DENILSON SOUZA BÉLO

Programação e execução de aplicações distribuídas baseadas em tarefas em sistemas heterogêneos

São Paulo
2023

DENILSON SOUZA BÉLO

Programação e execução de aplicações distribuídas baseadas em tarefas em sistemas heterogêneos

Versão corrigida

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Ciências.

São Paulo
2023

DENILSON SOUZA BÉLO

Programação e execução de aplicações distribuídas baseadas em tarefas em sistemas heterogêneos

Versão corrigida

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Ciências.

Área de concentração:
Engenharia de Computação

Orientadora:
Prof.ª Dr.ª Liria Matsumoto Sato

São Paulo
2023

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

São Paulo, 26 de Junho de 2023

Assinatura do autor: _____

Assinatura do orientador: _____

Catalogação-na-publicação

Bélo, Denilson Souza
Programação e execução de aplicações distribuídas baseadas em tarefas em sistemas heterogêneos / D. S. Bélo -- versão corr. -- São Paulo, 2023.
209 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Computação de Alto Desempenho 2.Programação Paralela 3.Paralelização de Tarefas 4.Arquitetura Heterogênea I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

Primeiramente à Deus, nosso Senhor. À minha amada esposa, Pamela. Aos meus pais, Daniel e Doralice. À minha sogra, Jocelizia. E à minha orientadora Professora Doutora Liria Matsumoto Sato.

AGRADECIMENTOS

Primeiramente a Deus Pai Todo Poderoso por ter entregue essa tarefa em minhas mãos. "*Porque Dele, e por Ele, e para Ele são todas as coisas; glória, pois, a Ele eternamente. Amém!*"(Rm 11,36).

À Pamela, minha amada e sábia esposa que edifica o nosso lar, por sua paciência, apoio moral durante o período de mestrado e pelos momentos de exortação. Muito obrigado por estar ao meu lado.

Aos meus familiares, especialmente meus pais Daniel e Doralice, e a minha sogra Jocelizia. Vocês são a inspiração para minha vida. Os seus ensinamentos e apoio me trouxeram até aqui.

Um agradecimento muito especial à minha orientadora, Liria Matsumoto Sato, por sua longaminidade. Não tenho palavras para agradecer o cuidado e seu apoio. Deus a abençoe.

Aos amigos do *Laboratory of Architecture and High Performance Computing* (LAHPC) do Departamento de Engenharia de Computação e Sistemas Digitais (PCS) da Escola Politécnica da Universidade de São Paulo (EPUSP) pela parceria e pelos belos momentos.

E também aos professores e funcionários da Universidade de São Paulo, da Escola Politécnica da USP e do Departamento de Engenharia de Computação e Sistemas Digitais (PCS) pelo apoio técnico, humano e conhecimento. Meu muito obrigado!

“Isso de querer ser exatamente aquilo que a gente é ainda vai nos levar além”

Paulo Leminski

RESUMO

BÉLO, Denilson Souza. **Programação e execução de aplicações distribuídas baseadas em tarefas em sistemas heterogêneos**. 209p. Dissertação (Mestrado em Ciências) – Escola Politécnica, Universidade de São Paulo, 2023.

Atualmente há uma crescente demanda pelo uso da Computação de Alto Desempenho. Uma forma para explorar essa capacidade de processamento é a utilização de computadores conectados em uma rede de alta velocidade. Cada computador poderá possuir processadores com um número crescente de núcleos. E para melhorar ainda mais a performance se acrescentam aceleradores de *hardware* como GPGPUs. O desenvolvimento de aplicações em HPC é uma tarefa complexa, pois exige o entendimento das estratégias de paralelização, e a utilização de recursos heterogêneos intensifica ainda mais essa dificuldade. Este trabalho tem como objetivo propor um modelo de programação baseada em tarefas paralelas para sistemas distribuídos com arquiteturas heterogêneas compostas por CPUs e aceleradores de *hardware*, através de uma sintaxe simples e direta e que garanta um desempenho equivalente aos modelos atuais de programação distribuída. São apresentados os testes de avaliação da funcionalidade, desempenho e de comparação com o uso do sistema OpenMPI, amplamente utilizado em sistemas distribuídos. Os principais resultados alcançados foram: (1) a facilidade de utilização independente da arquitetura empregada, quer sejam CPUs e/ou aceleradores de *hardware*; (2) mostraram desempenho muito equivalente, em alguns casos superiores, aos obtidos com o uso do sistema OpenMPI. Estes resultados comprovam a viabilidade da solução proposta.

Palavras-chaves: Programação paralela. Paralelização de tarefas. Arquitetura heterogênea.

ABSTRACT

BÉLO, Denilson Souza. **Programming and running task-based distributed applications on heterogeneous systems**. 209p. Dissertation (Master in Science) – Escola Politécnica, Universidade de São Paulo, 2023.

Currently, there is a growing demand for the use of High Performance Computing. One way to exploit this processing capacity is used by of computers connected in a high-speed network. Each computer has processors with an increasing number of cores. And hardware accelerators like GPGPUs are added to further improve performance. The development of applications in HPC is a challenging task, as it needs an understanding of parallelization strategies, and the use of heterogeneous resources intensifies this difficulty even more. This work presents a programming model based on parallel tasks for distributing systems with heterogeneous architecture composed of CPUs and hardware accelerators, through a simple and direct syntax. Functional evaluation tests, performance, and comparison with the OpenMPI system, widely used in distributed systems, are presented. The main results achieved were: (1) user-friendliness regardless of the architecture employed, whether CPUs and/or hardware accelerators; (2) showed performance very similar, in some cases higher, than those obtained using the OpenMPI system. The results achieved confirm the feasibility of the proposed solution.

Keywords: Parallel programming. Task parallelization. Heterogeneous architecture.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API *Application Programming Interface*.

ARM *Advanced RISC Machine*.

BoT *Bag Of Tasks*.

CC *Communication Cost*.

CPU *Central Processing Unit*.

DL *Dynamic Loading*.

DTT *Data Transmission Time*.

FPGA *Field Programmable Gate Arrays*.

GCC *GNU Compiler Collection*.

GPGPU *General Propose Graphic Processor Unit*.

HPC *High Performance Computing*.

MIC *Many Integrated Core*.

MIMD *Multiple Instruction, Multiple Data*.

MISD *Multiple Instruction, Single Data*.

MPI *Message Passing Interface*.

POSIX *Portable Operating System Interface*.

RTE *Remote Tasks Execute*.

SIMD *Single Instruction, Multiple Data*.

SISD *Single Instruction, Single Data*.

SMP *Simetric Multiprocess*.

STF *Sequential Task Flow*.

TCP *Transmission Control Protocol*.

TLP *Task/Thread Level Paralelism*.

TVM *Task Vector Machine*.

SUMÁRIO

1 INTRODUÇÃO

Os últimos anos apresentaram a utilização cada vez maior de *High Performance Computing* (HPC). A popularização da Ciência dos Dados é um exemplo dessa demanda de recursos de alto desempenho computacional (SCHMIDT; HILDEBRANDT, 2017). Para supri-la, computadores com um número crescente de núcleos por *Central Processing Unit* (CPU) e coprocessadores vêm sendo apresentados. Tais computadores podem estar conectados em uma rede, compondo um ambiente distribuído, aumentando o desempenho computacional. Para melhorar a performance utilizam recursos heterogêneos como aceleradores de *hardware*. Tais recursos podem ser *General Propose Graphic Processor Unit* (GPGPU) (WEN-MEI; KIRK; HAJJ, 2022), e ou *Many Integrated Core* (MIC) (JEFFERS; REINDERS; SODANI, 2016). A lista dos supercomputadores mais rápidos do mundo[1] apresenta computadores que utilizam esses recursos para melhorar o desempenho. O supercomputador brasileiro Santos Dumont[2] possui servidores com arquitetura heterogênea CPU, GPGPU e MIC e são interconectados por uma rede de alta velocidade, fazendo uso dos recursos simultaneamente.

Para explorar adequadamente todos os recursos de processamento disponíveis em um supercomputador ou em um ambiente distribuído são necessárias ferramentas de programação e execução.

1.1 Objetivo

O principal objetivo deste trabalho é apresentar uma solução simples e de fácil uso no desenvolvimento de aplicações HPC através de uma codificação simplificada composta por poucas funções com uma sintaxe que pode ser utilizada tanto em processadores quanto nos aceleradores de *hardware*. Propõe um modelo de programação e execução e a respectiva biblioteca, denominada de RTE, que proveem uma forma de paralelização de tarefas, em que cada tarefa é executada por um dos recur-

[1] <https://www.top500.org/lists/top500/2022/11/>
[2] <https://sdumont.lncc.br/machine.php?pg=machine#>

sos de processamento. Esta paralelização permite utilizar de maneira eficiente cada recurso disponível nas arquiteturas heterogêneas sejam núcleos das CPUs dos computadores, que compõem um ambiente distribuído, ou coprocessadores, como MIC ou GPGPUs, neles presentes. Ou seja, múltiplas tarefas podem ser chamadas pelo *Host* para serem executadas simultaneamente pelos núcleos de processamento ou coprocessadores presentes no *Host* ou nos computadores remotos. Por exemplo, a pedido do programa no *Host* uma tarefa pode ser executada pela GPGPU de um computador remoto.

Para a compreensão da solução proposta são utilizadas as seguintes convenções:

***Host*:** Nó mestre que possuí a aplicação principal que realiza as chamadas para as execuções das tarefas nos *Guests*;

***Guest*:** Refere-se aos computadores ou dispositivos, conectados ao *Host* que receberão a solicitação para as execuções das tarefas.

Para efetuar a verificação de funcionalidade e análises do comportamento e de desempenho do modelo de programação proposto implementou-se um protótipo da API RTE.

1.1.1 Objetivos específicos

- Desenvolver um modelo de programação *Task/Thread Level Paralelism* (TLP) baseado em *Bag Of Tasks* (BoT) para sistemas distribuídos com arquiteturas heterogêneas;
- Definir a estratégia para usar de maneira direta os recursos heterogêneos como CPU e aceleradores de *hardware*, através de uma sintaxe simples ao desenvolvedor de aplicações em HPC;
- Implementar uma ferramenta de programação para validar o modelo proposto.

1.2 Justificativa

O uso de recursos heterogêneos é uma das técnicas utilizadas para melhorar o desempenho em HPC (AGOSTA *et al.*, 2018; CARPENTER *et al.*, 2022). A sua utilização requer adaptações dos métodos de programação existentes para cada tecnologia

(STRINGHINI; GONÇALVES; GOLDMAN, 2012). Algumas soluções oferecem diretivas de programação `#pragma` juntamente com bibliotecas de funções como as propostas pela HMPP (DOLBEAU; BIHAN; BODIN, 2007) e pelo OmpSs–2 (OmpSs–2, 2020) semelhantes à sintaxe oferecida pela OpenMP (OpenMP, 2021). Outras porém, apresentam bibliotecas mas com uma sintaxe própria como é o caso de QUARK (YARKHAN; KURZAK; DONGARRA, 2011), StarPU (INRIA, 2020) e OneAPI (Intel, 2020). Essas soluções apresentam conjuntos de funções individualizadas para cada recurso.

Outro ponto importante é a integração com sistemas distribuídos. Alguns casos disponibilizam conjuntos de funções que fazem acesso à API MPI, como OmpSs–2 e OneAPI. Essas soluções necessitam da instalação dessa API.

Este trabalho apresenta como contribuição principal um modelo de programação orientado a tarefas de forma a obter acesso aos recursos computacionais híbridos disponíveis. Sejam eles uma CPU e/ou aceleradores de *hardware*, num mesmo *Host* e/ou em cada *Guest*.

A API RTE foi desenvolvida com base no modelo de programação proposto como uma ferramenta que visa possibilitar escrever aplicações HPC com acesso aos recursos disponíveis de uma maneira simples e direta através de uma sintaxe simples. Sua implementação foi feita com recursos da linguagem de programação C, (LANGSAM; AUGENSTEIN; TENENBAUM, 1990; RITCHIE; KERNIGHAN; LESK, 1988), devido a sua ampla utilização no desenvolvimento de soluções para Computação de Alto Desempenho. E para a interconexão do sistemas distribuídos foi baseada na *Application Programming Interface* (API) POSIX *Sockets* (STEVENS; FENNER; RUDOFF, 2004; COMER; STEVENS, 1999). A intenção é não apresentar perda no desempenho nas execuções em comparação às soluções aqui apesentadas.

1.3 Método de pesquisa

O método utilizado segue a proposta de Wazlawick (2014), realizando as seguintes etapas:

1. **Pesquisa científica no tema proposto**: Iniciando com uma busca na literatura

científica como os anais dos principais congressos e artigos em revistas relevantes da área de HPC realizada através das palavras chaves como paralelização de tarefas, aceleradores de *hardware*, GPGPU, entre outras, para que pudesse ser definido o escopo desse trabalho e também seus conceitos e interações com os trabalhos relacionados;

2. **Definições da Solução Proposta**: Após análise dos resultados obtidos, foi realizada a definição do escopo da proposta. Definiu-se como estratégia um sistema distribuído que pudesse de maneira simples e direta utilizar recursos em arquiteturas heterogêneas e ao mesmo tempo em sistemas distribuídos;
3. **Implementação do modelo de programação e processamento de aplicações paralelas baseada em tarefas remotas assíncronas**: Desenvolvimento de um protótipo para gerenciamento de tarefas paralelas em cada *Guest*;
4. **Adaptação do protótipo para integração com SMP no *Host***: Para essa etapa há a necessidade de adequação do código para que o sistema possa ter melhor aproveitamento com recursos SMPs nos múltiplos núcleos de sua CPU;
5. **Adaptação do protótipo para integração com aceleradores de *hardware***: Há a necessidade de adaptar o protótipo às arquiteturas dos aceleradores de *hardware* MIC e GPGPU para melhorar o aproveitamento desses recursos heterogêneos;
6. **Definição dos testes do modelo de programação**: Assim que completou-se o desenvolvimento do protótipo, foram definidos os testes para avaliar a sua funcionalidade e desempenho e a comparação com um o modelo de programação comumente utilizada em sistemas distribuídos, a API MPI;
7. **Execução dos testes e análise dos resultados e da viabilidade da solução proposta**: Por fim, realizou-se a avaliação, analisando os tempos obtidos em cada teste juntamente com as comparações com o uso de MPI.

1.4 Organização do trabalho

Juntamente com a apresentação do problema, objetivo, justificativa e o método da pesquisa aqui mostrados essa dissertação está organizada com os seguintes capítulos:

Capítulo 2: Apresenta os conceitos adotados para dar melhor entendimento na descrição e no escopo deste trabalho;

Capítulo 3: Traz um resumo dos trabalhos que estabelecem vínculos à essa pesquisa;

Capítulo 4: Mostra a proposta de uma solução para programação e processamento de aplicações paralelas baseada em tarefas remotas assíncronas e o detalhamento de sua arquitetura;

Capítulo 5: São apresentados os aspectos da implementação do protótipo baseada em BoT conforme a proposta no Capítulo 4;

Capítulo 6: Apresenta os testes efetuados na avaliação do modelo de programação proposto, bem como os resultados e análises experimentais mostrando a viabilidade de sua utilização;

Capítulo 7: Descreve as considerações finais dessa pesquisa, suas contribuições e propõe sugestões de trabalhos futuros.

2 CONCEITOS

Este capítulo apresenta os conceitos adotados para dar melhor entendimento na descrição e no escopo do trabalho proposto.

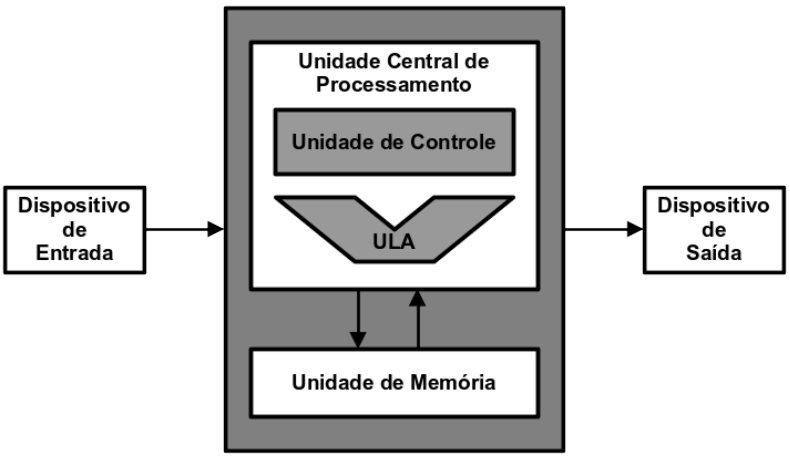
2.1 Arquiteturas em computação de alto desempenho

O termo HPC foi originalmente usado para descrever os primeiros supercomputadores (BRAZELL; BETTERSWORTH, 2008). Com o crescimento da escala das aplicações para computação de alto-desempenho, a sua definição foi modificada para também incluir qualquer sistema com a combinação de características como capacidade de aceleração computacional, alto *throughput* de dados e a habilidade de agregar o poder da computação distribuída.

Uma classificação de arquiteturas de computadores amplamente usada é a proposta por Flynn (FLYNN, 1972). Essa taxonomia baseia-se nos conceitos de fluxo de instruções e fluxo de dados identificando as seguintes categorias:

***Single Instruction, Single Data* (SISD):** É a mais simples das arquiteturas, onde um processador executa uma única instrução em um único fluxo de dados. Essa arquitetura pode ser representada pelo Modelo de von Neumann como mostra a Figura 2.1;

Figura 2.1: Modelo de von Neumann.



Fonte: Próprio Autor

***Single Instruction, Multiple Data* (SIMD):** Essa categoria é capaz de executar uma única instrução em múltiplos dados. Isto é possível pois sua arquitetura é composta por diversos processadores, ou *cores*, com compartilhamento de níveis de memória;

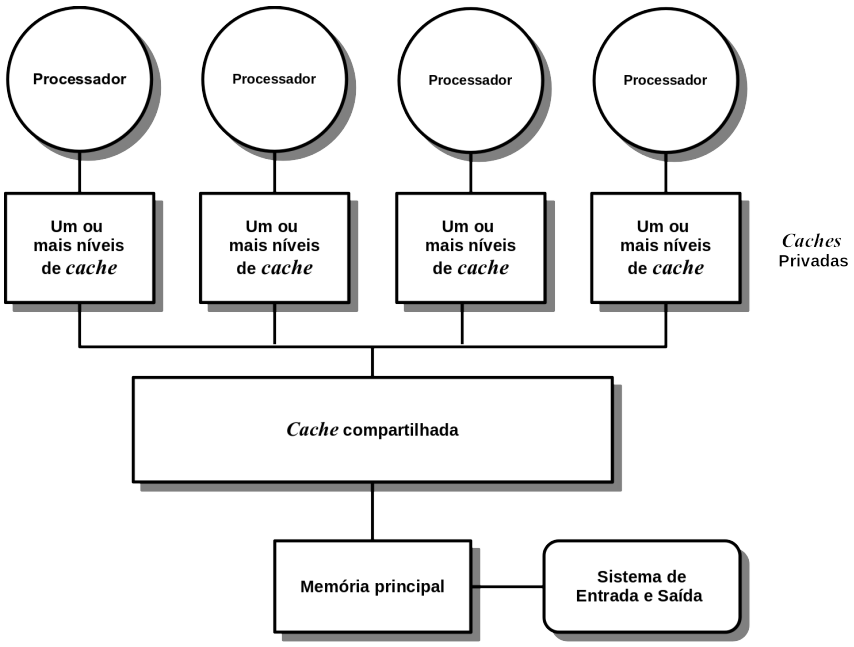
***Multiple Instruction, Single Data* (MISD):** Essa arquitetura é meramente conceitual, o próprio Flynn descarta a utilização e a existência. Ela consiste em diversos processadores executando suas instruções em um único fluxo de dados.

***Multiple Instruction, Multiple Data* (MIMD):** É a arquitetura mais complexa, pois são processadas múltiplas instruções em múltiplos dados.

2.1.1 Multiprocessamento simétrico (SMP)

O *Simetric Multiprocess* (SMP) é um sistema composto por conjunto de processadores fortemente acoplados que compartilham a memória principal através de um barramento de interconexão (HENNESSY; PATTERSON, 2017). Cada processador possui o mesmo custo de acesso à memória e aos dispositivos de entrada e saída. A Figura 2.2 apresenta a representação de uma arquitetura SMP.

Figura 2.2: Estrutura básica do SMP centralizado em processadores *multicores*.

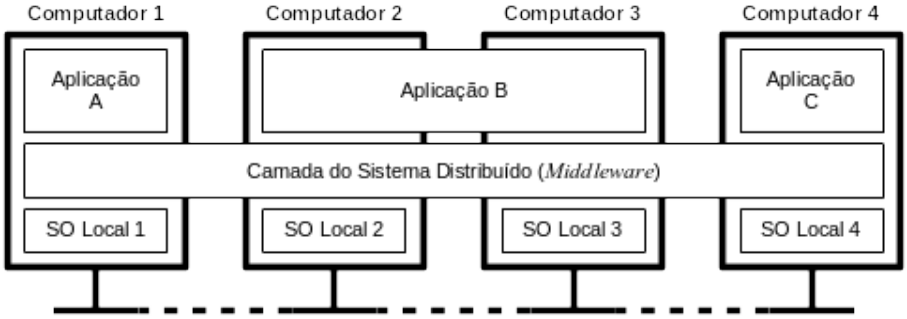


Fonte: Adaptado de Hennessy e Patterson (2017)

2.1.2 Sistemas distribuídos

É uma arquitetura com grande escalabilidade e com a possibilidade de agregação de computadores comuns interligados numa rede (TANENBAUM; STEEN, 2017). São ambientes constituídos tanto por máquinas homogêneas quanto por heterogêneas. Na Figura 2.3 observa-se as características de um sistema distribuído.

Figura 2.3: Organização de um sistema distribuído



Fonte: Adaptado de Tanenbaum e Steen (2017)

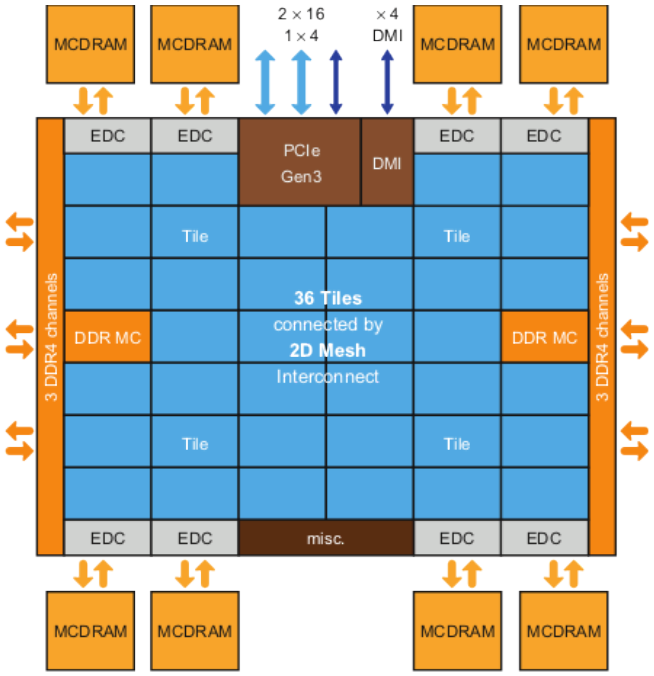
2.1.3 Computação heterogênea

A computação heterogênea é definida pelo uso de diferentes arquiteturas de processadores em um mesmo nó computacional (STRINGHINI; GONÇALVES; GOLDMAN, 2012). Normalmente são utilizados aceleradores de *hardware* como GPGPUs, FPGAs e atualmente processadores de baixo custo e consumo de energia como *Advanced RISC Machine*s (ARMs) (SATO, 2020).

2.1.3.1 Intel® XEON Phi™

Essa arquitetura destina-se a soluções de HPC voltada a aplicações que fazem uso massivo de processamento paralelo (JEFFERS; REINDERS; SODANI, 2016). Nessa arquitetura é possível explorar paralelismo com *thread* e SIMD. A Figura 2.4 mostra a arquitetura do acelerador de *hardware* Intel® XEON Phi™.

Figura 2.4: Arquitetura Intel® XEON Phi™.

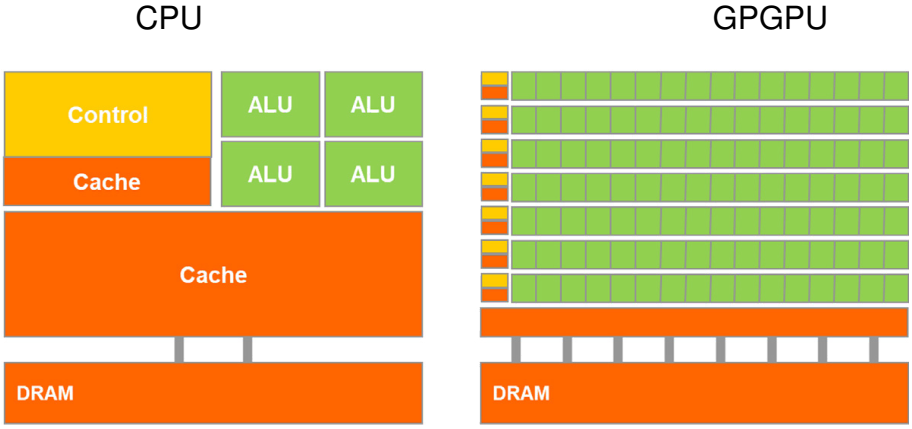


Fonte: Jeffers, Reinders e Sodani (2016)

2.1.3.2 GPGPUs

As GPGPUs são aceleradores que atuam junto às CPUs (WEN-MEI; KIRK; HAJJ, 2022), suas capacidades de processamento de dados são muito maiores do que as CPUs, a Figura 2.5 apresenta as diferenças entre as duas arquiteturas. A GPGPU explora o paralelismo do tipo SIMD.

Figura 2.5: Diferença entre as arquiteturas CPUs e GPGPUs.



Fonte: Wen-Mei, Kirk e Hajj (2022)

2.2 Modelos de programação em computação de alto desempenho

Os modelos de programação HPC são abstrações de suas arquiteturas que expressam algoritmos na composição em seus programas (BARNEY *et al.*, 2017a). Um modelo de programação pode utilizar uma biblioteca invocada a partir de linguagens sequenciais.

2.2.1 Memória compartilhada

Em SMP processos paralelos compartilham um espaço de endereço global no qual eles leem e gravam de forma assíncrona. Esse comportamento pode levar a uma condição de corrida, *race condition*, em que a utilização de bloqueios, semáforos e monitores, ajuda a evitar. Aqui utiliza-se o paralelismo de *threads* que podem ser feitas com PThreads POSIXs *Threads* (BARNEY, 2017b), ou OpenMP (OpenMP, 2021).

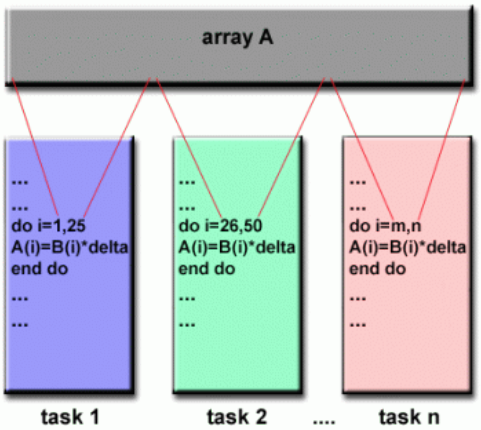
2.2.2 Passagem de mensagens

O modelo de passagem de mensagens em processamento paralelo provê a troca de dados passando mensagens entre nós ou processos (Message Passing Interface Forum, 2021). A APIs MPI possui várias funções para troca de mensagens e é responsável pela comunicação e sincronização em um *cluster* paralelo. As comunicações podem ser síncronas, em que o receptor deve estar pronto antes do envio da mensagem, ou assíncronas. Um programa MPI consiste em vários processos autônomos, executando seu próprio código em uma arquitetura MIMD.

2.2.3 Paralelismo de dados

Um modelo paralelo de dados concentra-se na execução de operações em um conjunto de dados (BARNEY *et al.*, 2017a). Um conjunto de tarefas operará com esses dados, mas independentemente em partições separadas. Esse modelo de programação explora arquitetura SIMD. A Figura 2.6 apresenta o modelo de paralelismo de dados.

Figura 2.6: Modelo de paralelismo de Dados.

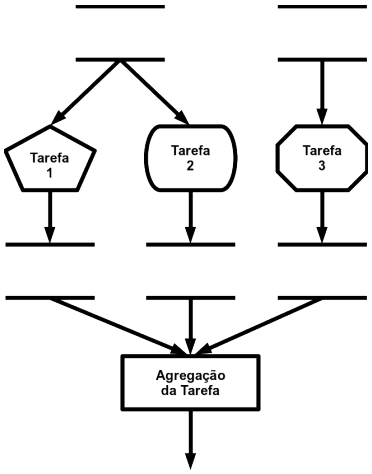


Fonte: Barney *et al.* (2017a)

2.2.4 Paralelismo de tarefas

Ao contrário do paralelismo de dados, enfatiza a distribuição de processos ou funções entre servidores remotos (SCHMIDT *et al.*, 2017). A Figura 2.7 apresenta o modelo de paralelismo de tarefas em que os processos possuem comportamentos distintos. Uma forma de explorar esse modelo de programação paralela é através da *Bag Of Tasks* (BoT), que são tarefas independentes, sem nenhuma interface de comunicação entre si, executadas fora da ordem (STAVRINIDES; KARATZA, 2021).

Figura 2.7: Modelo de paralelismo de tarefas.



Fonte: Adaptado de Tröger e Feinbube (2014)

Este trabalho propõe a execução paralela de tarefas em processadores remotos, coprocessadores ou núcleos do próprio *Host* para melhorar o desempenho.

3 TRABALHOS RELACIONADOS

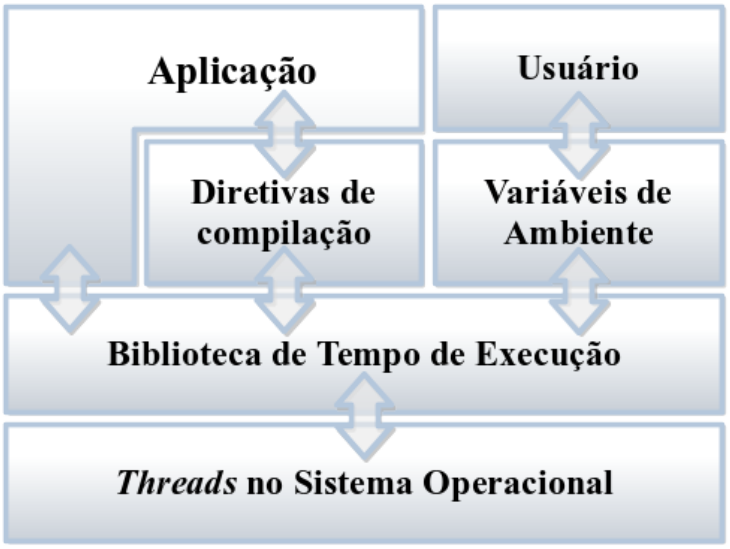
A busca por uma solução híbrida para execução em arquiteturas tão diversificadas torna-se um desafio à ciência da computação. Existem várias técnicas com algumas eficiências tanto no desenvolvimento das aplicações quanto nas execuções. A ideia é explorar as vantagens de cada recurso do processamento para conseguir um melhor desempenho.

Neste capítulo são apresentados alguns trabalhos relacionados à pesquisa proposta.

3.1 OpenMP

OpenMP é uma API que apresenta funcionalidades para desenvolvimento de aplicações com SMP com o paralelismo de nível de tarefa / *threads* (TLP), *Task/Thread Level Paralelism*, através de suas diretivas de compilação como `#pragma omp` e bibliotecas com conjunto de funções, constantes e variáveis de ambiente utilizado em linguagens como C/C++ e Fortran (CHAPMAN; JOST; PAS, 2007; OpenMP, 2021). A Figura 3.1 apresenta a arquitetura OpenMP.

Figura 3.1: Arquitetura OpenMP.



Fonte: Adaptado de Süleyman e AKGÜN (2016)

Um recurso oferecido pela API OpenMP são as chamadas de funções *Offloading* em dispositivos conectados, tais como os aceleradores de *hardware* GPGPU e MIC, numa única placa mãe através da diretiva de compilação `#pragma omp target` que especifica clausulas como: `device` que cria um ambiente num dispositivo, `map` que especifica uma área de memória mapeada no dispositivo, e `data` que especifica um ambiente de dados no dispositivo. Essa funcionalidade está presente desde a versão 4.0, inicialmente disponível somente para o acelerador de *hardware* Intel® XEON Phi™ (CRAMER *et al.*, 2012). As versões mais recentes 4.5, 5.0, 5.1 e 5.2 disponibilizam chamadas *Offloading* também para GPGPUs (LI, 2016; BAK *et al.*, 2022).

A Figura 3.2 apresenta um exemplo simples de um trecho de programa OpenMP com acesso a aceleradores de *hardware*.

Figura 3.2: Exemplo de um trecho de código OpenMP com acesso à GPGPU.

```
1  #pragma omp target data map (to: pA[0:N*N],pB[0:N*N]) map (tofrom: pC[0:N*N])
2  #pragma omp target
3  #pragma omp teams distribute parallel for collapse(2) private(i,j,k)
4  for(i=0;i<N;i++)
5  {
6    for(j=0;j<N;j++)
7    {
8        for(k=0;k<N;k++)
9        {
10           pC(i,j)+=pA(i,k)*pB(k,j);
11       }
12   }
13 }
```

Fonte: Nitsure, Shrivastava e Dsouza (2019)

3.2 QUARK

O *QUeuing And Runtime for Kernels* (QUARK) é uma API simples superescalar dinâmica orientada a dados em tempo de execução para inserção de tarefa serial e permite a colocação de tarefas de baixo nível (YARKHAN, 2012; HAIDAR *et al.*, 2014).

Cada arquitetura é projetada para diferentes capacidades de processamento. Para ter uma melhor eficiência é explorado o paralelismo em cada uma delas. A GPGPU tem um maior grau de paralelismo em comparação à capacidade das CPUs e Intel® MIC™ possui capacidade intermediária entre a CPU e a GPGPU.

O controle de fluxo de dados e execução paralela em sistemas híbridos, bem como

o gerenciamento das cargas a cada recurso especifico faz-se através de um ambiente de execução leve, isso alivia no desenvolvimento eliminando a complexidade em lidar com várias bibliotecas de programação. QUARK disponibiliza um conjunto de variáveis de ambiente e de flags para especificar o ambiente em que será executada a tarefa, seja uma CPU, MIC ou GPGPU.

A Figura 3.3 apresenta um exemplo simples de um programa com o uso da API QUARK.

Figura 3.3: Exemplo de um programa básico com o uso da API QUARK

```
1  #include "quark.h"
2
3  /* This task unpacks and prints a string. At each call, the idx
4     character is replaced with an underscore, changing the string. */
5  void hello_world_task( Quark *quark ) {
6       int idx; char *str;
7       quark_unpack_args_2( quark, idx, str );
8       printf( "%s\n", str );
9       str[idx] = ' ';
10 }
11
12 /* A simple variation of "Hello World" */
13 main() {
14     int idx;
15     char str[] = "Hello World";
16     Quark *quark = QUARK_New( 2 );
17     for( idx = 0; idx < strlen( str ); idx++ )
18        QUARK_Insert_Task( quark, hello_world_task, NULL,
19             sizeof( int ), &idx, VALUE,
20             strlen( str ) * sizeof( char ), str, INOUT,
21             0 );
22     QUARK_Delete( quark );
23 }
```

Fonte: Yarkhan, Kurzak e Dongarra (2011)

3.3 HMPP

O *Hybrid Multicore Parallel Programming (HMPP)* é um padrão para programação de computação heterogênea composta por CPUs e aceleradores de *hardware* como GPGPUs, com um conjunto de diretivas de compilação semelhante ao OpenMP para o uso em linguagens como C e Fortran (DOLBEAU; BIHAN; BODIN, 2007; CONSORTIUM *et al.*, 2012).

O HMPP é baseado no conceito de funções que podem ser executadas em um acelerador de *hardware*, os `codelets`. As diretivas necessárias para implementá-las são `codelet` para indicar a implementação de uma função e `callsite` para indicar a

invocação para os aceleradores de *hardware*.

Em tempo de execução, o HMPP se encarrega de verificar os aceleradores de *hardware*. Ao indicar um `codelets` em um acelerador, verifica a disponibilidade do dispositivo e se há `codelets` correspondente à biblioteca compartilhada, então é feita a carga como um "*plug-in*", senão a versão nativa é executada na CPU do *Host* ou em uma *thread Guest* (ANDIÓN *et al.*, 2016).

No tempo de execução, o HMPP cuida de descobrir os aceleradores de *hardware* disponíveis. Quando um `codelet` é indicado para ser executado em um acelerador de *hardware*, se o dispositivo estiver disponível e se o `codelet` correspondente da biblioteca compartilhada estiver presente, o HMPP carrega apenas como um *plug-in* de *software*. Caso contrário, a versão nativa é executada na CPU do *host* ou em um *thread* de trabalho. A Figura 3.4 apresenta o exemplo de um programa com as diretivas do HMPP.

Figura 3.4: Exemplo de um programa com HMPP.

```
1  #pragma hmpp simple codelet, args[outv].io=inout, target=CUDA
2  static void matvec(int sn, int sm,
3                     float inv[sm], float inm[sn][sm],
4                     float *outv)
5  {
6      int i, j;
7      for (i = 0 ; i < sm ; i++) {
8        float temp = outv[i];
9        for (j = 0 ; j < sn ; j++) {
10         temp += inv[j] * inm[i][ j];
11       }
12       outv[i] = temp;
13 }
14
15 int main(int argc, char **argv) {
16     int n;
17   ........
18 #pragma hmpp simple callsite, args[outv].size={n} matvec(n, m, myinc, inm, myoutv);]
19   ........
20 }
```

Fonte: Bihan *et al.* (2009)

3.4 TREES

A *Task Runtime with Explicit Epoch Synchronization (TREES)*, (HECHTMAN; HILTON; SORIN, 2016), implementa uma ferramenta para desenvolvimento em sistemas híbridos, compostos por CPUs e GPGPUs com um sistema em tempo de execução

de tarefas paralelas baseada em *Task Vector Machine* (TVM) que fornece um modelo adaptativo de cálculo para o paralelismo de tarefas num sistema baseado em GPGPU. Trabalha com o princípio do *Cilk's "Work-First"* baseados em *Fork/Join*, mais adequado para as CPUs.

TREES permite utilizar sistemas híbridos compostos por CPU e GPGPU. O código fonte possuí uma sintaxe semelhante à C++, com as primitivas de paralelização de tarefas fornecidas pelo modelo de execução TVM. Seu compilador traduz o código em OpenCL e inclui o tempo de execução. A Figura 3.5 apresenta um exemplo de código com a sintaxe apresentada.

Figura 3.5: Exemplo de um código TREES.

```
1  void visit(node) {
2       // do visit
3  }
4  task void preorder(node) {
5       if(node != nil) {
6          visit (node);
7          fork preorder(node.right);
8          fork preorder(node.left);
9       }
10 }
11 task void postorder(node) {
12      if (node != nil) {
13         fork postorder(node.right);
14         fork postorder(node.left);
15         join visitAfter(node);
16      }
17 }
18 task void visitAfter (node) {
19      visit (node); }
```

Fonte: Hechtman, Hilton e Sorin (2016)

3.5 StarPU

StarPU é um sistema em tempo de execução unificado para arquiteturas heterogêneas *multicore* compostas por CPUs e aceleradores de *hardware* com suporte para linguagens como C/C++ e Fortran (AUGONNET *et al.*, 2009; AUGONNET, 2011; INRIA, 2020). Foi desenvolvido pelo *Institut National De Recherche En Sciences Et Technologies Du Numérique (INRIA)* na *Université de Bordeaux*.

O modelo de fluxo de tarefa sequencial (STF), *Sequential Task Flow* (AGULLO *et al.*, 2016), é utilizado pela aplicação StarPU para submeter sequencialmente as tarefas às execuções, e estas são escalonadas dinamicamente em tempo de execução. A

implementação de tarefas é dada através dos *codlets*. Cada *codlet* especifica um *kernel* para uma arquitetura como CPU ou um acelerador de *hardware* e seu tipo de acesso, leitura e/ou escrita (INRIA, 2020).

A Figura 3.6 apresenta uma implementação simples de um programa utilizando a sintaxe proposta pelo StarPU.

Figura 3.6: Exemplo de um código StarPU.

```
1  #include <stdlib.h>
2  #include <limits.h>
3  #include <starpu.h>
4
5  void func_cpu(void *buffers[], void *args)
6  {
7      printf("Hello World!\n");
8  }
9
10 struct starpu_codelet codelet_world =
11 {
12     .cpu_funcs = { func_cpu },
13     .nbuffers = 0,
14     .name = "hello_world",
15 };
16
17 int main(){
18     starpu_init(NULL);
19     starpu_topology_print(stdout);
20     starpu_task_insert(&codelet_world, 0);
21     starpu_task_wait_for_all();
22     starpu_shutdown();
23 }
```

Fonte: Nesi *et al.* (2020)

3.6 OmpSs–2

OmpSs–2 é um modelo de programação baseado em tarefas composto por diretivas e rotinas para utilização em conjunto com programação com linguagens de alto nível como C/C++ e/ou Fortran por exemplo para desenvolvimento de aplicações concorrentes. É a segunda geração do OmpSs desenvolvido no *Barcelona Supercomputing Center* (BSC) (OmpSs–2, 2020). Seu nome é a junção dos nomes dos modelos de programação OpenMP (OpenMP, 2021) e do ambiente de execução StarSs (PLANAS *et al.*, 2009). Este ambiente é responsável pela paralelização automática em tempo de execução da tarefas que podem ser paralelizadas.

