem iterações diferentes de um mesmo laço de repetição. Este é um dos principais construtores em termos de implementação para o ganho de desempenho nesse modelo de programação, isso porque implementa o modelo de arquiteturas de máquinas SIMD. Para o paralelismo de tarefas usa-se o construtor *sections*. Este que por sua vez implementa o modelo MIMD, ou seja, diferentes *threads* executam tarefas distintas (SENA; COSTA, 2008).

As funções de interface possibilitam recuperar diversas informações dentro de uma aplicação paralela. Elas estão divididas em três principais grupos: controle de ambiente de execução, sincronização e tempo de execução.

Neste trabalho, as aplicações contidas no *benchmark* utilizado apresentam as principais funções de interface, *omp_set_num_threads()* que configura a quantidade de *threads* em uma região paralela, e *omp_get_thread_num()*, que recupera a quantidade de *threads* na região paralela. Ambas funções pertencem ao grupo de controle de ambiente de execução (CHAPMAN; JOST; PAS, 2007; SENA; COSTA, 2008).

Variáveis de ambiente são mecanismos controladores das variáveis internas das aplicações, podendo ser acessadas em tempo de execução. Essas variáveis de ambiente permitem modificar diferentes políticas de escalonamento e seus respectivos tamanhos de tarefas, definem também, a quantidade de *threads* para processamento compartilhado e/ou aninhado (GONÇALVES et al, 2016; SENA; COSTA, 2008).

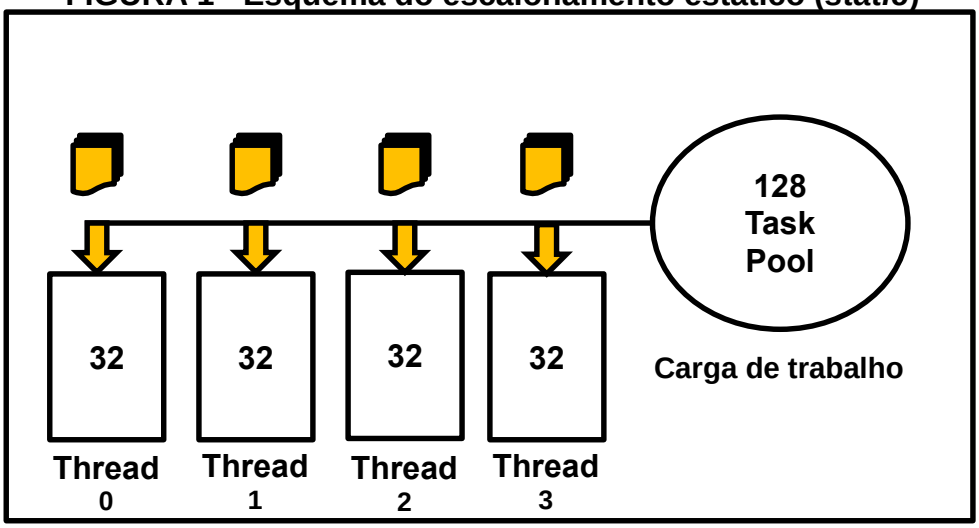
As cláusulas no OpenMP servem para definir o comportamento dos construtores em relação aos diferentes tipos de variáveis dentro de regiões paralelas. Existem diversas cláusulas que podem ser usadas ao paralelizar uma aplicação, logo, como este trabalho trata em específico das políticas de

escalonamento, a API aborda tanto o escalonamento estático quanto o dinâmico, porém, existem três principais políticas de escalonamento: *static, dynamic* e *guided*.

Para o escalonamento estático usa-se a cláusula *schedule*(*static*, *chunk_size*), sendo que, o parâmetro *chunk_size* estipula o tamanho da tarefa a ser executada por cada *thread*. Nesta política de escalonamento, cada thread recebe uma quantidade de tarefa já predefinida para execução de forma estática. Caso não seja estipulado o tamanho de tarefas no parâmetro *chunk_size*, o próprio OpenMP se encarrega de verificar e calcular a quantidade de *threads*, tamanho de tarefas e iterações (CHAPMAN; JOST; PAS, 2007; SENA; COSTA, 2008).

Desta forma, o tamanho da tarefa recebe aproximadamente o quociente entre o número de iterações dividido pelo número de *threads*. Por exemplo, em um processador com quatro *threads*, onde o número do laço de repetição for 128, o tamanho das tarefas para execução seriam de aproximadamente no tamanho de 32. A FIGURA 1 apresenta a ideia de funcionamento referente ao escalonamento estático:

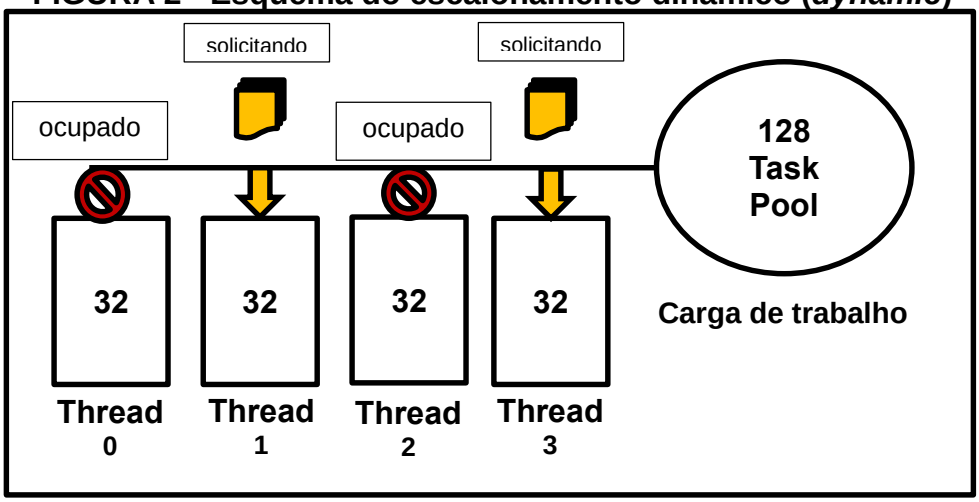
FIGURA 1 - Esquema do escalonamento estático (*static*)



Fonte: Elaborado pelo autor

Para as políticas de escalonamento dinâmico, estas possuem dois tipos, *dynamic* e *guided*. Ao selecionar modo *dynamic* de execução, as execuções das tarefas ocorrem de acordo com as solicitações pelas *threads* na medida que as mesmas vão finalizando as tarefas. Para esta opção de escalonamento, a FIGURA 2 ilustra seu funcionamento.

FIGURA 2 - Esquema do escalonamento dinâmico (*dynamic*)

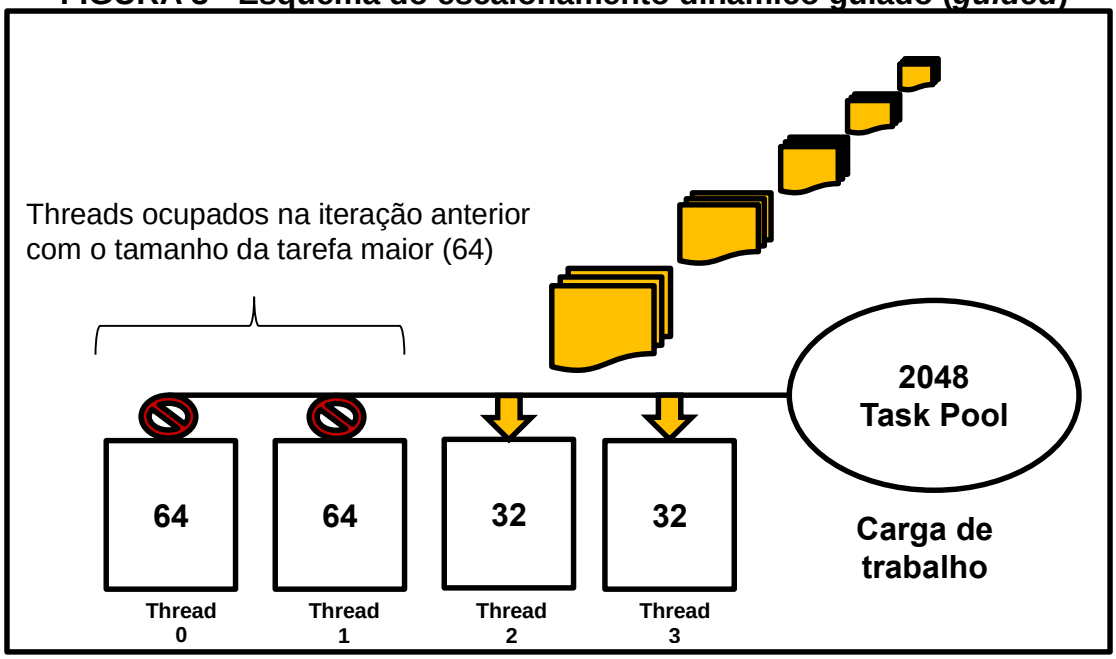


Fonte: Elaborado pelo autor

Essas solicitações entre as diferentes *threads* vão sendo requisitadas até que não haja mais tarefas para serem executadas.

O funcionamento da opção *guided* de escalonamento é semelhante ao *dynamic*, porém, o tamanho da tarefa vai diminuindo a cada iteração (FIGURA 3) do laço de repetição até chegar a opção estipulada no *chunk_size*, caso contrário o tamanho de tarefas das últimas iterações é assumido pelo valor 1.

FIGURA 3 - Esquema do escalonamento dinâmico guiado (*guided*)



Fonte: Elaborado pelo autor

É importante salientar que qualquer opção utilizando a cláusula *schedule* no OpenMP só funciona seguida do construtor *for*, ou seja, só é possível configurar políticas de escalonamento com construtores de trabalho de tarefas compartilhado (CHAPMAN; JOST; PAS, 2007).

2.6.2 Benchmark CAP-Bench

CAP-Bench (GITHUB, 2015) é um *benchmark* que possui vários *kernels* (aplicações paralelas) de diferentes comportamentos em termos de comunicação entre tarefas, sincronização, balanceamento de carga, e além disso, dá suporte ao OpenMP. Estes *kernels* utilizam padrões de programação paralela como *map*, *mapreduce*, *stencil*, frequentemente presente em aplicações paralelas. Seu principal objetivo é permitir a execução de seus *kernels* explorando o paralelismo em arquiteturas paralelas, sobre processadores com conjunto de instruções x86 e o processador MPPA (*University of* Grenoble) (GITHUB, 2015).

Deste benchmark foram utilizados 7 de seus *kernels* para realização dos testes:

a) *Features from Accelerated Segment Test* (FAST): algoritmo com o objetivo de detectar quinas no processamento de imagens.
b) *Friendly Numbers* (FN): algoritmo que calcula a relação entre números abundantes (que possuem a soma dos divisores anteriores próprios) gerando o mesmo resultado em comum.
c) *Gaussian Filter* (GF): aplica uma máscara bidimensional Gausiana sobre uma imagem para redução de ruído de imagem.
d) *Integer Sort* (IS): realiza a ordenação de números inteiros através de operações paralelas.
e) *K-means* (KM): é uma solução aplicada para avaliação de clusters calculando distâncias entre os centróides.
f) *Lower-Upper* (LU): usa esquema numérico para resolver matrizes triangulares superiores e inferiores com o objetivo de testar a comunicação bloqueante.
g) *Ray Trace* (RT): algoritmo para processamento de reflexo de luz, sombras e renderização de cenas ultra-realísticas em imagens 3D.

Em relação ao balanceamento de carga, algumas aplicações presentes no CAP-Bench são desbalanceadas, pois as tarefas possuem diferentes distribuições de trabalho tais como, IS, KM e FAST. Outras aplicações apresentam características de cargas balanceadas, como por exemplo, FN, GF e LU (GITHUB, 2015).

2.7 Trabalhos Relacionados

Nesta seção, os trabalhos relacionados são apresentados por um breve resumo de modo comparativo com a pesquisa apresentada nesta dissertação.

Em Qawasmeh; Malik; Chapman, 2014 foi proposta uma avaliação de escalonamento de tarefas, verificando o melhor tempo de execução das aplicações paralelas em OpenMP. Devido aos desafios da programação paralela tais como, balanceamento de carga, afinidade de memória, localidade de dados e altos custos em sobrecargas, foram propostas ferramentas de configurações e controle das filas de tarefas para execução da carga de trabalho implementada em OpenUH (compilador de código fonte aberto). Através do benchmark utilizado (Barcelona OpenMP Test Suite) e API de desempenho de tarefas (ORA) foi possível verificar pelos resultados obtidos que, o maior impacto de desempenho é dado pela organização nas filas de tarefas para execução em relação a ordem com que as tarefas são escalonadas.

Em Severo; Serpa; Schepke, 2013 foi realizada a avaliação de desempenho de diferentes políticas de escalonamento em uma aplicação desenvolvida para cálculos de dinâmica de fluídos computacional. A aplicação utilizada foi o Método de Lattice Bolstmann e ao ser paralelizado através do OpenMP, foi possível alterar as políticas de escalonamento e seus respectivos tamanhos das tarefas, verificando assim o comportamento em termos de desempenho da aplicação abordada. Os resultados apresentam que a melhor configuração entre as políticas de escalonamento utilizada foi o escalonamento guiado (*guided*) com *speedup* de 15x em relação à execução sequencial da aplicação.

Em Olivier et al., 2012, a maior preocupação é voltada para o escalonamento de tarefas em relação a localidade de memória. Neste trabalho correlato são apresentadas estratégias de escalonamento através da implementação de um escalonador que utiliza técnicas de roubo de tarefas, roubo de carga de trabalho por

núcleo de processamento, identificação de localidade de memória, entre outras técnicas embasadas na biblioteca *Qthreads*. A comparação realizada dessa abordagem foi comparada à configuração padrão de escalonamento do OpenMP em tempo de execução, sendo que, dos 7 benchmarks testados, 5 apresentaram resultados superiores ao padrão OpenMP.

Em Silva, 2011 é abordado o escalonamento de tarefas de acordo com o controle de granularidade de tarefas em OpenMP. O principal objetivo desse trabalho foi analisar o comportamento de aplicações paralelas como o cálculo da série de Fibonacci, Quicksort e NQueens, em termos do ganho de desempenho ao alterar a granularidade de tarefa das aplicações propostas. Os resultados mostram que esse controle de granularidade de tarefas também impacta no desempenho das aplicações e uma configuração errada pode causar queda no desempenho devido à sobrecarga de *threads* ou tarefas, de forma que, aplicações seriais obtivessem melhores resultados do que em versões paralelas.

Através de técnicas de aprendizado de máquina, (Wang; O'Boyle, 2009) propõe um compilador automático para mapear o paralelismo através do aprendizado de máquina no qual o compilador prevê a melhor configuração de número de *threads* e melhor política de escalonamento para a execução das aplicações. As técnicas de inteligência artificial têm sido constantemente usadas na programação paralela para predizer melhores escolhas e estratégias de escalonamento de tarefas. Em relação aos testes, foram utilizados 3 benchmarks (NAS, UTDPS e Mibench) e a abordagem proposta pelos autores atingem resultados melhores (17,5 vezes mais rápido) do que os modelos disponíveis de implementação automática para escalonamento com OpenMP.