A implementação é semelhante ao OpenMP com o uso de diretivas `#pragma`. É baseada na biblioteca em tempo de execução Nanos6 (ROBERT, 2019; MUÑOZ *et*

al., 2021) que é responsável por fornecer o gerenciamento das tarefas na aplicação do usuário, bem como componentes para gerenciamento de recursos com suporte à aceleradores de *hardware* GPGPUs proporcionando a heterogeneidade na programação; e no compilador fonte-a-fonte Mercurium (FERRER *et al.*, 2011; BOSCH *et al.*, 2018) que traduz diretivas de alto nível na versão paralela com chamadas de funções da biblioteca Nanos6.

OmpSs–2 é baseado na execução de tarefas síncronas ou assíncronas que podem conter dependências de dados, permitindo que o usuário determine o fluxo dos dados do programa. A Figura 3.7 apresenta um *kernel* otimizado com a sintaxe OmpSs–2.

Figura 3.7: Exemplo de um *kernel* OmpSs–2 otimizado.

```
1  void matmul(double *A, const double *B, const double *C, int dim, int ts)
2  {
3       int rowsPerNode = dim/nanos6_get_num_cluster_nodes();
4
5       for( int i = 0; i < dim; i += rowsPerNode ) {
6          #pragmaosstasklabel("weakmatvec")\
7             weakin(A[i * dim;rowsPerNode * dim]) weakin(B[0; dim * dim])\
8             weakout(C[i * dim; rowsPerNode * dim])
9             {
10                #pragma oss task for label("taskformatvec")\
11                   in(A[i * dim; rowsPerNode * dim]) in(B[0; dim * dim])\
12                   out(C[i * dim; rowsPerNode * dim])
13                for( intj = i; j < i+rowsPerNode; j += ts ) {
14                   cblas_dgemm( CblasRowMajor, CblasNoTrans, CblasNoTrans,
15                   ts, dim, dim, 1.0, &A[j * dim], dim,
16                   B,dim, 0.0, &C[j * dim], dim);
17                }
18             }
19       }
20 }
```

Fonte: Mena *et al.* (2022)

3.7 oneAPI

A oneAPI™ é uma interface que tem como finalidade simplificar o modelo de programação para diversas arquiteturas como CPUs, *Field Programmable Gate Arrays*s (FPGAs) e aceleradores de *hardware*, tais como GPGPUs. Esta API foi desenvolvida pela Intel (Intel, 2020).

A sua implementação é baseada em uma nova linguagem de programação projetada para dar suporte ao paralelismo de dados denominada Data Parallel C++ (DPC++) (REINDERS *et al.*, 2021; ASHBAUGH *et al.*, 2020). Esta linguagem oferece

uma forma de codificação mais simples em comparação à outras ferramentas, como OpenCL e CUDA. Ela é baseada em C++ com extensão para SYCL™ (KERYELL; ROVATSOU; HOWES, 2019).

Esta nova linguagem permite executar operações paralelas entre os diversos dispositivos, GPGPUs ou FPGAs, e o *Host* CPU. O *Host* solicita a execução de uma operação em um dispositivo e continua sua execução, não permanecendo em espera. Desse modo, operações nessas arquiteturas são executadas simultaneamente.

A OneAPI especifica os modelos de **Plataforma**, de **Execução**, de **Memória** e de **Kernel**, baseados em SYCL, para que o desenvolvedor possa utilizar um ou mais recursos de processamento como CPUs e/ou aceleradores de *hardware*.

A Figura 3.8 apresenta um trecho de código com a sintaxe do OneAPI.

Figura 3.8: Exemplo de um trecho de código OneAPI.

```
1  int blocks = N / 1024;
2
3  float* dev_v;
4
5  /* DPC++ */
6  void kernel(float *v, cl::sycl::nd_item<3> item_ct1)
7  {
8       int idx = item_ct1.get_group(0) * item_ct1.get_local_range().get(0) + item_ct1.
            get_local_id(0);
9
10      v[idx] = cl::sycl::sqrt(idx * 1.0);
11 }
12
13 *((void **)&dev_v) = cl::sycl::malloc_device(...);
14         dpct::get_default_queue_wait().submit([&](cl::sycl::handler &cgh) {
15               auto global_rng = cl::sycl::range<3>(blocks,1,1) * cl::sycl::range<3>(1024, 1,
                       1);
16               auto local_rng = cl::sycl::range<3>(1024, 1, 1);
17
18               cgh.parallel_for<dpct_kernel_name<class kernel_e321fab>>(cl::sycl::nd_range
                       <3>(...), [=](cl::sycl::nd_item<3> item_ct1) {
19                   kernel(dev_v, item_ct1);
20               });
21         }
22 }
23 cl::sycl::free(dev_v, ...);
```

Fonte: Adaptado de Christgau e Steinke (2020)

3.8 Considerações sobre os trabalhos apresentados

Os trabalhos apresentados têm suas contribuições para o desenvolvimento de aplicações paralelas em um computador com CPU e aceleradores de *hardware*. Alguns

apresentam soluções voltadas somente para CPU e GPGPUs e provêm paralelismo de tarefas. Todas as soluções apresentadas visam oferecer uma interface única e fácil de ser aplicada na programação de aplicações com múltiplas chamadas de tarefas a serem executadas nos dispositivos heterogêneos.

A proposta deste trabalho é apresentar ao desenvolvedor de aplicações paralelas um conjunto de soluções voltada para paralelismo de tarefas com uma única codificação independente de arquitetura.

As soluções apresentadas nos trabalhos HMPP e OmpSs–2 se assemelham muito à sintaxe de OpenMP pelo uso das diretivas de programação. QUARK, TREES, StarPU e OneAPI apresentam bibliotecas com conjunto funções para acesso aos dispositivos heterogêneos com uma sintaxe própria, assim como é a proposta pela RTE. As soluções aqui apresentadas disponibilizam formas bem particulares e distintas de acesso às arquiteturas heterogêneas. RTE tem a finalidade de facilitar o uso de tais recursos através de uma sintaxe simples e unificada independente da arquitetura seja ela CPU ou acelerador de *hardware*.

A Tabela 3.1 apresenta um comparativo entre as técnicas aqui apresentadas com a proposta desse trabalho. Como pode observar todas as soluções possuem as mesmas características como tarefas assíncronas, processamento simétrico nas CPUs e nos aceleradores de *hardware* e a execução de tarefas em sistemas híbridos. Porém somente QUARK, StarPU, OmpSs–2 e oneAPI apresentam recursos para acesso à sistemas distribuídos, mas essa solução é disponibilizada através de funções que usam internamente a API MPI. Já RTE apresenta acesso direto aos recursos remotos, não necessitando de uma ferramenta a mais de *software* instalado.

Tabela 3.1: Comparação entre RTE e os trabalhos relacionados.

	Tarefas assíncronas	SMPs nas CPUs	SMPs nos aceleradores	Processamento heterogêneo	Processamento distribuído	Sem uso de ferramentas externas
OpenMP	✓	✓	✓	✓		
QUARK	✓	✓	✓	✓	✓	
HMPP	✓	✓	✓	✓		
TREES	✓	✓	✓	✓		
StarPU	✓	✓	✓	✓	✓	
OmpSs–2	✓	✓	✓	✓	✓	
oneAPI	✓	✓	✓	✓	✓	
RTE	✓	✓	✓	✓	✓	✓

Fonte: Próprio Autor

4 SISTEMA DE PROGRAMAÇÃO PARALELA PARA TAREFAS ASSÍNCRONAS

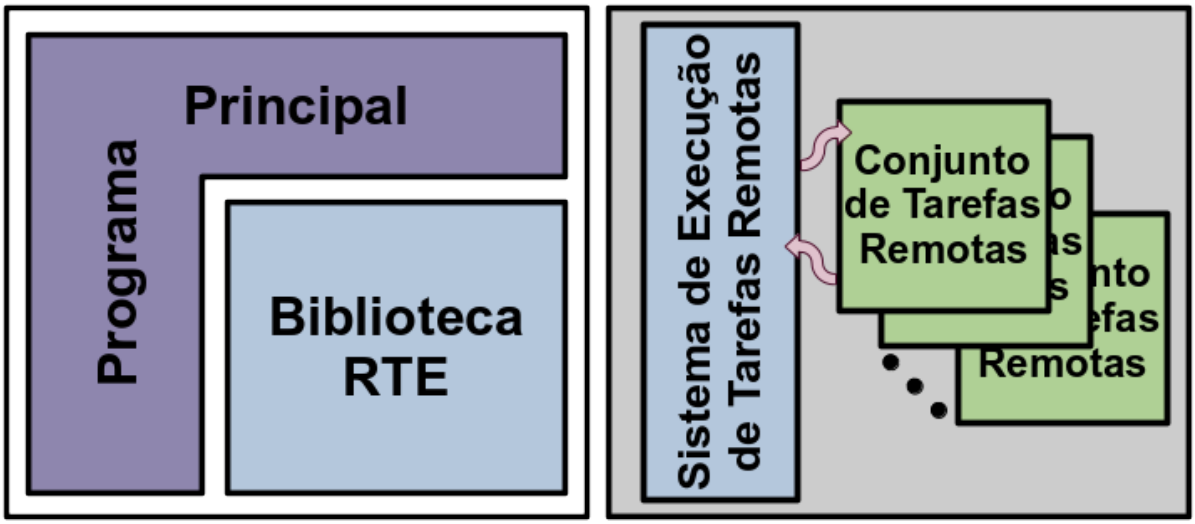
Este capítulo apresenta o sistema de programação e processamento de aplicações paralelas baseado em tarefas remotas assíncronas *Remote Tasks Execute* (RTE). Este sistema provê a execução de tarefas em paralelo em um sistema distribuído com arquiteturas de *hardware* heterogêneas. As tarefas são assíncronas e não bloqueantes, desta forma múltiplas tarefas podem ser executadas simultaneamente. As tarefas são executadas em cada *Guest* através de chamadas efetuadas por uma aplicação em um *Host*.

O computador principal, ou *Host*, executa a aplicação que efetua chamadas de tarefas para cada processador remoto, ou *Guest*. Cada *Guest* pode ser um computador com processadores multicore, um computador na nuvem ou aceleradores de *hardware* como uma GPGPU ou um Intel® XEON Phi™. O sistema transfere os dados de respostas para os respectivos endereços das variáveis de retorno, assim que recebidos.

A Figura 4.1 mostra a organização desse sistema. Uma aplicação paralela RTE é composta por um **Programa Principal** ligado à **Biblioteca RTE** e por um ou mais **Conjuntos de Tarefas Remotas**, cada conjunto contém as tarefas a serem processadas. A Figura 4.1a apresenta a organização em um *Host* onde devem estar o **Programa Principal** e a **Biblioteca RTE**, essa biblioteca provê a interface de comunicação com os **Conjuntos de Tarefas Remotas** nos *Guests*. A Figura 4.1b apresenta a organização em um *Guest* em que o **Sistema de Execução de Tarefas Remotas** é responsável pela carga dos arquivos contendo os **Conjuntos de Tarefas Remotas** para a execução das tarefas solicitadas pela aplicação no *Host*.

O programa principal em conjunto com a interface RTE, pode utilizar as diretivas de programação paralela oferecidas pelo OpenMP® (OpenMP, 2021), inclusive internamente nas tarefas. Se o *Guest* for um computador ou um coprocessador a ser utilizado como um computador remoto, como por exemplo um Intel® XEON Phi™, estas diretivas podem ser utilizadas internamente nas tarefas.

Figura 4.1: Organização do sistema.



(a) Organização do sistema no *Host*. (b) Organização do sistema no *Guest*.

Fonte: Próprio Autor

Restrições do Sistema RTE: O sistema não permite a execução aninhada de tarefas, ou seja, uma tarefa não pode ser chamada dentro de outra tarefa.

A organização deste capítulo está da seguinte forma: A Seção 4.1 descreve o modelo e a interface de programação propostos para execução de tarefas paralelas em ambientes com arquiteturas heterogêneas, com exemplos de compilação e execução de uma aplicação com o uso da ferramenta proposta. A Seção 4.2 mostra em detalhes a arquitetura desta ferramenta de programação.

4.1 Interface RTE e programação

Esta seção apresenta a interface e o modelo de programação RTE. Este modelo disponibiliza ao desenvolvedor acesso aos *Guests* necessário para execução de sua aplicação paralela e distribuída. Em um *Host* o usuário executa a aplicação paralela que faz as requisições de chamadas de tarefas em um ou mais *Guests*. Cada *Guest* possui um ou mais conjuntos de tarefas remotas.

4.1.1 Interface RTE

A execução das tarefas remotas é provida por um processo que executa o programa **rted**, que deve estar presente e iniciado em cada nó remoto. Para integrar o programa desenvolvido pelo usuário com esse processo, RTE provê uma biblioteca, **librte**, contendo as funções necessárias para integração e execução das tarefas remotas. Essas funções serão apresentadas a seguir.

4.1.1.1 Inicialização

Para a execução de uma aplicação com RTE é necessário realizar os procedimentos de inicialização, tais como:

- Prover as conexões com os nós remotos;
- Inicializar as listas de controle:
  - Lista de tarefas a serem executadas remotamente;
  - Lista com os endereços das respostas de cada tarefa.

A função **rte_init**, apresentada na Figura 4.2, providencia esta inicialização. Esta função provê as conexões com os nós remotos, que estão listados no arquivo de configuração e inicializa o gerenciador de chamadas de tarefas remotas e o receptor dos dados de retorno. O parâmetro `rte_cfg_file` contém o nome do arquivo de configuração com as informações para conexão com cada *Guest*.

Figura 4.2: Sintaxe da função **rte_init**.

```
int rte_init(const char* rte_cfg_file)

Parâmetro:
    rte_cfg_file: nome do arquivo de configuração.
```

Fonte: Próprio Autor

O arquivo de configuração contém a lista dos *Guests* que serão utilizados na execução da aplicação. A sintaxe do arquivo contém as seguintes variáveis:

guests – Lista dos nós remotos que serão utilizados. E cada item da lista é composto pelos seguintes campos:

Address: Endereço de rede do *Guest*;

Port: Porta destinada para a aplicação de gerenciamento das execuções das tarefas no *Guest*.

A Figura 4.3 mostra o exemplo de um arquivo contendo dois *Guests*, cada um com o endereço IP específico, mas ambos com a mesma porta de rede.

Figura 4.3: Exemplo de um arquivo de configuração de uma aplicação RTE.

```
guests = (
    {
        Address = "10.0.0.1";
        Port = 5000;
    },
    {
        Address = "10.0.0.2";
        Port = 5000;
    }
);
```

Fonte: Próprio Autor

4.1.1.2 Chamada de tarefa

A tarefa é executada remotamente em um *Guest* específico, de forma assíncrona e não bloqueante. A função **Chamada de Tarefa** insere a tarefa especificada na lista de execução. O processo de gerenciamento de tarefas remotas retira um item e o envia para o *Guest* especificado para sua execução. Após a inserção de uma tarefa na lista inicializa-se o controle de sincronização da sua resposta e a sequência que

Figura 4.4: Sintaxe da função **rte_tsk_call**.

int rte_tsk_call(**const unsigned int** gst_id, **const unsigned int** dvc, **const char*** tsk_name, **const char*** tsk_set_name, **const char*** argin, **const size_t** sz_argin, **const size_t** sz_resp, **void*** resp)

Parâmetros:

gst_id: o identificador do dispositivo;
dvc: tipo do dispositivo chamador;
tsk_name: nome da tarefa;
tsk_set_name: nome da biblioteca de tarefas;
argin: cadeia de *bytes* contendo os valores dos argumentos da tarefa;
sz_argin: tamanho em *bytes* de `argin`;
sz_resp: tamanho em *bytes* do retorno da tarefa;
resp: ponteiro contendo o endereço da variável de retorno.

Fonte: Próprio Autor

fez a chamada retorna para o seu fluxo de execução. A Figura 4.4 apresenta a sintaxe da função **rte_tsk_call** que é responsável pela chamada de uma tarefa.

4.1.1.3 Sincronização das tarefas

Para utilizar o resultado de uma tarefa remota deve-se aguardar a sua conclusão e o seu retorno. Isto é feito através da sincronização do fluxo de controle da execução de cada tarefa especificada pelo seu identificador.

A Figura 4.5 apresenta a sintaxe da função **rte_tsk_sync**. Esta função controla a sincronização do fluxo de execução da tarefa especificada pelo identificador que foi iniciada pela função `rte_tsk_call` em sua requisição de chamada. A sequência que fez a chamada da função `rte_tsk_sync` permanece bloqueada até o término da execução da tarefa especificada.

Figura 4.5: Sintaxe da função **rte_tsk_sync**.

```
int rte_tsk_sync(int tsk_id)

Parâmetro:
    tsk_id: identificador da tarefa.
```

Fonte: Próprio Autor

4.1.1.4 Finalização

Para encerrar a execução de uma aplicação RTE, após o término de todas as tarefas remotas e seus respectivos retornos, é necessário finalizar a conexão com cada *Guest*, e também, limpar os conteúdos das variáveis de controle do sistema.

A função **rte_finalize**, mostrada na Figura 4.6, é responsável por essa finalização.

Figura 4.6: Sintaxe da função **rte_finalize**.

```
void rte_finalize()
```

Fonte: Próprio Autor

4.1.2 Programa com tarefas remotas

Uma aplicação é composta por um programa principal a ser executado no *Host* e pelos **Conjuntos de Tarefas Remotas** contendo as funções do programa que serão executadas em um *Guest*. Os arquivos contendo os conjuntos de tarefas devem estar presentes no *Guest*.

4.1.2.1 Programa principal

O programa principal é um conjunto de linhas de código fonte escrito pelo desenvolvedor da aplicação paralela contendo as chamadas de tarefas remotas que deverão ser executadas em cada *Guest*. A aplicação, ou programa executável, é a composição do programa principal ligado com a `librte`. Além da utilização das funções mencionadas na Seção 4.1.1, é necessária a preparação dos parâmetros para envio ao *Guest*. Os argumentos deverão ser agrupados em uma cadeia de *bytes* independentemente da sua quantidade e seus tipos, respeitando unicamente a ordem de parametrização para a execução da tarefa remota.

No exemplo de programa, apresentado na Figura 4.7, são chamadas 2 tarefas, `tsk1` e `tsk2`, executadas remotamente. Na linha 3 a função `rte_init` inicia a execução, provendo as conexões com os *Guests*, cujos endereços encontram-se no arquivo `rte_example.cfg`. Aqui supõem-se que há dois *Guests*, como exemplificado na Figura 4.3. Nas linhas 23 à 29 são preparados os argumentos de entrada e o argumento de retorno para a tarefa 1, que é chamada na linha 39 e colocada em execução no *Guest*. O mesmo se repete nas linhas 31 à 37 para a tarefa 2. Tem-se então as execuções simultâneas das duas tarefas nos dois *Guests* nas linhas 39 e 40. Na linha 41, aguarda-se o término da tarefa 1 e o recebimento do resultado, podendo-se a seguir na linha 42 utilizar o conteúdo da variável. O mesmo ocorre para a tarefa 2, na linha 43 e 44. O sistema RTE é finalizado na linha 45 com a chamada da função `rte_finalize()`.

Figura 4.7: Programa a ser executado no computador principal

```
int main(int argc, char* argv[])
{
  if( rte_init("rte_exemple.cfg") != 0) {
    return -1;
  }

  list_type_dvc[0]=rte_dvc_cpu; // CPU

  rte_arg_list_t argin1;
  rte_ret_list_t argret1;
  rte_arg_list_t argin2;
  rte_ret_list_t argret2;

  int a, b, c, x, y;
  float z;
  unsigned int tsk1, tsk2;

  a = 2;
  b = 4;
  x = 3;
  y = 5;

  rte_init_args(2, &argin1, 1, &argret1);
  argin1.arg[0].size= sizeof(int);
  argin1.arg[0].arg=(char *)&a;
  argin1.arg[1].size= sizeof(int);
  argin1.arg[1].arg=(char *)&b;
  argret1.ret[0].size= sizeof(int);
  argret1.ret[0].ret=(char *)&c;

  rte_init_args(2, &argin2, 1, &argret2);
  argin2.arg[0].size= sizeof(int);
  argin2.arg[0].arg=(char *)&x;
  argin2.arg[1].size= sizeof(int);
  argin2.arg[1].arg=(char *)&y;
  argret2.ret[0].size= sizeof(float);
  argret2.ret[0].ret=(char *)&z;

  tsk1 = rte_tsk_call(0, 0, "soma", "librte_example.so", &argin1,  &argret1);
  tsk2 = rte_tsk_call(1, 0, "multiplica", "librte_example.so", &argin2,  &argret2);
  rte_sync(tsk1);
  printf("Resultado Soma A + B = C | %d + %d = %d\n", a, b, c);
  rte_sync(tsk2);
  printf("Produto da conta X * Y = Z | %d * %d = %f\n", x, y, z);
  rte_finalise();
  return 0;
}
```

Fonte: Próprio Autor

4.1.2.2 Conjunto de tarefas remotas

O código do **Conjunto de Tarefas Remotas** contém as funções especificadas pelo desenvolvedor da aplicação paralela a serem chamadas pelo **Programa Principal**. Esse conjunto deve estar presente no *Guest* em que as tarefas serão executadas. A carga da tarefa no *Guest* é feita em tempo de execução pela chamada da função **`rte_tsk_call`**.

A Figura 4.8 apresenta um exemplo de um código do **Conjunto de Tarefas Re-**

motas. Este exemplo é composto por duas tarefas. A primeira está especificada a partir da linha 1 e tem como finalidade retornar a soma de dois números inteiros. Já a segunda tarefa, especificada na linha 10, tem como finalidade retornar o resultado da multiplicação de dois números inteiros.

Figura 4.8: Exemplo de um conjunto de tarefas remotas.

```
1  void __attribute__((visibility("default"))) soma(void *arg)
2  {
3    rte_args_t *param = (rte_args_t*)arg;
4    int *result = (int *)param->resp;
5    int a = *(int *)param->args;
6    int b = *(int *)(param->args + sizeof(int));
7    *result = a + b;
8  }
9
10 void __attribute__((visibility("default"))) multiplica(void *arg)
11 {
12   rte_args_t *param = (rte_args_t *)arg;
13   float *result = (float *)param->resp;;
14   int a = *(int *)param->args;
15   int b = *(int *)(param->args + sizeof(int));
16   *result = a * b;
17 }
```

Fonte: Próprio Autor

4.1.2.3 Tarefas processadas em GPGPUs

O código de um **Conjunto de Tarefas** preparado para ser processado por uma GPGPU apresentam dois novos requisitos para que possam ser executadas corretamente. O primeiro é que cada tarefa deverá ter uma função que corresponde à tarefa propriamente dita e uma função contendo o código a ser processado em uma GPGPU. Já o segundo requisito é a necessidade de acrescentar as diretivas de compilação **`#ifdef __cplusplus`**, **`extern "C" {`** e **`#endif`** no inicio da definição das tarefas e as as diretivas de compilação **`#ifdef __cplusplus`**, **`}`** e **`#endif`** ao final da declaração. Isto se faz necessário pois o compilador utilizado para gerar o binário para ser executado na GPGPU é em C++ e a API RTE é desenvolvida no padrão C.

Na Figura 4.9 pode-se observar um exemplo do código de um **Conjunto de Tarefas Remotas** preparado para ser processado em uma GPGPU. Este exemplo é composto por duas funções. A primeira, **`rte_gpgpu_soma`**, especificada a partir

Figura 4.9: Exemplo do conjunto de tarefas para ser processado na GPGPU.

```
1  #include <stdio.h>
2  #include <rte.h>
3
4  __global__ void rte_gpgpu_soma( int* A, int* B, int* C, int* N )
5  {
6      int tID = blockIdx.x;
7      int n = *( int* )N;
8
9      if( tID < n ) {
10            c[ tID ] = A[ tID ] + B[ tID ];
11     }
12 }
13
14 #ifdef __cplusplus
15     extern "C" {
16 #endif
17 void __attribute__((visibility("default")))rte_task_example( void* arg )
18 {
19     int i, n;
20     int *a, *b, *c, *dA, *dB, *dC, *dN;
21
22     // enderecos de argumentos e endereco de resposta
23     rte_args_t* arg_task = ( rte_args_t* )arg;
24
25     n = ( int* )arg_task->args[ 0 ];
26
27     a = ( int* )arg_task->args[ 1 ];
28     b = ( int* )arg_task->args[ 2 ];
29     c = ( int* )arg_task->rets[ 0 ];
30
31     cudaMalloc( ( void** ) &dA, n * sizeof( int ) );
32     cudaMalloc( ( void** ) &dB, n * sizeof( int ) );
33     cudaMalloc( ( void** ) &dC, n * sizeof( int ) );
34     cudaMalloc( ( void** ) &dN, sizeof( int ) );
35
36     cudaMemcpy( dA, a, n * sizeof( int ), cudaMemcpyHostToDevice );
37     cudaMemcpy( dB, b, n * sizeof( int ), cudaMemcpyHostToDevice );
38     cudaMemcpy( dN, n, sizeof( int ), cudaMemcpyHostToDevice );
39
40     dim3 dimGrid( n / TILE_WIDTH, n / TILE_WIDTH, 1 );
41     dim3 dimBlock( TILE_WIDTH, TILE_WIDTH);
42
43     rte_gpgpu_soma<<<n,1>>>( dA, dB, dC, dN );
44
45     cudaMemcpy( c, dC, n * sizeof( int ), cudaMemcpyDeviceToHost );
46 }
47 #ifdef __cplusplus
48     }
49 #endif
```

Fonte: Próprio Autor

da linha 4 é uma função que é executada diretamente na GPGPU. Já a segunda, **rte_task_example**, especificada na linha 17 é uma tarefa padrão RTE com as direti-

vas para a chamada e execução da função interna na GPGPU.

4.1.3 Compilação e execução de um programa

Esta seção apresenta a utilização da ferramenta aqui proposta. Primeiramente é exemplificada a forma de compilação do programa principal e do conjunto de tarefas remotas nas Seções 4.1.3.1 e 4.1.3.2 respectivamente. A seguir é apresentada a execução da aplicação na Seção 4.1.3.4.

4.1.3.1 Compilação do programa principal

Como apresentado na Seção 4.1.2.1, uma aplicação trata-se de um executável composto pelo programa principal desenvolvido pelo usuário ligado com a biblioteca `librte`. A Figura 4.10 mostra a compilação de uma aplicação RTE simples.

Figura 4.10: Exemplo de compilação de um programa com RTE.

```
$ gcc -c -Wall -m64 -lrte rte_example.c -o rte_example
```

Fonte: Próprio Autor

4.1.3.2 Compilação do conjunto de tarefas remotas

O Conjunto de Tarefas Remotas é organizado no formato de uma biblioteca dinâmica. A Figura 4.11 mostra o exemplo de compilação de um Conjunto de Tarefas Remotas.

Figura 4.11: Exemplo de compilação do conjunto de tarefas RTE.

```
$ gcc -c -Wall -Werror -Wextra -pedantic -fpic rte_lib_example.c -shared
-o librte_example.so
```

Fonte: Próprio Autor

Após a compilação, uma cópia da biblioteca deverá estar em cada *Guest* em que for solicitada a sua execução.

4.1.3.3 Compilação do conjunto de tarefa a serem processadas numa GPGPU

Como já mencionado anteriormente na Seção 4.1.3.2, o Conjunto de Tarefas com a funcionalidade de ser processada em uma GPGPU também deve ser compilado na forma de uma biblioteca compartilhada. A Figura 4.12 mostra o exemplo da compilação do Conjunto de Tarefas RTE para ser chamada e executada em uma GPGPU.

Figura 4.12: Exemplo compilação do conjunto de tarefas processadas em uma GPGPU

```
$ nvcc -g -m64 -O3 -c --compiler-options "-fPIC" rte_gpgpu_soma.cu
$ nvcc -m64 -O3 rte_gpgpu_soma.o --shared --compiler-options "-fpic
-pedantic -Wall -Wextra" -o librte_gpgpu_soma.so
```

Fonte: Próprio Autor

Assim como mencionado na seção anterior, deverá ter uma cópia do conjunto de tarefas nos *Guests* que forem feitas as chamadas para a execução em suas respectivas GPGPUs.

4.1.3.4 Execução da aplicação no modelo RTE

O programa responsável pela execução do conjunto de tarefas na API RTE é **rted**. Este programa deve estar em execução em cada *Guest* que for utilizado na aplicação paralela. A Figura 4.13 mostra a execução do programa **rted** em dois *guests* distintos, node01 (Figura 4.13a) e node02 (Figura 4.13b) simultaneamente.

Figura 4.13: Execução do gerenciador de tarefas remotas **rted**.

(a) Execução no *Guest* **node01**

```
[node01]$ ./rted
```

(b) Execução no *Guest* **node02**

```
[node02]$ ./rted
```

Fonte: Próprio Autor

Em seguida, com cada *guest* executando seu **rted**, inicia-se o processamento da aplicação no *Host*, como é apresentado na Figura 4.14.

Figura 4.14: Exemplo da execução de uma aplicação RTE.

```
[node01]$ ./rte_example
```

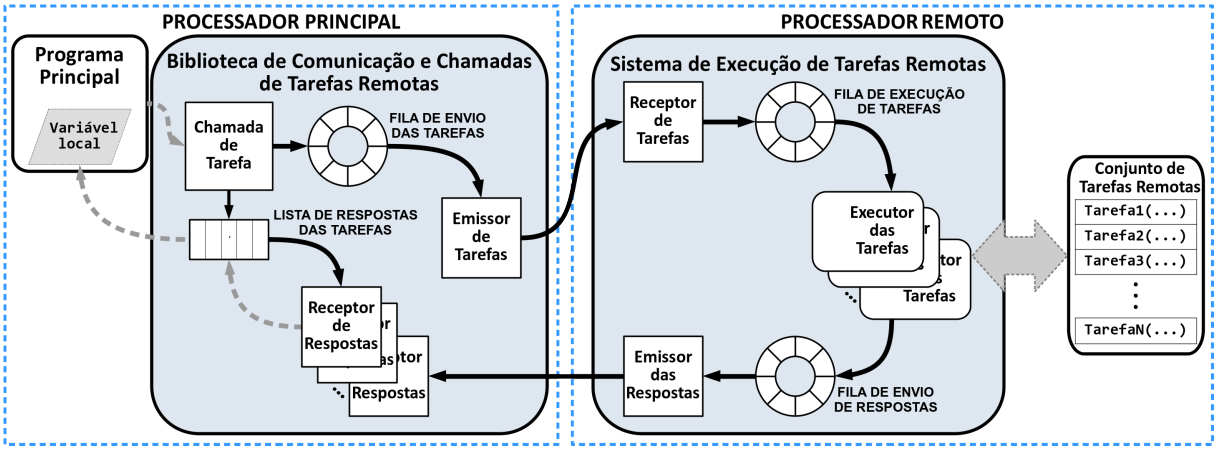
Fonte: Próprio Autor

4.2 Arquitetura do sistema RTE

A arquitetura do sistema RTE é composta por: Uma **Biblioteca de Comunicação e Chamadas de Tarefas Remotas**, que deve estar presente no *Host*, é responsável pela interface da aplicação paralela desenvolvida pelo usuário com um ou mais **Conjuntos de Tarefas Remotas** em cada **Processador Remoto**. O **Sistema de Execução de Tarefas Remotas**, deve estar presente em cada **Processador Remoto** que fará o processamento da tarefa selecionando-a num **Conjunto de Tarefas Remotas**.

A Figura 4.15 apresenta a arquitetura do sistema RTE. A aplicação do usuário é composta por um **Programa Principal** executado em um *Host*, aqui denominado **Processador Principal**, e por um ou mais **Conjuntos de Tarefas Remotas** contendo as funções correspondentes às tarefas a serem executadas em cada *Guest*, aqui denominado **Processador Remoto**.

Figura 4.15: Arquitetura do sistema RTE



Fonte: Próprio Autor

4.2.1 Biblioteca de comunicação e chamadas de tarefas remotas

Esta biblioteca contém o conjunto de funções de interface de comunicação executadas no **Processador Principal** para interagir com cada **Processador Remoto**,

provendo o envio das requisições de execução de tarefas remotas, gerenciamento do término de cada tarefa e recepção das respostas retornadas.

4.2.1.1 Procedimento de inicialização

Para garantir a execução remota de tarefas é necessária a execução de três procedimentos de inicialização:

1. Conexão com os **Processadores Remotos**: Para executar uma tarefa remota precisa-se definir cada **Processador Remoto** que fará parte desta execução paralela. A identificação e a respectiva especificação de cada **Processador Remoto** é feita através da **Lista de Processadores Remotos**. Essa lista é inicializada na função de inicialização com os dados do arquivo de configuração cujo nome é passado como parâmetro, como referido na Seção 4.1.1.1.
   A Figura 4.16 apresenta um ítem da **Lista de Processadores Remotos** contendo a estrutura responsável pelo armazenamento dos dados para a conexão com cada **Processador Remoto**.

Figura 4.16: Estrutura da **Lista de Processadores Remotos**.

Identificador da Conexão	Dados da Conexão		
	Tipo do Protocolo	Endereço de Rede	Porta da Conexão

Fonte: Próprio Autor

Identificador da Conexão [1]: Número inteiro positivo maior que zero gerado automaticamente através do módulo de inicialização da conexão;

Dados da Conexão de Rede Estrutura com as informações das conexões dos **Processadores Remotos**;

Tipo do Protocolo[1]: Protocolo de conexões do *socket*. O tipo utilizado neste protótipo é o *Transmission Control Protocol* (TCP) devido a garantia de conexão entre os nós e a confiabilidade no envio dos dados;

Endereço de Rede[2]: Estrutura composto de 4 octetos, 8 *bits* cada, contendo o endereço do **Processador Remoto**;

Porta da Rede[2]: Número da porta da rede para execução do **Sistema de Execução de Tarefas Remotas** no **Processador Remoto**;

[1]Dados gerados automaticamente. [2]Dados passados como parâmetro na inicialização.

2. Criação da **FILA DE ENVIO DE TAREFAS**: Essa fila é responsável pelo armazenamento das tarefas a serem executadas remotamente. A Figura 4.17 apresenta a estrutura de dados com as informações de um item desta fila.

Figura 4.17: Estrutura da **FILA DE ENVIO DE TAREFAS**.

Identificador do Processador Remota	Identificador do Dispositivo	Identificador da Tarefa	Nome da Tarefa	Nome do Conjunto de Tarefas	Lista de Argumentos	Lista de Respostas

Fonte: Próprio Autor

Identificador do Processador Remoto: Número inteiro positivo, maior ou igual a 0 (zero), contendo o índice de um elemento da **Lista de Processadores Remotos** indicando em qual **Processador Remoto** será enviada a tarefa a ser executada;

Identificador do Dispositivo: Número inteiro positivo, maior ou igual a 0 (zero), para identificação do dispositivo, CPU ou GPGPU por exemplo, em que a tarefa será executada no **Processador Remoto**;

Identificador da Tarefa: Número inteiro positivo, maior ou igual a 0 (zero), para identificação da tarefa;

Nome da Tarefa: Cadeia de caracteres para localização da tarefa no **Conjunto De Tarefas**;

Nome do Conjunto De Tarefas: Sequência de caracteres para identificação e localização da lista de tarefas pelo **Sistema de Execução de Tarefas Remotas** no **Processador Remoto**;

Lista de Argumentos: Endereço da área de memória reservada para armazenar a posição do primeiro elemento da lista de argumentos necessários para a execução da tarefa;

Lista de Respostas: Endereço da área de memória reservada para armazenar a posição do primeiro elemento da **Lista de Respostas da Tarefa**.

3. Criação da **LISTA DE RESPOSTAS DAS TAREFAS**: É uma lista criada para o controle e atualização de cada resposta da tarefa executada remotamente.
A Figura 4.18 apresenta a lista para armazenar e atualizar o valor da resposta.

O Identificador da Tarefa é utilizado como índice desta lista.

Figura 4.18: Estrutura da **LISTA DE RESPOSTAS DAS TAREFAS**.

1		**2**		**...**	**N**	
Controle de Resultado da Tarefa		**Controle de Res...**			**...sultado da Tarefa**	
Sincronização de Tarefa	**Endereço de Resposta**	**Sincronização de Tarefa**	**...**	**...**	**...**	**Endereço de Resposta**

Fonte: Próprio Autor

Sincronização de Tarefa: Um "semáforo" utilizado para garantir o sincronismo de uma tarefa e atualização da variável apontada pela **Endereço de Resposta**;

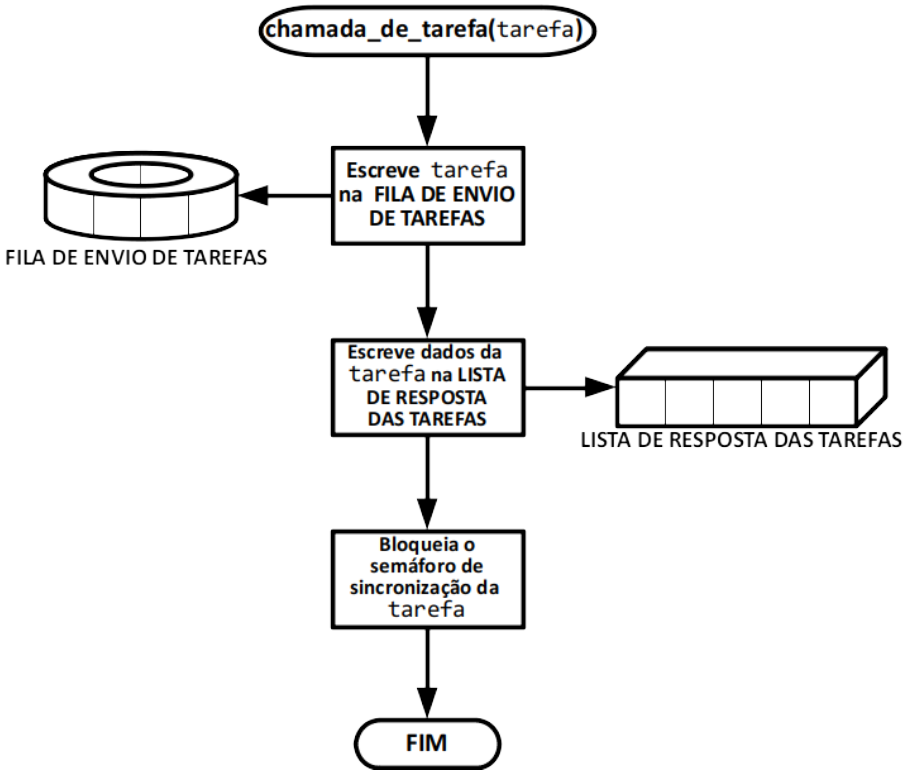
Endereço de Resposta: Endereço da variável de resposta para sua atualização após a execução da tarefa no **Processador Remoto**.

4.2.1.2 Chamada de tarefas

O **Programa Principal** ao chamar uma tarefa remota, através da função `rte_tsk_call` apresentada na Seção 4.1.1.2, armazena as informações da requisição desta tarefa na **FILA DE ENVIO DE TAREFAS**. Essa execução é assíncrona e não bloqueante, isto significa que logo após inserir uma tarefa na fila o programa continuará seu fluxo normal.

A Figura 4.19 mostra o fluxo de execução da função responsável pela chamada de uma tarefa remota. Ela recebe como parâmetro os dados necessários para execução de uma tarefa em um **Processador Remoto**. Executa a conversão dos dados e os armazena na **FILA DE ENVIO DE TAREFAS**. Também armazena na posição indicada pelo identificador da tarefa, conforme a estrutura apresentada na Figura 4.18, o endereço da variável de retorno na **LISTA DE RESPOSTAS DAS TAREFAS**, cria o semáforo responsável pela sincronização, armazena o seu endereço nessa lista e o bloqueia.

Figura 4.19: Execução da chamada de tarefas.



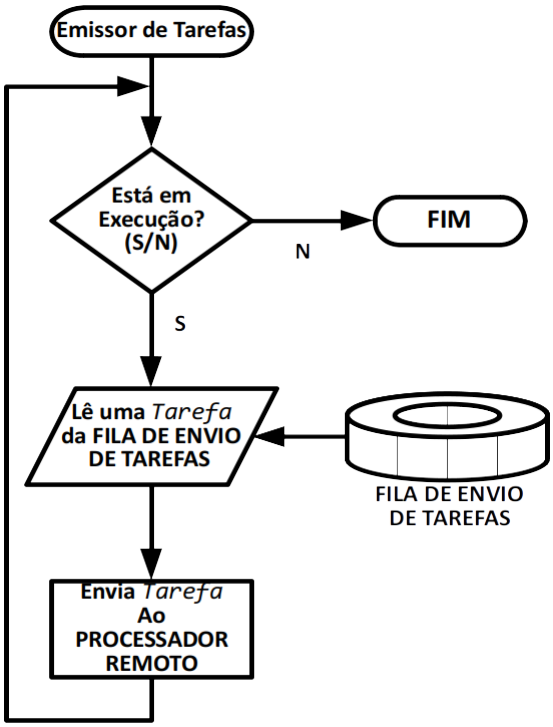
Fonte: Próprio Autor

4.2.1.3 Emissor de tarefas

O envio de uma tarefa para o **Processador Remoto** é feita pelo **Emissor de Tarefas**. É uma *thread* que aguarda a entrada de tarefas em uma fila de execução, faz a retirada de um item e encaminha ao **Processador Remoto** solicitado para seu processamento.

A Figura 4.20 mostra o fluxo de execução desta *thread*. Enquanto houver uma requisição na **FILA DE ENVIO DE TAREFAS**, retira uma tarefa e a envia ao **Processador Remoto** destinatário.

Figura 4.20: Emissor de tarefas.



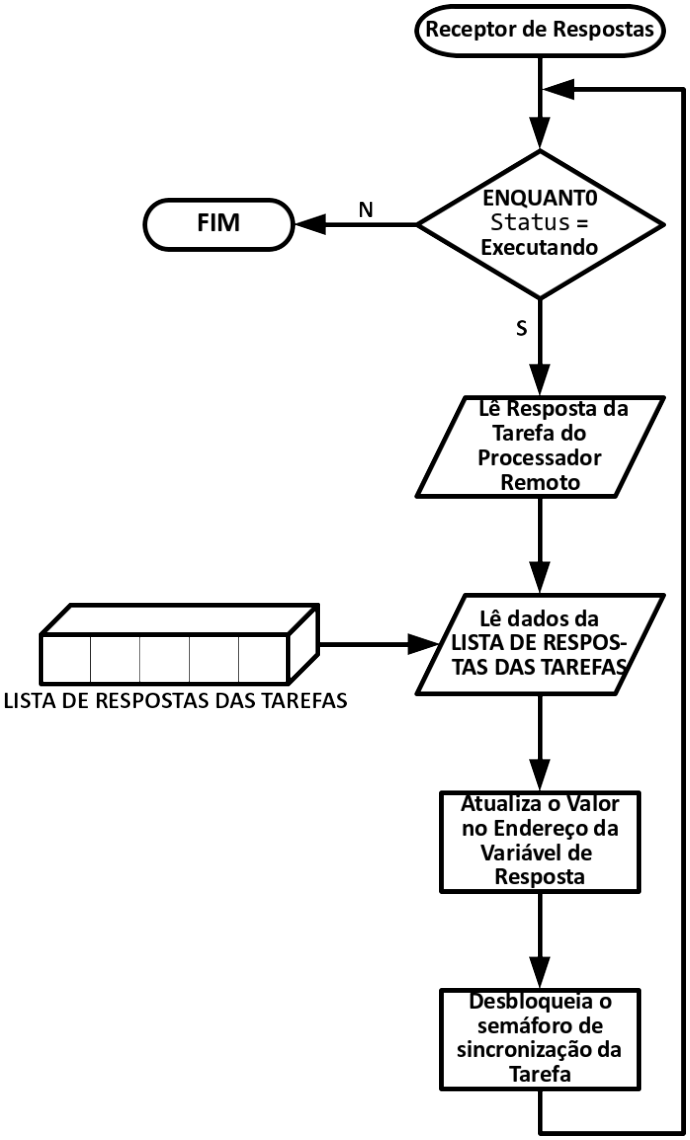
Fonte: Próprio Autor

4.2.1.4 Receptores de respostas

A recepção do resultado da execução de uma tarefa remota é feita através das *threads* que aguardam o envio da resposta de cada **Processador Remoto** que foi solicitado. O **Sistema de Comunicação e Chamadas de Tarefas Remotas** disponibiliza uma *thread* **Receptor de Respostas** exclusiva para cada **Processador Remoto**.

A Figura 4.21 apresenta o fluxo de execução destas *threads*. Ao receber a mensagem contendo os dados do resultado da execução da tarefa, atualiza o conteúdo no endereço presente na **LISTA DE RESPOSTAS DE TAREFAS** e o disponibiliza para sua utilização através do desbloqueio do seu semáforo de sincronização, liberando sequências que estão aguardando o término da tarefa.

Figura 4.21: Receptor de respostas.



Fonte: Próprio Autor

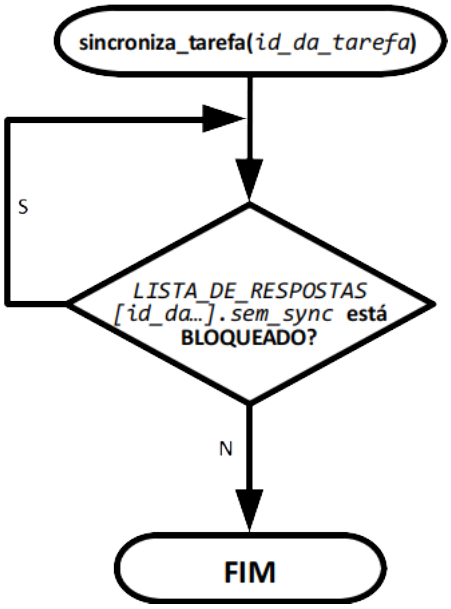
4.2.1.5 Sincronização de tarefas

Para o uso do resultado da execução de uma tarefa remota é necessário que esta tenha terminado seu processamento no **Processador Remoto** e o **Processador Principal** recebido o dado de retorno. Isto é feito através da sincronização do fluxo de controle da execução de cada tarefa, como referido na Seção 4.1.1.3.

A Figura 4.22 apresenta a função responsável por esta sincronização. Ela recebe, como argumento, o índice correspondente a sua posição na **LISTA DE RESPOS-TAS DE TAREFAS** e aguarda o desbloqueio do “semáforo”, especificado no campo

sem_sync, para prosseguir com a execução e utilização desta variável pelo **Programa Principal**.

Figura 4.22: Fluxo da execução da sincronização de tarefa.



Fonte: Próprio Autor

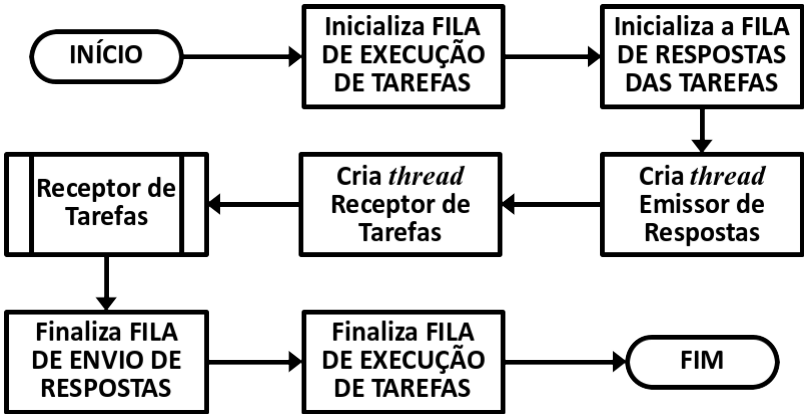
4.2.2 Sistema de execução de tarefas remotas

Este sistema, referido na Seção 4.1.1 como **rted**, deverá estar presente e em execução em cada **Processador Remoto** que recebe as requisições de processamento de tarefas de um **Processador Principal**.

A Figura 4.23 mostra o fluxo da execução deste sistema. O **Receptor de Tarefas** aguarda uma mensagem enviada pelo **Processador Principal** contendo os dados da requisição de execução de uma tarefa e os escreve na **FILA DE EXECUÇÃO DE TAREFAS** que é consumida pelo **Executor de Tarefas**. Depois de processar a tarefa, escreve o resultado na **FILA DE ENVIO DE RESPOSTAS**. O **Emissor de Respostas** lê o resultado dessa fila e o envia para ao **Processador Principal** que solicitou a execução.

Para a execução de uma tarefa, este sistema gerencia duas estruturas de dados. Uma para a entrada de dados e outra para a saída que são a **FILA DE EXECUÇÃO DE TAREFAS** e a **FILA DE ENVIO DE RESPOSTAS**, respectivamente, essas estruturas

Figura 4.23: Sistema de execução de tarefas remotas.



Fonte: Próprio Autor

são apresentadas a seguir:

1. **FILA DE EXECUÇÃO DE TAREFAS**: Guarda as tarefas a serem executadas em cada **Processador Remoto**. A Figura 4.24 mostra a estrutura desta fila responsável pelos dados necessários para a execução de uma tarefa.

Figura 4.24: Estrutura da **FILA DE EXECUÇÃO DE TAREFAS**.

Identificador da Conexão	Identificador do Dispositivo	Identificador da Tarefa	Nome da Tarefa	Nome do Conjunto de Tarefas	Lista de Argumentos	Lista de Respostas

Fonte: Próprio Autor

Identificador da Conexão: Número inteiro positivo, maior ou igual a 0 (zero), para identificar a conexão com o **PROCESSAR PRINCIPAL** que fez a solicitação da tarefa a ser executada;

Identificador do Dispositivo: Número inteiro positivo, maior ou igual a 0 (zero), para identificação do dispositivo, CPU ou GPGPU por exemplo, em que a tarefa será executada;

Identificador da Tarefa: Número inteiro positivo, maior ou igual a 0 (zero), para identificação da tarefa;

Nome da Tarefa: Cadeia de caracteres para localização da tarefa no **Conjunto de Tarefas**;

Nome do Conjunto de Tarefas: Sequência de caracteres para identificação e localização da lista de tarefas pelo **Sistema de Execução de Tarefas Re-**

motas;

Lista de Argumentos: Endereço da área de memória reservada para armazenar a posição do primeiro elemento da lista de argumentos necessários para a execução da tarefa;

Lista de Resposta: Endereço da área de memória reservada para armazenar a posição do primeiro elemento da **Lista de Respostas da Tarefa**.

2. **FILA DE ENVIO DE RESPOSTAS**: Armazena os resultados de cada tarefa processada para, após a finalização dessa tarefa, encaminhar a resposta ao **Processador Principal** que solicitou sua execução. A Figura 4.25 mostra a estrutura de dados de cada item desta fila.

Figura 4.25: Estrutura da **FILA DE ENVIO RESPOSTAS**.

Identificador da Conexão	Identificador da Tarefa	Lista de Respostas

Fonte: Próprio Autor

Identificador da Conexão: Número inteiro positivo, maior ou igual a 0 (zero), para identificador da conexão com o **PROCESSAR PRINCIPAL** que fez a solicitação da tarefa executada;

Identificador da Tarefa: Número inteiro positivo, maior ou igual a 0 (zero), para identificação da tarefa;

Lista de Respostas: Endereço da área de memória reservada para armazenar a posição do primeiro elemento dessa lista.

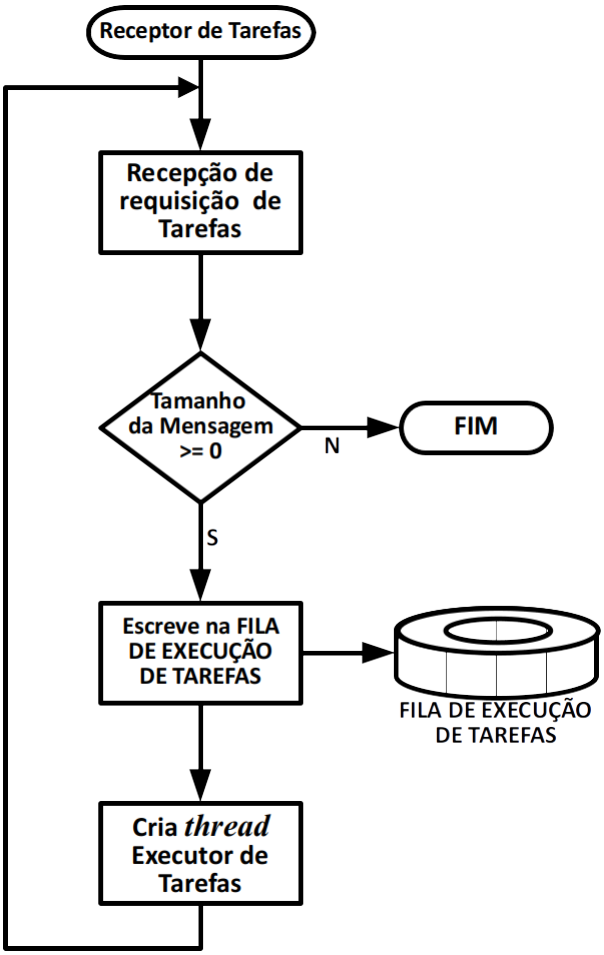
4.2.2.1 Receptor de tarefas

Esta função é responsável pela recepção de requisições de execução de tarefas enviadas pelo **Sistema de Comunicação e Execução de Tarefas Remotas**. A tarefa recebida é armazenada em uma fila para sua execução.

A Figura 4.26 mostra o fluxo de execução da *thread* **Receptor de Tarefas**. Ao receber uma mensagem faz a sua validação, verificando o tamanho e o seu conteúdo. Se o tamanho for menor que 1 *byte* então finaliza sua execução, pois o recebimento

de uma mensagem com conteúdo vazio indica que foi encerrado o processamento do **Programa Principal** no **Processador Principal**. Se esta condição não for satisfeita, lê os dados e os converte no formato para escrever na **FILA DE EXECUÇÃO DE TAREFAS**. Em seguida, cria a *thread* responsável pela execução da tarefa e volta a aguardar uma nova solicitação.

Figura 4.26: Receptor de tarefas.



Fonte: Próprio Autor

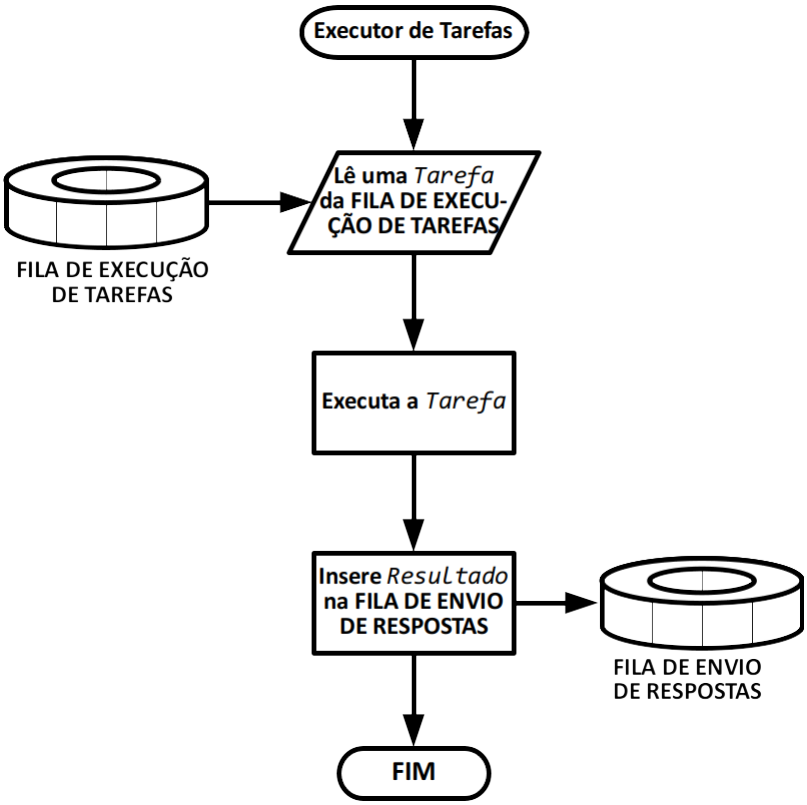
4.2.2.2 Executor de Tarefas

A execução de uma tarefa é feita através da *thread* **Executor de Tarefas**. A Figura 4.27 apresenta o fluxo de execução desta *thread*, como segue:

1. Retira um item com os dados necessários para execução da **FILA DE EXECUÇÃO DE TAREFAS**;

2. Carrega o **Conjunto de Tarefas Remotas** do arquivo designado pelo campo **Nome do Conjunto de Tarefas**;
3. Executa a tarefa indicada no campo **Nome da Tarefas**, passando como parâmetro os dados contidos no campo **Argumento(s) da Tarefa**;
4. Ao término da execução da tarefa converte segue resultado no formato para escrever na **FILA DE ENVIO DE RESPOSTAS**;
5. e Encerra o seu processamento.

Figura 4.27: Executor de tarefas.



Fonte: Próprio Autor

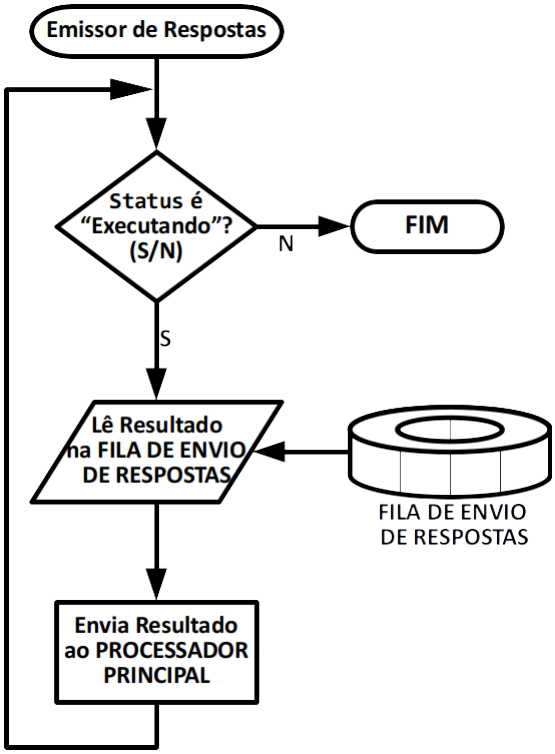
4.2.2.3 Emissor de respostas

Para o uso do resultado da execução de uma tarefa é necessário que esta seja enviada como resposta ao **Processador Principal** que a solicitou. A transmissão desta resposta é feita pela *thread* **Emissor de Respostas**.

A Figura 4.28 mostra o fluxo de execução desta *thread*. Enquanto o **Sistema de Execução de Tarefas Remotas** estiver em execução o **Emissor de Respostas**

retira um item da **FILA DE ENVIO DE RESPOSTAS**, converte os dados e envia-os ao **Processador Principal**.

Figura 4.28: Emissor de respostas.



Fonte: Próprio Autor

5 IMPLEMENTAÇÃO

Este capítulo apresenta os aspectos da implementação do protótipo conforme a proposta descrita no Capítulo 4. O *Remote Tasks Execute* (RTE) é um ambiente de programação e um sistema de execução de tarefas paralelas e distribuídas baseada em BoT que disponibiliza recursos de desenvolvimento para arquiteturas de *hardware* heterogêneas.

Para demonstrar o modelo de programação implementou-se o protótipo de um sistema escrito na linguagem de programação C, padrão ISO C99 (ISO, 1999), em ambiente GNU Linux. A escolha desses critérios é devido a sua ampla utilização para desenvolvimento em HPC.

A organização deste capítulo está disposta da seguinte forma: A Seção 5.1 apresenta as definições das estruturas de dados, constantes e tipos de dados propostos para o sistema **RTE**. A Seção 5.2 mostra a elaboração da camada de interface entre a **Biblioteca de Comunicação e Chamadas de Tarefas Remotas** e o **Sistema de Execução de Tarefas Remotas**. A Seção 5.3 exibe a implantação da **Biblioteca de Comunicação e Chamadas de Tarefas Remotas**. E a Seção 5.4 expõe em detalhes o desenvolvimento do **Sistema de Execução de Tarefas Remotas**.

5.1 Macros, constantes e estruturas gerais

O desenvolvimento do sistema RTE utilizou recursos disponíveis na linguagem de programação C (LANGSAM; AUGENSTEIN; TENENBAUM, 1990; RITCHIE; KERNIGHAN; LESK, 1988) para predefinir tipos de dados exclusivos e necessários para armazenar e manipular as informações contidas nos módulos do protótipo.

Essa seção apresenta as definições das macros, constantes e estruturas de dados que são utilizadas tanto no Sistema de Execução de Tarefas Remotas, **`rted`**, quanto na Biblioteca de Comunicação e Chamadas de Tarefas Remotas, **`librte`**. Estas definições são feitas para garantir a compatibilidade entre os módulos do sistema RTE.

5.1.1 Macros

Para as declarações das constantes gerais utilizadas no desenvolvimento do sistema RTE fez-se o uso de definições presentes em *GNU Compiler Collection* (GCC) (KERRISK, 2010; STALLMAN *et al.*, 2020) tais como:

`__attribute__((visibility("default")))`: É usado para especificar explicitamente a visibilidade da função dentro de uma biblioteca compartilhada;

`_POSIX_THREAD_THREADS_MAX`: Constante responsável por armazenar o número máximo de *threads* por processo;

`NAME_MAX`: Define o tamanho máximo, em *bytes*, do nome do arquivo.

5.1.2 Estruturas ou registros

Propõem-se tipos compostos para armazenar dados que conterão as informações necessárias para a execução do sistema RTE.

5.1.2.1 Estrutura de mensagem de tarefas

Uma tarefa, no sistema RTE, é representada pelas seguintes estruturas de dados com as informações necessárias para sua execução em um determinado *Guest*:

`rte_tsk_header_t`: Contendo as informações básicas de uma tarefa RTE. A Tabela 5.1 apresenta a definição proposta para esta estrutura;

Tabela 5.1: Estrutura **`rte_tsk_header_t`**.

Estrutura	Campos

`typedef struct {`
`  int msg_type;`
`  int tsk_id;`
`  char tsk_name[256];`
`  char tsk_set_name[NAME_MAX]`
`  size_t sz_resp;`
`  size_t sz_args;`
`} __attribute__( ( packed ) )`
`rte_tsk_header_t;`
	`msg_type`: Tipo de mensagem RTE;
	`tsk_id`: Identificador da tarefa;
	`tsk_name`: Nome da tarefa;
	`tsk_set_name`: Nome do Conjunto de Tarefas;
	`sz_resp`: Tamanho da(s) resposta(s) da(s) tarefa(s);
	`sz_args`: Tamanho do(s) argumento(s) da(s) Tarefa(s);

Fonte: Próprio Autor

rte_arg_t: A Tabela 5.2 mostra a estrutura com as informações referentes ao argumento de entrada RTE;

Tabela 5.2: Estrutura **rte_arg_t**.

Estrutura	Campos
`typedef struct { size_t size; char* arg; } __attribute__( ( packed ) ) rte_arg_t;`	**size**: Tamanho do argumento da tarefa; **arg**: Área de memória com "**size**" *bytes* do argumento de entrada.

Fonte: Próprio Autor

rte_arg_list_t: A Tabela 5.3 mostra a estrutura da lista de argumentos de entrada;

Tabela 5.3: Estrutura **rte_arg_list_t**.

Estrutura	Campos
`typedef struct { int n_args; rte_arg_t* args; } __attribute__( ( packed ) ) rte_arg_list_t;`	**n_args**: Número de elementos da lista; **args**: Endereço de memória do primeiro elemento da lista.

Fonte: Próprio Autor

rte_msg_t: Estrutura principal dos argumentos de entrada de uma tarefa RTE. Essa estrutura principal é mostrada na Tabela 5.4.

Tabela 5.4: Estrutura **rte_msg_t**.

Estrutura	Campos
`typedef struct { rte_tsk_header_t; rte_tsk_header; rte_arg_list_t* rte_arg_list; } __attribute__( ( packed ) ) rte_msg_t;`	**rte_tsk_header**: Informações básicas de uma tarefa; **rte_arg_list**: Endereço de memória da lista de argumentos.

Fonte: Próprio Autor

5.1.2.2 Estrutura de mensagem de respostas

O sistema RTE apresenta, também, uma estrutura para armazenar os resultados de cada tarefa. A Tabela 5.5 mostra a definição proposta para esta estrutura.

Tabela 5.5: Estrutura **rte_msg_recv_t**.

Estrutura	Campos

`typedef struct {`
`  int msg_type;`
`  int task_id;`
`  size_t n_ret;`
`} __attribute__( ( packed ) )`
`rte_msg_recv_t;`
	msg_type: Tipo de mensagem RTE;
	task_id: Identificador da tarefa;
	n_ret: Tamanho da resposta.

Fonte: Próprio Autor

5.2 Camada de interconexão do sistema RTE

A interconexão entre os módulos do sistema RTE, **librte** e **rted**, é feita através da camada de comunicação baseada na API POSIX *Sockets* (STEVENS; FENNER; RUDOFF, 2004; COMER; STEVENS, 1999), que especifica os protocolos de comunicação desse sistema.

Esta camada é apresentada como uma biblioteca, **librte_comm**, que deve ser ligada na compilação de cada módulo RTE. Esta biblioteca é composta pelos conjuntos de **Registros** e **Funções**, ambos apresentados a seguir.

5.2.1 Registros

A **librte_comm** define dois grupos de registros. O primeiro é responsável por armazenar as informações necessárias para estabelecer as conexões entre os nós do sistema distribuído RTE, já o segundo estabelece os formatos para a transmissão das mensagens entre os módulos.

5.2.1.1 Registros de conexões

A **Camada de Comunicação RTE** propõem tipos de dados necessários para armazenar as informações para a criação de um *Guest* e agrupar os elementos para obter a conexão com o *Host*.

O tamanho máximo em *bytes* do tipo de protocolo de rede utilizado em cada um dos nós e o nome dos *Guests* na rede, tanto no próprio *Guest* quanto em uma definição de conexão no *Host*, é dado pela macro **_POSIX_HOST_NAME_MAX**. Essa constante é definida na biblioteca de cabeçalho `limits.h`.

A Figura 5.1 apresenta as estruturas para estabelecer as conexões entre os módulos RTE. A Figura 5.1a mostra a estrutura para armazenar as informações para a criação de um *Guest*. Já a Figura 5.1b exibe a estrutura responsável por armazenar os dados da conexão entre o *Host* e o *Guest*.

Figura 5.1: Estruturas de comunicação RTE.

(a) ***Guest***(Servidor)

```
typedef struct {
   struct sockaddr_in rte_addr;
   int guest_id;
   int listen_id;
   char hname[_POSIX_HOST_NAME_MAX];
   short port;
   char prtcl[_POSIX_HOST_NAME_MAX];
   int lstn_bk_log;
} __attribute__((packed))
  rte_srv_guest_t;
```

(b) ***Host***(Cliente)

```
typedef struct {
   struct sockaddr_in rte_addr;
   int guest_conn_id;
   char hname[_POSIX_HOST_NAME_MAX];
   short port;
   char prtcl[_POSIX_HOST_NAME_MAX];
   int lstn_bk_log;
} __attribute__((packed))
  rte_guest_conn_t;
```

Fonte: Próprio Autor

5.2.1.2 Estruturas de envio de mensagens

O protocolo de transmissão de mensagens entre os nós é definido em duas partes simultâneas, que são:

Cabeçalho: Primeira informação enviada para que o receptor reserve espaço para o recebimento do “pacote” de dados. A Tabela 5.6 exibe a estrutura responsável por armazenar os dados do cabeçalho.

Tabela 5.6: Estrutura **rte_msg_head_t** .

Estrutura	Campos

`typedef struct {`
`  int tsk_id;`
`  long unsigned data_size;`
`} __attribute__( ( packed ) )`
`rte_msg_head_t;`
	tsk_id: Identificador da tarefa;
	data_sz: Tamanho do(s) dado(s) a serem enviados;

Fonte: Próprio Autor

Pacote de Dados: O envio de mensagens entre o *Host* e um *Guest* necessita da definição de como uma tarefa ou um resultado deve ser enviado. A Tabela 5.7 apresenta a estrutura responsável por guardar as informações a serem enviadas.

Tabela 5.7: Estrutura **rte_pkg_t**.

Estrutura	Campos

`typedef struct {`
`  unsigned int tsk_id;`
`  size_t rte_default_sz;`
`  size_t rte_total_sz;`
`  size_t gst_id;`
`  unsigned int rte_pkg_count;`
`  unsigned int rte_pkg_idx;`
`  void* rte_data;`
`} __attribute__( ( packed ) )`
`rte_pkg_t;`
	tsk_id: Identificador da tarefa;
	rte_default_sz: Tamanho padrão da mensagem;
	rte_total_sz: Tamanho total da mensagem;
	gst_id: Identificador do *Guest* destinatário;
	rte_pkg_count: Número Total de pacotes;
	rte_pkg_idx: Indice do pacote enviado;
	rte_data: Cadeia de *bytes* contendo os dados a serem enviados.

Fonte: Prório Autor

5.2.1.3 Fila de envio de pacotes

Cada **Estrutura de Envio de Mensagens** é colocada em uma fila. No caso do *Host*, ele armazena na fila para envio aos *Guests*. Esta estrutura é recebida pelo *Guest* e é armazenada na fila de execução.

A Tabela 5.8 apresenta a definição proposta para esta estrutura. Ela contém uma fila circular de tamanho fixo de elementos definido pela macro **RTE_MAX_RING_BUFFER**, com o tamanho virtual da fila e a posição atual do elemento para leitura/escrita.

Tabela 5.8: Estrutura **rte_ring_buffer_t**.

Estrutura	Campos

`typedef struct {`
`  rte_pkg_t buffer[RTE_MAX_RING_BUFFER];`
`  size_t sz;`
`  size_t pos;`
`} __attribute__( ( packed ) )`
`rte_ring_buffer_t;`
	buffer: Dados a serem enviados/executados;
	sz: Tamanho virtual do `buffer`;
	pos: Posição de Leitura e/ou Escrita.

Fonte: Prório Autor

5.2.2 Funções

A **librte_comm** também define um conjunto de funções, como uma abstração das funções da API *Portable Operating System Interface* (POSIX) *Sockets* Linux, para que possam ser utilizadas nos módulos do sistema RTE.

5.2.2.1 Criação do *Guest*

Esta função é responsável pela atribuição dos campos que compõem o endereço para a criação de um *Guest*. A Tabela 5.9 apresenta a sintaxe desta função. Ela retorna o valor lógico **Verdadeiro** caso a criação do *socket* tenha sucesso.

Tabela 5.9: Sintaxe da função **rte_comm_srv_guest_creat**.

Sintaxe	Parâmetro
bool `rte_comm_srv_guest_creat(` `rte_srv_guest_t *rte_srv_gst );`	**rte_srv_gst**: Registro do tipo `rte_srv_guest_t` contendo os dados para a criação do *socket Guest*.

Fonte: Prório Autor

A configuração do *socket* assincrono e não bloqueante é feita através das chamadas das sub-rotinas **setsockopt** e **fcntl** internamente nessa função como pode ser observado no trecho de código da Figura 5.2. Inicialmente são feitas duas chamadas da função **setsockopt**. A primeira, com os argumentos **SOL_SOCKET** e **SO_REUSEADDR** liga o modo de reutilização para o identificador do *socket* **rte_guest->listen_id**. Já a segunda chamada, com os argumentos **IPPROTO_TCP** e **TCP_NODELAY** ativa o *socket* para enviar as mensagens imediatamente sem a necessidade do preenchimento

total do *buffer* de envio do *socket*. Em seguida é feita chamada da função **fcntl** para obter as "*flags*" referentes ao identificador do descritor de arquivo do *socket* **rte_guest->listen_id** e em seguida liga-se o modo assíncrono através dos argumentos **F_SETFL** e **flags|O_ASYNC** para o mesmo identificador.

Figura 5.2: Parametrização do *socket Guest*.

```
1  const int on = 1;
2
3  if((setsockopt(rte_guest->listen_id, SOL_SOCKET, SO_REUSEADDR, (char*)&on , sizeof(on))==0) &&
4     (setsockopt(rte_guest->listen_id, IPPROTO_TCP, TCP_NODELAY, (char*)&on , sizeof(on))==0))
5  {
6     unsigned int flags = fcntl(rte_guest->listen_id, F_GETFL);
7     fcntl(rte_guest->listen_id, F_SETFL, flags|O_ASYNC); // set to asynchronous I/
```

Fonte: Prório Autor

5.2.2.2 Criação do *socket* de conexão do *Guest* no *Host*

Esta função é responsável pela atribuição dos campos que compõem o endereço para a criação de uma conexão com um *Guest*. A Tabela 5.10 apresenta essa sintaxe desta função.

Tabela 5.10: Sintaxe da função **rte_comm_guest_creat**.

Sintaxe	Parâmetro
bool rte_comm_guest_creat(rte_guest_conn_t* rte_guest);	**rte_guest**: Registro do tipo rte_guest_conn_t contendo os dados para criar a conexão com o *Guest*.

Fonte: Prório Autor

5.2.2.3 Função de conexão com o *Guest*

O *Host* deverá estabelecer uma conexão ao *socket Guest* já criado. A conexão do

Tabela 5.11: Sintaxe da função **rte_comm_guest_conn**.

Sintaxe	Parâmetro
int rte_comm_guest_conn(rte_guest_conn_t* rte_guest);	**rte_guest**: Registro do tipo rte_guest_conn_t contendo os dados para conectar com o *Guest*.

Fonte: Prório Autor

Host com o *Guest* é realizada através da função **rte_comm_guest_conn**, apresentada na Tabela 5.11.

5.2.2.4 Função de validação do *socket*

Após a criação de um *socket* deve-se validar seu identificador, tanto para a criação de um *Guest* quanto para estabelecer a conexão com o *Host*. Essa validação é feita através da função **rte_comm_is_valid**. A Tabela 5.12 apresenta a sintaxe desta função. Ela retorna o valor lógico **Verdadeiro** caso o *socket* possua um identificador válido.

Tabela 5.12: Sintaxe da função **rte_comm_is_valid**.

Sintaxe	Parâmetro
`bool rte_comm_is_valid( int rte_sckt_id );`	**rte_sckt_id**: Idetificador da conexão de rede.

Fonte: Prório Autor

5.2.2.5 Encerrar conexão

Ao finalizar as execuções das tarefas em um *Guest*, o *Host* solicitante encerra a conexão entre eles. A Tabela 5.13 mostra a sintaxe da função **rte_comm_close**.

Tabela 5.13: Sintaxe da função **rte_comm_close**.

Sintaxe	Parâmetro
`int rte_comm_close( int rte_gst_id );`	**rte_gst_id**: Identificador do *Guest*.

Fonte: Prório Autor

5.3 Biblioteca de chamadas de tarefas remotas

A implementação do módulo de chamadas para a execução das tarefas remotas segue a proposta descrita na Seção 4.1. É apresentada a biblioteca **librte** que deverá ser ligada na compilação ao programa desenvolvido pelo usuário para integração com o sistema RTE.

5.3.1 Tipos de dados

A **librte** faz uso de tipos de dados exclusivos para armazenar as informações necessárias para criação e conexão com cada *Guest*, e a utilização dos seus resultados.

5.3.1.1 Fila de *Guest*

O tipo **rte_gst_list_t** é definido como uma lista dos *Guests* que executarão as tarefas remotas. A Figura 5.3 apresenta a declaração do tipo de dados proposto. Este contém o campo **sz** que armazena o número total de itens presentes na lista, e lista dinâmica composta pelos elementos do tipo **rte_guest_conn_t** denominada **guests**.

Figura 5.3: Estrutura **rte_gst_list_t**.

```
typedef struct
{
    size_t sz;
    rte_guest_conn_t* guests;
} __attribute__((packed)) rte_gst_list_t;
```

Fonte: Próprio Autor

5.3.1.2 Sincronismo e resultados das tarefas remotas

Para armazenar cada resultado de uma tarefa executada remotamente propõem-se uma estrutura que receberá o resultado emitido de cada *Guest*.

A Figura 5.4 exibe a estrutura responsável por armazenar os resultados da execução das tarefas.

Figura 5.4: Estrutura **rte_wait_chk_t**.

```
typedef struct
{
    sem_t sync;
    void* resp;
} __attribute__((packed)) rte_wait_chk_t;
```

Fonte: Próprio Autor

5.3.2 Variáveis globais

Define-se um conjunto de variável globais para garantir a compatibilidade e compartilhamento de dados entre os componentes da **librte**.

A Tabela 5.14 apresenta a lista das declarações dessas variáveis globais.

Tabela 5.14: Lista de variáveis globais da **librte**.

Tipo	Nome	Descrição
`rte_gst_list_t`	`gsts_list;`	Lista encadeada responsável por armazenar os dados de cada *Guest* que será utilizado pela aplicação.
`rted_ring_buffer_t`	`snd_to_gsts;`	Fila circular, do tipo **rted_ring_buffer_t**, descrita na Seção 5.2.1.3, armazenará as tarefas para serem enviadas e executadas remotamente para cada *Guest* específico.
`rte_wait_chk_t`	`rte_recv_buffer[ _POSIX_THREAD_THREADS_MAX ]`	Lista estática responsável por armazenar os resultados recebidos de cada *Guest*.

Fonte: Próprio Autor

5.3.3 Funções

A **librte** também define um conjunto de funções que fazem a interface da aplicação paralela desenvolvida pelo usuário com os recursos de execução de tarefas remotas disponíveis no sistema RTE.

5.3.3.1 Inicialização

Como mostrado na Seção 4.1.1.1, para a execução de tarefas no sistema RTE há a necessidade de realizar o procedimento de inicialização. A Figura 5.5 apresenta a sintaxe desta função. O parâmetro **rte_cfg_file** contém o nome do arquivo de configuração, este arquivo armazena a lista dos *Guests* que serão responsáveis pelas

Figura 5.5: Sintaxe da função **rte_init**.

```
int rte_init(const char* rte_cfg_file)
```

Fonte: Próprio Autor

execuções remotas das tarefas definidas pelo usuário.

Primeiramente, a função de inicialização insere cada *Guest* na lista **gsts_list**, mostrada na Seção 5.3.2. Em seguida após criar cada conexão, com a função **rte_comm_guest_creat**, conecta cada *Guest*.

O procedimento de inicialização também cria para cada *Guest* as seguintes *threads*:

***Thread* de envio de tarefas**

Tem a finalidade de enviar ao *Guest* as tarefas a serem executadas. A Figura 5.6 mostra sintaxe desta *thread*. Enquanto houver uma requisição na fila **snd_to_gsts**, retira uma tarefa e a envia ao *Guest* destinatário.

Figura 5.6: Sintaxe da *thread* **rte_tasks_sender**.

```
void* rte_tasks_sender()
```

Fonte: Próprio Autor

***Thread* de Recepção de resultados**

Responsável pela recepção do resultado da execução da tarefa remota. Aguarda a recepção da resposta de cada Guest que foi solicitado. A Figura 5.7 apresenta a sintaxe desta *thread*. Ao receber o resultado da execução da tarefa, atualiza o conteúdo no endereço presente na lista **rte_recv_buffer** e o disponibiliza para sua utilização.

Figura 5.7: Sintaxe da *thread* **rte_recv**.

```
void* rte_recv(void* ptr_rte_connfd )
```

Fonte: Próprio Autor

5.3.3.2 Chamada de tarefa

A implementação da função **rte_tsk_call**, conforme sua definição na Seção 4.1.1.2, é apresentada nesta seção. A Figura 5.8 mostra a sintaxe desta função.

Figura 5.8: Sintaxe da função **rte_tsk_call**.

```
int rte_tsk_call( const unsigned int gst_id, const unsigned int dvc, const char*
    tsk_name, const char* tsk_set_name, const char* argin, const size_t sz_argin,
    const size_t sz_resp, void* resp )
```

Fonte: Próprio Autor

A **rte_tsk_call** empacota cada tarefa e em seguida faz a chamada da função estática para a inclusão de cada tarefa na lista **snd_to_gsts**. A Figura 5.9 mostra a sintaxe desta função.

Figura 5.9: Sintaxe da função **rte_write_tsk**.

```
static void rte_write_tsk( rted_pkg_t rted_pkg )
```

Fonte: Próprio Autor

5.3.3.3 Sincronização das tarefas

Conforme está definida na Seção 4.1.1.3, para a utilização do resultado de cada tarefa deve-se aguardar a sua conclusão e o seu retorno. Isto é feito através da sincronização do fluxo de controle da execução de cada tarefa na lista **rte_recv_buffer**.

A Figura 5.10 apresenta a sintaxe da função **rte_sync**, que recebe como parâmetro o identificador da tarefa na lista **rte_recv_buffer**. O controle da sincronização de cada tarefa é feita através do campo **sync**, "semáforo" responsável pela sincronização da tarefa.

Figura 5.10: Sintaxe da função **rte_sync**.

```
int rte_sync(int tsk_id)
```

Fonte: Próprio Autor

5.3.3.4 Finalização

Como mostrado na Seção 4.1.1.4, ao terminar as execuções das tarefas definidas pelo usuário e o recebimento de seus resultados, a aplicação RTE é finalizada. A função apresentada na Figura 5.11 é responsável por esta finalização.

Figura 5.11: Sintaxe da função **rte_finalize**.

```
void rte_finalize()
```

Fonte: Próprio Autor

Após encerrar as variáveis de controle é chamada a função **rte_comm_close**, descrita na Seção 5.2.2.5, para fechar a conexão com cada *Guest*.

5.4 Gerenciador de execução de tarefas remotas

Este módulo do sistema RTE, **rted**, é responsável pelo gerenciamento de tarefas enviadas pela aplicação paralela desenvolvida pelo usuário. O programa executável **rted** deve estar instalado em cada *Guest*, aguardando a solicitação para sua inicialização e execução.

5.4.1 Tipos de dados

O módulo **rted** define tipos de dados para armazenar as informações necessárias para a execução das tarefas e as informações dos seus resultados para envio como resposta ao *Host* solicitante.

5.4.1.1 Filas circulares

Propõem-se dois tipos de dados necessários para armazenar as informações para a execução da tarefa no *Guest* e guardar as suas respostas para enviar ao *Host*.

A Figura 5.12 apresenta as filas circulares utilizadas no módulo **rted**. A Fi-

Figura 5.12: Filas utilizadas no módulo **rted**.

(a) Fila de execução de tarefas

```
typedef struct {
   rte_pkg_t buffer[
      RTE_MAX_RING_BUFFER ];
   size_t sz;
   size_t pos
} __attribute__((packed))
  rte_ring_buffer_t;
```

(b) Fila de envio de respostas

```
typedef struct {
   rte_resp_pkg_t buffer[
      RTE_MAX_RING_BUFFER ];
   size_t sz;
   size_t pos;
} __attribute__((packed))
  rte_resp_ring_buffer_t;
```

Fonte: Próprio Autor

gura 5.12a mostra a estrutura para armazenar as tarefas a serem processadas. Já a Figura 5.12b exibe a estrutura responsável por guardar os resultados de cada tarefa executada para enviar ao *Host*.

As estruturas são muito parecidas. Possuem o campo **buffer** como uma fila estática, com o tamanho fixado pela macro **RTE_MAX_RING_BUFFER**. Na **Fila de Execução de Tarefas** esse campo é do tipo **rte_pkg_t**, e armazena as informações para execução de cada tarefa. E a **Fila de Envio de Respostas** o campo é do tipo **rte_resp_pkg_t**, e guarda os resultados para serem enviados como resposta ao *Host*. Ambos os tipos propostos possuem dois campos **sz** e **pos** responsáveis por atualizar, respectivamente, o tamanho virtual e a posição do próximo elemento na fila circular.

5.4.2 Variáveis gobais

Para manter o compartilhamento e garantir a compatibilidade das informações entre os componentes do **rted** define-se um conjunto de variáveis globais. A Tabela 5.15 apresenta a lista dessas variáveis no módulo **rted**.

Tabela 5.15: Lista de variáveis globais do **rted**.

Tipo	Nome	Descrição
rted_ring_buffer_t	rted_ring_buffer;	Armazena as tarefas para serem executadas.
rted_resp_ring_buffer_t	rted_resp_ring_buffer;	Armazena os resultados para serem enviados para cada *Host* que solicitou a execução da tarefa.