É possível notar que a preocupação em extrair melhor o desempenho em arquiteturas utilizando processadores com múltiplos núcleos é um tema ainda muito explorado e principalmente no contexto de diferentes opções de escalonamento de tarefas. Esse trabalho se diferencia das demais pesquisas relacionadas por se tratar de uma avaliação entre várias configurações de escalonamento dinâmico e estático, utilizando aplicações paralelas com diferentes características contidas no benchmark CAP-Bench.

3 METODOLOGIA

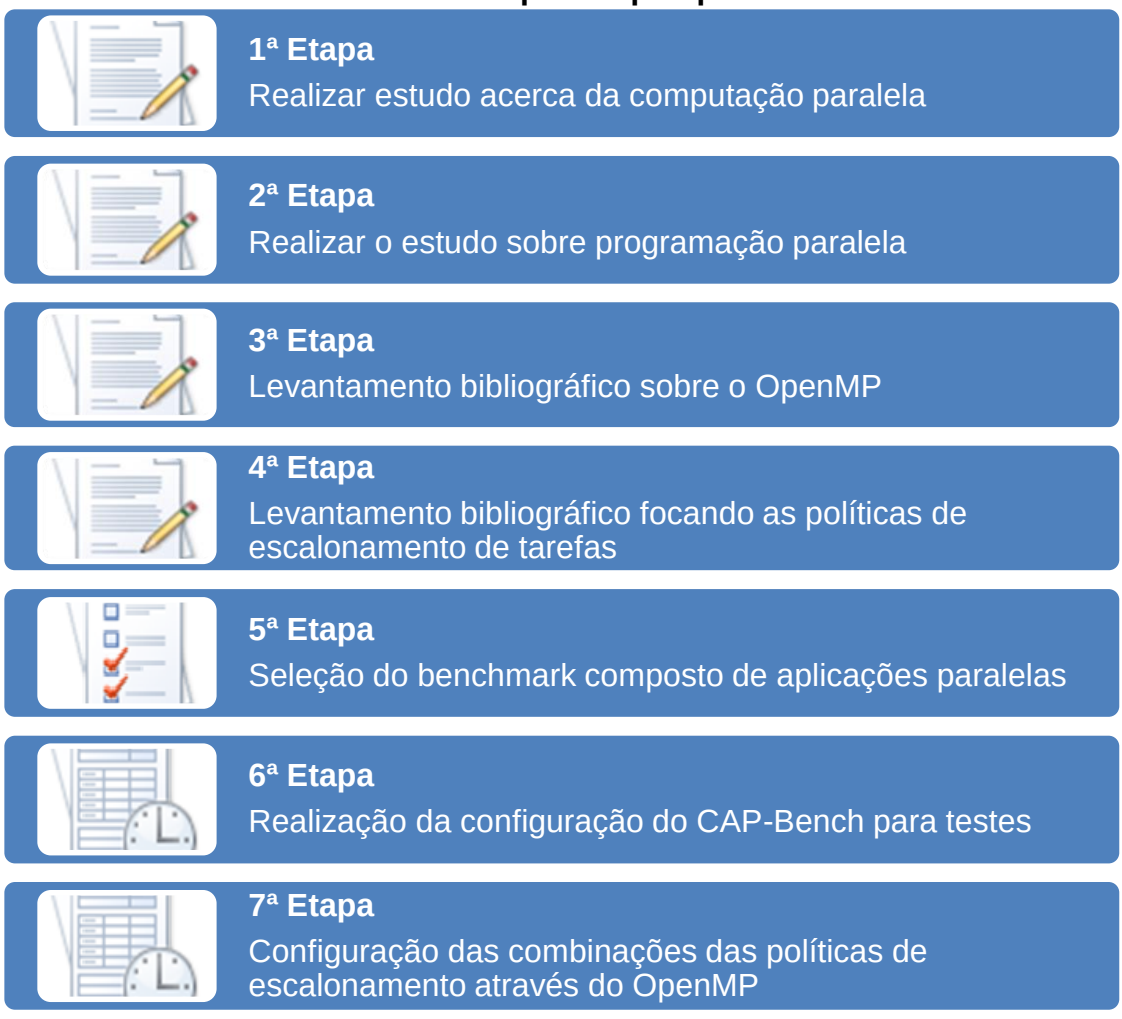
Este trabalho apresenta uma pesquisa quantitativa, do tipo experimental descritiva, onde os dados foram observados, registrados, analisados, classificados e interpretados, utilizando coleta de dados e observação sistemática com base na literatura científica.

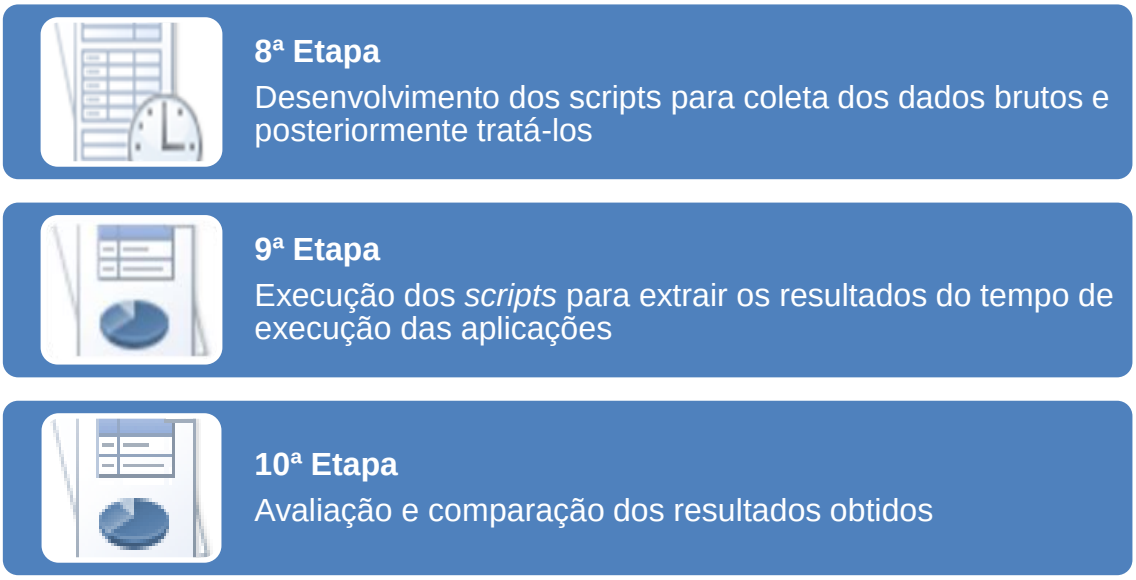
A metodologia deste trabalho possui mais 3 seções onde será apresentado no item 3.1 as Etapas da pesquisa, no item 3.2, o Ambiente experimental e por último item 3.3, a Carga de trabalho.

3.1 Etapas da Pesquisa

A FIGURA 4 apresenta de modo resumido as etapas da pesquisa.

FIGURA 4 - Etapas da pesquisa





Fonte: Elaborado pelo autor

A seguir são descritos de forma detalhada, os processos de cada uma das etapas utilizadas nesta pesquisa.

Primeiramente, foi realizado um levantamento bibliográfico sobre computação paralela. Nesse levantamento foi possível identificar a importância da computação paralela no dias atuais. Dessa forma, foi estudada a evolução dos processadores, os diferentes tipos de arquiteturas, modelos, paradigma de programação paralela e seus desafios.

Na segunda etapa, foi realizado um estudo sobre paradigma de programação paralela e seus desafios. Nesta etapa foi possível identificar os vários obstáculos impostos à paralelização de uma aplicação trabalhando-se com várias tarefas simultaneamente.

Na terceira etapa, realizou-se o levantamento bibliográfico sobre OpenMP, possibilitando compreender a programação contida em aplicações paralelas para processadores de múltiplos núcleos.

Na quarta etapa, foi feito um estudo sobre as políticas de escalonamento que permitiu trabalhar com aplicações OpenMP.

Na quinta etapa foi selecionado o *benchmark* que dispusesse de aplicações paralelas com várias características diferentes para que pudessem ser analisadas em termos de ganho de desempenho, e comportamento ao configurar políticas de escalonamento diferentes.

Na sexta etapa foi configurado o CAP-Bench para execução. Essa configuração consiste em adaptar o *benchmark* ao sistema operacional, em termos de compilação, execução dos *kernels*, além de entender seu funcionamento.

Na sétima etapa, foram inseridas as políticas de escalonamentos *static*, *dynamic* e *guided* com tamanhos de tarefas de 1, 32 e 1024 em cada um dos *kernels* do CAP-Bench.

A oitava etapa, consiste na criação dos *scripts* uma vez que a combinação das políticas de escalonamento com números de *threads* para execução eram muito grandes inviabilizando a execução manual de cada configuração por vez.

Na nona etapa, executou-se os scripts de coleta de dados e tempo médio de execução das aplicações.

Na décima etapa, foi realizada a avaliação dos resultados, selecionando as combinações das políticas de escalonamento, comparando valores de tempo de execução, plotagem de gráficos, etc.

3.2 Ambiente Experimental

Todos os experimentos foram submetidos ao sistema com processadores com múltiplos núcleos. As características e componentes deste sistema é descrito no QUADRO 1.

QUADRO 1 - Características e componentes do sistema utilizado

Componentes do sistema	Características
Processador	Intel Xeon 5645 (2x)
Clock	2.4GHz
Quantidade de Núcleos	6 (2x)
HyperThreading	24
Tamanho de memória	32GB
Memória Cache	12MB
Sistema Operacional	CentOS 6.5
Versão OpenMP	3.1
Compilador	GCC versão 4.7

Fonte: Elaborado pelo autor com dados extraídos no servidor Quadro, Instituto de Ciências Exatas e Informática PUC Minas, 2015.

O QUADRO 1 detalha as características dos processadores *multicore* e demais componentes do sistema. Foi utilizado como ambiente experimental um servidor de processamento, sendo este, composto por dois processadores de múltiplos núcleos, operando em uma frequência de 2.4GHz em seis núcleos físicos cada e possui 32 Gigabytes de memória RAM. Em relação aos *softwares* utilizados, o sistema operacional CentoOS 6.5 é compatível com GCC 4.7 e OpenMP versão 3.1.

Em paralelo, este computador possui uma GPU Nvidia Quadro 2000, porém as aplicações contidas no *benchmark* utilizado não explora o paralelismo neste coprocessador.

Com o objetivo de realizar a avaliação de aplicações paralelas com o OpenMP sobre os processadores, foram utilizadas aplicações contidas em *benchmarks*. O *benchmark* atribuído a este trabalho apresenta diferentes cargas de trabalho. Algumas de suas aplicações possuem desbalanceamento de carga, enquanto outras apresentam cargas mais balanceadas. Dessa forma, as aplicações tendem a mostrar diferentes resultados em termos de desempenho ao alternar entre as políticas de escalonamento.

3.3 Carga de Trabalho

No CAP-Benchmark, a distribuição da carga de trabalho é padronizada em termos qualitativos, sendo 5 diferentes opções de tamanho para processamento (*problem sizes*), muito pequeno (*tiny*), pequeno (*small*), médio (*standard*), grande (*large*) e muito grande (*huge*). Porém em termos quantitativos, cada aplicação tem sua própria proporção referente a carga de trabalho. A TABELA 1 descreve a carga de trabalho das aplicações paralelas em ambos os contextos: qualitativo e quantitativo.

TABELA 1 – Cargas de trabalho e suas respectivas aplicações

Cargas/Kernels	FAST	FN	GF	IS	KM	LU	RT
Tiny	2048	8004096	2048	8388606	4096	512	1280
Small	4096	8008192	4096	16777216	8192	1024	1600
Standard	8192	8016384	8192	33554432	16384	1536	1920

Large	16384	8032768	16384	671088647	32768	2048	2048
Huge	24576	8065536	32768	134217728	65536	2560	2560

Fonte: Elaborado pelo autor com dados extraídos da documentação do CAP-Bench.

Os valores expostos na TABELA 1 mostram valores que, por gerar uma grande quantidade de combinações junto as configurações de escalonamento, fez-se necessário definir uma única carga de trabalho para todas as aplicações. Para não atingir um tempo de espera muito alto nos experimentos, a carga escolhida foi a pequena (*small*) sendo repetida 10 vezes para cada política de escalonamento.

Inicialmente os testes foram realizados com 1 núcleo de processamento para registrar o desempenho em relação ao tempo de execução na versão sequencial das aplicações. Na primeira configuração, as aplicações sequenciais (OpenMP configurado para processar com uma única *thread*) foram executadas sem escalonamento e posteriormente, selecionadas as políticas de escalonamento estático, dinâmico e guiado. Com esses resultados foi possível comparar o tempo de execução da versão paralela e versão sequencial para calcular seus resultados em termos de ganho de desempenho.

Para executar o código em paralelo, foram utilizadas combinações entre opções de políticas de escalonamento do OpenMP, e tamanhos de tarefas entre 1, 32 e 1024. Em relação ao número de *threads*, foram variadas entre 1, 2, 4, 8, 12 e 24.

Porém, como todos esses atributos geram uma gama muito grande de combinações, foi fixada uma configuração no qual pudesse avaliar o desempenho entre as versões sequencial e paralelas com o máximo de núcleos e diferentes políticas de escalonamento, pois, nesses casos, as aplicações apresentavam melhor desempenho. A seleção desta configuração é apresentada seguida de suas siglas no QUADRO 2.

QUADRO 2 - Configuração de execução das aplicações e respectivas siglas

Quantidade de *threads*	Tipo de escalonamento	Tamanho da tarefa	Sigla
1	-	-	1T_SS
12	estático	32	12T_stc_32
	estático	1024	12T_stc_1024
	guiado	32	12T_gdd_32
	guiado	1024	12T_gdd_1024