Fonte: Próprio Autor

5.4.3 Conjunto de tarefas remotas

Conforme está descrito na Seção 4.1.2.2, um **Conjunto de Tarefas Remotas** no sistema RTE é uma biblioteca dinâmica desenvolvida pelo usuário da aplicação paralela, e deverá estar presente nos *Guests* em que as tarefas serão executadas.

Neste trabalho adotou-se o formato Unix como padrão de declaração dos conjuntos de tarefas. Neste ambiente as bibliotecas dinâmicas são denominadas de Objetos Compartilhados, *Shared Objects* em inglês, normalmente identificados com a extensão de seus arquivos com **.so**.

5.4.4 Sistema de execução de tarefas

O módulo **rted**, como mencionado anteriormente, é um programa que deverá estar em execução em cada *Guest* aguardando a conexão com o *Host* e a solicitação para o processamento de uma tarefa. A execução de uma tarefa é feita a través da carga dinâmica do Conjunto de Tarefas. Para essa finalidade foi utilizada a API *Dynamic Loading* (DL) (JONES, 2008)

5.4.4.1 Bloco principal

É o programa principal deste módulo. Esse bloco é responsável por criar o *socket* de cada *Guest*, através da chamada da função **rte_comm_srv_guest_creat** e, após a criação do socket, aguarda a solicitação de conexão com o *Host* através da função **rte_comm_host_accept**. O programa principal, também, inicializa o **Bloco de Execução de Tarefas** e o **Bloco de Envio de Repostas**, descritos a seguir.

5.4.4.2 Bloco de execução de tarefas

A finalidade deste bloco é de aguardar a solicitação e a execução de tarefas encaminhadas pelo *Host*. A Figura 5.13 mostra a sintaxe desta função.

Figura 5.13: Sintaxe da função **rted_listen**.

```
void* rted_listen( void* ptr_rte_connfd )
```

Fonte: Próprio Autor

A função **rted_listen** aguarda o envio de um pacote pelo *Host* e em seguida chama a função estática, apresentada na Figura 5.14, para a inclusão na lista **rted_ring_buffer**.

Figura 5.14: Sintaxe da função **rted_write_tsk**.

```
void* rted_write_tsk( rted_pkg_t rted_pkg )
```

Fonte: Próprio Autor

Esta função associa para cada tarefa uma *thread*, apresentada na Figura 5.15, para a execução. Esta *thread* faz a chamada para a API DL. Inicialmente abre o

conjunto de tarefas com a função **dlopen** que retorna um identificador. A função **dlsym** recebe como parâmetros o identificador e o nome da tarefa a ser executada, e retorna um ponteiro de função para o processamento da tarefa. Ao finalizar a execução da tarefa seu resultado é guardado na fila **rted_resp_ring_buffer**.

Figura 5.15: Sintaxe da *thread* **rted_tsk_exec**.

```
void* rted_tsk_exec( )
```

Fonte: Próprio Autor

5.4.4.3 Bloco de envio de repostas

Thread responsável de enviar ao *Host* os resultados das tarefas executadas. A Figura 5.16 mostra sintaxe desta *thread*. Enquanto a fila **rted_resp_ring_buffer** não estiver vazia, retira uma resposta, empacota e a envia ao *Host*.

Figura 5.16: Sintaxe da *thread* **rted_replay**.

```
void* rted_replay( )
```

Fonte: Próprio Autor

6 TESTES E ANÁLISES

Este capítulo apresenta os testes efetuados para avaliar o modelo de programação orientado a tarefas remotas proposto nesse trabalho. Os resultados e análises experimentais aqui apresentados mostram a viabilidade de sua utilização.

Plataforma de teste

Os testes foram realizados em um sistema distribuído composto por 5 computadores homogêneos com processador Intel Core i7 de 3,4GHz, quad-core, com 16GB de RAM, interconectados numa rede Gigabit Ethernet. Cada nó possui a seguinte configuração de *software*: sistema operacional Ubuntu 22.04.1 LTS com *kernel* 5.15.0-48-generic e compilador gcc versão 7.5.0-6.

Para os testes de paralelização foram utilizadas as diretivas do OpenMP 4.5 na versão do gcc 7.5.0-6 e nos testes com MPI os programas foram compilados e executados com o OpenMPI 4.1.4. Todos os executáveis utilizaram a *flag* de otimização -O3 em suas compilações.

Os testes cujas análises envolveram tempo de execução foram executados 12 vezes e, após a eliminação dos *outliers*, calculou-se as médias aritméticas, os desvios padrões. O cálculo do intervalo de confiança foi feito através da distribuição t de *Student* de 95% dos tempos de execução.

A organização deste capítulo está disposta da seguinte forma: A Seção 6.1 apresenta os testes e análises de funcionalidade dos recursos da API RTE. A Seção 6.2 mostra a avaliação do custo de comunicação. A Seção 6.3 mostra a avaliação de desempenho com alocação dinâmica de dados na execução de tarefas sequenciais. A Seção 6.4 também avalia o desempenho com alocação dinâmica de dados mas com a execução de tarefas paralelas. A Seção 6.5 apresenta o resultado da comparação de desempenho entre implementações com o uso de RTE e MPI. A Seção 6.6 analisa o desempenho em uma aplicação real utilizando recursos de programação distribuída da ferramenta RTE, e também compara às outras implementações MPI. E por fim a

Seção 6.7 avalia a funcionalidade dos recursos da API RTE em um ambiente com arquiteturas heterogêneas.

6.1 Testes de funcionalidade

O objetivo dos testes realizados nessa seção é avaliar o funcionamento no uso dos recursos oferecidos pelo modelo de programação proposto nesse trabalho e a API RTE.

6.1.1 Descrição dos testes

Foram realizados testes visando verificar o funcionamento e o comportamento do modelo de programação através de implementações utilizando a API RTE para efetuar a chamada e a execução de uma tarefa em um *Guest*, local ou remoto, com o uso de variáveis escalares e vetoriais. Os cenários de testes utilizados na análise do funcionamento são descritos a seguir:

Soma Multiplicação Escalar Local(*Host*): O objetivo deste cenário é apresentar o funcionamento correto de múltiplas chamadas de tarefas a serem executadas simultaneamente de forma local pelos múltiplos núcleos disponíveis nos processadores do *Host*. Executou-se o programa que realiza uma soma e uma multiplicação de duas variáveis escalares do tipo inteiro através da chamada e execução de uma *thread* criada pela API RTE internamente no próprio *Host*;

Soma Multiplicação Escalar Remota: Nesse cenário o objetivo é semelhante ao cenário anterior, porém verifica o funcionamento em ambiente distribuído através das chamadas das tarefas pelo *Host* para serem executadas no *Guest*. O código do programa **`soma-mult_escalar_rem.c`** está no Apêndice A.1;

Multiplicação de Matrizes Local(*Host*): O objetivo deste cenário é mostrar o funcionamento correto do modelo de programação com a alocação estática de dados. É feita a chamada da tarefa, da mesma forma como no primeiro cenário, para a execução da tarefa através de uma *thread* nos múltiplos núcleos disponíveis nos processadores do *Host*;

Multiplicação de Matrizes Remota: O objetivo nesse cenário é semelhante ao anterior, porém a execução é feita em um ambiente distribuído. No *Host* o programa

faz a chamada da tarefa de multiplicação das duas matrizes para a execução no *Guest*;

Multiplicação de Matrizes Remota com Alocação Dinâmica de Dados: Nesse cenário o objetivo é mostrar o funcionamento correto da alocação dinâmica de dados através de cinco execuções distintas do programa. Em cada execução é passado como argumentos duas matrizes de tamanho dinâmico variando de 100×100, 1000×1000, 2000×2000, 3000×3000 e 4000×4000. É realizada a chamada da tarefa pelo *Host* a ser executada em um *Guest* e o retorno dessa execução ao *Host* é a matriz resultante com o mesmo tamanho das matrizes passadas nos argumentos de entrada.

6.1.2 Análise dos resultados

Os testes realizados em todos os cenários mostraram o funcionamento conforme o esperado, alcançando os objetivos apresentados nessa seção.

6.2 Avaliação do custo de comunicação

O caso de teste descrito nessa seção tem como objetivo avaliar o custo de comunicação referente ao uso de uma aplicação distribuída utilizando a API RTE.

6.2.1 Descrição dos testes

Para avaliar o custo de comunicação no modelo de programação proposto definiu-se aplicações sequenciais de multiplicação de matrizes com duas dimensões quadradas com o tamanho de 2000×2000.

A análise de funcionamento e a comparação de tempos de comunicação foram realizadas em quatro cenários distintos, como descrito a seguir:

Local(*Host*): Faz a chamada da tarefa de multiplicação das duas matrizes em uma *thread* no *Host*;

Execução Remota: Nesse cenário de teste o programa realiza a chamada da tarefa responsável pela multiplicação das duas matrizes de forma distribuída. Assim sendo o *Host* envia a tarefa para ser executada no *Guest*;

Execução Remota(*Host*): Este cenário é semelhante ao anterior, porém faz o envio da tarefa do *Host* para ser executada localmente no próprio *Host*. O código do programa encontra-se no Apêndice B.1;

Execução Sequencial: O programa executa uma função sequencial, sem o uso dos recursos de comunicação da API RTE, para multiplicação das duas matrizes estáticas com o tamanho de 2000×2000 e tem como resposta a matriz resultante com o mesmo tamanho. O código do programa encontra-se no Apêndice B.2.

Tarefa Sequencial de Multiplicação de 2 Matrizes: A tarefa RTE, na forma de função sequencial, faz a multiplicação de duas matrizes e retorna a matriz resultante ao *Host*.

6.2.2 Análise dos resultados

A análise dos resultados visa verificar o comportamento do modelo de programação, quanto ao desempenho e em especial o custo de comunicação. Os tempos de execução são medidos em segundos. A Tabela 6.1 apresenta a média e o desvio padrão dos tempos obtidos em cada versão dos testes de custo de comunicação. Em cada teste é chamada uma única tarefa, que internamente é sequencial. Esta tarefa, nos diversos cenários, foi executada exclusivamente em um único e mesmo nó do ambiente de teste. Foi realizado desta forma para garantir a compatibilidade das execuções em todos cenários.

Tabela 6.1: Tempo de execução em cada teste.

Teste	Média	Desvio Padrão
Execução Local(Host)	4,211189	0,010599
Execução Remota	5,035680	0,009067
Execução Remota(Host)	4,237085	0,007884
Execução Sequencial	4,192104	0,011553

Fonte: Próprio Autor

Tem-se os seguintes cálculos de tempo para as execuções:

Local(*Host*): $t_{lc} = t_{tsk} + t_{lc\_tsk\_call}$ (6.1a)

Remota: $t_{rem} = t_{tsk} + t_{rem\_tsk\_call} + t_{rem\_comm}$ (6.1b)

Remota(*Host*): $t_{rem\_lc} = t_{tsk} + t_{rem\_tsk\_call} + t_{lc\_comm}$ (6.1c)

Sequencial: $t_{seq} = t_{func}$ (6.1d)

Onde:

$\boldsymbol{t_{tsk}}$**:** tempo de execução da tarefa;

$\boldsymbol{t_{lc\_tsk\_call}}$**:** tempo da chamada e controle para execução da tarefa localmente no *Host*;

$\boldsymbol{t_{rem\_tsk\_call}}$**:** tempo da chamada e inserção da solicitação da tarefa na fila de envio para nós remotos;

$\boldsymbol{t_{rem\_comm}}$**:** tempo do envio das informações e dos parâmetros da tarefa para o nó remoto e tempo do recebimento da resposta;

$\boldsymbol{t_{lc\_comm}}$**:** tempo do envio das informações e dos parâmetros da tarefa para o próprio *Host* e do recebimento da resposta;

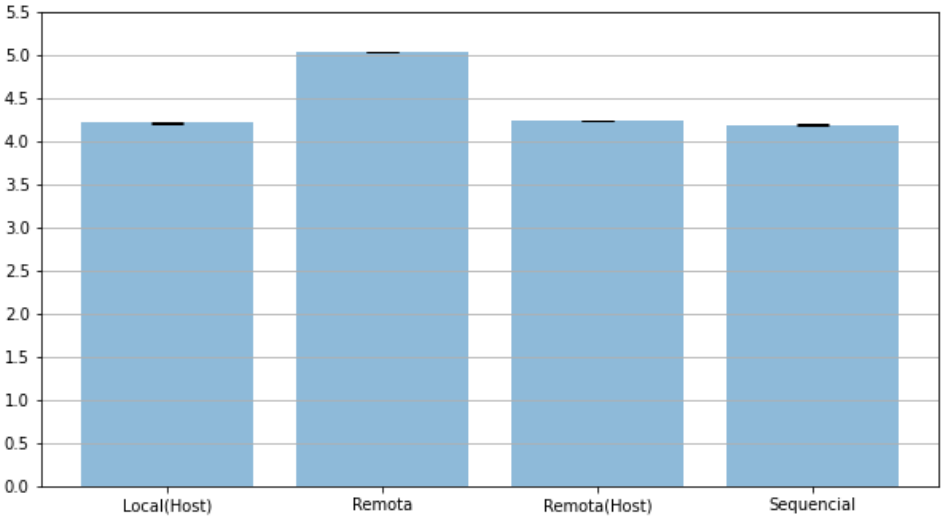
$\boldsymbol{t_{func}}$**:** tempo de execução da função sequencial.

Pelos tempos apresentados na Tabela 6.1, verifica-se um aumento de 0,46% entre os testes **Execução Local(*Host*)** e **Execução Sequencial**. Tal aumento se deve ao custo do gerenciamento da chamada e o controle da execução da tarefa (cálculo 6.1a). A diferença entre o tempo da **Execução Remota(*Host*)** e o da **Execução Local(*Host*)** é de 0,61%. Já a diferença que ocorre na comparação dos tempos dos testes **Execução Remota** e **Execução Sequencial** é de 20,12%. Num cenário em que não são executadas tarefas simultaneamente, o uso do RTE não é vantajoso, apresentando ganhos em cenários em que existam esta simultaneidade, o que é apresentado e analisado na Seção 6.3.

Conforme mostra a Figura 6.1 e a Tabela 6.1 tem-se uma diferença de 18,84% entre os tempos de **Execução Remota** (t_{rem}) e **Execução Remota(*Host*)** (t_{rem_lc}), que se deve ao tempo gasto para o envio dos parâmetros da tarefa (t_{rem_comm}), uma

vez que o t_{lc_comm} é práticamente nulo, presentes nos cálculos (6.1b) e (6.1c). No teste **Execução Remota** são enviados o total de *bytes* de cada uma das matrizes de entrada e seus elementos e o total de *bytes* da matriz resultante.

Figura 6.1: Tempo de execução em cada teste.



Fonte: Próprio Autor

6.3 Análises do custo de comunicação e do desempenho com chamadas de tarefas sequenciais

O objetivo desta seção é avaliar o comportamento do modelo de programação através da API RTE quanto ao custo de comunicação e o desempenho da execução de aplicações que executam tarefas sequenciais variando a quantidade de dados dos parâmetros das tarefas e o número de chamadas das execuções simultâneas nos múltiplos *Guests*.

6.3.1 Descrição e resultado dos testes

As avaliações foram feitas em implementações com chamadas de tarefas a serem executadas simultaneamente por múltiplos *Guests*, sendo em cada tarefa processada uma versão sequencial da multiplicação de matrizes quadradas. O tamanho das matrizes são alocadas dinamicamente em suas respectivas áreas de memória durante o processamento. Os dez cenários de testes, com os seus respectivos programas escritos em linguagem C, são mostrados a seguir:

Teste 1 **1 Tarefa Local(*Host*)**: É feita a chamada da tarefa de multiplicação das matrizes em uma *thread* internamente no *Host*;

Teste 2 **1 Tarefa Remota**: Assim como no cenário anterior executa-se uma tarefa, mas essa é feita através da chamada remota ao *Guest* pelo *Host*;

Teste 3 **2 Tarefas Remotas**: Executa 2 chamadas simultâneas, uma para cada *Guest* específico com duas matrizes em cada;

Teste 4 **3 Tarefas Remotas**: Este cenário é semelhante ao anterior, porém faz três chamadas paralelas de multiplicação de seis matrizes;

Teste 5 **3 Tarefas Remotas + 1 Tarefa Local(*Host*)**: Além das três execuções simultâneas remotas é feita a execução de uma tarefa em uma *thread* interna do próprio *Host*;

Teste 6 **4 Tarefas Remotas**: Faz 4 chamadas paralelas em cada *Guest* para a multiplicação simultânea de oito matrizes, duas para cada, o código do programa encontra-se no Apêndice C.1;

Teste 7 **1 Função Sequencial**: É executada a chamada de uma função para a multiplicação de duas matrizes;

Teste 8 **2 Funções Sequenciais**: Executa-se 2 chamadas sequenciais distintas da função de multiplicação de matrizes;

Teste 9 **3 Funções Sequenciais**: Executa-se 3 chamadas sequenciais distintas da função de multiplicação de matrizes;

Teste 10 **4 Funções Sequenciais**: Executa-se 4 chamadas sequenciais distintas da função de multiplicação de matrizes, o código do programa encontra-se no Apêndice C.2.

Cada cenário foi executado em nove versões com os tamanhos das matrizes variando de 100×100, 1000×1000, 2000×2000, 3000×3000, 4000×4000, 5000×5000, 6000×6000, 7000×7000 e 8000×8000. A Tabela 6.2 mostra os tempos médios das execuções de tarefas sequenciais. As análises referentes aos dados obtidos nessa tabela serão apresentadas nas próximas seções.

Tabela 6.2: Tempos de execuções das tarefas sequenciais.

	100×100		1000×1000		2000×2000		3000×3000		4000×4000		5000×5000		6000×6000		7000×7000		8000×8000	
Teste	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão
1	0,001261	0,000220	0,467166	0,003406	4,416204	0,006525	15,461110	0,025806	36,545740	0,038065	71,601110	0,133818	123,455885	0,204765	196,184725	0,201421	292,247588	0,238153
2	0,005637	0,000597	0,680867	0,003444	5,107348	0,039420	16,835062	0,025024	38,851212	0,084878	74,911742	0,289555	128,202974	0,611525	202,218370	0,956351	301,167701	1,558637
3	0,009752	0,004057	0,821396	0,003459	5,587263	0,014518	18,052770	0,016257	40,954425	0,071118	78,195031	0,277857	134,109372	0,217759	210,719008	0,275543	310,391060	1,342522
4	0,010803	0,000579	0,955536	0,001361	6,181536	0,047132	19,317586	0,149287	43,397167	0,174370	81,799362	0,322674	138,041476	0,563150	214,933517	0,351248	318,617948	1,793198
5	0,012295	0,004449	0,954324	0,001395	6,209007	0,038743	19,385045	0,161538	43,258094	0,149219	81,599527	0,272463	138,048180	0,581482	215,454753	0,713957	318,363864	1,314421
6	0,015728	0,005207	1,093691	0,003666	6,764420	0,031884	20,549070	0,049214	45,276543	0,079852	84,981007	0,300193	142,365613	0,602905	222,643545	1,201105	326,047007	1,205835
7	0,000705	0,000085	0,463018	0,003109	4,394522	0,002966	15,434059	0,007897	36,555860	0,012171	71,633516	0,079060	123,321819	0,125872	196,088216	0,193948	291,839177	0,319970
8	0,001440	0,000202	0,913150	0,005874	8,788330	0,016614	30,865529	0,011933	73,173406	0,038380	143,656616	0,182989	248,115628	0,076768	394,948622	0,538700	590,712144	0,915429
9	0,002355	0,000158	1,370421	0,005861	13,017765	0,078864	46,253939	0,055553	109,717091	0,037767	215,354872	0,265495	372,914636	0,550127	593,368839	0,468244	887,282428	1,117003
10	0,002987	0,000143	1,829854	0,003620	17,572611	0,035565	61,761592	0,052602	146,540000	0,203292	287,630802	0,444075	498,349920	0,721440	795,316963	1,481918	1188,911717	1,609411

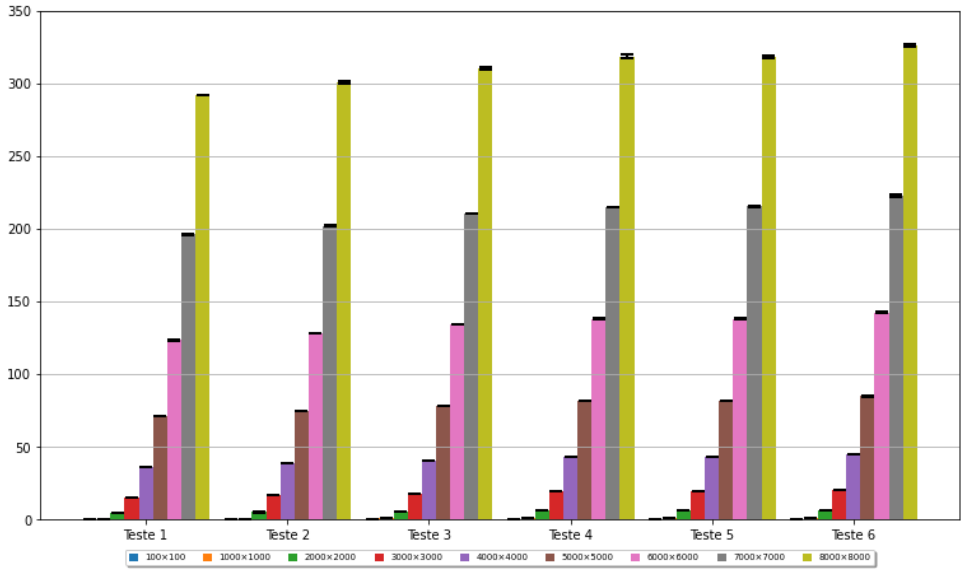
Fonte: Próprio Autor

6.3.2 Análise do comportamento do custo de comunicação com a variação da quantidade de dados dos parâmetros da tarefa

Tem-se aqui, o objetivo de analisar o tempo gasto na transmissão de dados com a variação da quantidade de dados dos parâmetros da tarefa. É analisado também o impacto deste tempo gasto no total da execução.

Nota-se na Figura 6.2 que no Teste 1, em que uma tarefa é chamada e executada no *Host*, o tempo de execução é um pouco menor do que o do Teste 2, em que esta tarefa é executada no *Guest*. Esta diferença é devido ao tempo gasto na transmissão das informações da tarefa e dos dados dos parâmetros pelo *Host* para o *Guest*.

Figura 6.2: Tempos de execuções das tarefas sequenciais.



Fonte: Próprio Autor

Como o Teste 1 não possui transmissão de dados, esse tempo é praticamente nulo pois sua execução é local. Assim sendo, pode-se assumir que o Tempo de Transmis-

são de Dados (DTT) entre o *Host* e um *Guest* é dado pela diferença dos tempos de execuções dos cenários de testes **"1 Tarefa Local(*Host*)"** e **"1 Tarefa Remota"** e é definido pela seguinte fórmula:

$$t_{dt} = t_{rem} - t_{lc} \tag{6.2}$$

O Custo de Comunicação (CC) de uma tarefa remota em um teste será adotado como o percentual do gasto na comunicação dos dados da tarefa remota (t_{rem}) executada no nó remoto em relação ao tempo de execução do teste executado localmente no *Host* (t_{lc}). Esta métrica permite avaliar o impacto do tempo total gasto em comunicação pela tarefa remota sobre o tempo de execução propriamente dito da tarefa, ou seja, sem incluir o tempo gasto com a comunicação dos dados referentes aos parâmetros da tarefa. É calculado pela seguinte fórmula:

$$CC(t_{rem}) = \frac{t_{rem} - t_{lc}}{t_{lc}} \cdot 100 \tag{6.3}$$

Onde:

t_{rem}**:** tempo de execução da chamada e execução da tarefa remota;

t_{lc}**:** refere-se ao tempo da chamada e execução da tarefa no *Host*.

A Tabela 6.3 apresenta os tempos de transmissão dos dados e os custos de comunicação calculados através da Fórmula 6.4 para os tamanhos das matrizes usados neste cenário de teste.

Tabela 6.3: Tempos de transmissão de dados.

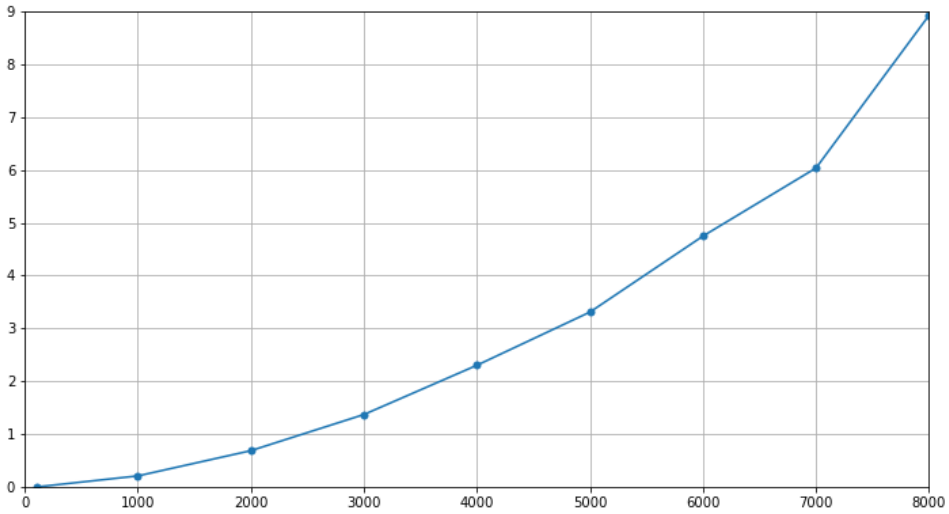
Tamanho	Teste 1	Teste 2	DTT	CC
100	0.001261	0.005637	0,0043762	346,986996511
1000	0.467166	0.680867	0,2137017	45,744314222
2000	4.416204	5.107348	0,6911441	15,650185800
3000	15.461110	16.835062	1,3739523	8,886505087
4000	36.545740	38.851212	2,3054724	6,308457292
5000	71.601110	74.911742	3,3106319	4,623715902
6000	123.455885	128.202974	4,7470883	3,845169705
7000	196.184725	202.218370	6,0336450	3,075491734
8000	292.247588	301.167701	8,9201135	3,052245383

Fonte: Próprio Autor

No gráfico apresentado na Figura 6.3, nota-se um crescimento do tempo gasto na

transmissão de dado com a variação do tamanho n das matrizes quadradas de $O(n^2)$.

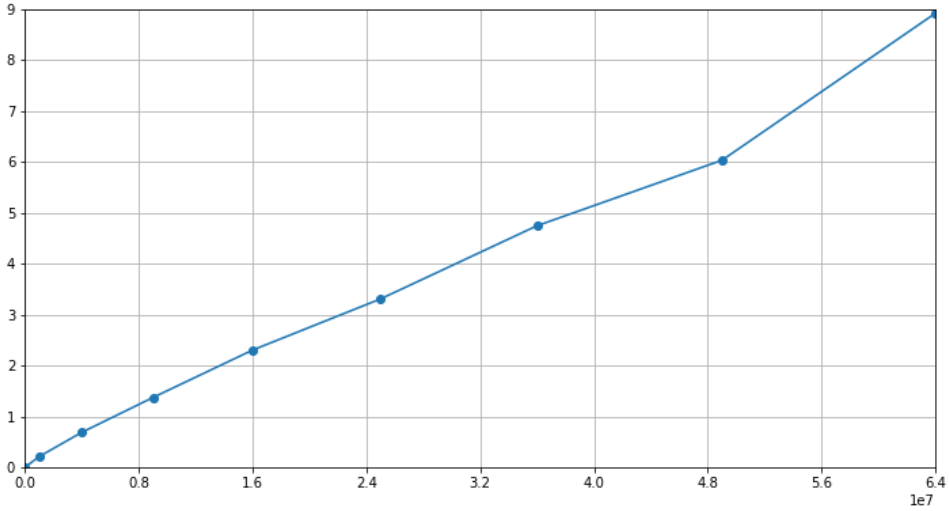
Figura 6.3: Tempos de transmissão de dados por tamanho das matrizes.



Fonte: Próprio Autor

Se considerar o total de elementos, ou seja $n$elementos = $n \times n$, o crescimento do tempo gasto será de $O(n\text{elementos})$, conforme mostra a Figura 6.4.

Figura 6.4: Tempos de transmissão de dados por número de elementos.



Fonte: Próprio Autor

Dado que a multiplicação de matrizes é da ordem de $O(n^3)$ o impacto do tempo gasto na comunicação dos dados é reduzido com o aumento de n, o que é confirmado

pelos valores $\boldsymbol{CC}$ apresentados na Tabela 6.3. Cabe ressaltar que na aplicação executada neste teste, o tempo gasto com a transmissão de dados é bem menor do que o tempo total da execução, o que torna o uso do modelo aqui proposto adequado para esta aplicação.

6.3.3 Análise do comportamento do custo de comunicação com a variação do número de tarefas remotas e a variação da quantidade de dados de parâmetros

Assim como na Seção 6.3.2, o objetivo aqui é analisar o tempo gasto na transmissão de dados com a variação da quantidade de dados dos parâmetros das tarefas, porém é analisado também o impacto deste tempo gasto no total da execução de múltiplas tarefas remotas.

A métrica do CC de múltiplas tarefas remotas executadas simultaneamente nos *Guests* será, adotada aqui, como o percentual do tempo gasto na comunicação dos dados do conjunto de tarefas remotas executadas simultaneamente em relação ao tempo de execução de uma tarefa executada no *Host*. É considerado que os tempos de execução propriamente de cada tarefa, ou seja, sem incluir a comunicação de dados, são iguais. Esta métrica permite avaliar o impacto do tempo total gasto em comunicação pelas múltiplas tarefas sobre o tempo de execução propriamente dito da tarefa. É calculado pela seguinte fórmula:

$$CC(t_{iTasks}) = \frac{t_{iTasks} - t_{lc}}{t_{lc}} \cdot 100 \tag{6.4}$$

Onde:

t_{iTasks}: tempo de execução das chamadas e execução das múltiplas tarefas remotas, em que i representa o número de tarefas;

t_{lc}: referente ao tempo de execução de uma tarefa chamada e executada no *Host*.

A Tabela 6.4 apresenta os Custos de Comunicação calculados para os tamanhos das matrizes usados neste cenário de teste para as execuções de múltiplas tarefas, de 1 até 4, executadas simultaneamente.

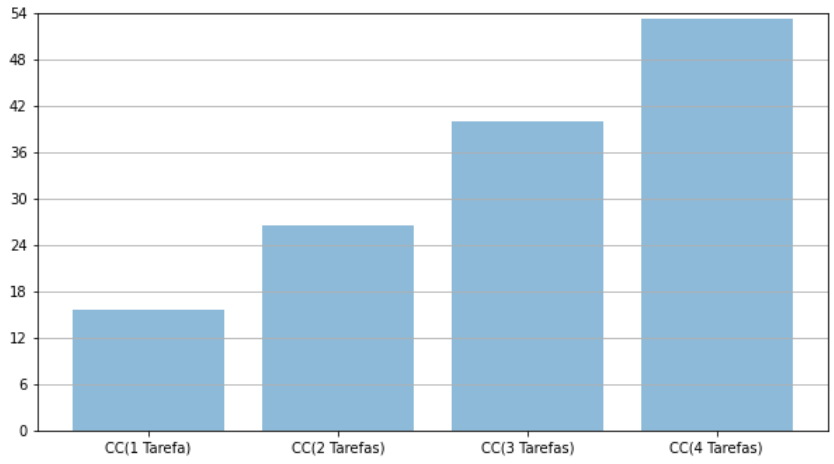
Tabela 6.4: Custo de comunicação de múltiplas tarefas.

	$CC(1Tarefa)$	$CC(2Tarefas)$	$CC(3Tarefas)$	$CC(4Tarefas)$
100	346,986996511	673,200126863	756,604820806	1147,050428164
1000	45,744314222	75,825339023	104,538968623	134,112143531
2000	15,650185800	26,517333734	39,973970404	53,172731748
3000	8,886505087	16,762444936	24,943077016	32,908121077
4000	6,308457292	12,063470872	18,747539385	23,890071729
5000	4,623715902	9,209245106	14,243148086	18,686717060
6000	3,845169705	8,629387067	11,814415218	15,316991534
7000	3,075491734	7,408468320	9,556703459	13,486686948
8000	3,052245383	6,208253881	9,023294569	11,565337520

Fonte: Próprio Autor

Considerando o tamanho de matrizes de 2000×2000 e a variação do número de tarefas pode-se observar na Figura 6.5 o aumento do impacto da comunicação dos dados na transmissão de dados do Teste 4, com 3 tarefas sendo chamadas e executadas em relação à chamada e execução de uma tarefa no Teste 2. Nota-se, também, que o aumento do custo de comunicação do Teste 4 com 3 tarefas, em relação ao do Teste 2, é 2 vezes maior do que o do Teste 3 com 2 tarefas, em relação ao do Teste 2. Tem-se, assim um crescimento linear do impacto da comunicação de dados em função do número de tarefas. Isto ocorre devido à serialização no envio dos pacotes de dados do *Host* para os nós remotos. Se no Teste 2 são enviados $n$dados então no Teste 3 são 2 vezes $n$dados e no Teste 4 são 3 vezes $n$dados.

Figura 6.5: Custo de comunicação de múltiplas tarefas para $n = 2000$.

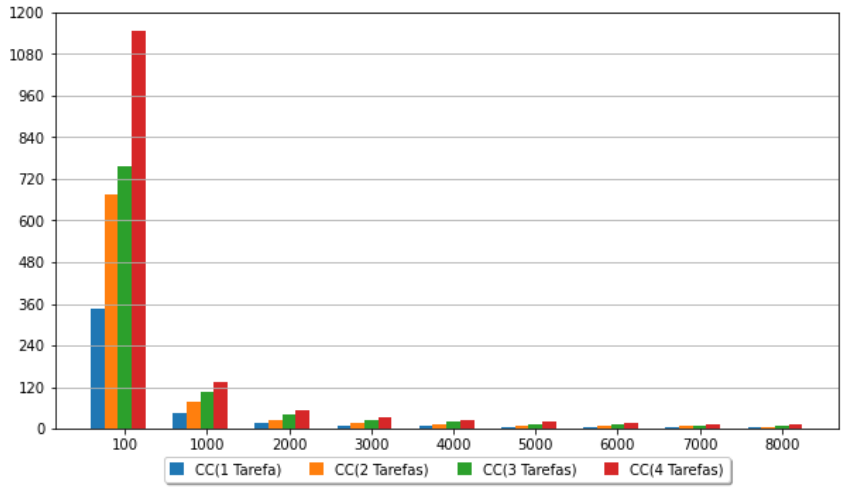


Fonte: Próprio Autor

Há apenas uma pequena diferença entre o impacto da comunicação dos dados entre as chamadas para 3 tarefas no Teste 4 em comparação com 4 tarefas no Teste 6, não acompanhando o comportamento anterior. Isto ocorre pois a quarta tarefa, embora chamada na forma remota, é executada no próprio *Host*, sem envio de dados através da rede de comunicação.

Analisando a variação do número de tarefas para matrizes de diferentes tamanhos observa-se no gráfico da Figura 6.6 o mesmo comportamento do custo de comunicação de dados com a variação do número de tarefas para todos os tamanhos de matrizes, daquele observado para o caso de $n = 2000$. A única diferença que se nota é a redução deste custo com o aumento de n, comportamento constatado anteriormente, na Seção 6.3.2.

Figura 6.6: Custo de comunicação de múltiplas tarefas.



Fonte: Próprio Autor

6.3.4 Análise de desempenho

Para avaliar o ganho no desempenho do modelo de programação realizou-se a comparação das execuções remotas com as execuções sequenciais equivalentes executadas no *Host*. Usou-se como parâmetros a variação do número de tarefas entre 1 e 4 e a quantidade de dados com o n variando de 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000 e 8000. Foram realizadas as comparações Teste 2 $\times$ Teste 7, Teste 3 $\times$ Teste 8, Teste 4 $\times$ Teste 9 e Teste 6 $\times$ Teste 10 com 1, 2, 3 e 4 tarefas respectivamente.

Para o cálculo de ganho de desempenho utilizou-se a seguinte fórmula:

$$S(t_{rem_i}) = \frac{t_{rem_i}}{t_{seq_i}} \tag{6.5}$$

Onde:

$\boldsymbol{t_{rem_i}}$**:** tempo de execução das chamadas e execução das múltiplas tarefas;

$\boldsymbol{t_{seq_i}}$**:** tempo de execução sequencial das funções, ou seja as funções são chamadas sequencialmente;

$\boldsymbol{i}$**:** número de tarefas ou execuções sequenciais.

A Tabela 6.5 apresenta os ganhos de desempenho encontrados nas comparações entre as execuções remotas e as execuções sequencias utilizando a Equação 6.5.

Tabela 6.5: Ganhos de desempenho.

	1 Tarefa	**2 Tarefas**	**3 Tarefas**	**4 Tarefas**
100	0,125022173	0,147668075	0,218021937	0,189937563
1000	0,680041177	1,111705871	1,434190999	1,673098737
2000	0,860431262	1,572922042	2,105911172	2,597800092
3000	0,916780627	1,709739232	2,394395383	3,005566252
4000	0,940919406	1,786703308	2,528208604	3,236554495
5000	0,956238828	1,837157847	2,632720688	3,384648089
6000	0,961926358	1,850099104	2,701468035	3,500493624
7000	0,969685473	1,874290438	2,760708734	3,572153706
8000	0,969025482	1,903122287	2,784784828	3,646442660

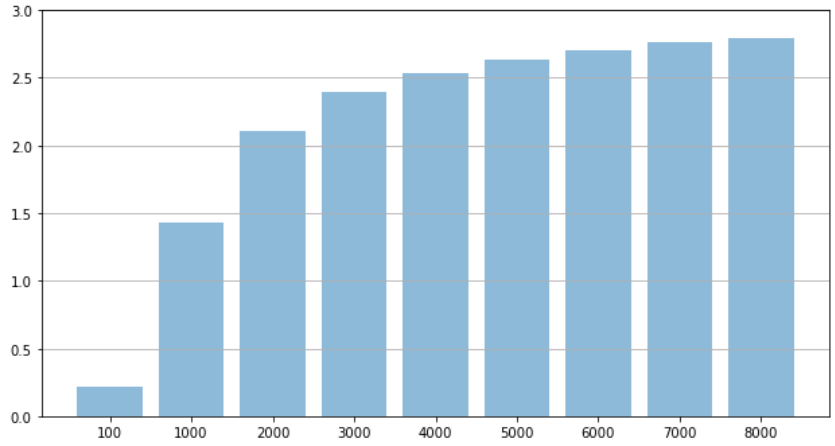
Fonte: Próprio Autor

Ao avaliar a variação do número de dados (n), verifica-se que o ganho de desempenho aumenta proporcionalmente ao valor do aumento do n em cada teste. A análise do gráfico, apresentada na Figura 6.7, referente a execução de 3 tarefas simultâneas pode-se constatar esse aumento do desempenho.

Mas ao comparar as execuções de cada n observa-se, tanto na Tabela 6.5 quanto na Figura 6.7, que o ganho de desempenho tende a uma aproximação a medida em que n é cada vez maior. Essa ocorrência é devido à complexidade do envio dos pacotes de dados, como mencionado na Seção 6.3.2, que é de $O(n^2)$ em comparação à complexidade do processamento da multiplicação de matrizes que é de $O(n^3)$. Constata-se que CC diminui sua relevância a medida que n aumenta, ou seja reduzindo o impacto do tempo gasto na comunicação no tempo da execução de cada

tarefa.

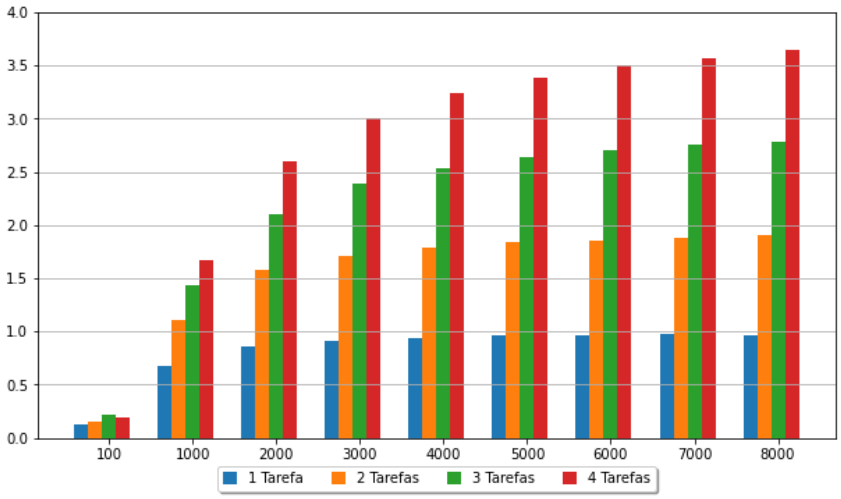
Figura 6.7: Ganhos de desempenho na execução de 3 tarefas simultâneas.



Fonte: Próprio Autor

Na Figura 6.8 nota-se o mesmo comportamento da evolução dos ganhos de desempenho obtidos, independente do número de tarefas e dos respectivos *Guests* utilizados.

Figura 6.8: Ganho de desempenho em cada teste.

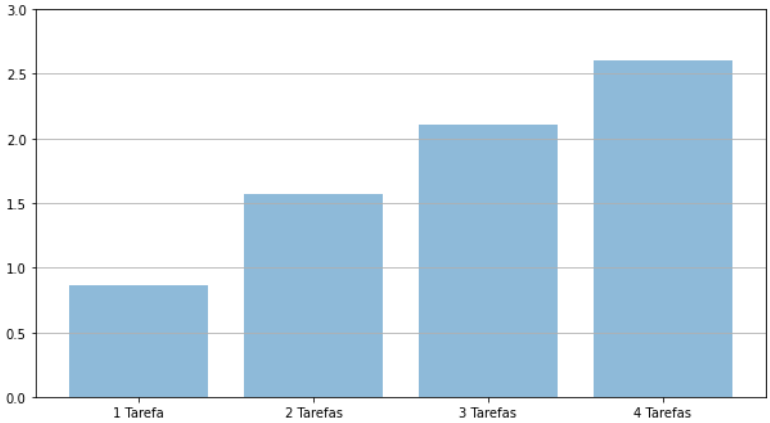


Fonte: Próprio Autor

A análise variando o número de tarefas foi realizada comparando as execuções de 1, 2, 3, e 4 tarefas com o $n = 2000$. Na Figura 6.9 observa-se um ganho no desempenho mais expressivo a medida em que se aumenta o número de tarefas. Portanto deduz-se que CC tem uma menor relevância quando se aumenta o número

de *Guests*.