	dinâmico	32	12T_dyn_32
	dinâmico	1024	12T_dyn_1024

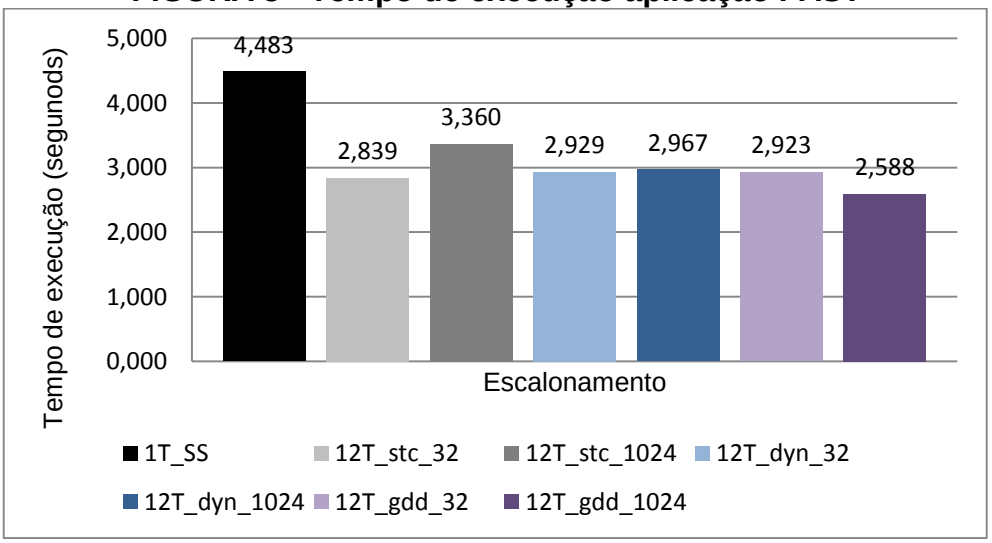
Fonte: Dados da pesquisa

Os testes permitem avaliar através de seus resultados, o tempo de execução em relação a versão sequencial das aplicações paralelizadas. Cada aplicação foi executada 10 vezes para cada configuração de política de escalonamento, tamanho de tarefas e número de *threads* diferentes. Após o registro desses dados, foi calculada a média aritmética do tempo de execução.

4 RESULTADOS EXPERIMENTAIS

A primeira aplicação contida no CAP-Bench submetida à avaliação foi o FAST. Nesta aplicação por possuir características de desbalanceamento de carga, e por isto o escalonamento dinâmico é mais adequado nesse caso. Em relação à diferença do tempo de execução entre a versão sequencial (OpenMP com 1 *thread*) e paralela desta aplicação, não se obtém um ganho significativo de desempenho quando a carga de trabalho do FAST é pequena. A FIGURA 5 mostra o tempo de execução do FAST com as políticas de escalonamento testadas.

FIGURA 5 - Tempo de execução aplicação FAST



Fonte: Dados da pesquisa

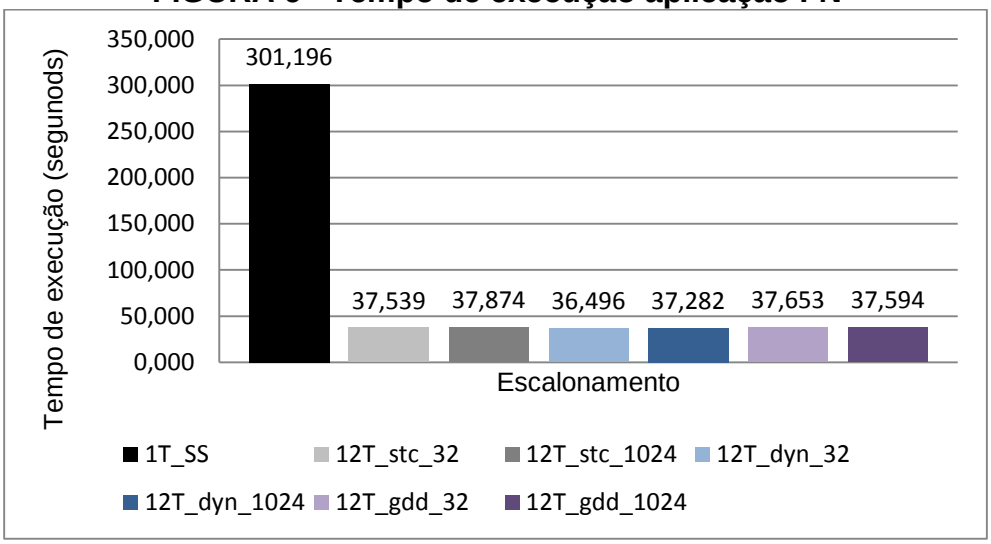
O escalonamento guiado, por se tratar de um escalonamento dinâmico, fez com que a aplicação FAST alcançasse o melhor desempenho de 2,588 segundos, sendo, escalonamento guiado e o tamanho da tarefa de 1024. Por outro lado, a aplicação em sua versão sequencial proporciona o pior desempenho, atingindo 4,483 segundos para ser executado.

Em relação à diferença entre as configurações de escalonamentos dinâmicos, sobre o tempo de execução, o escalonamento guiado executou a aplicação com uma diferença de 12,8% mais rápido do que o dinâmico. O tamanho da tarefa influencia mais no escalonamento guiado com 1024. Já o escalonamento estático foi mais eficiente com tarefa pequena de tamanho 32, ao invés de 1024. Entre o

estático e o guiado com tarefas maiores de 1024, houve uma diferença de 23% melhor no guiado.

Ao contrário da aplicação FAST, FN exige mais em termos de processamento. A execução do código sequencial gasta em torno de 301 segundos para ser finalizada. A versão paralela desta aplicação com 12 *threads* provê um *speedup* de até 8,2 vezes com escalonamento dinâmico de tarefas e tarefas com tamanhos pequenos de 32. O melhor tempo de execução com as configurações de escalonamento foi de 36,496 segundos. As demais configurações de escalonamento apresentaram um comportamento semelhante em relação ao melhor tempo de execução, mantendo-se na faixa 37,5 segundos. A FIGURA 6 apresenta estes resultados.

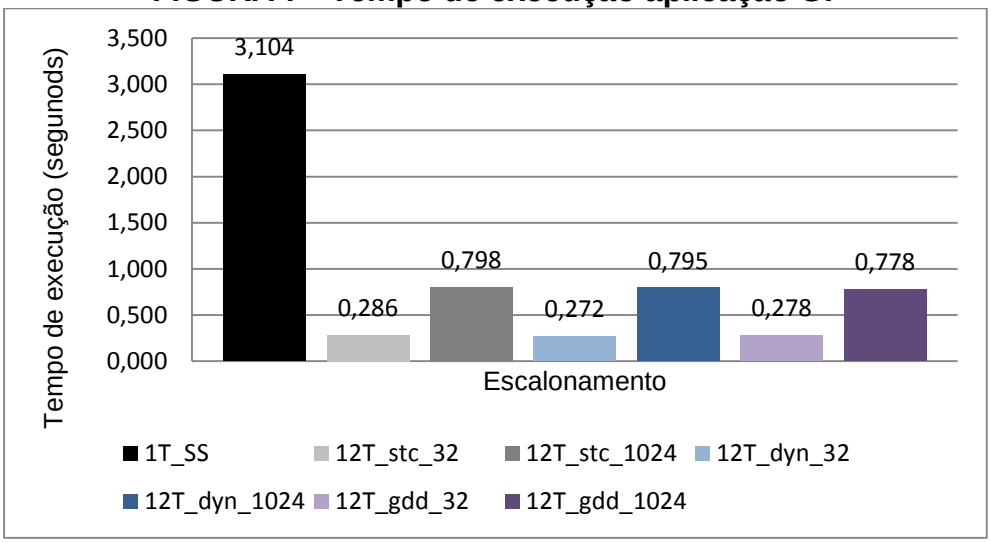
FIGURA 6 - Tempo de execução aplicação FN



Fonte: Dados da pesquisa

A aplicação GF (próxima página) mostra os melhores resultados quando a configuração do tamanho da tarefa é de 32 (0,272 segundos), sendo que, a escolha da politica de escalonamento não altera tanto entre escalonamentos dinâmicos e estático, porém, ao selecionar tamanho de tarefa de 1024 (0,795 segundos) o desempenho é afetado. Esta diferença chega a 65,8% entre tarefas de tamanho 32 e 1024 considerando o mesmo número de *threads*, como mostrado na FIGURA 7.

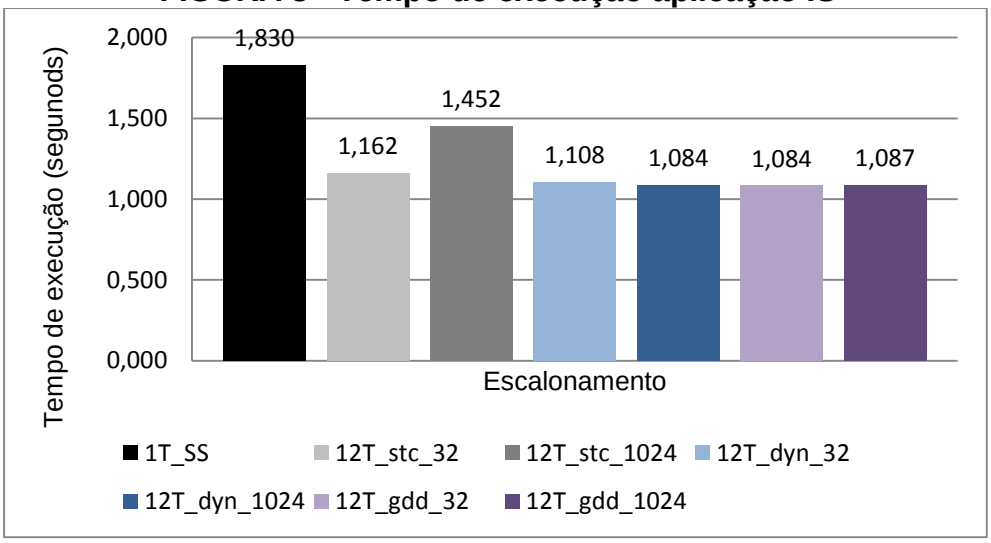
FIGURA 7 - Tempo de execução aplicação GF



Fonte: Dados da pesquisa

A avaliação dos resultados da aplicação IS mostra que o tempo de alocar tarefas às *threads* é mais regular, pois, a variação entre escalonamentos dinâmicos é mínima. No caso do escalonamento estático, este apresenta queda de desempenho em relação à outras configurações com o tempo de execução em 1,452 segundos, onde quase se iguala a versão sequencial com apenas 1 *thread*. A FIGURA 8 mostra os resultados com as políticas de escalonamento da aplicação IS.

FIGURA 8 - Tempo de execução aplicação IS

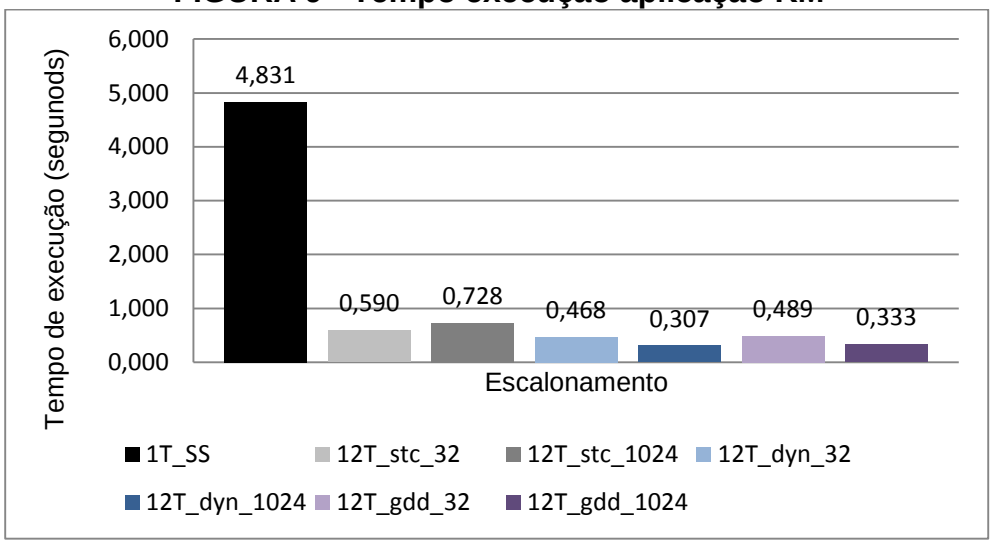


Fonte: Dados da pesquisa

A aplicação KM apresenta um resultado 93,6% melhor em tempo de execução na versão paralela com 12 *threads*, escalonamento dinâmico e o valor de

1024 no tamanho das tarefas. Neste caso, as políticas de escalonamento demonstram melhor desempenho com configurações dinâmicas e tarefas grandes de 1024 em comparação ao escalonamento estático. KM possui características irregulares referente a carga de trabalho (aplicação desbalanceada) das tarefas para a execução das *threads*. E por esse motivo, executar esta aplicação com tarefas pequenas de tamanho 32 pode ocasionar sobrecarga por ter que requisitar pequenas tarefas a todo momento para executar o processamento. A FIGURA 9 apresenta estes resultados.

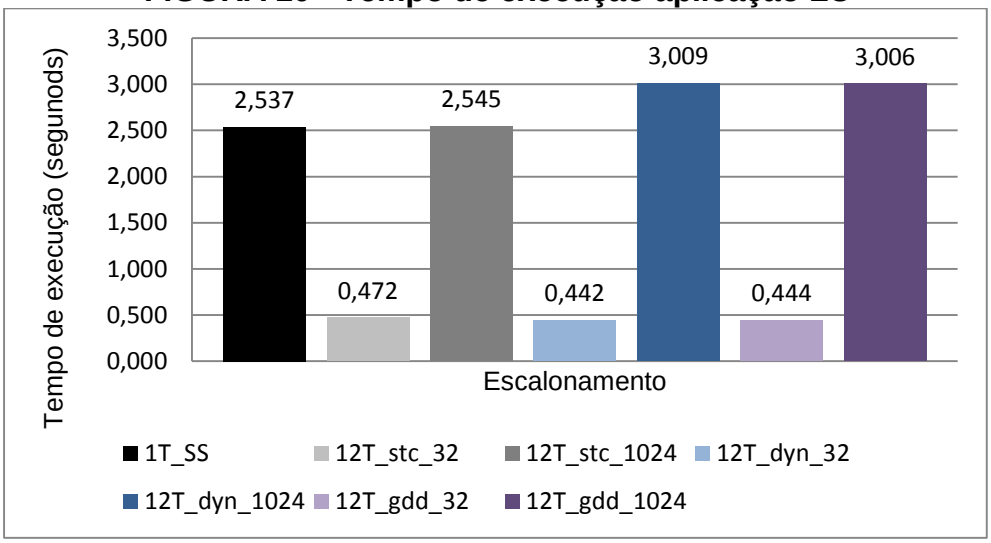
FIGURA 9 - Tempo execução aplicação KM



Fonte: Dados da pesquisa

O tempo de execução da aplicação LU é melhor executada com escalonamentos de tarefas pequenas, tanto na configuração estático como no dinâmico. Essa diferença atinge proporções de até 85,3% com tarefas de tamanho 32, onde o melhor tempo de execução foi 0,442 segundos, sendo ainda 5,74 vezes mais rápido do que a execução sequencial da aplicação. A FIGURA 10 mostra o desempenho das configurações referente às políticas de escalonamento.