Figura 6.9: Ganhos de desempenho na execução de tarefas com $n = 2000$.



Fonte: Próprio Autor

Conclui-se que o modelo proposto de tarefas remotas executadas em processamento remoto deve ser usado em aplicações em que as tarefas tenham um nível baixo de quantidade de dados de parâmetros de entrada e resposta a serem transmitidos em relação ao processamento envolvido. No caso da aplicação deste teste envolvendo multiplicação de matrizes este modelo, mesmo considerando o crescimento da quantidade de dados a serem transmitidos conforme aumenta o tamanho das matrizes, se mostra adequado.

6.4 Análises do custo de comunicação e do desempenho com chamadas de tarefas paralelas

Como apresentado na Seção 4.1.1 e implementados na Seção 5.3.3, a API RTE faz uso de duas *threads* de comunicação responsáveis pelo envio das tarefas e pelo recebimento das respostas.

Para avaliar o custo do processamento em paralelo em ambientes *multicores* de uma aplicação com paralelização interna define-se dois casos de testes, o primeiro para alocações de quatro *threads* e outro com alocação de duas *threads*. Cada caso analisou o custo de processamento em paralelo junto com a utilização das *threads* de comunicação, e assim como foram avaliados no caso de teste da Seção 6.3, serão analisados o comportamento de aplicações variando a quantidade de dados dos parâ-

metros das tarefas e o número de chamadas das execuções simultâneas nos múltiplos Guests, porém a execução de cada tarefa é feita com paralelização interna. Os dois casos de testes são necessários, também, para avaliaram a sobrecarga no processamento computacional, CPUs por exemplo. A paralelização interna dentro de cada tarefa foi feita através da API OpenMP.

O custo de comunicação é similar ao apresentado na Seção 6.3.2, na análise dos testes com tarefas sequenciais.

6.4.1 Descrição e resultado dos testes

A análise do comportamento e da funcionalidade com a variação da quantidade de dados dos parâmetros das tarefas e o número de chamadas das execuções simultâneas de tarefas remotas com paralelização interna foi realizada através de aplicações que fazem chamadas simultâneas de tarefas que executam a multiplicação de matrizes quadradas. O tamanho das matrizes são alocados em suas respectivas áreas de memória dinamicamente durante a execução.

Os dez cenários de testes são mostrados a seguir:

1. **1 Tarefa Local(*Host*)**: A chamada da tarefa é feita por uma *thread* internamente no *Host*;
2. **1 Tarefa Remota**: Tal qual no cenário anterior, faz a chamada e a execução de uma tarefa, porém é feita a chamada remota pelo *Host* ao *Guest*;
3. **2 Tarefas Remotas**: O programa executa 2 chamadas paralelas, uma para cada *Guest* específico, com duas matrizes em cada;
4. **3 Tarefas Remotas**: Neste cenário são feitas três chamadas paralelas de multiplicação de seis matrizes;
5. **3 Tarefas Remotas + 1 Tarefa Local(*Host*)**: Além das três execuções simultâneas remotas, uma tarefa é executada em uma *thread* interna da RTE no *Host*;
6. **4 Tarefas Remotas**: São feitas quatro chamadas para a execução de tarefas simultâneas com duas matrizes para cada *Guest*, o programa utilizado nesse teste encontra-se no Apêndice D.1;
7. **1 Função Paralela**: O programa faz a chamada da função com paralelização interna;

8. **2 Funções Paralelas**: Executa-se o programa com 2 chamadas sequenciais distintas da função com paralelização interna;
9. **3 Funções Paralelas**: Executa-se o programa com 3 chamadas sequenciais distintas da função com paralelização interna;
10. **4 Funções Paralelas**: Executa-se o programa com 4 chamadas sequenciais distintas da função com paralelização interna, o código do programa encontra-se no Apêndice D.2.

Utiliza-se o conjunto de tarefas no padrão do sistema RTE, exclusivamente para os cenários de 1 ao 6, para realizar a multiplicações de duas matrizes e responder a matriz resultante ao *Host*. Essa tarefa também utiliza paralelização interna, o programa encontra-se no Apêndice D.3.

Cada cenário foi executado em nove versões com os tamanhos das matrizes variando de 100×100, 1000×1000, 2000×2000, 3000×3000, 4000×4000, 5000×5000, 6000×6000, 7000×7000 e 8000×8000. Os cenários são os mesmos em ambos os casos de testes desta seção. Para realizar o caso de teste com duas *threads* substituiu o valor da macro **NTHREADS** de 4 para 2 nos programas **mult-par-dynamic.c**, **mult-2par-dynamic.c**, **mult-3par-dynamic.c**, **mult-4par-dynamic.c**, **parallel_matrix_mult_task.c**.

A Tabela 6.6 apresenta os tempos alcançados com as execuções de tarefas para-

Tabela 6.6: Tempos das execuções de tarefas com paralelização interna.

(a) Com 4 *Threads*.

	100×100		1000×1000		2000×2000		3000×3000		4000×4000		5000×5000		6000×6000		7000×7000		8000×8000	
Teste	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**
1	0,000974	0,000113	0,138258	0,004197	1,261299	0,017577	4,451461	0,008771	10,570328	0,026065	20,859878	0,274900	35,961453	0,288674	57,217254	0,404397	85,993765	0,862746
2	0,005296	0,000482	0,345281	0,002321	2,099526	0,005061	6,271699	0,001827	14,535210	0,367379	27,059220	1,163874	47,663989	0,964300	77,394803	2,163704	117,265024	2,298921
3	0,007713	0,000446	0,483626	0,001311	2,617379	0,010618	7,452695	0,027710	17,243815	0,780934	30,966510	1,623159	49,610384	0,686116	78,772050	2,492585	116,539053	3,654354
4	0,013165	0,005710	0,628075	0,002840	3,172556	0,015145	8,688710	0,023519	19,149283	0,767899	32,431566	0,619583	52,350537	0,420836	81,216928	1,433667	121,132402	2,899335
5	0,011898	0,004529	0,628101	0,003171	3,164578	0,008772	8,663612	0,004251	19,321664	0,716299	33,055268	0,648290	53,013009	1,239858	83,041345	2,688248	122,527327	4,475837
6	0,014324	0,004634	0,772231	0,003599	3,833832	0,059289	10,215821	0,169502	20,565544	0,184119	36,253034	0,743245	57,636672	0,387601	85,997109	0,233218	123,065223	1,327118
7	0,000313	0,000023	0,133363	0,000456	1,281833	0,001169	4,488888	0,007938	10,627337	0,019033	20,890039	0,109284	36,411436	0,220226	59,690543	4,272600	87,517323	0,706736
8	0,000532	0,000047	0,266358	0,001128	2,562274	0,001386	8,965749	0,004632	21,999837	1,103370	42,280581	0,670343	72,709347	0,276126	116,411556	0,437245	175,674781	0,890928
9	0,000623	0,000105	0,397666	0,001430	3,842224	0,001753	13,457422	0,019452	31,997618	0,157037	62,963221	0,158746	109,723939	0,516650	175,447559	0,435179	262,867582	0,932962
10	0,000828	0,000078	0,526826	0,001830	5,122435	0,002379	17,945867	0,008637	43,199518	0,532833	84,141385	0,270959	146,857771	1,431858	234,557681	2,820160	352,556286	3,340386

(b) Com 2 *Threads*.

	100×100		1000×1000		2000×2000		3000×3000		4000×4000		5000×5000		6000×6000		7000×7000		8000×8000	
Teste	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**
1	0,001031	0,000048	0,244328	0,000791	3,259751	0,018861	11,239958	0,041937	26,770962	0,210291	53,383120	0,189194	92,286293	0,253062	146,473725	0,272039	219,254432	0,654667
2	0,005879	0,000185	0,459948	0,000888	4,046362	0,025063	12,713709	0,114729	29,641907	0,194054	58,389962	0,265263	100,326321	0,244856	158,607096	0,363000	236,657632	0,464587
3	0,008385	0,000241	0,597561	0,001366	4,190008	0,076253	14,334637	0,081784	32,745841	0,074366	62,034227	0,269866	104,547992	0,276375	165,804880	0,224734	244,952403	1,331107
4	0,010854	0,000159	0,735338	0,001562	5,186924	0,024413	15,448134	0,109078	34,713120	0,133354	65,422628	0,149836	109,542442	0,248357	172,698719	0,431783	254,251451	0,299275
5	0,010630	0,000281	0,735729	0,000784	5,197523	0,006559	15,724049	0,020725	34,135145	0,170156	65,310274	0,303859	109,160778	0,401852	172,240964	0,449894	254,020311	0,215468
6	0,013181	0,000193	0,879354	0,001953	5,707358	0,073819	16,961802	0,059959	36,531019	0,198525	69,409186	0,268150	114,943534	0,220444	180,501163	0,525820	263,023271	0,330285
7	0,000419	0,000004	0,244676	0,000586	3,262441	0,009044	11,343103	0,028831	27,019469	0,101468	53,005351	0,178126	92,124527	0,149559	146,524548	0,365206	221,511160	0,426925
8	0,000836	0,000002	0,485933	0,001486	6,510072	0,029039	22,503420	0,290757	53,693231	0,442012	105,956876	0,714877	157,993642	1,221227	251,971035	1,339259	432,977734	1,189482
9	0,001209	0,000004	0,724101	0,002342	9,314452	0,021979	33,061911	0,086777	79,850835	0,380270	158,234800	0,407491	276,145376	0,523657	441,992327	0,526942	665,043775	0,877119
10	0,001463	0,000005	0,960234	0,001720	12,758107	0,226820	41,772119	0,928201	107,612776	0,168992	210,784599	0,518663	321,532839	4,460059	462,129534	2,175191	690,860249	6,318663

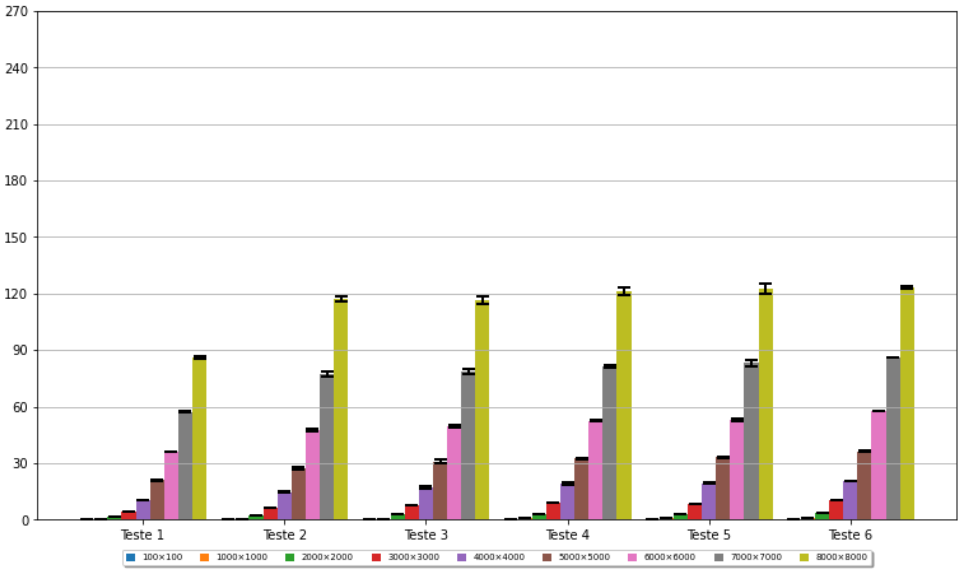
Fonte: Próprio Autor

lelas com alocação dinâmica de dados. Os resultados obtidos com a execução com as 4 *threads* são mostrados na Tabela 6.6a e a Tabela 6.6b mostra os resultados com 2 *threads*.

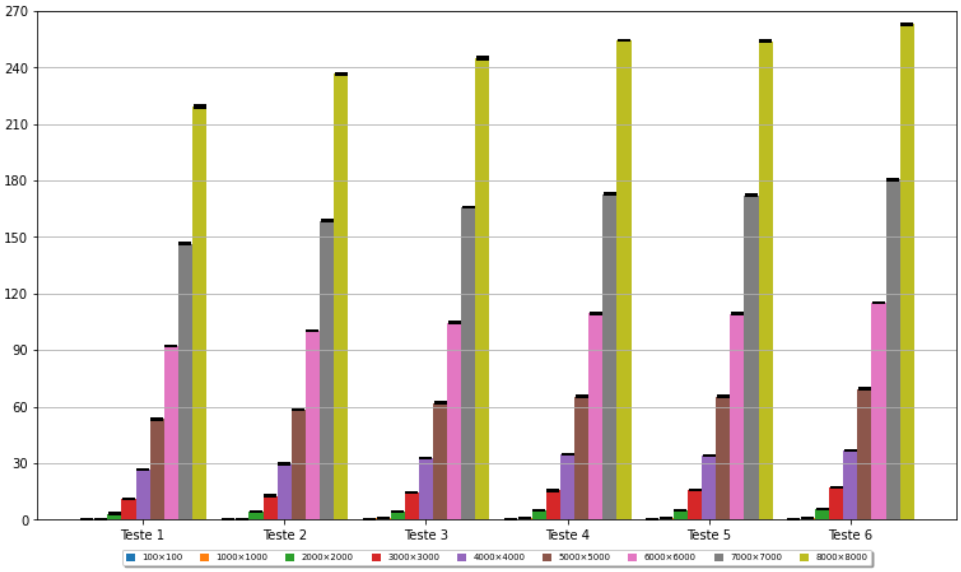
Na Figura 6.10 e nas Tabelas 6.6a e 6.6b nota-se o aumento do tempo do Teste 2 para o Teste 3, do Teste 3 para o Teste 4, do Teste 4 para o Teste 5. Já os tempos para execução de cada tarefa do Teste 5 e do Teste 6 são praticamente similares. Esta similaridade é devido à chamada da quarta tarefa, no Teste 6, que embora chamada

Figura 6.10: Tempos das execuções de tarefas com paralelização interna.

(a) 4 *Threads*.



(b) 2 *Threads*.



Fonte: Próprio Autor

na forma remota é executada no próprio *Host* sem o envio de dados através da rede de comunicação, como é explicado na Seção 6.3.3.

O tempo total gasto na comunicação de dados aumenta proporcionalmente ao número de tarefas enviados para os nós remotos. Mas esse aumento na diferença possuí um menor impacto entre as chamadas para 3 tarefas no Teste 4 em comparação com 4 tarefas no Teste 5, e esse comportamento é semelhante nas chamadas para 3 tarefas no Teste 4 em comparação com 4 tarefas no Teste 6. Como mencionado anteriormente, isto ocorre pois a quarta tarefa é executada localmente no Teste 5 e no Teste 6 embora a chamada da tarefa seja feita na forma remota é executada no próprio *Host*, sem envio de dados através da rede de comunicação. Esse comportamento é o mesmo, tanto nas execuções com 4 *threads* quanto nas execuções com 2 *threads*. Observa-se que o tempo de execução total é maior para as execuções com 2 *threads*, como o tempo de envio de dados é praticamente o mesmo, e independe do número de *threads*, conclui-se que esse aumento de tempo é devido ao menor número de *threads* executando concorrentemente.

6.4.2 Análise de desempenho

A avaliação do ganho de desempenho com paralelização interna executadas remotamente realizadas nos programas descritos na Seção 6.4.1 foram feitas de forma semelhante às da Seção 6.3.4. Os parâmetros como o número de tarefas entre 1 e 4 e a quantidade de dados com o n variando de 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000 até 8000 foram utilizados em ambos os casos de testes.

O cálculo do ganho de desempenho na execução de tarefas remotas com paralelização interna é feita através da mesma Fórmula (6.5) apresentada na Seção 6.3.4. Foram realizadas as comparações do Teste 2 $\times$ Teste 7, Teste 3 $\times$ Teste 8, Teste 4 $\times$ Teste 9 e Teste 6 $\times$ Teste 10 com 1, 2, 3 e 4 tarefas respectivamente, tanto para 4 quanto para 2 *threads*. A Tabela 6.7 apresenta os ganho de desempenho obtidos.

Tabela 6.7: Ganhos de desempenho em tarefas com paralelização interna.

Tamanho $n \times n$	1 Tarefa		2 Tarefas		3 Tarefas		4 Tarefas	
	4 *Threads*	2 *Threads*	4 *Threads*	2 *Threads*	4 *Threads*	2 *Threads*	4 *Threads*	2 *Threads*
100×100	0,05913562	0,07120622	0,06898653	0,09968992	0,04733656	0,11136089	0,05782441	0,11102597
1000×1000	0,38624385	0,53196516	0,55075342	0,81319444	0,63315008	0,98471891	0,68221331	1,09197743
2000×2000	0,61053459	0,80626524	0,97894661	1,55371335	1,21108125	1,79575652	1,33611359	2,23537857
3000×3000	0,71573723	0,89219461	1,20302091	1,56986325	1,54884008	2,14018797	1,75667409	2,46271702
4000×4000	0,73114439	0,91152940	1,27581029	1,63969621	1,67095644	2,30030704	2,10057743	2,94579180
5000×5000	0,77201187	0,90778192	1,36536476	1,70803895	1,94141786	2,41865554	2,32094743	3,03684009
6000×6000	0,76391920	0,91824884	1,46560742	1,51120685	2,09594677	2,52089848	2,54799186	2,79731124
7000×7000	0,77124743	0,92382088	1,47782819	1,51968407	2,16023389	2,55932603	2,72750657	2,56025793
8000×8000	0,74632077	0,93599838	1,50743272	1,76759945	2,17008478	2,61569314	2,86479217	2,62661264

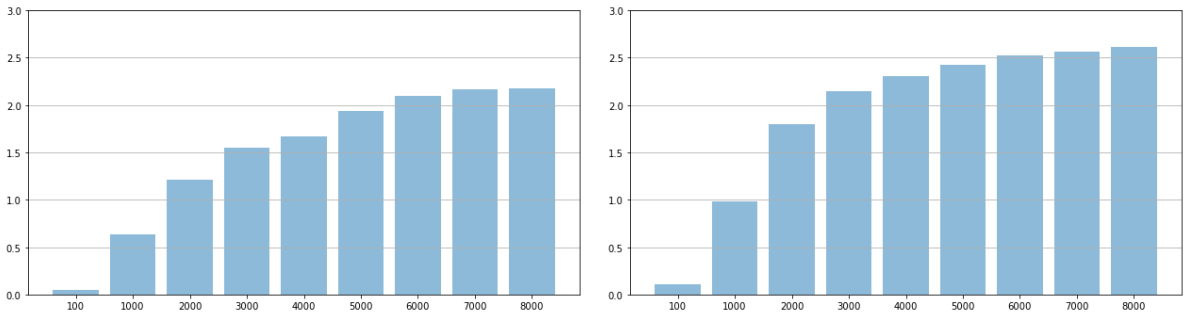
Fonte: Próprio Autor

6.4.2.1 Análise do desempenho em relação à variação do número de tarefas

Da mesma forma que ocorre na Seção 6.3.4, pode-se verificar que há uma evolução no ganho de desempenho na medida em que o n aumenta na execução em cada teste. Ao analisar do gráfico apresentado na Figura 6.11, referente a execução de 3 tarefas simultâneas, pode-se constatar essa evolução.

Figura 6.11: Ganhos de desempenho com 3 tarefas com paralelização interna.

(a) 4 *Threads*. (b) 2 *Threads*.



Fonte: Próprio Autor

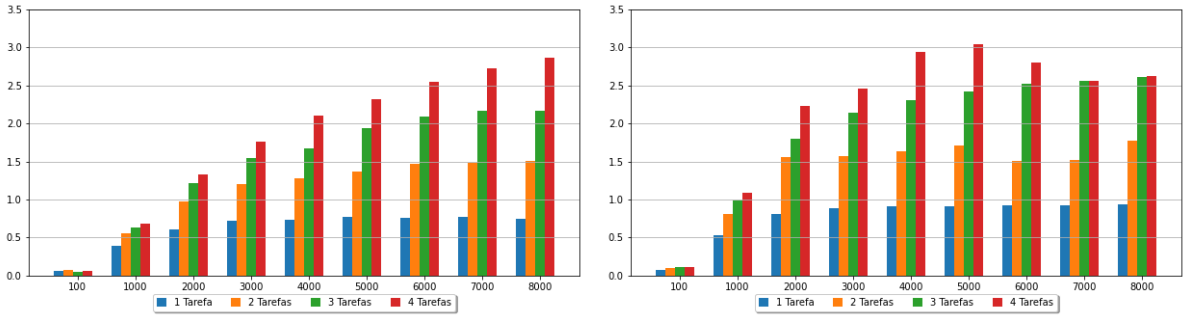
Ainda pode-se observar que as execuções com 4 *threads*, Figura 6.11a, possuem um desempenho um pouco menor às com 2 *threads* apresentado na Figura 6.11b. Isto é devido a uma menor sobrecarga dos recursos de processamento utilizados nas execuções com 2 *threads* em comparação das com 4 *threads*.

Assim como na Seção 6.3.4, nota-se que o ganho de desempenho aumenta a medida em que n aumenta. Essa ocorrência é devido à complexidade do processamento da multiplicação de matrizes que é de $O(n^3)$.

Como pode ser observado na Figura 6.12, independente do número de *Guests* empregados, nota-se o mesmo comportamento da evolução dos ganhos de desempenho. Ao comparar o desempenho de cada teste nas execuções com 4 e 2 *threads*, Figuras 6.12a e 6.12b respectivamente, observa-se a mesma diferença de grandeza das execuções com 4 *threads* em relação as com 2 *threads*.

Figura 6.12: Ganho de desempenho em cada teste com paralelização interna.

(a) 4 *Threads*. (b) 2 *Threads*.

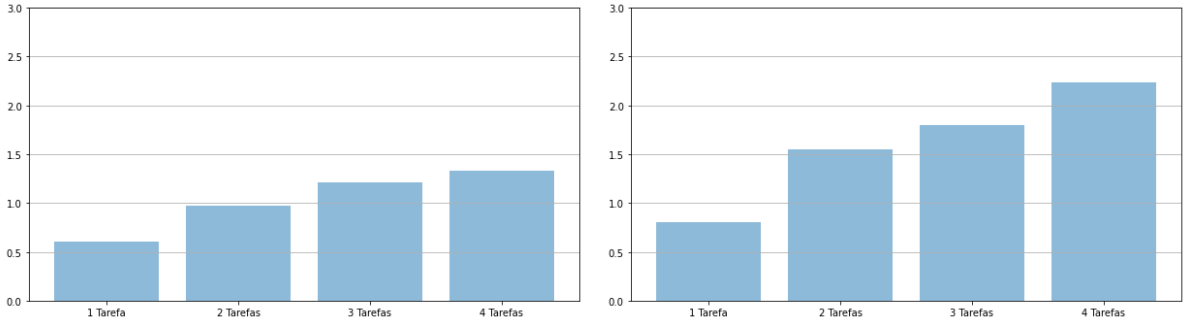


Fonte: Próprio Autor

Ao avaliar a variação do número de tarefas nota-se que o ganho de desempenho tem uma evolução mais expressiva ao aumentar o número de *Guests*. A Figura 6.13 mostra essa evolução a medida que o número de tarefas aumenta de 1, 2, 3 até 4 com o $n = 2000$. Observa-se, também, que as execuções com 4 *threads*, Figura 6.13a, têm um desempenho menor comparado às execuções com 2 *threads*, Figura 6.13b, devido ao fato da concorrência interna nos *Guests* ser menor nas execuções com 2 *threads*.

Figura 6.13: Ganho de desempenho com paralelização interna com o $n = 2000$.

(a) 4 *Threads*. (b) 2 *Threads*.

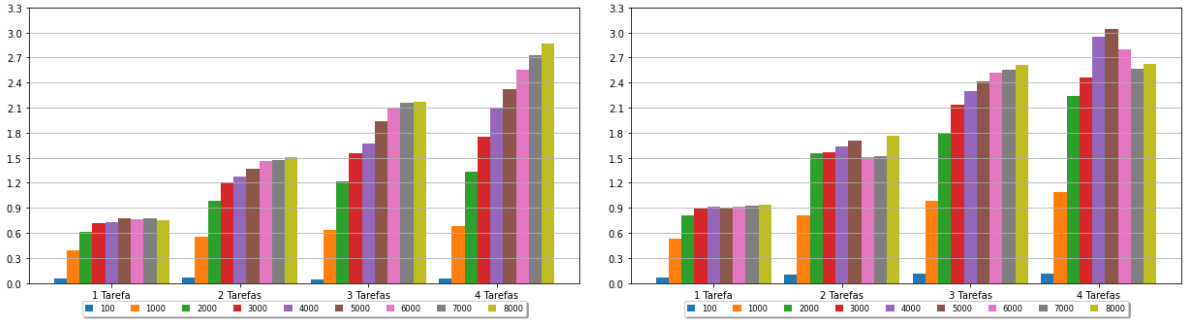


Fonte: Próprio Autor

Na Figura 6.14 constata-se o mesmo comportamento na evolução nos ganhos de desempenho a medida que aumenta o tamanho das matrizes. Também as execuções com 2 *threads*, Figura 6.14b possuem ganhos maiores às execuções com 4 *threads*, Figura 6.14a.

Figura 6.14: Ganho de desempenho em tarefas com paralelização interna

(a) 4 *Threads*. (b) 2 *Threads*.



Fonte: Próprio Autor

6.4.2.2 Análise do desempenho em relação à variação do número de tarefas com o paralelismo interno em cada tarefa

Realizou-se a avaliação do ganho de desempenho referente ao tempo de execução paralela de todas as tarefas com paralelismo interno em relação ao tempo de execução sequencial de todas as tarefas sem paralelização interna. Comparou-se os tempos obtidos nos testes 1, 2, 3, 4, 5 e 6, apresentados na Tabela 6.6a da Seção 6.4.1 com os tempos dos testes 7, 8, 9 e 10 da Seção 6.3.1 que encontram-se na Tabela 6.2. Utilizou-se a mesma Fórmula (6.5), mas sendo o tempo referente à execução sequencial o tempo da execução da versão correspondente com chama-

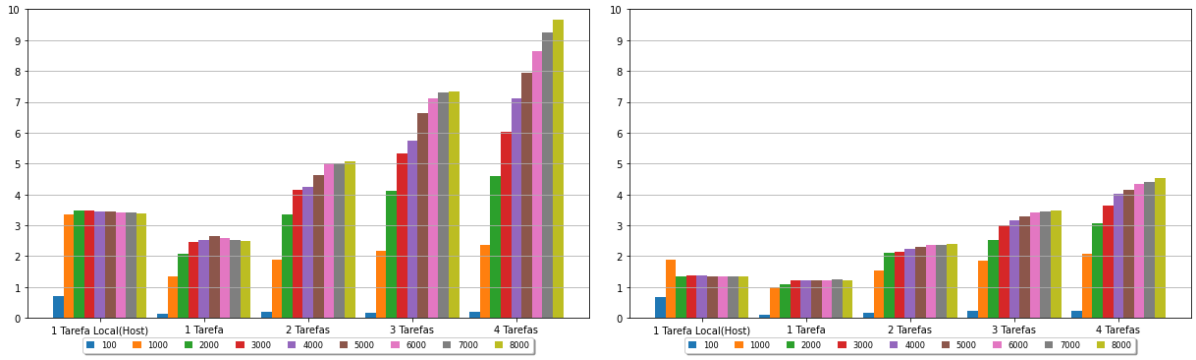
Tabela 6.8: Ganho de desempenho total em tarefas com paralelização interna.

Tamanho $n \times n$	1 Tarefa Local(Host)		1 Tarefa Remota		2 Tarefas Remotas		3 Tarefas Remotas		4 Tarefas Remotas	
	4 *Threads*	2 *Threads*	4 *Threads*	2 *Threads*	4 *Threads*	2 *Threads*	4 *Threads*	2 *Threads*	4 *Threads*	2 *Threads*
100×100	0,7239113	0,6834093	0,1330740	0,1198905	0,1866954	0,1717352	0,1789097	0,2170096	0,2085463	0,2266420
1000×1000	3,3489501	1,8950688	1,3409880	1,0066751	1,8881349	1,5281291	2,1819376	1,8636607	2,3695681	2,0809073
2000×2000	3,4841246	1,3481158	2,0931023	1,0860426	3,3576834	2,0974492	4,1032414	2,5097276	4,5835632	3,0789394
3000×3000	3,4671893	1,3731420	2,4609056	1,2139698	4,1415255	2,1532131	5,3234532	2,9941441	6,0456810	3,6412164
4000×4000	3,4583468	1,3655042	2,5149867	1,2332493	4,2434580	2,2345863	5,7295664	3,1606807	7,1255105	4,0113855
5000×5000	3,4340333	1,3418758	2,6472868	1,2268122	4,6390961	2,3157638	6,6402860	3,2917491	7,9339789	4,1439875
6000×6000	3,4292780	1,3362962	2,5873164	1,2292070	5,0012842	2,3732223	7,1234157	3,4042936	8,6464035	4,3356064
7000×7000	3,4270819	1,3387262	2,5336096	1,2363143	5,0138167	2,3820084	7,3059749	3,4358613	9,2481825	4,4061598
8000×8000	3,3937248	1,3310526	2,4887146	1,2331704	5,0687913	2,4115385	7,3248975	3,4897831	9,6608261	4,5201769

Fonte: Próprio Autor

das sequenciais das funções sem paralelismo interno, para o cálculo dos ganhos de desempenho apresentados na Tabela 6.8.

Nota-se na Tabela 6.8 e na Figura 6.15 que o ganho de desempenho tem uma melhora expressiva ao aumentar o número de *Guests* e é possível observar a evolução a medida que o número de tarefas aumenta de 1, 2, 3 até 4. Observa-se que as execuções com 4 *threads*, Figura 6.15a, tem um desempenho maior comparado às execuções com 2 *threads* na Figura 6.15b devido ao fato dos tempos nas execuções nos testes com 4 *threads* serem menores se comparado às execuções com 2 *threads*.



(a) 4 *Threads*. (b) 2 *Threads*.

Figura 6.15: Ganho de desempenho total em tarefas com paralelização interna.

6.5 Comparação de desempenho de uma aplicação utilizando RTE e MPI

Apresenta os resultados obtidos ao comparar o comportamento de uma aplicação utilizando o modelo de programação proposto através da API RTE com o padrão de programação distribuída com a API MPI.

6.5.1 Experimentos com paralelização entre tarefas sequenciais

Realiza a análise comparativa do ganho de desempenho na execução de aplicações que executam tarefas sequenciais entre o modelo de programação através da API RTE, apresentadas na Seção 6.3 com aplicações empregando recursos da API MPI.

6.5.1.1 Descrição dos experimentos

Para realizar a comparação com os resultados alcançados pelo RTE, apresentados na Seção 6.3.1, desenvolveu-se a implementação de quatro cenários de testes com o uso da API MPI. Aqui, também, as chamadas de tarefas a serem executadas simultaneamente por múltiplos recursos, processam uma versão sequencial da multiplicação de matrizes quadrada em que o tamanho é definido durante sua execução com a alocação dinâmica de áreas de memória. Os quatro cenários de testes MPI são mostrados a seguir:

1 Tarefa Remota: Executa uma tarefa remotamente através das diretivas da API MPI;

2 Tarefas Remotas: Executa 2 chamadas simultâneas, uma para cada nó específico, com duas matrizes em cada;

3 Tarefas Remotas: Este cenário é semelhante ao anterior mas com três chamadas paralelas de multiplicação de seis matrizes;

4 Tarefas Remotas: São feitas quatro chamadas paralelas para a multiplicação simultânea de oito matrizes, duas para cada nó, o código encontra-se no Apêndice E.1.

Os mesmos critérios adotados na Seção 6.3, em que cada cenário é executado nas nove versões com os tamanhos das matrizes variando de 100×100, 1000×1000, 2000×2000, 3000×3000, 4000×4000, 5000×5000, 6000×6000, 7000×7000 e 8000×8000, foram utilizados nas execuções com MPI. A Tabela 6.9 apresenta os tempos alcançados.

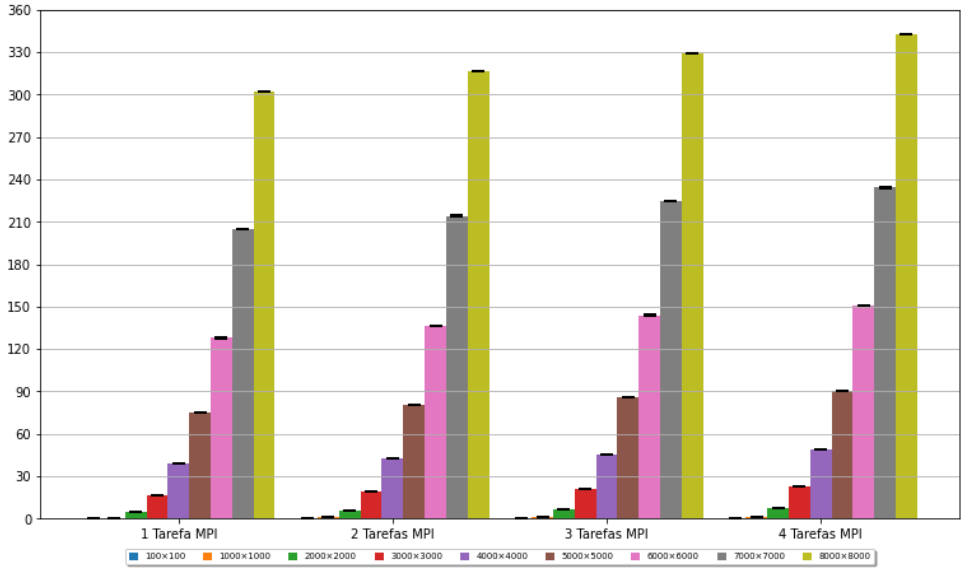
Tabela 6.9: Tempos de execuções MPI.

Tamanho	1 Tarefa		2 Tarefas		3 Tarefas		4 Tarefas	
$n \times n$	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão
100×100	0,021714	0,003989	0,015225	0,000233	0,017011	0,000584	0,019128	0,000767
1000×1000	0,671157	0,003634	0,871986	0,006454	1,075356	0,009949	1,273914	0,004100
2000×2000	5,072100	0,004930	5,919037	0,027416	6,692828	0,014281	7,509291	0,014988
3000×3000	16,845233	0,013969	18,870067	0,015716	20,644010	0,015715	22,480652	0,015096
4000×4000	38,745017	0,037059	42,494249	0,034803	45,762624	0,073567	48,977157	0,047332
5000×5000	75,052699	0,168958	80,692964	0,069455	85,730239	0,167683	90,718668	0,081952
6000×6000	128,007010	0,478041	136,463285	0,388754	143,949720	0,284526	151,144277	0,127587
7000×7000	205,220517	0,112134	214,362256	0,123043	224,450345	0,153409	234,223595	0,168829
8000×8000	302,065664	0,084606	316,323714	0,140371	329,585145	0,173331	342,852377	0,174464

Fonte: Próprio Autor

Na Figura 6.16 e na Tabela 6.9 nota-se o aumento do tempo do teste de 1 Tarefa para 2 Tarefas, do teste com 2 Tarefas para 3 Tarefas, e do teste com 3 Tarefas para 4 Tarefas. Esse comportamento é semelhante ao observado na Seção 6.3.2. As avaliações comparativas do comportamento entre RTE e MPI são apresentadas a seguir.

Figura 6.16: Tempos de execuções MPI por número de tarefas.



Fonte: Próprio Autor

6.5.1.2 Resultados dos testes e análises de desempenho

O ganho de desempenho é calculado pelo tempo da execução paralela das tarefas em relação a execução das correspondentes funções chamadas sequencialmente.

As avaliações do ganho de desempenho com o uso da API MPI foram realizadas entre os tempos dos testes 7, 8, 9 e 10 na Tabela 6.2 da Seção 6.3.4 com os tempos das execuções com MPI de 1, 2, 3 e 4 tarefas respectivamente, e com a quantidade de dados n variando de 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000 e 8000. O ganho de desempenho nas execuções com MPI são calculados através da fórmula:

$$S\left(t_{mpi_i}\right) = \frac{t_{mpi_i}}{t_{seq_i}} \tag{6.6}$$

Onde:

t_{mpi_i}: tempo de execução MPI;

t_{seq_i}: tempo de execução das funções chamadas sequencialmente;

i: número de tarefas ou execuções sequenciais.

Análise de ganho de desempenho

A comparação é feita com os resultados obtidos na Seção 6.3.4. A Tabela 6.10 apresenta os ganhos de desempenho obtidos nas execuções RTE e MPI.

Tabela 6.10: Comparação do ganhos de desempenho entre RTE e MPI.

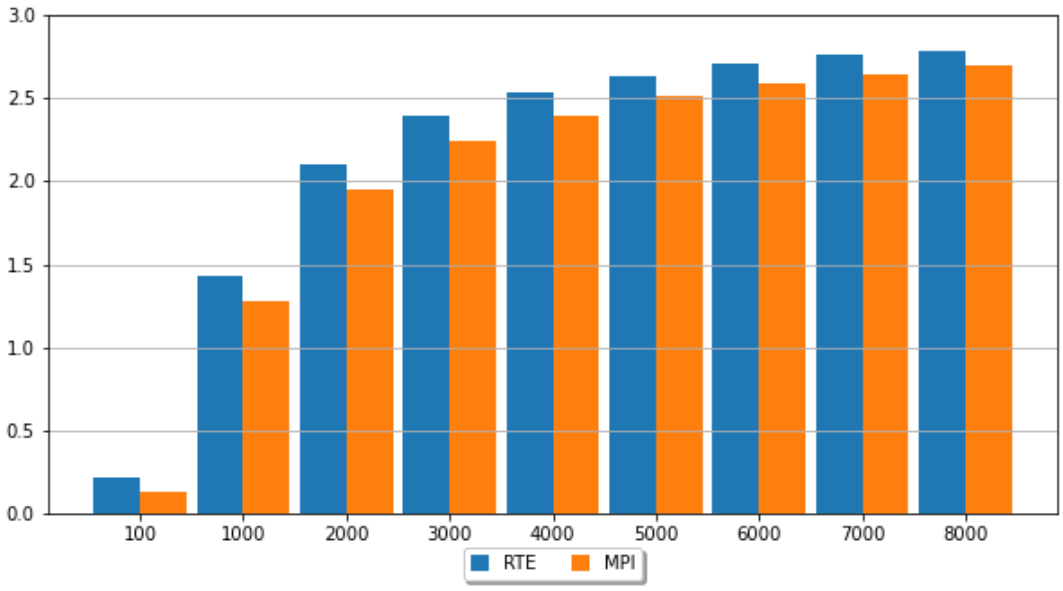
Tamanho	1 Tarefa		2 Tarefas		3 Tarefas		4 Tarefas	
$n \times n$	RTE	MPI	RTE	MPI	RTE	MPI	RTE	MPI
100×100	0,125022173	0,032458621	0,147668075	0,094581902	0,218021937	0,138466603	0,189937563	0,156175011
1000×1000	0,680041177	0,689879499	1,111705871	1,047207409	1,434190999	1,274387924	1,673098737	1,436402963
2000×2000	0,860431262	0,866410717	1,572922042	1,484756609	2,105911172	1,945032158	2,597800092	2,340115820
3000×3000	0,916780627	0,916227078	1,709739232	1,635687317	2,394395383	2,240550115	3,005566252	2,747322144
4000×4000	0,940919406	0,943498352	1,786703308	1,721960210	2,528208604	2,397526237	3,236554495	2,992007078
5000×5000	0,956238828	0,954442908	1,837157847	1,780286759	2,632720688	2,512005977	3,384648089	3,170580097
6000×6000	0,961926358	0,963398954	1,850099104	1,818185957	2,701468035	2,590589513	3,500493624	3,297180205
7000×7000	0,969685473	0,955500058	1,874290438	1,842435462	2,760708734	2,643653045	3,572153706	3,395545886
8000×8000	0,969025482	0,966144820	1,903122287	1,867429211	2,784784828	2,692118993	3,646442660	3,467707375

Fonte: Próprio Autor

Análise do ganho de desempenho em relação à variação do número de dados

Como era de se esperar ambas as execuções, RTE e MPI, apresentam o mesmo

Figura 6.17: Comparação no ganho de desempenho entre RTE e MPI com 3 tarefas.

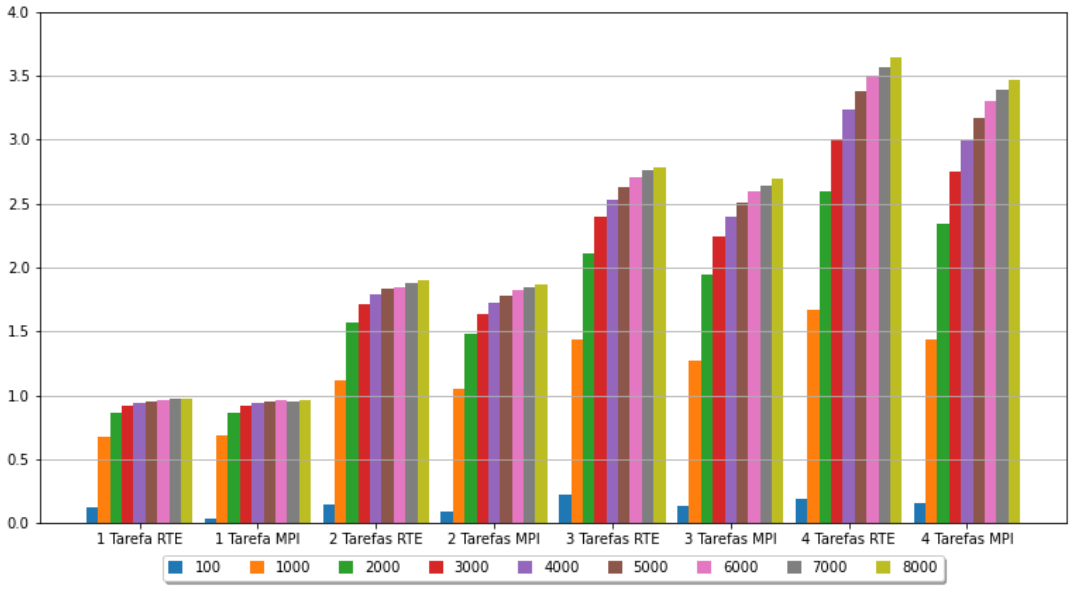


Fonte: Próprio Autor

comportamento da evolução no ganho de desempenho ao aumentar o n de 100 para 1000, 2000, 3000, 4000, 5000, 6000, 7000 até 8000. É possível observar na Figura 6.17, para 3 tarefas simultâneas, que as execuções RTEs começam a apresentar ganhos de desempenho superiores às versões com MPI a medida que o n começa a aumentar a partir de 1000. A principal hipótese é que o impacto do custo de comunicação é menor em aplicações com RTE em relação ao custo de comunicação nas execuções com MPI, já que as tarefas paralelizadas são praticamente as mesmas.

Independente do número de tarefas e de seus respectivos nós MPI e *Guests* RTE nota-se o mesmo comportamento no ganho de desempenho como se pode constatar na Figura 6.18.

Figura 6.18: Comparação no ganho de desempenho entre RTE e MPI em cada teste.



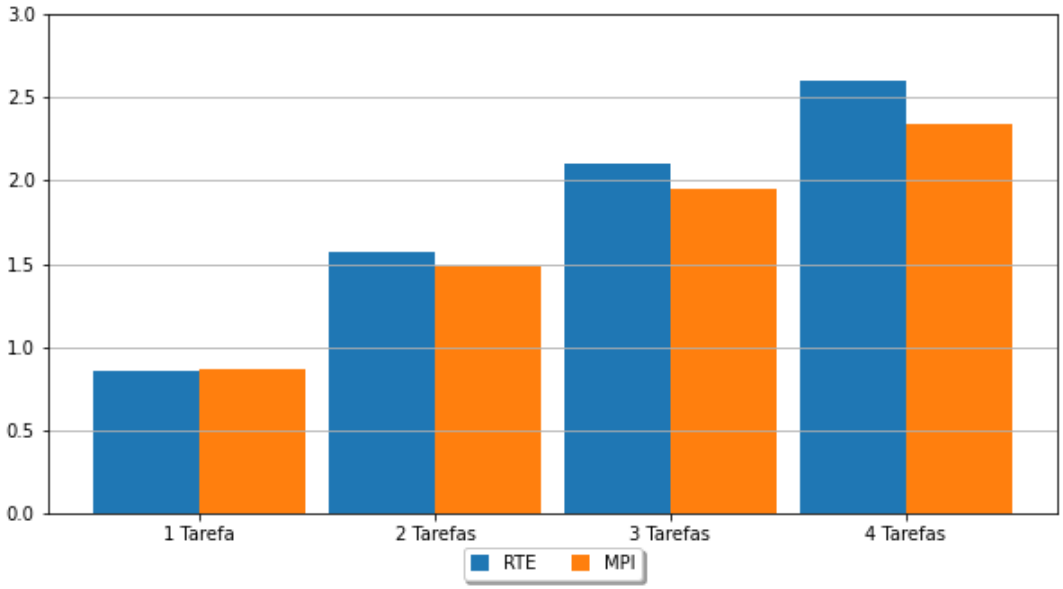
Fonte: Próprio Autor

Análise do ganho de desempenho variando o número de tarefas

Na comparação variando o número de tarefas o comportamento também mantém a similaridade nas execuções com RTE e MPI. Na Figura 6.19 observa nas execuções de 1, 2, 3, e 4 tarefas com o $n = 2000$ que os ganhos de desempenho são mais expressivos para RTE em relação às execuções com MPI. Portanto deduz-se que o custo de comunicação tem uma menor relevância quando se aumenta o número de

Guests no RTE em relação ao custo no gerenciamento de comunicação ao aumentar o número de nós para as execuções com MPI.

Figura 6.19: Comparação no ganho de desempenho entre RTE e MPI com $n = 2000$.



Fonte: Próprio Autor

6.5.2 Experimentos com paralelização interna de tarefas

Os testes apresentados nessa seção realizam a comparação do ganho de desempenho na execução de aplicações que executam tarefas distribuídas com paralelização interna e alocação dinâmica de dados entre o modelo de programação proposto nesse trabalho através da API RTE, apresentadas na Seção 6.4 com implementações utilizando recursos da API MPI.

6.5.2.1 Descrição dos experimentos

Desenvolveu-se uma implementação com quatro cenários de testes com o uso da API MPI para comparar com os resultados alcançados pelo RTE, apresentados na Seção 6.4.1. As chamadas das tarefas também são executadas simultaneamente por múltiplos recursos e processam a versão paralela de multiplicação de matrizes quadradas em que o tamanho é definido na execução com a alocação dinâmica de áreas de memória. Os quatro cenários de testes MPI com paralelização interna e

seus respectivos programas escritos em linguagem C são mostrados a seguir:

1 Tarefa Remota: Executa uma tarefa remota através das diretivas da API MPI;

2 Tarefas Remotas: O programa executa 2 chamadas simultâneas, uma para cada nó específico com duas matrizes em cada;

3 Tarefas Remotas: Este cenário é semelhante ao anterior, porém é feita três chamadas paralelas de multiplicação de seis matrizes;

4 Tarefas Remotas: Faz 4 chamadas paralelas para as multiplicações simultâneas de oito matrizes, duas para cada nó. O código encontra-se no Apêndice E.2,

Os critérios adotados na Seção 6.4 em que cada cenário executa nove versões com os tamanhos das matrizes variando de 100×100, 1000×1000, 2000×2000, 3000×3000, 4000×4000, 5000×5000, 6000×6000, 7000×7000 e 8000×8000 também foram utilizados aqui. A Tabela 6.11 apresenta os tempos alcançados com as execuções de tarefas paralelas. Os resultados obtidos com a execução com as 4 *threads* é mostrado na Tabela 6.11a e a Tabela 6.11b mostra os resultados com 2 *threads*.

Tabela 6.11: Tempos de execuções MPI em tarefas com paralelização interna.

(a) Com 4 *Threads*.

Tamanho	1 Tarefa		2 Tarefas		3 Tarefas		4 Tarefas	
$n \times n$	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão
100×100	0,015929	0,003934	0,016252	0,002444	0,017084	0,003478	0,015011	0,003101
1000×1000	0,623607	0,001071	0,556322	0,001667	0,752624	0,004178	0,959142	0,005300
2000×2000	4,764839	0,143787	2,924486	0,030184	3,707124	0,008176	4,514243	0,005000
3000×3000	15,527845	0,402379	8,103559	0,007360	9,876850	0,010054	11,712700	0,018823
4000×4000	36,123066	0,365658	17,914919	0,564827	20,723212	0,401663	23,603987	0,368930
5000×5000	71,088073	0,398451	31,212842	1,037959	35,350879	0,018725	40,421705	0,015762
6000×6000	121,293264	0,567015	49,667635	0,494484	56,655760	0,010733	64,111211	0,318813
7000×7000	195,129570	0,435654	80,102379	1,731209	88,005709	1,452237	95,945088	1,012804
8000×8000	295,808521	1,089361	121,440656	1,857108	129,959813	2,136138	139,583477	2,524510

(b) Com 2 *Threads*.

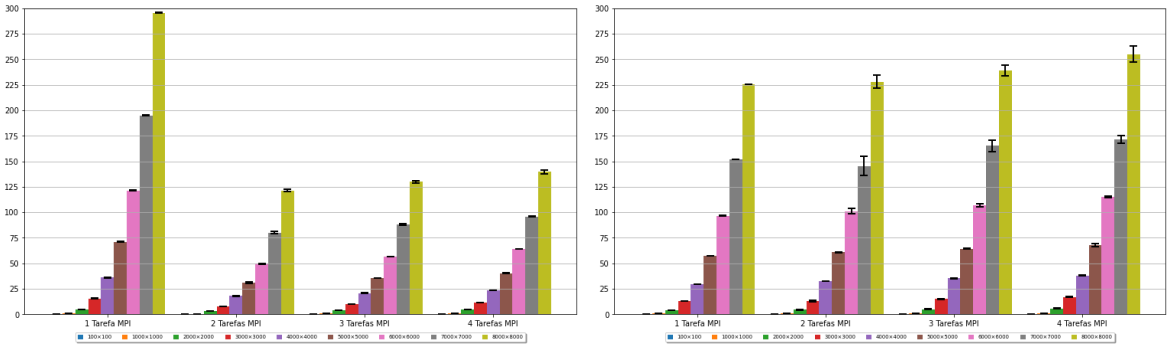
Tamanho	1 Tarefa		2 Tarefas		3 Tarefas		4 Tarefas	
$n \times n$	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão
100×100	0,014920	0,002995	0,016311	0,002464	0,017260	0,004979	0,014642	0,002406
1000×1000	0,624618	0,001807	0,662766	0,002435	0,857615	0,002702	1,058226	0,003006
2000×2000	4,160813	0,003737	4,436517	0,209993	5,046797	0,231679	5,813979	0,146845
3000×3000	13,120702	0,030304	13,171059	1,104606	15,184558	0,538511	17,026983	0,399274
4000×4000	29,859559	0,012173	32,483916	0,463355	35,188344	0,567847	38,054451	0,835648
5000×5000	57,019135	0,010129	60,933294	0,712196	64,426759	0,588851	67,874109	2,211444
6000×6000	96,742093	0,036914	101,038330	4,443189	106,865416	2,031251	115,225831	1,639361
7000×7000	152,186981	0,053017	145,374952	15,367142	165,221715	9,030884	171,630375	6,288006
8000×8000	225,328551	0,085373	227,954602	10,191740	239,071799	8,980743	255,045561	12,593204

Fonte: Próprio Autor

Na Figura 6.20 e nas Tabelas 6.11a e 6.11b nota-se o aumento do tempo do Teste 1 para o Teste 2, do Teste 2 para o Teste 3, do Teste 3 para o Teste 4. As figuras 6.20a e 6.20b, para 4 e 2 *threads* respectivamente, apresentam comportamentos semelhantes aos descritos na Seção 6.4.1.

Figura 6.20: Tempos de execuções MPI em tarefas com paralelização interna.

(a) 4 *Threads*. (b) 2 *Threads*.



Fonte: Próprio Autor

As comparações do comportamento entre RTE e MPI com chamadas de tarefas com paralelização interna são apresentados a seguir.

6.5.2.2 Resultados dos testes e análises de desempenho

As avaliações do ganho de desempenho com o uso da API MPI foram realizadas entre os tempos dos testes 7, 8, 9 e 10 na Tabela 6.6 da Seção 6.4.2 através das tabelas 6.6a e 6.6b, para 4 e 2 *threads* respectivamente, com os tempos das execuções com MPI de 1, 2, 3 e 4 tarefas respectivamente, e com o n variando de 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000 e 8000. O cálculo do ganho de desempenho nas execuções com MPI foi feito através da Fórmula 6.6 presentes na Seção 6.5.1.2.

Análise de ganho de desempenho

A comparação é feita com os resultados obtidos na Seção 6.4.2. A Tabela 6.12 apresenta os ganhos de desempenho obtidos nas execuções RTE e MPI. Os desempenhos obtidos nas execuções com 4 *threads* é mostrado na Tabela 6.12a e a Tabela 6.12b mostra os ganhos de desempenho com 2 *threads*.

Tabela 6.12: Comparação nos ganhos de desempenho com paralelização interna entre RTE e MPI.

(a) Com 4 *Threads*.

Tamanho	1 Tarefa		2 Tarefas		3 Tarefas		4 Tarefas	
$n \times n$	RTE	MPI	RTE	MPI	RTE	MPI	RTE	MPI
100×100	0,059135623	0,019662868	0,068986529	0,032740183	0,047336559	0,036477722	0,057824411	0,055178800
1000×1000	0,386243846	0,213856964	0,550753424	0,478784589	0,633150080	0,528372608	0,682213312	0,549268028
2000×2000	0,610534589	0,269019174	0,978946610	0,876145022	1,211081253	1,036443125	1,336113589	1,134727404
3000×3000	0,715737230	0,289086374	1,203020907	1,106396427	1,548840077	1,362521689	1,756674094	1,532171694
4000×4000	0,731144387	0,294198097	1,275810288	1,228017650	1,670956437	1,544047198	2,100577431	1,830178867
5000×5000	0,772011874	0,293861381	1,365364760	1,354589265	1,941417864	1,781093505	2,320947426	2,081589216
6000×6000	0,763919196	0,300193391	1,465607420	1,463918046	2,095946773	1,936677561	2,547991858	2,290672224
7000×7000	0,771247425	0,305902087	1,477828195	1,453284624	2,160233887	1,993592920	2,727506572	2,444707553
8000×8000	0,746320770	0,295858018	1,507432717	1,446589527	2,170084776	2,022683604	2,864792168	2,525773771

(b) Com 2 *Threads*.

Tamanho	1 Tarefa		2 Tarefas		3 Tarefas		4 Tarefas	
$n \times n$	RTE	MPI	RTE	MPI	RTE	MPI	RTE	MPI
100×100	0,071206219	0,028056488	0,099689922	0,051247310	0,111360893	0,070029374	0,111025970	0,099946728
1000×1000	0,531965163	0,391721494	0,813194438	0,733189250	0,984718911	0,844320203	1,091977427	0,907400558
2000×2000	0,806265238	0,784087337	1,553713353	1,467383601	1,795756521	1,845616441	2,235378573	2,194384682
3000×3000	0,892194608	0,864519499	1,569863245	1,708550486	2,140187973	2,177337642	2,462717025	2,453289529
4000×4000	0,911529396	0,904885094	1,639696206	1,652917434	2,300307037	2,269241056	2,945791800	2,827863067
5000×5000	0,907781916	0,929606363	1,708038951	1,738899517	2,418655536	2,456041591	3,036840092	3,105522885
6000×6000	0,918248835	0,952269320	1,511206851	1,563700062	2,520898480	2,584048110	2,797311241	2,790457964
7000×7000	0,923820883	0,962792918	1,519684068	1,733249311	2,559326029	2,675146704	2,560257932	2,692585937
8000×8000	0,935998381	0,983058557	1,767599454	1,899403347	2,615693138	2,781774249	2,626612637	2,708771896

Fonte: Próprio Autor

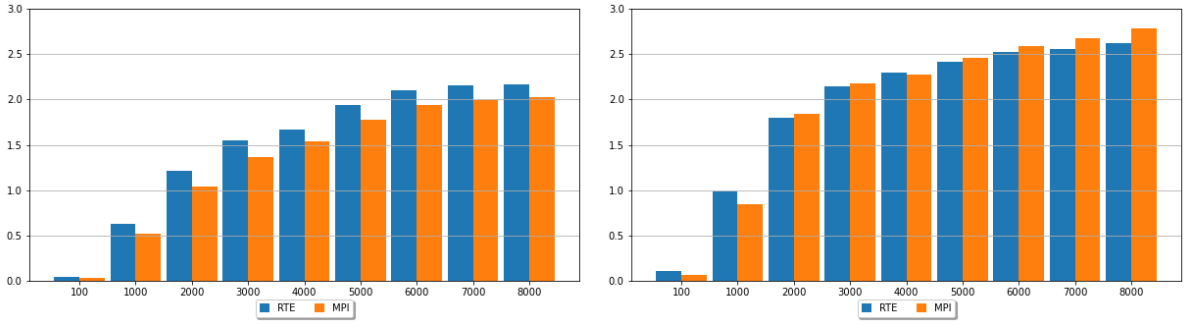
Análise do ganho de desempenho em relação à variação do número de dados

O comportamento na evolução no ganho de desempenho ao aumentar o n se repete nas execuções com MPI. Ao comparar as execuções com 3 tarefas simultâneas, Figura 6.21, atesta-se que nas execuções com 4 *threads*, Figura 6.21a, o ganho de desempenho é menor do que com 2 *threads* da Figura 6.21b. Como visto na Seção 6.4.2.1 a sobrecarga dos recursos nas execuções com 2 *threads* é maior em relação as com 4 *threads*.

Na Figura 6.21b observa-se que executando a aplicação com 2 *threads* com RTE os ganhos de desempenho são similares aos com MPI e, em alguns casos, menores. Este comportamento não ocorre nas execuções com 4 *threads*. Observa-se na Figura 6.21a que o ganho de desempenho é maior nas aplicações com RTE em relação à execução com MPI. Isto ocorre devido ao controle dos mecanismos de comunicação de cada API. É possível que o mecanismo de comunicação em MPI faça uso de recurso de processamento para o gerenciamento de comunicação mais intensivamente do que em RTE, mostrando maior eficiência nas execuções com duas *threads* pois tem

Figura 6.21: Comparação dos ganhos de desempenho em 3 tarefas com paralelização interna.

(a) 4 *Threads*. (b) 2 *Threads*.



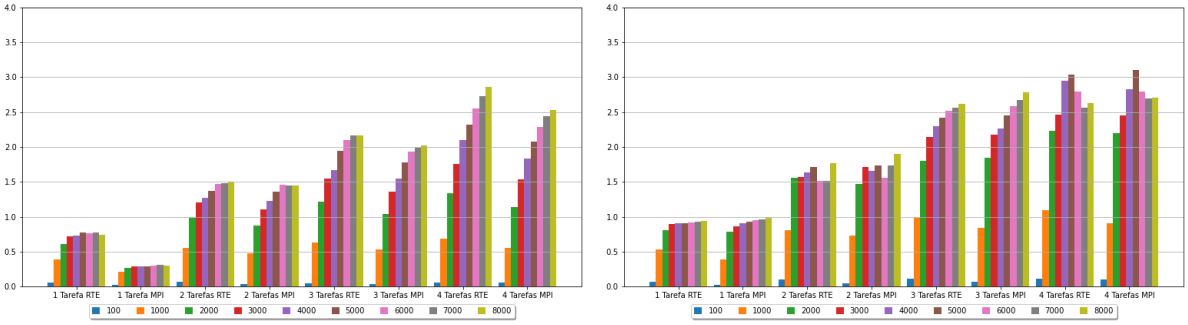
Fonte: Próprio Autor

maior quantidade de *cores* livres para a comunicação. Já o mecanismo de comunicação da API RTE é assíncrono e não bloqueante, apresentando maior eficiência do que o MPI, quando a aplicação utiliza todos os recursos de processamento disponíveis, como no caso de testes aqui apresentado da execução com 4 *threads*.

Na Figura 6.22 constata-se o mesmo comportamento no ganho de desempenho, independente do número de tarefas e de seus respectivos nós MPI e *Guests* RTE. Ao relacionar o desempenho de cada teste para as execuções com 4 e 2 *threads*, Figuras 6.22a e 6.22b, observa-se a mesma diferença de grandeza vistas nas execuções com 3 tarefas simultâneas. As execuções RTE possuem desempenhos melhores que as execuções com MPI, devido aos critérios já discutidos nesta seção.

Figura 6.22: Comparação de desempenho com paralelização interna em cada teste.

(a) 4 *Threads*. (b) 2 *Threads*.



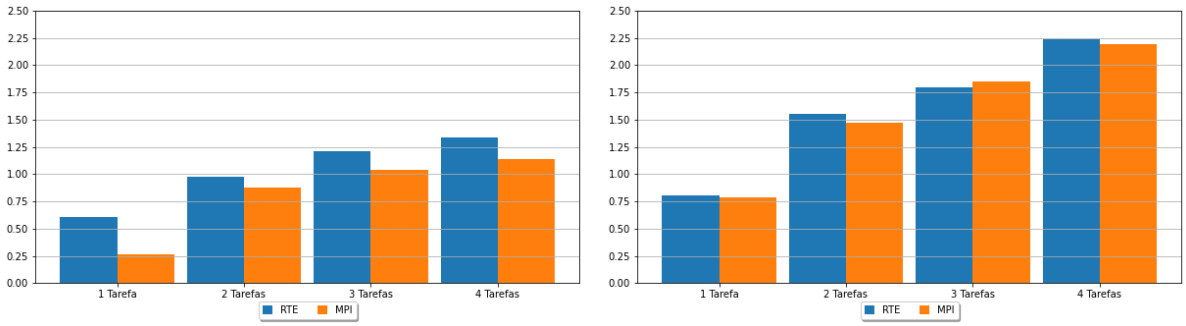
Fonte: Próprio Autor

Análise do ganho de desempenho variando o número de tarefas

Ao comparar os desempenhos nas execuções com RTE e MPI variando o número de tarefas de 1, 2, 3, e 4 com o $n = 2000$, mostrados na Figura 6.23, os ganhos de desempenho são maiores para RTE que MPI, comprovando que o custo de comunicação tem uma menor relevância ao aumentar o número de *Guests* no RTE em relação ao custo no gerenciamento de comunicação ao aumentar o número de nós para as execuções com MPI. Confirma também que o desempenho é menor nas execuções com 4 *threads*, Figura 6.23a, em relação às com 2 *threads*, Figura 6.23b, devido à sobrecarga dos recursos de processamento ser menor nas execuções com 2 *threads*.

Figura 6.23: Comparação de desempenho na paralelização interna com $n = 2000$.

(a) 4 *Threads*. (b) 2 *Threads*.

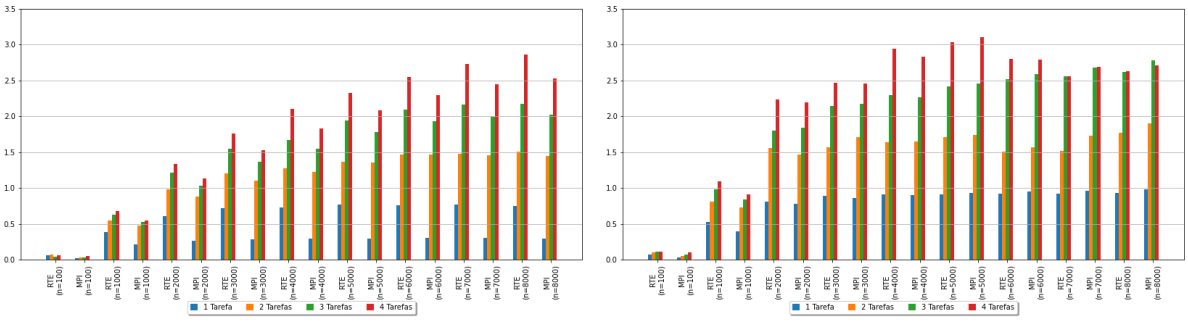


Fonte: Próprio Autor

Na Figura 6.24 constata-se que independente do n empregado, ao aumentar o

Figura 6.24: Comparação do desempenho com paralelização interna por número de tarefas entre RTE e MPI.

(a) 4 *Threads*. (b) 2 *Threads*.



Fonte: Próprio Autor

número de tarefas apresenta a mesma evolução no ganho de desempenho. Para as execuções com 4 *threads*, Figura 6.24a, o desempenho é menor que às com 2 *threads*, Figura 6.24b, devido à sobrecarga dos recursos ser menor para as execuções de 2 *threads*. Os ganhos de desempenho são maiores para RTE que MPI, devido ao fato já mencionado de que o custo no gerenciamento de comunicação nas execuções MPI tem maior impacto que o custo de comunicação nas execuções RTE.

Análise do desempenho em relação à variação do número de tarefas e ao paralelismo interno em cada tarefa

As comparações entre as implementações com RTE e MPI foram realizadas com os tempos dos testes com 1, 2, 3 e 4 tarefas para avaliar o ganho real na execução de tarefas com paralelização interna, tanto com RTE quanto com MPI, em relação aos tempos das execuções sequenciais dos testes 7 a 10 da Tabela 6.2 na Seção 6.3. O cálculo do ganho de desempenho é feito através da mesma Fórmula (6.5). Os resultados dos cálculos dos ganhos reais de desempenho encontrados são mostrados na Tabela 6.13, em que as comparações entre as execuções com 4 *threads* estão na Tabela 6.13a e a Tabela 6.13b apresenta a comparação com 2 *threads*.

Tabela 6.13: Comparação de desempenho real com paralelização interna.

(a) Com 4 *Threads*.

Tamanho	1 Tarefa		2 Tarefas		3 Tarefas		4 Tarefas	
$n \times n$	RTE	MPI	RTE	MPI	RTE	MPI	RTE	MPI
100×100	0,133074033	0,044247732	0,186695362	0,088603389	0,178909710	0,137868465	0,208546257	0,199004743
1000×1000	1,340987966	0,742483327	1,888134931	1,641405876	2,181937638	1,820857512	2,369568092	1,907802106
2000×2000	2,093102270	0,922281316	3,357683368	3,005084790	4,103241440	3,511553308	4,583563223	3,892704063
3000×3000	2,460905561	0,993960121	4,141525519	3,808885622	5,323453170	4,683066066	6,045681000	5,273044857
4000×4000	2,514986691	1,011981098	4,243457982	4,084495435	5,729566443	5,294405537	7,125510464	6,208273247
5000×5000	2,647286787	1,007672778	4,639096143	4,602484270	6,640286040	6,091924133	7,933978866	7,115751377
6000×6000	2,587316386	1,016724390	5,001284169	4,995519297	7,123415655	6,582113360	8,646403455	7,773210173
7000×7000	2,533609593	1,004912867	5,013816749	4,930547957	7,305974943	6,742390262	9,248182514	8,289293186
8000×8000	2,488714606	0,986581374	5,068791343	4,864204147	7,324897474	6,827360011	9,660826122	8,517567696

(b) Com 2 *Threads*.

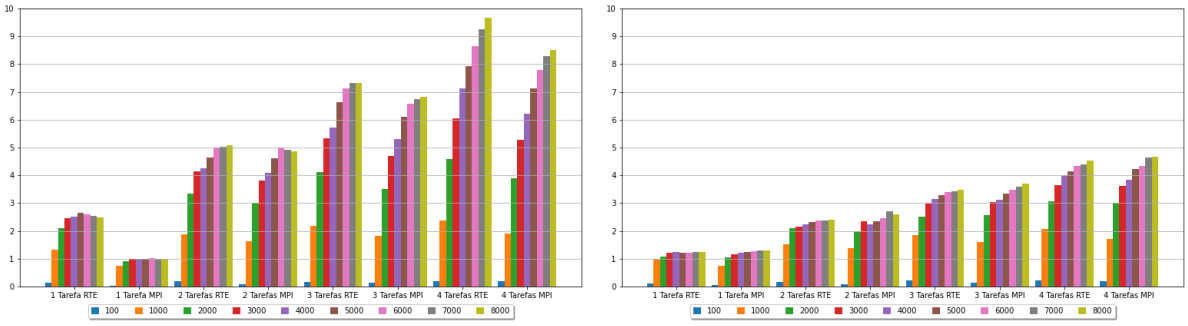
Tamanho	1 Tarefa		2 Tarefas		3 Tarefas		4 Tarefas	
$n \times n$	RTE	MPI	RTE	MPI	RTE	MPI	RTE	MPI
100×100	0,119890452	0,047238923	0,171735242	0,088283439	0,217009554	0,136466608	0,226641984	0,204025461
1000×1000	1,006675108	0,741282145	1,528129099	1,377785899	1,863660675	1,597944695	2,080907253	1,729171644
2000×2000	1,086042646	1,056168921	2,097449160	1,980907544	2,509727645	2,579411267	3,078939390	3,022475708
3000×3000	1,213969775	1,176313476	2,153213145	2,343435568	2,994144076	3,046116830	3,641216404	3,627277510
4000×4000	1,233249262	1,224259886	2,234586279	2,252604236	3,160680747	3,117995294	4,011385502	3,850797911
5000×5000	1,226812180	1,256306597	2,315763771	2,357604609	3,291749051	3,342630836	4,143987529	4,237710158
6000×6000	1,229207033	1,274748304	2,373222315	2,455658455	3,404293612	3,489572684	4,335606396	4,324984371
7000×7000	1,236314267	1,288469056	2,382008428	2,716758406	3,435861261	3,591348982	4,406159775	4,633893991
8000×8000	1,233170359	1,295171764	2,411538468	2,591358707	3,489783148	3,711363827	4,520176907	4,661566003

Fonte: Próprio Autor

A Tabela 6.13 e a Figura 6.25 mostram que o ganho de desempenho tem uma evolução muito mais expressiva ao aumentar o número de *Guests* RTE e nós MPI se comparadas as execuções da Figura 6.22. A medida que o número de tarefas aumenta de 1, 2, 3 até 4 nota-se a evolução no ganho de desempenho nas execuções com 4 *threads*, Figura 6.15a, são superiores se comparado às execuções com 2 *threads* na Figura 6.15b, devido ao fato dos tempos nas execuções com 4 *threads* serem menores aos tempos das execuções com 2 *threads*. Aqui constata-se também que os ganhos de desempenho reais são maiores em RTE em comparação à MPI confirmando o fato, já evidenciado nesta seção, de que o custo de comunicação RTE ter um menor impacto em relação ao custo de gerenciamento de comunicação MPI.

Figura 6.25: Comparação de desempenho real com paralelização interna.

(a) 4 *Threads*. (b) 2 *Threads*.



Fonte: Próprio Autor

6.6 Análise de uma aplicação real

A avaliação de desempenho do sistema RTE em uma aplicação real foi efetuada em uma implementação de Bioinformática que faz a contagem da quantidade de ocorrências de sequências de bases nitrogenadas, DNAs, nos genomas da Baleia Orca, do Bicho Preguiça, do Cavalo, do Leão, do Pavão Azul, do Peixe Betta, do Peixe Boi e do Tigre. Os arquivos apresentam tamanhos entre 440MB à 3,1GB e foram extraídos do banco de dados do *National Center for Biotechnology Information (NCBI)*[3].

[3] <https://www.ncbi.nlm.nih.gov/genome/>

6.6.1 Análises de desempenho de uma aplicação real

O objetivo é avaliar o modelo de programação orientado à tarefas distribuídas através da API RTE quanto ao desempenho e a escalabilidade na execução em uma aplicação real.

Implementaram-se quatro casos de testes específicos para avaliar uma aplicação real utilizando recursos do modelo de programação proposto. Cada caso de teste possui seu respectivo cenário em duas versões, uma com chamadas de tarefas sequenciais e outra com tarefas com paralelização interna através de OpenMP.

6.6.1.1 Contagem de DNAs em genomas com tamanhos aproximados

Tem a finalidade de avaliar o comportamento com a variação do número de tarefas através da busca do número de ocorrências da sequência de DNA específica com 8 bases nitrogenadas em genomas distintos com arquivos de tamanhos aproximados. Os cenários em que variam o número de tarefas são:

1 Tarefa: Faz a contagem da sequência de DNA no genoma da Baleia Orca em um *Guest*;

2 Tarefas: Busca do número de ocorrências da sequência de DNA nos genomas da Baleia Orca e do Tigre, executando cada uma num *Guest* específico;

3 Tarefas: Realiza a contagem da sequência em três *Guests* para os respectivos genomas do Leão, da Baleia Orca e do Tigre;

4 Tarefas: Conta o número de ocorrências da sequência nos genomas do Leão, da Baleia Orca, do Tigre e do Cavalo em quatro *Guests* específico. Os códigos dos programas sequenciais e com paralelização interna encontra-se nos apêndices F.1.1 e F.1.2 respectivamente.

Análise dos resultados

Ao avaliar os resultados obtidos nesse caso de teste, observa-se que o sistema RTE apresenta estabilidade na execução e boa escalabilidade ao aumentar o número de tarefas paralelizadas remotamente.

Os resultados dos tempos obtidos nos cenários são apresentados na Tabela 6.14.

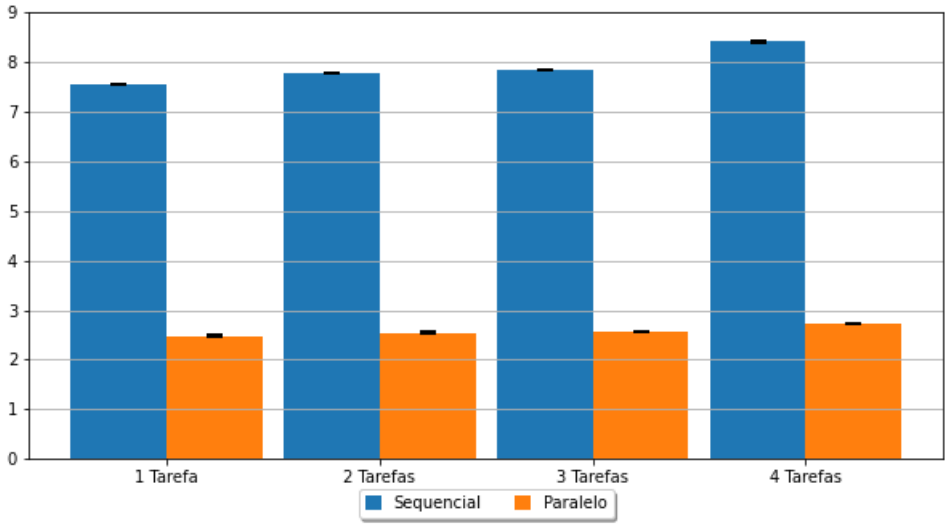
Tabela 6.14: Tempos das execuções variando a quantidade de tarefas com busca do mesmo DNA.

	Sequencial		Paralelo	
Teste	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**
1 Tarefa	7,530206	0,002412	2,480721	0,013339
2 Tarefas	7,764583	0,002651	2,536211	0,023534
3 Tarefas	7,830535	0,003766	2,554523	0,001519
4 Tarefas	8,397757	0,004283	2,714519	0,000938

Fonte: Próprio Autor

A Figura 6.26 apresenta os tempos de execuções das duas versões em que verifica-se que a média dos tempos aumenta de cerca de 3% a cada incremento de uma tarefa. Este aumento se deve ao custo de comunicaçãos dos dados, referentes aos parâmetros e resposta das tarefas. Já entre os tempos de Teste 3 e Teste 4, nota-se um aumento de 7%. Isto deve ser devido ao fato de um número maior de ocorrências da sequência pesquisada ser maior no genoma respectivo à tarefa adicionada comparada às demais tarefas. Verifica-se que o ganho de desempenho com

Figura 6.26: Tempos das execuções variando a quantidade de tarefas com busca do mesmo DNA.



Fonte: Próprio Autor

a paralelização interna das tarefas se mantem com o aumento do número de tarefas, sendo de 3 com o uso de 4 *threads*.

6.6.1.2 Contagem de DNAs de tamanho fixo em um genoma específico

É analisado o comportamento ao variar o número de tarefas, porém é feita a contagem de ocorrências das sequências de DNA distintas com o mesmo número de 8 bases nitrogenadas no genoma do Peixe Boi. Os cenários são descrito a seguir:

1 Tarefa: Faz a contagem de ocorrência de uma sequência em um *Guest*;

2 Tarefas: Realiza a contagem das ocorrências de duas sequências, cada sequência em um *Guest* específico;

3 Tarefas: Conta o número de ocorrências das três sequências. Cada sequência é enviada para um *Guests* juntamente com o nome do arquivo contendo o genoma;

4 Tarefas: Utilizou-se as 4 sequências, sendo uma para cada *Guest* específico. O código com tarefas sequenciais está no Apêndice F.1.3 e o Apêndice F.1.4 encontra-se o código com paralelização interna de tarefas.

Análise dos resultados

A avaliação nesse caso de teste é semelhante à feita no caso anterior. A Tabela 6.15 apresenta as médias de para cada cenário.

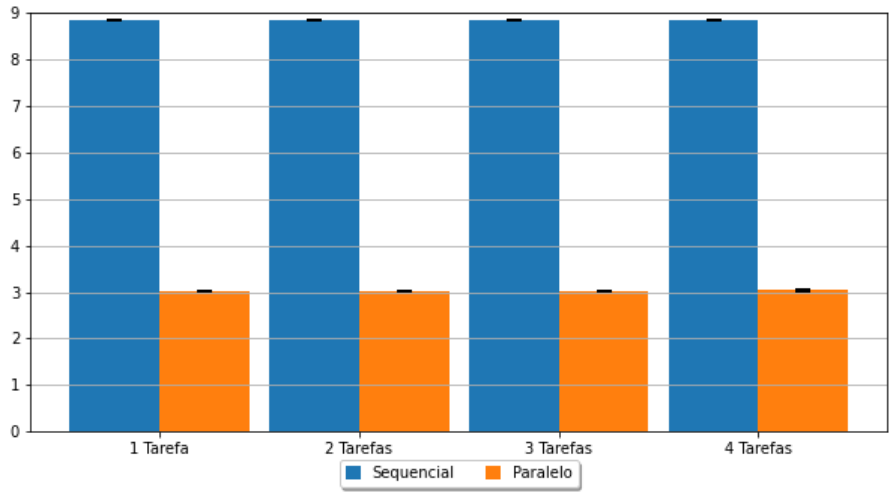
Tabela 6.15: Tempos ao variar a quantidade de tarefas buscando DNAs distintos.

	Sequencial		Paralelo	
Teste	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**
1 Tarefa	8,819048	0,001436	3,028447	0,001486
2 Tarefas	8,819961	0,002259	3,029785	0,001611
3 Tarefas	8,822537	0,002691	3,030735	0,002161
4 Tarefas	8,822511	0,002820	3,036937	0,019812

Fonte: Próprio Autor

Ao observar os resultados apresentados na Tabela 6.15 e na Figura 6.27 nota-se na variação dos tempos devido ao fato de todos os cenários utilizarem o mesmo genoma para a contagem das sequências de DNAs. Verifica-se um pequeno aumento

Figura 6.27: Tempos ao variar a quantidade de tarefas buscando DNAs distintos.



Fonte: Próprio Autor

de tempo. O tempo total de execução é equivalente ao tempo da tarefa com maior custo. Se as 4 tarefas executassem sequencialmente o tempo total seria a soma de todos os tempos.

6.6.1.3 Contagens simultâneas da ocorrência de 4 DNAs distintos de mesmo tamanho em um genoma

Visa verificar a escalabilidade da aplicação aumentando o tamanho da sequência de DNAs a ser encontrada. Para isso realiza contagens simultâneas de ocorrências de uma mesma sequência de DNA em 4 *Guests*, cada qual com um genoma distinto. Os quatro arquivos possuem o tamanho aproximado de 2,4GB. Cada *Guest* executa a contagem de ocorrências em seu genoma específico. Os cinco cenários variam o tamanho da sequência de DNA pesquisada, e são apresentados a seguir:

Teste 1 Faz a contagem de ocorrências da sequência com 10 bases nitrogenadas nos genomas;

Teste 2 Cada *Guest* conta as ocorrências de uma sequência com 50 bases nitrogenadas;

Teste 3 Cada *Guest* realiza a contagem de ocorrências da sequência com 100 bases nitrogenadas;

Teste 4 As quatro execuções paralelas realizam a contagem de ocorrências da sequência com 150 bases nitrogenadas para cada genoma;

Teste 5 Verifica o número de ocorrências de uma sequência com 200 bases nitrogenadas em 4 *Guests*. Os códigos com tarefas sequenciais e com paralelização interna encontram-se nos respectivos apêndices F.1.5 e F.1.6;

Análise dos resultados

A avaliação da escalabilidade em função ao aumento do tamanho das sequências de DNA constata que o sistema RTE apresenta estabilidade na execução ao aumentar o número de tarefas paralelizadas. A Tabela 6.16 mostra os tempos obtidos.

Tabela 6.16: Tempos das execuções variando tamanho da sequência a ser buscada.

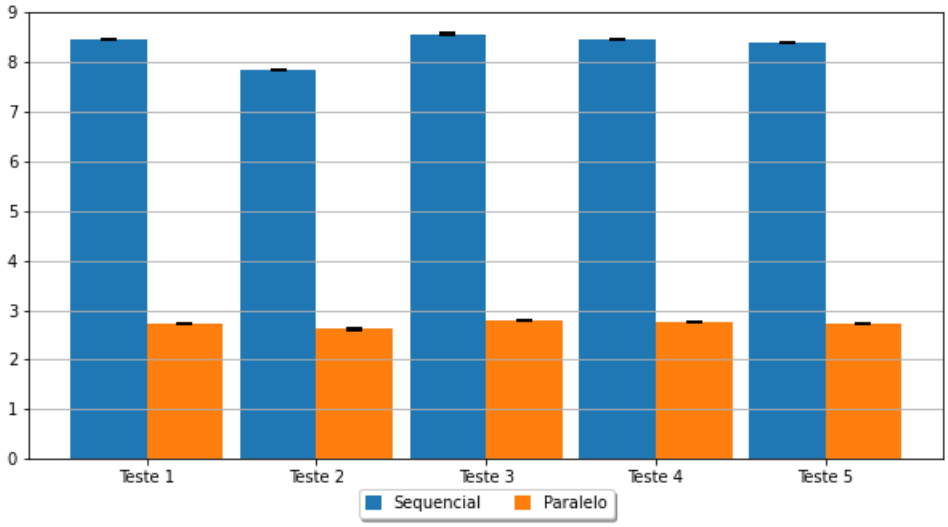
	Sequencial		**Paralelo**	
Teste	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**
1	8,451101	0,005193	2,729492	0,006166
2	7,836638	0,007636	2,613593	0,005018
3	8,555510	0,009383	2,776800	0,003141
4	8,442041	0,012649	2,761093	0,007580
5	8,381357	0,006653	2,737599	0,005189

Fonte: Próprio Autor

A Figura 6.28 apresenta os gráficos dos tempos para as duas versões dos cenários testados. Observa-se pouca variação no tempo de execução em função ao aumento do tamanho da sequência de DNA.

Como cada *Guest* realiza a contagem da sequência de DNA em um genoma diferente, essa variação é dada pela estratégia adotada pelo algoritmo que ao encontrar uma base nitrogenada diferente no genoma retorna ao laço principal para iniciar uma nova busca.

Figura 6.28: Tempos das execuções variando tamanho da sequência a ser buscada.