FIGURA 10 - Tempo de execução aplicação LU



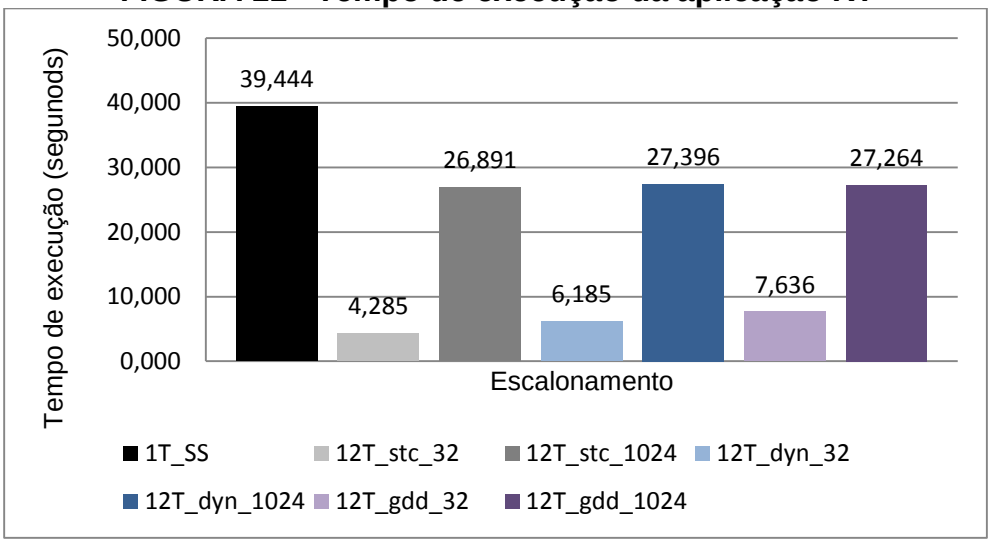
Fonte: Dados da pesquisa

O tempo de execução da aplicação RT é melhor executada com escalonamento de pequenas tarefas, tanto na configuração estático como no dinâmico. A avaliação que pode ser feita no escalonamento estático, mostra que a diferença no tamanho entre tarefas influencia drasticamente no ganho de desempenho. Essa diferença chega ser de 84,1%, sendo o escalonamento estático e tamanho de tarefas 32, onde se obteve o melhor tempo de execução.

Em relação a ambos escalonamentos dinâmicos com tarefas de 1024, o tempo de execução também aumenta, ficando em média de 27 segundos, mas, ao configurar o tamanho de tarefas para 32, é possível notar que o tempo de execução cai em torno de 72% à 77,4%, nas políticas de escalonamento guiado e dinâmico respectivamente.

O ganho de desempenho foi de 9,21 vezes mais rápido do que a execução sequencial da aplicação. Este ganho foi proporcionado pela configuração de escalonamento estático com tamanho de tarefas de 32. A FIGURA 11 mostra o tempo de execução da aplicação RT com as configurações de escalonamento.

FIGURA 11 - Tempo de execução da aplicação RT



Fonte: Dados da pesquisa

Em síntese, a avaliação mostra que as aplicações possuem diferentes comportamentos ao alterar as políticas de escalonamento através do OpenMP. Esta proposição consolida e confirma através dos resultados experimentais questões levantadas entre a teoria e a prática. Estas questões são referentes ao balanceamento de carga, afinidade de memória, sobrecarga de criação de *threads,* e o quanto essas características impactam em termos de desempenho ao selecionar uma política de escalonamento inadequada de acordo com essas questões.

A partir desses aspectos, aplicações como GF, LU, RT apresentaram resultados satisfatórios e de melhor desempenho ao selecionar tarefas pequenas de tamanho 32, para a execução das *threads* ao configurar as políticas de escalonamento no OpenMP. Esta afirmação é válida tanto para o escalonamento estático, quanto para os escalonamentos dinâmicos.

As aplicações IS e FN são mais estáveis na diferença entre políticas de escalonamento e tamanho das tarefas para execução, não apresentando grandes diferenças em desempenho. FAST e IS são equivalentes ao se tratar de escalonamento estático e tamanho de tarefas 1024 proporcionando piores resultados de desempenho e se igualando, quando referenciadas as demais configurações.

Em relação aos escalonamentos estático e dinâmico, as aplicações RT e LU apresentaram melhores resultados com a política de escalonamento de tarefas estática, cláusula *static* do OpenMP. Por outro lado, aplicações como, FAST e KM

demostraram melhor desempenho com a configuração *guided*, enquanto a configuração *dynamic* também proporcionou bom desempenho para GF e KM. As aplicações IS e FN mostraram em seus resultados que a política de escalonamento não influencia tanto quando as demais, isso pode ter ocorrido devido ao fato de serem mais balanceadas do que as outras aplicações ou possuir mais afinidade de memória. O QUADRO 3 mostra um comparativo entre os melhores resultados obtidos das aplicações:

QUADRO 3 - Menores tempos de execução por aplicação

Aplicação	Política de Escalonamento	Tempo de execução
FAST	guided_1024	2,588 segundos
FN	dynamic_32	36,496 segundos
GF	dynamic_32	0,272 segundos
IS	guided_32 e dynamic_1024	1,084 segundo (ambos)
KM	dynamic_1024	0,307 segundos
LU	dynamic_32	0,442 segundos
RT	static_32	4,285 segundos

Fonte: Elaborado pelo autor com dados extraídos no servidor Quadro, Instituto de Ciências Exatas e Informática PUC Minas, 2015.

5 CONCLUSÃO

Esse trabalho apresentou uma avaliação comparativa entre as políticas de escalonamento de tarefas via OpenMP das aplicações (ou *kernels*) contidos no CAP-Benchmark *suite*, com o principal objetivo de analisar o comportamento do desempenho destas aplicações alterando as políticas de escalonamento e o tamanho das tarefas de cada aplicação.

Os resultados mostram que as configurações de escalonamento com tamanhos de tarefas grandes para aplicações que possuem carga de trabalho maiores provocam queda no desempenho. Dessa forma, o balanceamento de carga tem relação entre o tamanho da tarefa estabelecida no *chunk_size* que é diretamente dependente do número de iterações na estrutura de repetição na região paralelizada, podendo até causar a ociosidade de outros elementos de processamento.

De modo geral, obteve-se melhores resultados ao configurar tarefas pequenas de 32 em comparação a tarefas de 1024 na maioria das aplicações, mas, quando os resultados mostravam que tarefas maiores eram melhores do que tarefas pequenas, essa diferença em termos de desempenho não era tão significativa. Desta forma, como principal **contribuição**, tem-se uma avaliação original documentada do CAP-Bench onde, pretende-se confeccionar um artigo científico para publicação do mesmo.

A avaliação alcançou seu principal objetivo de discernir o quão é importante realizar uma boa escolha entre as políticas de escalonamento disponíveis para OpenMP, além de mostrar que as diferentes configurações do escalonamento de tarefas, de fato, alteram o desempenho das aplicações paralelas. Como **trabalhos futuros** existe a possibilidade de realizar uma nova pesquisa no intuito de aprimorar estes testes, coletando mais informações de hardware para melhor conhecer as características das aplicações e até mesmo desenvolver mecanismos de mapeamento automático de tarefas.

REFERÊNCIAS

AJI, Ashwin M. **Programming high-performance clusters with heterogeneous computing devices**. 2015. 137f. Tese (Doutorado) – Virginia State University, Faculty of the Virginia Polytechnic Institute, Virginia, 2015.

ALICE IN CHAINS. Black Gives Way to Blue. Black Gives Way to Blue. Warner Bros, 2009. Faixa 11. Compact Disk.