Fonte: Próprio Autor

6.6.1.4 Contagem de sequência de DNA em genomas distintos

Avalia-se a escalabilidade quanto ao tamanho do genoma em que uma sequência de DNA com 10 bases hidrogenadas é pesquisada. Cada um dos quatro *Guests* possui uma sequência de DNA distinta e o arquivo do genoma em que esta será pesquisada. Os cenários em que variam os tamanhos dos genomas são:

Teste 1 Conta as ocorrências da sequência de DNA no genoma do Peixe Beta num arquivo de 440MB;

Teste 2 Conta a quantidade de ocorrências no genoma do Pavão Azul em um arquivo com 1GB;

Teste 3 Realiza a contagem de ocorrências em um arquivo com 1,5GB contendo o genoma do Bicho Preguiça;

Teste 4 Cada *Guest* conta a quantidade da sequência de DNA no arquivo de 2,4GB contendo o genoma do Leão;

Teste 5 Utilizou-se o arquivo com 3,1GB de dados do genoma do Peixe Boi. Nos apêndice F.1.7 e F.1.8 apresentam os código com tarefas sequenciais e com paralelização interna de tarefas, respectivamente.

Análise dos resultados

Observa-se que o aumento do tempo das execuções é diretamente proporcional ao do tamanho dos genomas. Como é apresentado na Tabela 6.17.

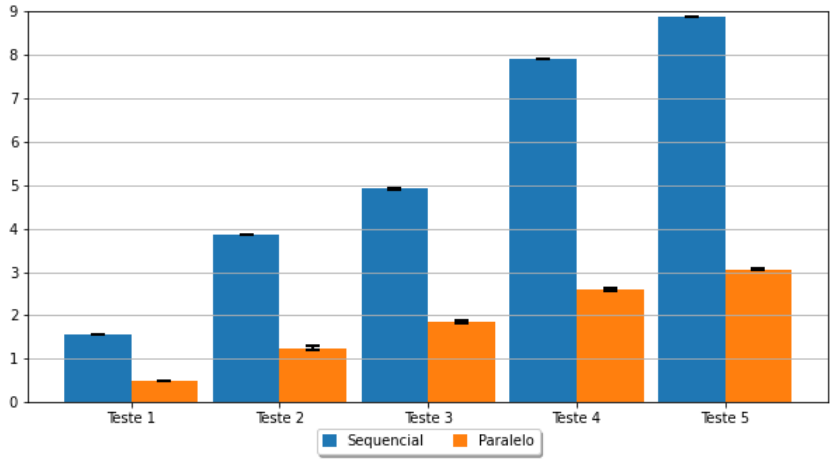
Tabela 6.17: Tempos das execuções ao variar o tamanho dos genomas.

	Sequencial		Paralelo	
Teste	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**
1	1,554774	0,003543	0,504584	0,001994
2	3,843214	0,003347	1,253578	0,071909
3	4,907061	0,003726	1,861693	0,053088
4	7,883078	0,003255	2,594914	0,030561
5	8,869813	0,007393	3,063382	0,038610

Fonte: Próprio Autor

Os resultados apresentados na tabela 6.17 e na Figura 6.29 mostram um desempenho com um comportamento aproximadamente linear, tanto na versão sequencial quanto na versão com paralelização interna, verificando a escalabilidade em função dos tamanhos de cada genoma.

Figura 6.29: Tempos das execuções ao variar o tamanho dos genomas.



Fonte: Próprio Autor

6.6.2 Comparações entre RTE e MPI numa aplicação real

A finalidade é realizar a comparação entre a aplicação real utilizando o modelo de programação proposto, através da API RTE, com outra usando o padrão de programação distribuída com a API MPI.

6.6.2.1 Descrição dos testes

Implementou-se quatro cenários de testes que realizam a mesma contagem de uma sequência composta 8 bases hidrogenadas no mesmo genoma do Peixe Boi. Essa abordagem foi utilizada para garantir a equivalência nas execuções. Em cada cenário foram codificados em dois modos um com o uso da API RTE e outra com a API MPI, e também cada modo possuí duas versões a primeira com chamadas de tarefas sequenciais e segunda com paralelização interna de tarefas através da API OpenMP. Os cenários utilizados são mostrados a seguir:

1 Tarefa: Faz a contagem de ocorrência da sequência de DNA em um *Guest* RTE ou num nó MPI;

2 Tarefas: Realiza duas tarefas distribuídas, uma para cada *Guest* RTE ou cada nó MPI, dependendo do modo;

3 Tarefas: Conta o número de ocorrências da sequência, em três *Guests* RTE ou em três nós MPI, no genoma mencionado;

4 Tarefas: Utilizou-se 4 *Guests* RTE ou 4 nós MPI, cada um realiza a contagem da sequência no genoma mencionado. Os Apêndices F.2.1, F.2.2, F.2.3 e F.2.4 encontram-se os códigos das versões de tarefas sequenciais com RTE, tarefas sequenciais com MPI, paralelização interna de tarefas com RTE e com paralelização interna de tarefas com MPI respectivamente.

6.6.2.2 Análise dos resultados

Os resultados alcançados na comparação mostram a viabilidade na implementação de uma aplicação real usando recursos da API RTE. A Tabela 6.18 apresenta a média dos tempos alcançados nas execuções com RTE e com MPI. As médias dos tempos obtidos com tarefas sequenciais estão na Tabela 6.18a e a Tabela 6.18b mos-

tra as médias dos tempos com paralelização interna de tarefas.

Tabela 6.18: Comparação dos tempos da aplicação real entre RTE e MPI.

(a) Com tarefas sequenciais

	Média		Desvio Padrão	
Teste	**RTE**	**MPI**	**RTE**	**MPI**
1 Tarefa	7,096958	7,489265	0,001587	0,002749
2 Tarefas	7,181146	7,221288	0,040438	0,024064
3 Tarefas	7,101541	7,380830	0,006069	0,004079
4 Tarefas	7,101444	7,845821	0,003132	0,007651

(b) Com paralelização interna

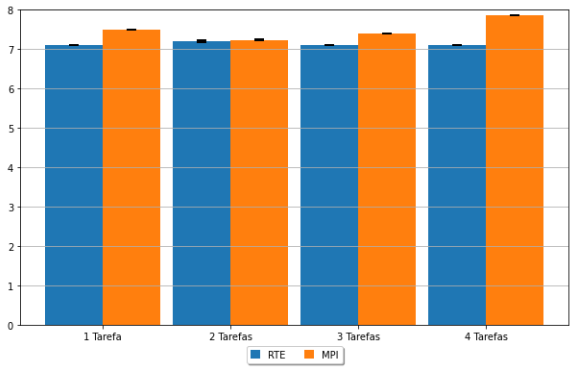
	Média		Desvio Padrão	
Teste	**RTE**	**MPI**	**RTE**	**MPI**
1 Tarefa	2,451141	4,830104	0,131385	0,004448
2 Tarefas	2,499581	3,063760	0,103233	0,068408
3 Tarefas	2,553188	4,307436	0,011524	0,044427
4 Tarefas	2,563202	4,267549	0,047007	0,030027

Fonte: Próprio Autor

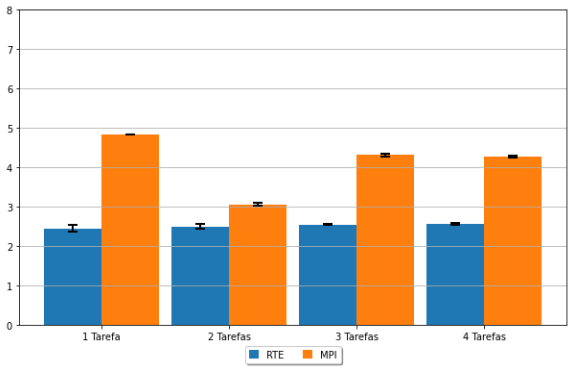
As tabelas 6.18a e 6.18b e a Figura 6.30 verifica-se a diferença nos dois modos de execuções. Pois apresentam tempos no mínimo equivalentes, mas na maioria dos casos RTE são melhores. As diferenças de tempos observados nas execuções com tarefas sequenciais, 6.18, são muito próximos. Já os tempos das execuções com paralelização interna de tarefas com RTE, 6.18b, estão aproximadamente entre 50% a 80% menores que as execuções com MPI. Uma hipótese é de que o impacto do processamento é menor na aplicação com RTE em relação ao custo de processamento nas execuções com MPI, já que para implementação do modelo de programação optou-se pela utilização de processamento “leve” através das *threads* para a paralelização dos conjuntos de tarefas que foram utilizadas como bibliotecas dinâmicas nesse protótipo.

Figura 6.30: Comparação dos tempos da aplicação real entre RTE e MPI.

(a) Com Tarefas sequenciais



(b) Com paralelização interna



Fonte: Próprio Autor

6.7 Análise em ambientes com arquiteturas heterogêneas

A finalidade é avaliar o modelo de programação proposto nesse trabalho através da API RTE quanto ao seu funcionamento em arquiteturas heterogêneas.

6.7.1 Execução de tarefa no co-processador Intel® XEON Phi™

Avalia-se o modelo de programação proposto quanto ao seu funcionamento correto utilizando o co-processador Intel® XEON Phi™.

6.7.1.1 Descrição dos testes

A realização da análise de funcionamento e do comportamento do modelo de programação através de uma implementação utilizando a API RTE para efetuar chamadas e execuções de tarefas distribuídas em um ambiente com arquiteturas heterogêneas composto por processadores x86 e o co-processador Intel® XEON Phi™ é feita através da implementação de uma aplicação simples descrita logo a seguir.

Ambiente de teste

Para realizar esse caso de teste utilizou-se como *Host* um servidor com 4 processadores Intel® XEON®, com 8 cores por processador e com duas *threads* por core. E como *Guest* utilizou-se um acelerador de *hardware* Intel® XEON Phi™ 5100 com 240 processadores conectado no barramento PCIx desse servidor. O sistema operacional utilizado é o Linux CentOS 7, compilador gcc versão 7.3.

Cenário de teste

A implementação realiza duas operações matemáticas básicas, soma e multiplicação, de forma a utilizar simultaneamente recursos da CPU e do acelerador de *hardware* através de chamadas de tarefas da API RTE. O cenário de teste e o conjunto de tarefas utilizados são descritos a seguir:

1 Tarefa Local(*Host*) e 1 Tarefa Remota: Para avaliar o funcionamento correto de chamadas e execuções simultâneas em arquiteturas heterogêneas

implementou-se um programa que realiza duas operações matemáticas de duas variáveis escalares do tipo inteiro. A primeira operação é uma soma feita de forma local no *Host*. Já a segunda operação realiza a chamada da função de multiplicação para ser exutada remotamente no acelerador de *hardware* Intel® MIC (*Guest*). O código do programa encontra-se no Apêndice G.1;

Conjunto de tarefas de operações matemáticas: Um conjunto de tarefas, na forma de uma biblioteca dinâmica proposto na arquitetura do sistema RTE, com tarefas que realizam as operações básicas da matemática como soma, subtração, multiplicação e divisão entre dois argumentos de entrada e devolvem o resultado da operação. O código do conjunto de tarefas encontra-se no Apêndice G.2.

6.7.1.2 Análise dos resultados

A avaliação mostra a viabilidade na implementação do modelo de paralelização de tarefas proposto neste trabalho em aplicações com recursos de arquiteturas heterogêneas. O sistemam demonstra uma facilidade tanto na conexão entre o servidor (*Host*) e o acelerador de *hardware* (*Guest*) quanto no envio das requisições e execuções de tarefas em ambientes híbridos. Independente em qual arquitetura foi realizada a chamada, CPU e/ou Intel® MIC, as execuções são equivalentes.

6.7.2 Execução de tarefa em GPGPU local

Tem-se a finalidade de avaliar a aplicabilidade do modelo de programação proposto nesse trabalho através da API RTE com chamadas de tarefas com acelerador de *hardware* GPGPU localmente no *Host*.

6.7.2.1 Descrição dos testes

A análise do funcionamento e comportamento é feito através de uma implementação utilizando o acelerador de *hardware* GPGPU com a API RTE em uma aplicação que pudesse explorar os recursos desse acelerador *hardware*.

Ambiente de teste

Na realização dos testes utilizou-se como *Host* o servidor com a configuração citada na Capítulo 6. Utilizou-se, também, o acelerador de *hardware* GPGPU NVIDIA® GTX™ com 33 *cores* e 2,4GB de RAM conectado no barramento PCIx. A criação do conjunto de tarefas RTE foi feita pelo compilador nvcc da NVIDIA® na versão Cuda *Compilation Release* V10.2.89. Todos os binários utilizaram a *flag* de otimização -O3 em suas compilações.

Cenário de teste

Os testes foram feitos em implementações com chamadas de tarefas de multiplicação de matrizes a serem executadas internamente em uma *thread* no *Host*. Os tamanhos das matrizes são alocadas dinamicamente em suas respectivas áreas de memória durante o processamento, variando de 128×128, 1024×1024, 2048×2048, 3072×3072, 4096×4096, 5120×5120, 6144×6144, 7168×7168 e 8192×8192 para cada execução. O código do programa encontra-se no Apêndice G.3. Implementou-se o conjunto de tarefas RTE que realiza a multiplicação de duas matrizes em uma GPGPU e responde a matriz resultante ao *Host*, o código do conjunto de tarefas encontra-se no Apêndice G.4.

6.7.2.2 Análise dos resultados

A avaliação da escalabilidade em função ao aumento do tamanho dos dados processados constata que o sistema RTE apresenta estabilidade na chamada de execuções em uma GPGPUs. A Tabela 6.19 apresenta os tempos obtidos.

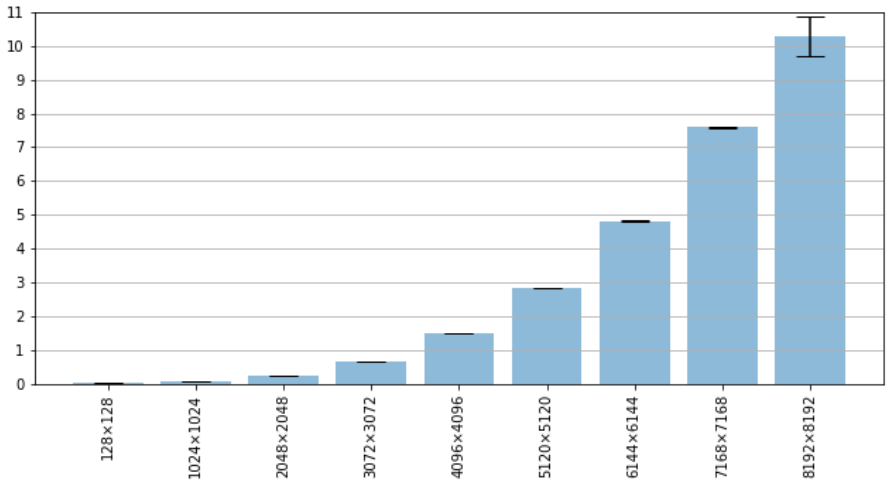
Tabela 6.19: Tempos de execução em GPGPU ao variar o tamanho dos dados.

Tamanho	Média	Desvio Padrão
128×128	0,044538	0,006214
1024×1024	0,078838	0,008017
2048×2048	0,240421	0,007931
3072×3072	0,673170	0,005531
4096×4096	1,495209	0,001742
5120×5120	2,834952	0,002262
6144×6144	4,819858	0,008076
7168×7168	7,588626	0,012109
8192×8192	10,276083	0,955800

Fonte: Próprio Autor

A Figura 6.31 apresenta os gráficos dos tempos para o cenário testado. Observa-se a variação no tempo de execução em função ao aumento do tamanho das matrizes. Essa variação é dada pelo fato de que cada execução realiza a multiplicação de matrizes quadradas de diferentes tamanhos.

Figura 6.31: Tempos de execuções em GPGPU ao variar o tamanho dos dados.



Fonte: Próprio Autor

6.7.3 Execução de tarefa em GPGPU remota

O objetivo é avaliar o comportamento do modelo de programação através da API RTE em aplicações com chamadas de execuções de tarefas remotas utilizando a GPGPU de cada *Guest*.

6.7.3.1 Descrição dos testes

Essa avaliação de funcionamento e do comportamento do modelo de programação é feita através de uma implementação que o *Host* realiza chamadas para a execução de tarefas utilizando a GPGPU em um ou mais *Guests*. Avalia-se a escalabilidade no aumento do número de tarefas simultâneas no ambiente distribuído para tamanho de matrizes de 8192×8192.

Ambiente de teste

Os testes utilizaram os computadores citados no início desse capítulo na página 87, Seção **Plataforma de Teste**. Para a validação das execuções em GPGPU utilizou-se o acelerador de *hardware* GPGPU NVIDIA® GTX™ com 33 *cores* e 2,4GB de RAM conectado no barramento PCIx no *Host* e em cada *Guest*.

Foi utilizado o mesmo conjunto de tarefas RTE da Seção 6.7.2 localizado no Apêndice G.4, porém foi disponibilizado uma cópia nos respectivos *Guests*.

Cenário de teste

Os testes foram feitos em implementações com chamadas de tarefas de multiplicação de matrizes quadradas pelo *Host* para serem executadas em um ou mais *Guests* simultaneamente, sendo que cada tarefa é processada na GPGPU de cada *Guest*. O tamanho das matrizes são alocadas dinamicamente em suas respectivas áreas de memória durante o processamento. Os cenários de testes são mostrados a seguir:

Teste 0 **1 Tarefa Local(*Host*)**: É o mesmo teste executado na Seção 6.7.2, em que é feita a chamada da tarefa de multiplicação das matrizes em uma *thread* internamente no *Host*;

Teste 1 **1 Tarefa Remota**: Nesse cenário o *Host* faz a chamada remota ao *Guest* para a execução de uma tarefa com o uso da GPGPU;

Teste 2 **2 Tarefas Remotas**: Executa 2 chamadas simultâneas, uma para cada *Guest* específico com duas matrizes em cada e em suas GPGPUs;

Teste 3 **3 Tarefas Remotas**: Este cenário é semelhante ao anterior, porém faz três chamadas paralelas de multiplicação de seis matrizes nas GPGPUs de cada *Guest*;

Teste 4 **4 Tarefas Remotas**: Faz 4 chamadas paralelas em cada *Guest* para a multiplicação simultânea de oito matrizes, duas para cada GPGPU em seu *Guest* específico;

Teste 5 **4 Tarefas Remotas + 1 Tarefa Local(*Host*)**: Além das quatro execuções simultâneas remotas é feita a execução de uma tarefa em uma *thread* interna do próprio *Host*, o código do programa está no Apêndice G.5;

6.7.3.2 Análise dos resultados

Ao observar a Tabela 6.20 é possível notar o aumento do tempo a medida que aumenta-se o número de tarefas a serem executadas. Como consta na Seção 6.3.2 pode considerar que, para este cenário, o tempo do *Host* para gerenciamento da requisição de execução de uma tarefa em um nó remoto é de cerca de 4,67 segundos e o tempo de transmissão dos dados é de cerca de 10 segundos.

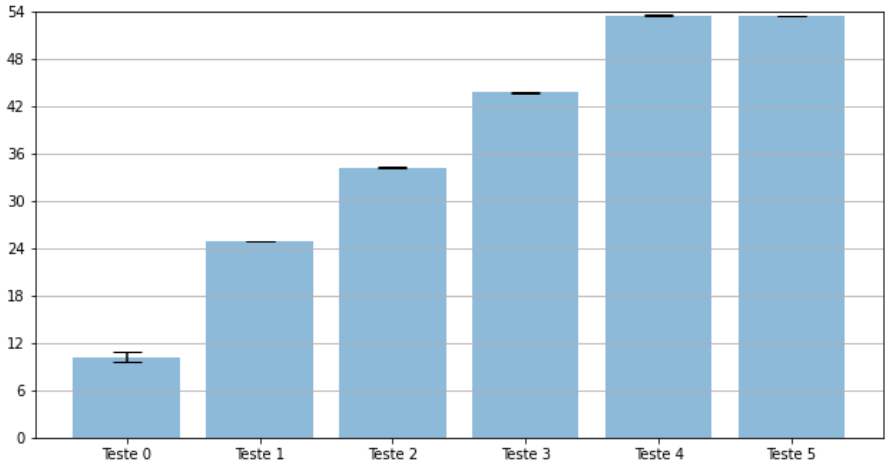
Tabela 6.20: Tempos de execuções remotas com GPGPUs.

Teste	Média	Desvio Padrão
0	10,276083	0,955800
1	24,941862	0,025445
2	34,251445	0,019401
3	43,748995	0,149028
4	53,472928	0,093072
5	53,457072	0,066063

Fonte: Próprio Autor

Ao analisar o gráfico apresentado na Figura 6.32 é possível observar o aumento dos tempos a medida em que o número de tarefas aumenta. Como cada execução realiza a multiplicação de matrizes quadradas com o mesmo tamanho de 8192×8192 pode-se afirmar que essa variação é devido ao envio dos dados das matrizes do *Host* para cada *Guest* e das respostas de cada *Guest* ao *Host*. Tem-se neste teste, que o tempo de processamento da tarefa é similar ao de transmissão de dados, não havendo ganho de desempenho em relação a uma execução sequencial das tarefas no *Host*. A execução de tarefas remotas só é vantajosa quando o tempo de transmissão dos dados é menor do que o tempo de processamento.

Figura 6.32: Tempos de execuções remotas com GPGPUs.



Fonte: Próprio Autor

Análise de desempenho

A avaliação do ganho de desempenho foi feita entre os tempos das execuções remotas na API RTE usando GPGPUs com as suas execuções sequenciais equivalentes, com os tempos dos testes 7, 8, 9 e 10 da Tabela 6.2 na Seção 6.3.1, executadas no *Host*. Usou-se como parâmetros a variação do número de tarefas entre 1 e 4 e a quantidade de dados com o n de 8192×8192. Para os cálculos de ganho de desempenho utilizou-se a seguinte fórmula:

$$S(t_{gpgpu_i}) = \frac{t_{seq_i}}{t_{gpgpu_i}} \tag{6.7}$$

Onde:

$\boldsymbol{t_{gpgpu_i}}$**:** tempo de execução das chamadas e execução das múltiplas tarefas com uso de GPGPU;

$\boldsymbol{t_{seq_i}}$**:** tempo de execução das funções sequenciais;

$\boldsymbol{i}$**:** número de tarefas ou execuções sequenciais.

A Tabela 6.21 apresenta os ganhos de desempenho encontrados nas comparações entre as execuções remotas com o uso da GPGPU e as execuções sequencias dada pela Equação 6.7.

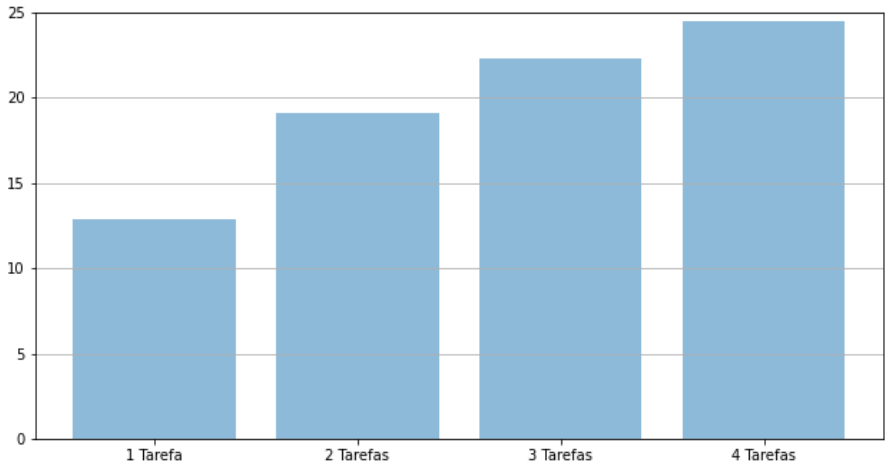
Tabela 6.21: Ganho de desempenho com GPGPU.

Teste	Médias em Tarefas na GPGPU	Médias em Tarefas Sequenciais	Ganhos de Desempenho
1 Tarefa	24,941862	320,296181	12,841711
2 Tarefas	34,251445	654,610624	19,111913
3 Tarefas	43,748995	977,117188	22,334620
4 Tarefas	53,472928	1308,478370	24,469922

Fonte: Próprio Autor

Ao analisar o gráfico apresentado na Figura 6.33 que o ganho de desempenho tem uma melhora expressiva ao aumentar o número de tarefas, mostrando a viabilidade da ferramenta RTE com o uso de aceleradore de *hardware* como GPGPU.

Figura 6.33: Ganho de desempenho com GPGPU.



Fonte: Próprio Autor

7 CONCLUSÃO

Este trabalho apresentou um modelo de programação *Task/Thread Level Parallelism* (TLP) baseado em *Bag Of Tasks* (BoT) em sistemas distribuídos compostos por arquiteturas heterogêneas. O modelo e a sua implementação propõem soluções para reduzir a complexidade ao lidar com o desenvolvimento de aplicações para execução com recursos heterogêneos como CPUs, aceleradores de hardware como GPGPUs, entre outros, em ambientes distribuídos provendo o alto desempenho.

Os excelentes resultados obtidos através dos testes utilizando a biblioteca RTE proposta e implementada mostram que a solução é funcional e apresenta escalabilidade ao aumentar o volume de dados e a quantidade de tarefas em diversos *Guests*. Apresentou vantagens, tais como, a facilidade de desenvolvimento e a utilização de recursos heterogêneos com uma mesma sintaxe, independente da arquitetura. Concluise que o modelo proposto pode ser usado tanto em aplicações em que as tarefas têm alta quantidade de dados nos parâmetros de entrada e resposta, considerando a transmissão destes dados, quanto nos casos com baixa quantidade. Estes dois casos são verificados primeiramente nas análises do custo de comunicação e desempenho, em seguida nos testes com uma aplicação real em que há pouca transmissão de dados. Contudo, a execução de uma tarefa em um recurso remoto, envolvendo alta quantidade de dados de parâmetros e resposta, só é vantajosa quando o custo com a transmissão de dados é menor do que o tempo restrito de processamento da tarefa, como mostrado no caso do teste em que a tarefa que executa uma multiplicação de matrizes em CPU. A facilidade de integração com outros modelos de programação, OpenMP por exemplo, é outro ponto forte dessa solução. A heterogeneidade possuí dois pontos bem relevantes. O primeiro é a facilidade no desenvolvimento de soluções que possam utilizar tais recursos. Outro é a utilização de GPGPUs em diversos *Guests*, nesse caso a vantagem está em usar essa funcionalidade nas situações em que o tempo de processamento seja maior que o tempo de envio dos dados.

Essa solução apresenta facilidades na implementação de soluções em HPC na paralelização de tarefas e ainda possibilita a execução de aplicações com ganho desempenho comprovando assim sua eficiência.

7.1 Publicação

Os primeiros resultados experimentais desse trabalho foram publicados no artigo do 19º WPeformance (BÉLO; SATO; MIDORIKAWA, 2020) o qual tinha como objetivo avaliar o desempenho do modelo de programação utilizado em uma aplicação, através da biblioteca RTE, que necessitasse de alto poder computacional e compará-lo a uma versão utilizando uma implementação da interface MPI. Estes testes apresentaram a viabilidade com características de estabilidade e escalabilidade no uso do modelo de programaçao e com um excelente desempenho comparado com MPI.

7.2 Trabalhos futuros

A evolução dessa pesquisa apresenta novos desafios a serem alcançados, em função disso são propostos os seguintes trabalhos futuros:

1. **Integração com mais Arquiteturas Heterogêneas**: Adaptar a solução à outras tecnologias de aceleradores de *hardware* em HPC, tais como FPGA, ARM e outros modelos de GPGPUs;
2. **Ferramenta de Análise para Seleção do Recurso**: Desenvolvimento de uma função para definir qual a melhor forma de distribuição de tarefas para cada recurso através de uma equação matemática para definir o grau de processamento;
3. **Comparação com Outras Ferramentas de Programação Paralela**: Para avaliar o modelo deve-se realizar uma comparação com outros modelos de programação e ferramentas utilizadas, como as apresentadas em Trabalhos Relacionados;
4. **Novos Testes para Explorar a Paralelização de Tarefas**: Apresentar novos testes que explorem melhor o modelo de programação em aplicações reais com paralelismo de tarefas.

REFERÊNCIAS

AGOSTA, G. *et al.* Managing heterogeneous resources in hpc systems. In: ____. **Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms**. New York, NY, USA: Association for Computing Machinery, 2018. p. 7–12. ISBN 9781450364447. Disponível em: <https://doi.org/10.1145/3183767.3183769>.

AGULLO, E. *et al.* Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. **Acm transactions on mathematical software (toms)**, ACM New York, NY, USA, v. 43, n. 2, p. 1–22, 2016. Disponível em: <https://dl.acm.org/doi/pdf/10.1145/2898348>. Acesso em: 11/07/2022.

ANDIÓN, J. M. *et al.* Locality-aware automatic parallelization for gpgpu with openhmpp directives. **International Journal of Parallel Programming**, Springer, v. 44, n. 3, p. 620–643, 2016.

ASHBAUGH, B. *et al.* Data parallel c++ enhancing sycl through extensions for productivity and performance. In: **Proceedings of the International Workshop on OpenCL**. [S.l.: s.n.], 2020. p. 1–2.

AUGONNET, C. **Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective**. 2011. Tese (Doutorado) — Bordeaux 1, 2011. Disponível em: <https://www.theses.fr/2011BOR14460/document>. Acesso em: 11/07/2022.

AUGONNET, C. *et al.* StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In: SPRINGER. **European Conference on Parallel Processing**. 2009. p. 863–874. Disponível em: <https://link.springer.com/content/pdf/10.1007/978-3-642-03869-3_80.pdf?pdf=inline%20link>. Acesso em: 11/07/2022.

BAK, S. *et al.* Openmp application experiences: Porting to accelerated nodes. **Parallel Computing**, v. 109, p. 102856, 2022. ISSN 0167-8191. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167819121001009>.

BARNEY, B. Posix threads programming. **Lawrence Livermore National Laboratory**, 2017b. Disponível em: <https://hpc-tutorials.llnl.gov/posix/>. Acesso em: 01/06/2022.

BARNEY, B. *et al.* Introduction to parallel computing. **Lawrence Livermore National Laboratory**, 2017a. Disponível em: <https://computing.llnl.gov/tutorials/parallel_comp/>. Acesso em: 01/06/2022.

BÉLO, D. S.; SATO, L.; MIDORIKAWA, E. Avaliação de desempenho de um sistema de programação paralela baseado em tarefas assíncronas. In: **Anais do XIX Workshop em Desempenho de Sistemas Computacionais e de Comunicação**. Porto Alegre, RS, Brasil: SBC, 2020. p. 49–60. ISSN 2595-6167. Disponível em: <https://sol.sbc.org.br/index.php/wperformance/article/view/11105>.

BIHAN, S. *et al.* Directive-based heterogeneous programming–a gpu-accelerated rtm use case. In: **Proceedings of the 7th international conference on computing, communications and control technologies**. [S.l.: s.n.], 2009. v. 40.

BOSCH, J. *et al.* Application acceleration on fpgas with ompss@ fpga. In: IEEE. **2018 International Conference on Field-Programmable Technology (FPT)**. 2018. p. 70–77. Disponível em: <https://engineering.purdue.edu/Cetus/cetusworkshop/papers/3-2.pdf>. Acesso em: 13/07/2022.

BRAZELL, J.; BETTERSWORTH, M. High performance computing. **Technical brief, Texas State Technical College–TSTC Forecasting**, 2008.

CARPENTER, P. *et al.* Heterogeneous high performance computing. 2022. Disponível em: <https://www.etp4hpc.eu/pujades/files/ETP4HPC_WP_Heterogeneous-HPC_20220216.pdf>. Acesso em: 07/11/2022.

CHAPMAN, B.; JOST, G.; PAS, R. V. D. **Using OpenMP: portable shared memory parallel programming**. [S.l.]: MIT press, 2007.

CHRISTGAU, S.; STEINKE, T. Porting a legacy cuda stencil code to oneapi. In: IEEE. **2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [S.l.], 2020. p. 359–367.

COMER, D. E.; STEVENS, D. L. **Internetworking with Tcp/ip: Design, Implementation, and Internals**. [S.l.]: PRENTICE HALL, 1999.

CONSORTIUM, O. *et al.* **OpenHMPP Concepts and Directives**. [S.l.]: Version, 2012.

CRAMER, T. *et al.* Openmp programming on intel r xeon phi tm coprocessors: An early performance comparison. In: **Proc. Many Core Appl. Res. Community (MARC) Symp**. [S.l.: s.n.], 2012. p. 38–44.

DOLBEAU, R.; BIHAN, S.; BODIN, F. Hmpp™: A hybrid multi-core parallel programming environment. v. 28, 2007. Disponível em: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=f4c946e493ac96bc600fe1f40c57c2df73a75397>. Acesso em: 18/01/2020.

FERRER, R. *et al.* Mercurium: Design decisions for a s2s compiler. In: **Cetus Users and Compiler Infastructure Workshop in conjunction with PACT**. [s.n.], 2011. v. 2011. Disponível em: <https://engineering.purdue.edu/Cetus/cetusworkshop/papers/3-2.pdf>. Acesso em: 13/07/2022.

FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE transactions on computers**, IEEE, v. 100, n. 9, p. 948–960, 1972.

HAIDAR, A. *et al.* Unified development for mixed multi-GPU and multi-coprocessor environments using a lightweight runtime environment. p. 491–500, 2014. ISSN 23321237. Disponível em: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6877282>.

HECHTMAN, B. A.; HILTON, A. D.; SORIN, D. J. Trees: A cpu/gpu task-parallel runtime with explicit epoch synchronization. **arXiv preprint arXiv:1608.00571**, 2016. Disponível em: <https://arxiv.org/pdf/1608.00571.pdf>. Acesso em: 10/06/2022.

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture: a quantitative approach**. 6th. ed. [S.l.]: Elsevier, 2017.

INRIA. StarPU Handbook. Version 1.3.7, 2020. Disponível em: <https://files.inria.fr/starpu/starpu-1.3.7/starpu_dev.pdf>. Acesso em: 05/01/2021.

Intel. Intel® oneAPI™ Programming Guide. Version Beta, 2020. Disponível em: <https://software.intel.com/sites/default/files/oneAPIProgrammingGuide_9.pdf>. Acesso em: 27/03/2020.

ISO. ISO - ISO/IEC 9899:1999 - Programming languages — C. 1999. Disponível em: <https://www.iso.org/standard/29237.html>. Acesso em: 12/02/2015.

JEFFERS, J.; REINDERS, J.; SODANI, A. **Intel® Xeon Phi™ Processor High Performance Programming: Knights Landing Edition**. [S.l.]: Morgan Kaufmann, 2016.

JONES, M. T. Anatomy of linux dynamic libraries. **IBM developer-Works**, 2008.

KERRISK, M. **The Linux programming interface: a Linux and UNIX system programming handbook**. [S.l.]: No Starch Press, 2010.

KERYELL, R.; ROVATSOU, M.; HOWES, L. Sycl™ Specification: Sycl™ integrates OpenCL™ devices with modern C++. **Khronos Working Group**, Version 1.2.1, 2019. Disponível em: <https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf>. Acesso em: 27/03/2020.

LANGSAM, Y.; AUGENSTEIN, M.; TENENBAUM, A. M. **Data Structures using C**. [S.l.]: Prentice Hall New Jersey, 1990. v. 1.

LI, K. Openmp accelerator support for gpu. **OpenPOWER Summit**, p. 12, 2016.

MENA, J. A. *et al.* Ompss-2@ cluster: Distributed memory execution of nested openmp-style tasks. In: **European Conference on Parallel Processing: Euro-Par 2022, 2022**. [s.n.], 2022. Disponível em: <http://paul-carpenter.org/aguilar2022europar.pdf>. Acesso em: 10/10/2022.

Message Passing Interface Forum. **MPI: A Message-Passing Interface Standard Version 4.0**. [S.l.], 2021. Disponível em: <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>.

MUÑOZ, A. N. *et al.* Combining dynamic concurrency throttling with voltage and frequency scaling on task-based programming models. In: **50th International Conference on Parallel Processing**. [s.n.], 2021. p. 1–11. Disponível em: <https://dl.acm.org/doi/pdf/10.1145/3472456.3472471>. Acesso em: 15/07/2022.

NESI, L. L. *et al.* Introdução ao desenvolvimento de aplicações paralelas com o paradigma orientado a tarefas e o runtime starpu. **Sociedade Brasileira de Computação**, 2020.

NITSURE, A.; SHRIVASTAVA, H.; DSOUZA, P. Archived | gpu programming made easy with openmp on ibm power – part 1. 2019. Disponível em: <https://developer.ibm.com/articles/gpu-programming-with-openmp/>. Acesso em: 18/07/2022.

OmpSs–2. Ompss-2 specification. 2020. Disponível em: <https://pm.bsc.es/ftp/ompss-2/doc/spec/OmpSs-2-Specification.pdf>. Acesso em: 16/01/2021.

OpenMP. Openmp application program interface. Version 5.2, 2021. Disponível em: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf>. Acesso em: 06/06/2022.

PLANAS, J. *et al.* Hierarchical task-based programming with starss. **The International Journal of High Performance Computing Applications**, Sage Publications Sage UK: London, England, v. 23, n. 3, p. 284–299, 2009.

REINDERS, J. *et al.* **Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL**. [S.l.]: Springer Nature, 2021.

RITCHIE, D. M.; KERNIGHAN, B. W.; LESK, M. E. **The C programming language**. [S.l.]: Prentice Hall Englewood Cliffs, 1988.

ROBERT, D. Á. **Improving nanos6 dependency subsystem**. 2019. Dissertação (B.S. thesis) — Universitat Politècnica de Catalunya, 2019. Disponível em: <https://upcommons.upc.edu/bitstream/handle/2117/170075/137710.pdf?sequence=1&isAllowed=y>. Acesso em: 15/07/2022.

SATO, M. The supercomputer “fugaku” and arm-sve enabled a64fx processor for energy-efficiency and sustained application performance. In: IEEE. **2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)**. [S.l.], 2020. p. 1–5.

SCHMIDT, B. *et al.* **Parallel programming: concepts and practice**. [S.l.]: Morgan Kaufmann, 2017.

SCHMIDT, B.; HILDEBRANDT, A. Next-generation sequencing: big data meets high performance computing. **Drug discovery today**, Elsevier, v. 22, n. 4, p. 712–717, 2017.

STALLMAN, R. M. *et al.* Using the gnu compiler collection. **For GCC version**, v. 8.4.0, 2020. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc-8.4.0/gcc.pdf>. Acesso em: 10/03/2020.

STAVRINIDES, G. L.; KARATZA, H. D. Dynamic scheduling of bags-of-tasks with sensitive input data and end-to-end deadlines in a hybrid cloud. **Multimedia Tools and Applications**, Springer, v. 80, n. 11, p. 16781–16803, 2021.

STEVENS, W. R.; FENNER, B.; RUDOFF, A. M. **UNIX Network Programming: The Sockets Networking API**. [S.l.]: Addison-Wesley Professional, 2004. v. 1.

STRINGHINI, D.; GONÇALVES, R. A.; GOLDMAN, A. Introdução à computação heterogênea. In: **Anais da XXXI Jornada de Atualização em Informática do XXXII Congresso da Sociedade Brasileira de Computação [Internet]**. [S.l.: s.n.], 2012. p. 16–19.

SÜLEYMAN, U.; AKGÜN, D. Performance evaluations for openmp accelerated training of separable image filter. **International Journal of Applied Mathematics Electronics and Computers**, n. Special Issue-1, p. 90–94, 2016.

TANENBAUM, A. S.; STEEN, M. V. **Distributed systems: principles and paradigms**. [S.l.]: Prentice-Hall, 2017.

TRÖGER, P.; FEINBUBE, M. S. F. Parallel programming concepts what kind of programming model can bridge the gap? 2014.

WAZLAWICK, R. S. **Metodologia de Pesquisa para Ciência da Computação**. [S.l.]: Elsevier, 2014. v. 2.

WEN-MEI, W. H.; KIRK, D. B.; HAJJ, I. E. **Programming Massively Parallel Processors: A Hands-on Approach**. [S.l.]: Morgan Kaufmann, 2022.

YARKHAN, A. **Dynamic Task Execution on Shared and Distributed Memory Architectures**. 2012. Tese (Dissertation for the Doctor of Philosophy) — University of Tennessee, 2012. Disponível em: <https://trace.tennessee.edu/cgi/viewcontent.cgi?article=2774&context=utk_graddiss>.

YARKHAN, A.; KURZAK, J.; DONGARRA, J. Quark users' guide. **Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tennessee**, v. 268, 2011. Disponível em: <https://icl.utk.edu/projectsfiles/plasma/pubs/56-quark_users_guide.pdf>. Acesso em: 17/08/2021.