Advanced Micro Devices AMD. AMD Radeon R9 Series Graphics Cards with High-Bandwidth Memory. Disponível em: <http://www.amd.com/en-us/products/graphics/desktop/r9>. Acesso em: 5 fev. 2016.

CHANDRA Rohit et al. **Parallel programming in openmp**. San Francisco: Morgan kaufmann, 2001. 231 p.

CHAPMAN, Barbara; JOST, Gabriele; PAS, Ruud van der. **Using OpenMP:** Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). United States of America: MIT Press, 2007. 353 p.

COUDER-CASTANEDA, Carlos et al. Performance of a code migration for the simulation of supersonic ejector flow to SMP, MIC, and GPU using OpenMP, OpenMP+ LEO, and OpenACC directives. **Scientific Programming**, v. 2015, p. 17, 2015.

DE ROSE, César A. F.; NAVAUX, Philippe O. A. **Arquiteturas paralelas.** Porto Alegre: Bookman, 2008. 152 p. (Série livros didáticos; v.15).

DETONI, Gabriel Girardello. **MTC:** modelo de programação paralela baseado na perspectiva conexionista. 2010. 129 f. Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação, Rio Grande do Sul, 2010.

DIAZ, Javier; MUÑOZ-CARO; NIÑO, Alfonso. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. vol. 23, no. 8, August 2012. IEEE Transactions on parallel and distributed systems.

DIENER, Matthias. **Automatic task and data mapping in shared memory architectures**. 2015. 187f. Tese (Doutorado) - Technische Universität Berlin, Fakultät 4 Elektrotechnik und Informatik, Berlin, 2015.

FARKAS, Rakesh Kumar et al. A multi-core approach to addressing the energy-complexity problem in microprocessors. In: Workshop on Complexity-Effective Design (WCED). **Proceedings…** California, 2003.

FLYNN, Michel J. Very high-speed computings systems. In: **Proceedings of the IEEE**. v.54. n. 12, p. 1901-1909, 1966.

GITHUB. **Cart nAnosim gPpd (CAP) Benchmark suite for manycore processors**. Disponível em: <https://github.com/cart-pucminas/cap-benchmarks>. Acesso em: 14 set. 2015.

GÓES, Luís Fabrício Wanderley. **Proposta e desenvolvimento de um algoritmo reconfigurável de escalonamento paralelo de tarefas**. 2004. 99 f. Dissertação (Mestrado) - Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte, 2004.

GÓES, Luís Fabrício Wanderley. et al. Automatic Skeleton-Driven Memory Affinity for Transactional Worklist Applications. **International Journal of Parallel Programming**, v. 42, n. 2, p. 365-382, 2014.

GONÇALVES, Rogério Aparecido. et al. OpenMP is not as easy as it appears. **49th Hawaii International Conference on System Sciences (HICSS)**, Koloa, HI, 2016, p. 5742-5751, 2016.

HENNESSY, John L.; PATTERSON, David A. **Arquitetura de computadores:** uma abordagem quantitativa. Rio de Janeiro: Elsevier, c2008. xxii, 494 p.

HENNESSY, John L.; PATTERSON, David A. **Arquitetura de computadores:** uma abordagem quantitativa. 5. ed. Rio de Janeiro: Elsevier, Campus. c2014. 744 p.

JOST, Gabriele et al. Comparing the OpenMP, and hybrid programming paradigms on an SMP cluster. In: **Fifth European Workshop on OpenMP** (EWOMP03), Aachen, Germany, Sep. 2003.

KHRONOS Group. **OpenCL - The open standard for parallel programming of heterogeneous systems.** Disponível em: <http://www.khronos.org/opencl/>. Acesso em: 20 jan. 2016.

MUDDUKRISHNA, Ananya. **Improving OpenMP Productivity with Data Locality Optimizations and High-resolution Performance Analysis**. 2016. Tese (Doutorado) – KTH School of Information and Communication Technology, Kista, Stockholm, 2016.

NIKOLOPOULOS, Dimitrius S. et al. **Exploiting memory affinity in OpenMP through schedule reuse**. Volume 29, pages 49 – 55. ACM SIGARCH Computer Architecture News. New York, 2001.

NVIDIA. **Placa de vídeo GeForce GTX 980**. Disponível em: <http://www.nvidia.com.br/object/geforce-gtx-980-br.html#pdpContent=1>. Acesso em: 4 fev. 2016.

OLIVIER, Stephen L. et al. OpenMP task scheduling strategies for multicore NUMA systems. In: International Journal of High Performance Computing Applications, 2012.

OPEN Multi-Processing OPENMP. **The OpenMP API specification for parallel programming**. Disponível em: <http://openmp.org/wp/2016/02/>. Acesso em: 6 fev. 2016.

QAWASMEH, Ahmad; MALIK, Abid M.; CHAPMAN, Barbara M. OpenMP task scheduling analysis via OpenMP runtime API and tool visualization. In: Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International. IEEE, 2014. p. 1049-1058.

RAINEY, Michael Alan. **Effective scheduling techniques for high-level parallel programming languages**. 2010. 135 f. Tese (Doutorado) – University of Chicago, Department of Computer Science, Chicago, Illinois, 2010.

RAUBER Thomas, RÜNGER, Gudula. Parallel programming for multicore and cluster systems. 2. ed. Berlin: Springer-Verlag, 2013. 516 p.

SATO, L.M.; MIDORIKAWA, E.T.; SENGER, H. Introdução à programação paralela e distribuída. In: *JAI 1996 - Jornada de Atualização em Informática, XVI Congresso da Sociedade Brasileira de Computação* - Ed. Depto. de Informática - Universidade Federal de Pernambuco. **Anais...** Recife / UFPE: DI-UFPE, 1996.

SENA, M. C. R.; COSTA J. A. C. "**Tutorial OpenMP C/C++**". Programa Campus Ambassador HPC. Maceió. Março 2008.

SEVERO, E. B., SERPA M. da S. e SCHEPKE C. "Avaliando a Performance das Políticas de Escalonamento de OpenMP no Método de Lattice Boltzmann". In: proceeding of XII SIRC – Simpósio de Informática. Outubro, 2013.

SILVA, Fernando. PPD: scheduling and Load Balancing. Anotações de aulas, Computer Science Department. Center of Research in Advanced Computing Systems. Universidade do Porto, FCUP. [20--]. Disponível em: <http://www.dcc.fc.up.pt/~fds/aulas/PPD/1112/loadbalancing.pdf>. Acesso em: 20 nov. 2015.

SILVA Márcio de Oliveira da. **Controle de granularidade de tarefas em OpenMP**. 2011. 66f. Monografia (Bacharel) – Universidade Federal do Rio Grande do Sul, Instituto de Informática, Porto Alegre, 2011.

SIVANANDAN, Vinaya; KUMAR, Vikas; MEHER, Srisai. Designing a parallel algorithm for Heat conduction using MPI, OpenMP and CUDA. In: Parallel Computing Technologies (PARCOMPTECH), 2015 National Conference on. IEEE, 2015. p. 1-7.

STALLINGS, William. **Arquitetura e organização de computadores:** projeto para o desempenho. 8. ed. São Paulo: Pearson, 2010. 624 p.

TREW, Arthur; WILSON, Greg. **Past, present, parallel:** a survey available parallel computing systems. Berlin: Springer-Verlag, 2012. 387 p.

WANG, Ke et al. Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales. **Concurrency and Computation: Practice and Experience**, v. 28, n. 1, p. 70-94, 2016.

WANG, Zheng; O'BOYLE, Michael FP. Mapping parallelism to multi-cores: a machine learning based approach. In: **ACM Sigplan notices**. ACM, 2009. p. 75-84.