APÊNDICE A – EXEMPLO DE TESTE DE FUNCIONALIDADE

A.1 Soma e multiplicação escalar de duas variávies inteiras executadas remotamente

```
1  //soma escalar, c=a + b
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/time.h>
5
6  #include <rte.h>
7
8  #include <mede_time.h>
9
10 rte_dvcs_types_t list_type_dvc[ 128 ];
11 int a,b,c,d;
12
13 int main()
14 {
15         int i,j;
16         int id_task;
17
18         list_type_dvc[0] = rte_dvc_cpu;  // CPU
19
20         rte_init_dvcs_lst( list_type_dvc, 1 );
21
22         printf("INICIO SOMA_ESCALAR\n");
23         fflush(stdout);
24
25         if ( rte_init( "soma-mult_escalar.cfg" ) != 0 ) {
26                 printf( "Nao inicializar o Processador Remoto .\n" );
27                 return 1;
28         }
29
30         a=7;
31         b=5;
32
33         printf("a=%d  b=%d  \n", a,b);
34         fflush(stdout);
35
36         rte_arg_list_t argin;
37         rte_ret_list_t argret;
38
39         rte_init_args( 2, &argin, 2, &argret );  //2=numero de
               parametros  argin=entrada
40
41         argin.arg[0].size= sizeof(int);
42         argin.arg[0].arg=(char *)&a;
43         argin.arg[1].size= sizeof(int);
```

```
44          argin.arg[1].arg=(char *)&b;
45          argret.ret[0].size= sizeof(int);
46          argret.ret[0].ret=(char *)&c;
47          argret.ret[1].size= sizeof(int);
48          argret.ret[1].ret=(char *)&d;
49
50          TIMER_CLEAR;
51          TIMER_START;
52
53          id_task=rte_tsk_call( 0, 0, "soma_mult\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libsoma-mult_escalar.so\0", &
                argin, &argret );
54
55          printf("id_task=%d\n",id_task);
56          fflush(stdout);
57
58          rte_sync(id_task); // Synk Task
59
60          TIMER_STOP;
61
62          printf("RESULT c=%d d=%d\n",c,d);
63          printf("TERMINOU task1\n");
64          printf("Tempo: %f \n", TIMER_ELAPSED);
65          fflush(stdout);
66
67          rte_finalize();
68          printf("FIM\n");
69          fflush(stdout);
70
71          return 0;
72  }
```

APÊNDICE B – EXEMPLOS DE TESTES DE AVALIAÇÃO DO CUSTO DE COMUNICAÇÃO

B.1 Teste de execução com chamada remota *localhost*

```
1  //multiplicacao de matrizes N x N (alocacao estatica), c=a x b
2  //versao 1 tarefa  - mult sequencial
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/time.h>
6
7  #include <rte.h>
8  #include <mede_time.h>
9
10 #define N 2000
11 #define MAX_DEVICES 100
12
13 int list_type_dvc[MAX_DEVICES];
14
15 int size_matrix;
16 double a[N][N];
17 double b[N][N];
18 double c[N][N];
19
20 int main()
21 {
22         int i,j;
23         int id_task;
24         list_type_dvc[0]=0;  // CPU
25         printf("INICIO\n");
26         fflush(stdout);
27         int size_matrix=N;
28
29         if ( rte_init( "mult-0-static.cfg" ) != 0 ) {
30                 printf( "Nao inicializar o Processador Remoto .\n" );
31                 return 1;
32         }
33
34         printf("\nTamanho em Bytes Matriz A = %zu\n", sizeof(a));
35         fflush(stdout);
36         for (i = 0; i < size_matrix; i++)
37                 for (j=0; j < size_matrix; j++){
38                         a[i][j] = 1.0;
39                         b[i][j] = 1.0;
40                 }
41         printf("\nTamanho em Bytes Matriz A = %d\n", size_matrix);
42         fflush(stdout);
```

```
43          rte_arg_list_t  argin1;
44          rte_ret_list_t argret1;
45          rte_init_args(3,&argin1,1,&argret1);
46          argin1.arg[0].size= sizeof(int);
47          argin1.arg[0].arg=(char *)&size_matrix;;
48          argin1.arg[1].size= size_matrix*size_matrix*sizeof(double);
49          argin1.arg[1].arg=(char *)a;
50          argin1.arg[2].size= size_matrix*size_matrix*sizeof(double);
51          argin1.arg[2].arg=(char *)b;
52          argret1.ret[0].size= size_matrix*size_matrix*sizeof(double);
53          argret1.ret[0].ret=(char *)c;
54
55          printf("\nENDERECO SIZE_MATRIX  %p %p\n",argin1.arg[0].arg,&
                size_matrix);
56          printf("\nENDERECO C  %p\n",c);
57          fflush(stdout);
58
59          TIMER_CLEAR;
60          TIMER_START;
61
62          id_task=rte_tsk_call(0, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-0-static.so\0", &
                argin1,  &argret1);
63          printf("id_task=%d\n",id_task);
64          fflush(stdout);
65          rte_sync(id_task); // Synk Task
66          TIMER_STOP;
67          printf("Tempo: %f\n", TIMER_ELAPSED);
68
69          printf("matrix_c[0][0]=%f\n",c[0][0]);
70          printf("TERMINOU task1\n");
71          printf("matrix_c[999][999]=%f\n",c[999][999]);
72          fflush(stdout);
73          rte_finalize();
74
75          printf("FIM\n");
76          return 0;
77  }
```

B.2 Teste de um programa com execução sequencial

```
1   //multiplicacao de matrizes N x N (alocacao estatica), c=a x b
2   //versao  mult sequencial
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <sys/time.h>
6   #include <mede_time.h>
7   #define N 2000
8   int size_matrix=N;
9   double a[N*N];
10  double b[N*N];
11  double c[N*N];
12
13  void mult_matrix()
14  {
15          int i,k,j;
16          for (i = 0; i < size_matrix; i++)
17                  for (j=0; j < size_matrix; j++)
18                          c[i*size_matrix+j] = 0.0;
19          // multiplica
20          for (i = 0; i < size_matrix; i++)
21                  for (k=0; k < size_matrix; k++)
22                          for (j=0; j < size_matrix; j++)
```

```
23                                      c[i*size_matrix+j] += a[i*size_matrix+k]
                                            * b[k*size_matrix+j];
24  }
25
26  int main()
27  {
28          int i,j;
29          int id_task;
30          printf("INICIO\n");
31          printf("\nTamanho em Bytes Matriz A = %ld\n", sizeof(a));
32          fflush(stdout);
33          for (i = 0; i < size_matrix; i++)
34                  for (j=0; j < size_matrix; j++){
35                          a[i*size_matrix+j] = 1.0;
36                          b[i*size_matrix+j] = 1.0;
37                  }
38          printf("\nTamanho em Bytes Matriz A = %d\n", size_matrix);
39          fflush(stdout);
40
41          TIMER_CLEAR;
42          TIMER_START;
43
44          mult_matrix();
45
46          TIMER_STOP;
47
48          printf("Tempo: %f \n", TIMER_ELAPSED);
49          printf("matrix_c[0][0]=%f\n",c[0]);
50          printf("matrix_c[999][999]=%f\n",c[999*999]);
51          printf("FIM\n");
52          fflush(stdout);
53          return 0;
54  }
```

APÊNDICE C – TESTES DO CUSTO DE COMUNICAÇÃO E DO DESEMPENHO COM CHAMADAS DE TAREFAS SEQUENCIAIS

C.1 Teste de execução de 4 tarefas sequenciais em ambiente distribuído com o uso da API RTE

```
1  //multiplicacao de matrizes N x N (alocacao dinamica), c=a x b
2  //versao 1 tarefa  - mult sequencial
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/time.h>
6
7  #include <rte.h>
8
9  #include <mede_time.h>
10
11 #define NTHREADS 8
12
13 rte_dvcs_types_t list_type_dvc[ 128 ];
14
15 int size_matrix;
16 double* a1;
17 double* b1;
18 double* c1;
19
20 double* a2;
21 double* b2;
22 double* c2;
23
24 double* a3;
25 double* b3;
26 double* c3;
27
28 double* a4;
29 double* b4;
30 double* c4;
31
32 int main(int argc,char **argv)
33 {
34         int i,j;
35         int id_task1, id_task2, id_task3, id_task4;
36         list_type_dvc[0]=rte_dvc_cpu;   // CPU
37
```

```
38          rte_arg_list_t  argin1;
39          rte_ret_list_t argret1;
40
41          rte_arg_list_t argin2;
42          rte_ret_list_t argret2;
43
44          rte_arg_list_t argin3;
45          rte_ret_list_t argret3;
46
47          rte_arg_list_t argin4;
48          rte_ret_list_t argret4;
49
50          printf("INICIO\n");
51          fflush(stdout);
52          if (argc==1) {
53                  printf("FALTOU ESPECIFICAR N: TAMANHO DAS MATRIZES\n");
54                  fflush(stdout);
55                  return 1;
56          }
57
58          if ( rte_init( "mult-0-dynamic.cfg" ) != 0 ) {
59                  printf( "Nao inicializar o Processador Remoto .\n" );
60                  fflush(stdout);
61                  return 1;
62          }
63
64          rte_init_dvcs_lst( list_type_dvc, 1 );
65
66          size_matrix= atoi(argv[1]);
67          printf ("size_matrix=%d\n",size_matrix);
68
69          a1 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
70          b1 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
71          c1 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
72
73          a2 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
74          b2 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
75          c2 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
76
77          a3 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
78          b3 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
79          c3 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
80
81          a4 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
82          b4 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
83          c4 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
84
85          printf("INICIO\n");
86          fflush(stdout);
87
88          /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
                >>>>>>> */
89          for (i = 0; i < size_matrix; i++)
90                  for (j=0; j < size_matrix; j++){
91                          a1[i*size_matrix+j] = 1.0;
92                          b1[i*size_matrix+j] = 1.0;
93
94                          a2[i*size_matrix+j] = 2.0;
95                          b2[i*size_matrix+j] = 2.0;
96
97                          a3[i*size_matrix+j] = 3.0;
98                          b3[i*size_matrix+j] = 3.0;
99
100                         a4[i*size_matrix+j] = 4.0;
```

```
101                     b4[i*size_matrix+j] = 4.0;
102             }
103
104             rte_init_args(3,&argin1, 1,&argret1);
105
106             argin1.arg[0].size= sizeof(int);
107             argin1.arg[0].arg=(char *)&size_matrix;;
108             argin1.arg[1].size= size_matrix*size_matrix*sizeof(double);
109             argin1.arg[1].arg=(char *)a1;
110             argin1.arg[2].size= size_matrix*size_matrix*sizeof(double);
111             argin1.arg[2].arg=(char *)b1;
112             argret1.ret[0].size= size_matrix*size_matrix*sizeof(double);
113             argret1.ret[0].ret=(char *)c1;
114
115             printf("\nENDERECO SIZE_MATRIX 1 %u %u\n",argin1.arg[0].arg,&
                    size_matrix);
116             printf("\nENDERECO C1 %u\n", c1);
117             fflush(stdout);
118
119             rte_init_args(3,&argin2, 1,&argret2);
120
121             argin2.arg[0].size= sizeof(int);
122             argin2.arg[0].arg=(char *)&size_matrix;;
123             argin2.arg[1].size= size_matrix*size_matrix*sizeof(double);
124             argin2.arg[1].arg=(char *)a2;
125             argin2.arg[2].size= size_matrix*size_matrix*sizeof(double);
126             argin2.arg[2].arg=(char *)b2;
127             argret2.ret[0].size= size_matrix*size_matrix*sizeof(double);
128             argret2.ret[0].ret=(char *)c2;
129
130             printf("\nENDERECO SIZE_MATRIX %u %u\n",argin2.arg[0].arg, &
                    size_matrix);
131             printf("\nENDERECO C2 %u \n",c2);
132             fflush(stdout);
133
134             rte_init_args(3,&argin3, 1,&argret3);
135
136             argin3.arg[0].size= sizeof(int);
137             argin3.arg[0].arg=(char *)&size_matrix;;
138             argin3.arg[1].size= size_matrix*size_matrix*sizeof(double);
139             argin3.arg[1].arg=(char *)a3;
140             argin3.arg[2].size= size_matrix*size_matrix*sizeof(double);
141             argin3.arg[2].arg=(char *)b3;
142             argret3.ret[0].size= size_matrix*size_matrix*sizeof(double);
143             argret3.ret[0].ret=(char *)c3;
144
145             printf("\nENDERECO SIZE_MATRIX 3 %u %u\n", argin3.arg[0].arg, &
                    size_matrix);
146             printf("\nENDERECO C3 %u\n", c3);
147             fflush(stdout);
148
149             rte_init_args(3,&argin4, 1,&argret4);
150
151             argin4.arg[0].size= sizeof(int);
152             argin4.arg[0].arg=(char *)&size_matrix;;
153             argin4.arg[1].size= size_matrix*size_matrix*sizeof(double);
154             argin4.arg[1].arg=(char *)a4;
155             argin4.arg[2].size= size_matrix*size_matrix*sizeof(double);
156             argin4.arg[2].arg=(char *)b4;
157             argret4.ret[0].size= size_matrix*size_matrix*sizeof(double);
158             argret4.ret[0].ret=(char *)c4;
159
160             printf("\nENDERECO SIZE_MATRIX %u %u\n",argin4.arg[0].arg, &
                    size_matrix);
```

```
161         printf("\nENDERECO C4 %u \n",c4);
162         fflush(stdout);
163
164         TIMER_CLEAR;
165         TIMER_START;
166
167         id_task1=rte_tsk_call(0, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-0-dynamic.so\0", &
                argin1,  &argret1);
168         printf("id_task1=%d\n",id_task1);
169         fflush(stdout);
170
171         id_task2=rte_tsk_call(1, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-0-dynamic.so\0", &
                argin2,  &argret2);
172         printf("id_task2=%d\n",id_task2);
173         fflush(stdout);
174
175         id_task3=rte_tsk_call(2, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-0-dynamic.so\0", &
                argin3,  &argret3);
176         printf("id_task3=%d\n",id_task3);
177         fflush(stdout);
178
179         id_task4=rte_tsk_call(3, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-0-dynamic.so\0", &
                argin4,  &argret4);
180         printf("id_task4=%d\n",id_task4);
181         fflush(stdout);
182
183         rte_sync(id_task1); // Synk Task 1
184         printf("Aps rte_sync [1]\n");
185         printf("matrix_c1[0][0]=%f\n",c1[0]);
186         printf("TERMINOU task1\n");
187         printf("matrix_c1[%d][%d]=%f\n\n",size_matrix-1,size_matrix-1,c1
                [(size_matrix-1)*(size_matrix-1)]);
188         fflush(stdout);
189
190         rte_sync(id_task2); // Synk Task 2
191         printf("Aps rte_sync [2]\n");
192         printf("matrix_c2[0][0]=%f\n",c2[0]);
193         printf("TERMINOU task2\n");
194         printf("matrix_c2[%d][%d]=%f\n\n",size_matrix-1,size_matrix-1,c2
                [(size_matrix-1)*(size_matrix-1)]);
195         fflush(stdout);
196
197         rte_sync(id_task3); // Synk Task 3
198         printf("Aps rte_sync [3]\n");
199         printf("matrix_c3[0][0]=%f\n",c3[0]);
200         printf("TERMINOU task3\n");
201         printf("matrix_c3[%d][%d]=%f\n\n",size_matrix-1,size_matrix-1,c3
                [(size_matrix-1)*(size_matrix-1)]);
202         fflush(stdout);
203
204         rte_sync(id_task4); // Synk Task 4
205         printf("Aps rte_sync [4]\n");
206         printf("matrix_c4[0][0]=%f\n",c4[0]);
207         printf("TERMINOU task4\n");
208         printf("matrix_c4[%d][%d]=%f\n",size_matrix-1,size_matrix-1,c4[(
                size_matrix-1)*(size_matrix-1)]);
209         fflush(stdout);
210
211         TIMER_STOP;
212
```

```
213         printf("Tempo: %f \n", TIMER_ELAPSED);
214         fflush(stdout);
215
216         rte_finalize();
217
218         printf("FIM\n");
219         fflush(stdout);
220         return 0;
221 }
```

C.2 Teste de execução de chamadas distintas de 4 funções sequenciais

```
 1 //multiplicacao de matrizes N x N (alocacao estatica), c=a x b
 2 //versao  mult sequencial
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5 #include <sys/time.h>
 6
 7 #include <mede_time.h>
 8
 9 #define N 4000
10 double* a1;
11 double* b1;
12 double* c1;
13
14 double* a2;
15 double* b2;
16 double* c2;
17
18 double* a3;
19 double* b3;
20 double* c3;
21
22 double* a4;
23 double* b4;
24 double* c4;
25
26 int size_matrix;
27
28 void mult_matrix(double* a, double* b, double* c)
29 {
30         int i,k,j;
31         for (i = 0; i < size_matrix; i++)
32                 for (j=0; j < size_matrix; j++)
33                         c[i*size_matrix+j] = 0.0;
34     // multiplica
35         for (i = 0; i < size_matrix; i++)
36                 for (k=0; k < size_matrix; k++)
37                         for (j=0; j < size_matrix; j++)
38                                 c[i*size_matrix+j] += a[i*size_matrix+k]
                                        * b[k*size_matrix+j];
39 }
40
41 int main(int argc,char **argv)
42 {
43         int i,j;
44         int id_task;
45         printf("INICIO\n");
46         fflush(stdout);
47         if (argc==1) {
48                 printf("FALTOU ESPECIFICAR N: TAMANHO DAS MATRIZES\n");
49                 return 1;
50         }
```

```
51          size_matrix= atoi(argv[1]);
52          printf ("size_matrix=%d\n",size_matrix);
53          a1=(double*)calloc(size_matrix*size_matrix,sizeof(double));
54          b1=(double*)calloc(size_matrix*size_matrix,sizeof(double));
55          c1=(double*)calloc(size_matrix*size_matrix,sizeof(double));
56
57          a2=(double*)calloc(size_matrix*size_matrix,sizeof(double));
58          b2=(double*)calloc(size_matrix*size_matrix,sizeof(double));
59          c2=(double*)calloc(size_matrix*size_matrix,sizeof(double));
60
61          a3=(double*)calloc(size_matrix*size_matrix,sizeof(double));
62          b3=(double*)calloc(size_matrix*size_matrix,sizeof(double));
63          c3=(double*)calloc(size_matrix*size_matrix,sizeof(double));
64
65          a4=(double*)calloc(size_matrix*size_matrix,sizeof(double));
66          b4=(double*)calloc(size_matrix*size_matrix,sizeof(double));
67          c4=(double*)calloc(size_matrix*size_matrix,sizeof(double));
68
69          printf("INICIO\n");
70          fflush(stdout);
71          for (i = 0; i < size_matrix; i++)
72                  for (j=0; j < size_matrix; j++){
73                          a1[i*size_matrix+j] = 1.0;
74                          b1[i*size_matrix+j] = 1.0;
75
76                          a2[i*size_matrix+j] = 2.0;
77                          b2[i*size_matrix+j] = 2.0;
78
79                          a3[i*size_matrix+j] = 3.0;
80                          b3[i*size_matrix+j] = 3.0;
81
82                          a4[i*size_matrix+j] = 4.0;
83                          b4[i*size_matrix+j] = 4.0;
84                  }
85          printf("\nTamanho em Bytes Matriz A = %d\n", size_matrix);
86          fflush(stdout);
87
88          TIMER_CLEAR;
89          TIMER_START;
90
91          mult_matrix(a1,b1,c1);
92          printf("matrix_c1[0][0]=%f\n",c1[0]);
93          printf("matrix_c1[%d][%d]=%f \n",(size_matrix-1),(size_matrix-1)
                ,c1[(size_matrix-1)*(size_matrix-1)]);
94          fflush(stdout);
95
96          mult_matrix(a2,b2,c2);
97          printf("matrix_c2[0][0]=%f\n",c1[0]);
98          printf("matrix_c2[%d][%d]=%f \n",(size_matrix-1),(size_matrix-1)
                ,c2[(size_matrix-1)*(size_matrix-1)]);
99          fflush(stdout);
100
101         mult_matrix(a3,b3,c3);
102         printf("matrix_c3[0][0]=%f\n",c3[0]);
103         printf("matrix_c3[%d][%d]=%f \n",(size_matrix-1),(size_matrix-1)
                ,c3[(size_matrix-1)*(size_matrix-1)]);
104         fflush(stdout);
105
106         mult_matrix(a4,b4,c4);
107         printf("matrix_c4[0][0]=%f\n",c4[0]);
108         printf("matrix_c4[%d][%d]=%f \n",(size_matrix-1),(size_matrix-1)
                ,c4[(size_matrix-1)*(size_matrix-1)]);
109         fflush(stdout);
110
```

```
111             TIMER_STOP;
112
113             printf("Tempo: %f \n", TIMER_ELAPSED);
114             printf("FIM\n");
115             fflush(stdout);
116
117             return 0;
118     }
```

APÊNDICE D – TESTES DO CUSTO DE COMUNICAÇÃO E DO DESEMPENHO COM CHAMADAS DE TAREFAS COM PARALELIZAÇÃO INTERNA

D.1 Teste de execução de 4 tarefas com paralelização interna em ambiente distribuído com o uso da API RTE

```
1  //multiplicacao de matrizes N x N (alocacao dinamica), c=a x b
2  //versao 4 tarefas  - mult paralela (openmp)
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/time.h>
6
7  #include <rte.h>
8
9  #include <mede_time.h>
10
11 #define MAX_DEVICES 100
12
13 rte_dvcs_types_t list_type_dvc[ MAX_DEVICES ];
14
15 int size_matrix;
16 double* a1;
17 double* b1;
18 double* c1;
19
20 double* a2;
21 double* b2;
22 double* c2;
23
24 double* a3;
25 double* b3;
26 double* c3;
27
28 double* a4;
29 double* b4;
30 double* c4;
31
32 int main(int argc,char **argv)
33 {
34         int i,j;
```

```
35          int id_task1, id_task2, id_task3, id_task4;
36          list_type_dvc[0]=rte_dvc_cpu;   // CPU
37
38          rte_arg_list_t  argin1;
39          rte_ret_list_t argret1;
40
41          rte_arg_list_t argin2;
42          rte_ret_list_t argret2;
43
44          rte_arg_list_t argin3;
45          rte_ret_list_t argret3;
46
47          rte_arg_list_t argin4;
48          rte_ret_list_t argret4;
49
50          printf("INICIO\n");
51          fflush(stdout);
52          if (argc==1) {
53                  printf("FALTOU ESPECIFICAR N: TAMANHO DAS MATRIZES\n");
54                  fflush(stdout);
55                  return 1;
56          }
57
58          if ( rte_init( "mult-0-dynamic-par.cfg" ) != 0 ) {
59                  printf( "Nao inicializar o Processador Remoto .\n" );
60                  fflush(stdout);
61                  return 1;
62          }
63
64          rte_init_dvcs_lst( list_type_dvc, 1 );
65
66          size_matrix= atoi(argv[1]);
67          printf ("size_matrix=%d\n",size_matrix);
68
69          a1 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
70          b1 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
71          c1 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
72
73          a2 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
74          b2 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
75          c2 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
76
77          a3 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
78          b3 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
79          c3 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
80
81          a4 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
82          b4 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
83          c4 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
84
85          printf("INICIO\n");
86          fflush(stdout);
87
88          /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
               >>>>>>> */
89          for (i = 0; i < size_matrix; i++)
90                  for (j=0; j < size_matrix; j++){
91                          a1[i*size_matrix+j] = 1.0;
92                          b1[i*size_matrix+j] = 1.0;
93
94                          a2[i*size_matrix+j] = 2.0;
95                          b2[i*size_matrix+j] = 2.0;
96
97                          a3[i*size_matrix+j] = 3.0;
```

```
 98                      b3[i*size_matrix+j] = 3.0;
 99
100                      a4[i*size_matrix+j] = 4.0;
101                      b4[i*size_matrix+j] = 4.0;
102          }
103
104          rte_init_args(3,&argin1, 1,&argret1);
105
106          argin1.arg[0].size= sizeof(int);
107          argin1.arg[0].arg=(char *)&size_matrix;;
108          argin1.arg[1].size= size_matrix*size_matrix*sizeof(double);
109          argin1.arg[1].arg=(char *)a1;
110          argin1.arg[2].size= size_matrix*size_matrix*sizeof(double);
111          argin1.arg[2].arg=(char *)b1;
112          argret1.ret[0].size= size_matrix*size_matrix*sizeof(double);
113          argret1.ret[0].ret=(char *)c1;
114
115          printf("\nENDERECO SIZE_MATRIX 1 %u %u\n",argin1.arg[0].arg,&
                 size_matrix);
116          printf("\nENDERECO C1 %u\n", c1);
117          fflush(stdout);
118
119          rte_init_args(3,&argin2, 1,&argret2);
120
121          argin2.arg[0].size= sizeof(int);
122          argin2.arg[0].arg=(char *)&size_matrix;;
123          argin2.arg[1].size= size_matrix*size_matrix*sizeof(double);
124          argin2.arg[1].arg=(char *)a2;
125          argin2.arg[2].size= size_matrix*size_matrix*sizeof(double);
126          argin2.arg[2].arg=(char *)b2;
127          argret2.ret[0].size= size_matrix*size_matrix*sizeof(double);
128          argret2.ret[0].ret=(char *)c2;
129
130          printf("\nENDERECO SIZE_MATRIX %u %u\n",argin2.arg[0].arg, &
                 size_matrix);
131          printf("\nENDERECO C2 %u \n",c2);
132          fflush(stdout);
133
134          rte_init_args(3,&argin3, 1,&argret3);
135
136          argin3.arg[0].size= sizeof(int);
137          argin3.arg[0].arg=(char *)&size_matrix;;
138          argin3.arg[1].size= size_matrix*size_matrix*sizeof(double);
139          argin3.arg[1].arg=(char *)a3;
140          argin3.arg[2].size= size_matrix*size_matrix*sizeof(double);
141          argin3.arg[2].arg=(char *)b3;
142          argret3.ret[0].size= size_matrix*size_matrix*sizeof(double);
143          argret3.ret[0].ret=(char *)c3;
144
145          printf("\nENDERECO SIZE_MATRIX 3 %u %u\n", argin3.arg[0].arg, &
                 size_matrix);
146          printf("\nENDERECO C3 %u\n", c3);
147          fflush(stdout);
148
149          rte_init_args(3,&argin4, 1,&argret4);
150
151          argin4.arg[0].size= sizeof(int);
152          argin4.arg[0].arg=(char *)&size_matrix;;
153          argin4.arg[1].size= size_matrix*size_matrix*sizeof(double);
154          argin4.arg[1].arg=(char *)a4;
155          argin4.arg[2].size= size_matrix*size_matrix*sizeof(double);
156          argin4.arg[2].arg=(char *)b4;
157          argret4.ret[0].size= size_matrix*size_matrix*sizeof(double);
158          argret4.ret[0].ret=(char *)c4;
```

```
159
160         printf("\nENDERECO SIZE_MATRIX %u %u\n",argin4.arg[0].arg, &
                size_matrix);
161         printf("\nENDERECO C4 %u \n",c4);
162         fflush(stdout);
163
164         TIMER_CLEAR;
165         TIMER_START;
166
167         id_task1=rte_tsk_call(0, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-0-dynamic-par-4thrds.
                so\0", &argin1,  &argret1);
168         printf("id_task1=%d\n",id_task1);
169         fflush(stdout);
170
171         id_task2=rte_tsk_call(1, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-0-dynamic-par-4thrds.
                so\0", &argin2,  &argret2);
172         printf("id_task2=%d\n",id_task2);
173         fflush(stdout);
174
175         id_task3=rte_tsk_call(2, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-0-dynamic-par-4thrds.
                so\0", &argin3,  &argret3);
176         printf("id_task3=%d\n",id_task3);
177         fflush(stdout);
178
179         id_task4=rte_tsk_call(3, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-0-dynamic-par-4thrds.
                so\0", &argin4,  &argret4);
180         printf("id_task4=%d\n",id_task4);
181         fflush(stdout);
182
183         rte_sync(id_task1); // Synk Task 1
184         printf("Aps rte_sync [1]\n");
185         printf("matrix_c1[0][0]=%f\n",c1[0]);
186         printf("TERMINOU task1\n");
187         printf("matrix_c1[%d][%d]=%f\n\n",size_matrix-1,size_matrix-1,c1
                [(size_matrix-1)*(size_matrix-1)]);
188         fflush(stdout);
189
190         rte_sync(id_task2); // Synk Task 2
191         printf("Aps rte_sync [2]\n");
192         printf("matrix_c2[0][0]=%f\n",c2[0]);
193         printf("TERMINOU task2\n");
194         printf("matrix_c2[%d][%d]=%f\n\n",size_matrix-1,size_matrix-1,c2
                [(size_matrix-1)*(size_matrix-1)]);
195         fflush(stdout);
196
197         rte_sync(id_task3); // Synk Task 3
198         printf("Aps rte_sync [3]\n");
199         printf("matrix_c3[0][0]=%f\n",c3[0]);
200         printf("TERMINOU task3\n");
201         printf("matrix_c3[%d][%d]=%f\n\n",size_matrix-1,size_matrix-1,c3
                [(size_matrix-1)*(size_matrix-1)]);
202         fflush(stdout);
203
204         rte_sync(id_task4); // Synk Task 4
205         printf("Aps rte_sync [4]\n");
206         printf("matrix_c4[0][0]=%f\n",c4[0]);
207         printf("TERMINOU task4\n");
208         printf("matrix_c4[%d][%d]=%f\n",size_matrix-1,size_matrix-1,c4[(
                size_matrix-1)*(size_matrix-1)]);
209         fflush(stdout);
```

```
210
211             TIMER_STOP;
212
213             printf("Tempo: %f \n", TIMER_ELAPSED);
214             fflush(stdout);
215
216             rte_finalize();
217
218             printf("FIM\n");
219             fflush(stdout);
220             return 0;
221     }
```

D.2 Teste de execução sequencial de chamadas distintas de 4 funções com paralelização interna

```
1   //multiplicacao de matrizes N x N (alocacao estatica), c=a x b
2   //versao  mult sequencial
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <sys/time.h>
6
7   #include <omp.h>
8
9   #include <mede_time.h>
10
11  #define N 4000
12  #define NTHREADS 4
13
14  double* a1;
15  double* b1;
16  double* c1;
17
18  double* a2;
19  double* b2;
20  double* c2;
21
22  double* a3;
23  double* b3;
24  double* c3;
25
26  double* a4;
27  double* b4;
28  double* c4;
29
30  int size_matrix;
31
32  void mult_matrix(double* a, double* b, double* c)
33  {
34          int i,k,j;
35          for (i = 0; i < size_matrix; i++)
36                  for (j=0; j < size_matrix; j++)
37                  c[i*size_matrix+j] = 0.0;
38          // multiplica
39          #pragma omp parallel num_threads(NTHREADS) private(i, j, k)
40          {
41                  #pragma omp for
42                  for (i = 0; i < size_matrix; i++)
43                  {
44                          for (k=0; k < size_matrix; k++)
45                                  for (j=0; j < size_matrix; j++)
46                                          c[i*size_matrix+j] += a[i*
                                                size_matrix+k] * b[k*
```

```
                                                size_matrix+j];
47                  }
48          }
49  }
50
51  int main(int argc,char **argv)
52  {
53          int i,j;
54          int id_task;
55          printf("INICIO\n");
56          fflush(stdout);
57          if (argc==1) {
58                  printf("FALTOU ESPECIFICAR N: TAMANHO DAS MATRIZES\n");
59                  return 1;
60          }
61          size_matrix= atoi(argv[1]);
62          printf ("size_matrix=%d\n",size_matrix);
63          a1=(double*)calloc(size_matrix*size_matrix,sizeof(double));
64          b1=(double*)calloc(size_matrix*size_matrix,sizeof(double));
65          c1=(double*)calloc(size_matrix*size_matrix,sizeof(double));
66
67          a2=(double*)calloc(size_matrix*size_matrix,sizeof(double));
68          b2=(double*)calloc(size_matrix*size_matrix,sizeof(double));
69          c2=(double*)calloc(size_matrix*size_matrix,sizeof(double));
70
71          a3=(double*)calloc(size_matrix*size_matrix,sizeof(double));
72          b3=(double*)calloc(size_matrix*size_matrix,sizeof(double));
73          c3=(double*)calloc(size_matrix*size_matrix,sizeof(double));
74
75          a4=(double*)calloc(size_matrix*size_matrix,sizeof(double));
76          b4=(double*)calloc(size_matrix*size_matrix,sizeof(double));
77          c4=(double*)calloc(size_matrix*size_matrix,sizeof(double));
78
79          printf("INICIO\n");
80          fflush(stdout);
81          for (i = 0; i < size_matrix; i++)
82                  for (j=0; j < size_matrix; j++){
83                          a1[i*size_matrix+j] = 1.0;
84                          b1[i*size_matrix+j] = 1.0;
85
86                          a2[i*size_matrix+j] = 2.0;
87                          b2[i*size_matrix+j] = 2.0;
88
89                          a3[i*size_matrix+j] = 3.0;
90                          b3[i*size_matrix+j] = 3.0;
91
92                          a4[i*size_matrix+j] = 4.0;
93                          b4[i*size_matrix+j] = 4.0;
94                  }
95          printf("\nTamanho em Bytes Matriz A = %d\n", size_matrix);
96          fflush(stdout);
97
98          TIMER_CLEAR;
99          TIMER_START;
100
101         mult_matrix(a1,b1,c1);
102         printf("matrix_c1[0][0]=%f\n",c1[0]);
103         printf("matrix_c1[%d][%d]=%f \n",(size_matrix-1),(size_matrix-1)
                ,c1[(size_matrix-1)*(size_matrix-1)]);
104         fflush(stdout);
105
106         mult_matrix(a2,b2,c2);
107         printf("matrix_c2[0][0]=%f\n",c1[0]);
108         printf("matrix_c2[%d][%d]=%f \n",(size_matrix-1),(size_matrix-1)
```

```
            ,c2[(size_matrix-1)*(size_matrix-1)]);
109         fflush(stdout);
110
111         mult_matrix(a3,b3,c3);
112         printf("matrix_c3[0][0]=%f\n",c3[0]);
113         printf("matrix_c3[%d][%d]=%f \n",(size_matrix-1),(size_matrix-1)
            ,c3[(size_matrix-1)*(size_matrix-1)]);
114         fflush(stdout);
115
116         mult_matrix(a4,b4,c4);
117         printf("matrix_c4[0][0]=%f\n",c4[0]);
118         printf("matrix_c4[%d][%d]=%f \n",(size_matrix-1),(size_matrix-1)
            ,c4[(size_matrix-1)*(size_matrix-1)]);
119         fflush(stdout);
120
121         TIMER_STOP;
122
123         printf("Tempo: %f \n", TIMER_ELAPSED);
124         printf("FIM\n");
125         fflush(stdout);
126
127         return 0;
128 }
```

D.3 Conjunto de tarefas com paralelização interna no padrão do sistema RTE

```
 1 //multiplicacao de matrizes - versao paralela
 2 //matrizes a e b
 3 //matriz resultante c
 4 #include <math.h>
 5 #include <stdio.h>
 6 #include <string.h>
 7 #include <omp.h>
 8
 9 #define NTHREADS 4
10 #include <rte.h>
11
12 void __attribute__((visibility ("default"))) mult_matrix(void *arg)
13 {
14         int i, j, k;
15         printf("TESTE\n");
16         fflush(stdout);
17         int * size;
18         double *a;
19         double *b;
20         double *c;
21         rte_args_t *arg_task=(rte_args_t *)arg;  // enderecos de
               argumentos e endereco de resposta
22         size=(int*)arg_task->args[0];
23         c=(double *)arg_task->rets[0];
24         int size_matrix=*size;
25         a=(double *)arg_task->args[1];
26         b=(double *)arg_task->args[2];
27         printf("tamanho sz=%d\n",size_matrix);
28         fflush(stdout);
29         printf("a[0][0]=%f  \n",a[0]);
30         for (i = 0; i < size_matrix; i++)
31                 for (j=0; j < size_matrix; j++)
32                         c[i*size_matrix+j] = 0.0;
33         // multiplica
34         #pragma omp parallel num_threads(NTHREADS) private(i, j, k)
35         {
36                 #pragma omp for
```

```
37                  for (i = 0; i < size_matrix; i++)
38                  {
39                          for (k=0; k < size_matrix; k++)
40                                  for (j=0; j < size_matrix; j++)
41                                          c[i*size_matrix+j] += a[i*
                                                size_matrix+k] * b[k*
                                                size_matrix+j];
42                  }
43          }
44
45          printf("FUNCAO mat_c[%d][%d]=%f \n",size_matrix-1,size_matrix-1,
                c[(size_matrix-1)*(size_matrix-1)]);
46          fflush(stdout);
47
48          printf("FIM mult\n");
49          fflush(stdout);
50  }
```

APÊNDICE E – TESTES DO CUSTO DE COMUNICAÇÃO E DO DESEMPENHO COM CHAMADAS DE TAREFAS SEQUENCIAIS COM O USO DA API MPI

E.1 Teste de execução de 4 tarefas sequenciais em ambiente distribuído com o uso da API MPI

```
1  //multiplicacao de matrizes N x N (alocacao dinamica), c=a x b
2  //versao 1 tarefa  - mult sequencial
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/time.h>
6  #include <mede_time.h>
7  #include <mpi.h>
8  #include <stdio.h>
9
10 int size_matrix;
11 double *a;
12 double *b;
13 double *x;
14 double *c;
15 double *d;
16 double *y;
17 double *e;
18 double *f;
19 double *z;
20 double *g;
21 double *h;
22 double *w;
23
24 void  mult_matrix(int size_matrix, double *a,double *b,double *c)
25 {
26         int i, j, k;
27 //printf("tamanho sz=%d\n",size_matrix);
28 //fflush(stdout);
29 //        printf("a[0][0]=%f  \n",a[0]);
30         for (i = 0; i < size_matrix; i++)
31             for (j=0; j < size_matrix; j++)
32                   c[i*size_matrix+j] = 0.0;
33         for (i = 0; i < size_matrix; i++)
34             {
```

```
35                      for (k=0; k < size_matrix; k++)
36                              for (j=0; j < size_matrix; j++)
37                                      c[i*size_matrix+j] += a[i*size_matrix+k]
                                            * b[k*size_matrix+j];
38                  }
39  //printf("FUNCAO mat_c[%d][%d]=%f \n",size_matrix-1,size_matrix-1,c[(
        size_matrix-1)*(size_matrix-1)]);
40  //fflush(stdout);
41  //  printf("FIM mult\n");
42  //  fflush(stdout);
43  }
44  int main(int argc,char **argv)
45  {
46          int             n,n_nos, rank;
47          MPI_Status      status;
48          MPI_Request req;
49          int i,j;
50          int id_task;
51          MPI_Init(&argc, &argv);
52          MPI_Comm_size(MPI_COMM_WORLD, &n_nos);
53          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
54          printf("INICIO\n");
55          fflush(stdout);
56          size_matrix= atoi(argv[1]);
57          printf ("size_matrix=%d\n",size_matrix);
58          a=(double *)calloc(size_matrix*size_matrix,sizeof(double));
59          b=(double *)calloc(size_matrix*size_matrix,sizeof(double));
60          x=(double *)calloc(size_matrix*size_matrix,sizeof(double));
61          c=(double *)calloc(size_matrix*size_matrix,sizeof(double));
62          d=(double *)calloc(size_matrix*size_matrix,sizeof(double));
63          y=(double *)calloc(size_matrix*size_matrix,sizeof(double));
64          e=(double *)calloc(size_matrix*size_matrix,sizeof(double));
65          f=(double *)calloc(size_matrix*size_matrix,sizeof(double));
66          z=(double *)calloc(size_matrix*size_matrix,sizeof(double));
67          g=(double *)calloc(size_matrix*size_matrix,sizeof(double));
68          h=(double *)calloc(size_matrix*size_matrix,sizeof(double));
69          w=(double *)calloc(size_matrix*size_matrix,sizeof(double));
70          if (rank == 0) {
71                  //TIMER_CLEAR;
72                  //TIMER_START;
73                  for (i = 0; i < size_matrix; i++)
74                          for (j=0; j < size_matrix; j++){
75                                  a[i*size_matrix+j] = 1.0;
76                                  b[i*size_matrix+j] = 1.0;
77                                  c[i*size_matrix+j] = 2.0;
78                                  d[i*size_matrix+j] = 2.0;
79                                  e[i*size_matrix+j] = 3.0;
80                                  f[i*size_matrix+j] = 3.0;
81                                  g[i*size_matrix+j] = 4.0;
82                                  h[i*size_matrix+j] = 4.0;
83                          }
84                  TIMER_CLEAR;
85                  TIMER_START;
86                  MPI_Isend(a,size_matrix*size_matrix,MPI_DOUBLE,1,10,
                        MPI_COMM_WORLD,&req);
87                  MPI_Isend(b,size_matrix*size_matrix,MPI_DOUBLE,1,11,
                        MPI_COMM_WORLD,&req);
88                  MPI_Isend(c,size_matrix*size_matrix,MPI_DOUBLE,2,12,
                        MPI_COMM_WORLD,&req);
89                  MPI_Isend(d,size_matrix*size_matrix,MPI_DOUBLE,2,13,
                        MPI_COMM_WORLD,&req);
90                  MPI_Isend(e,size_matrix*size_matrix,MPI_DOUBLE,3,14,
                        MPI_COMM_WORLD,&req);
91                  MPI_Isend(f,size_matrix*size_matrix,MPI_DOUBLE,3,15,
```

```
                    MPI_COMM_WORLD,&req);
92                  MPI_Isend(g,size_matrix*size_matrix,MPI_DOUBLE,4,16,
                        MPI_COMM_WORLD,&req);
93                  MPI_Isend(h,size_matrix*size_matrix,MPI_DOUBLE,4,17,
                        MPI_COMM_WORLD,&req);
94                  for (i=0;i<4;i++){
95                  MPI_Probe(MPI_ANY_SOURCE,9,MPI_COMM_WORLD,&status);
96                  if (status.MPI_SOURCE == 1)
97                          MPI_Recv(x,size_matrix*size_matrix,MPI_DOUBLE
                            ,1,9,MPI_COMM_WORLD,&status);
98                  else if(status.MPI_SOURCE == 2)
99                          MPI_Recv(y,size_matrix*size_matrix,MPI_DOUBLE
                            ,2,9,MPI_COMM_WORLD,&status);
100                         else if (status.MPI_SOURCE == 3)
101                                 MPI_Recv(z,size_matrix*size_matrix,
                                    MPI_DOUBLE,3,9,MPI_COMM_WORLD,&status
                                    );
102                                 else
103                                         MPI_Recv(w,size_matrix*
                                            size_matrix,MPI_DOUBLE,4,9,
                                            MPI_COMM_WORLD,&status);
104                 }
105                 TIMER_STOP;
106                 printf("Tempo=%f\n",TIMER_ELAPSED);
107                 printf("matrix_x[0][0]=%f\n",x[0]);
108                 printf("matrix_x[%d][%d]=%f\n",size_matrix-1,size_matrix
                        -1,x[(size_matrix-1)*(size_matrix-1)]);
109                 printf("matrix_y[0][0]=%f\n",y[0]);
110                 printf("matrix_y[%d][%d]=%f\n",size_matrix-1,size_matrix
                        -1,y[(size_matrix-1)*(size_matrix-1)]);
111                 printf("matrix_z[0][0]=%f\n",z[0]);
112                 printf("matrix_z[%d][%d]=%f\n",size_matrix-1,size_matrix
                        -1,z[(size_matrix-1)*(size_matrix-1)]);
113                 printf("matrix_w[0][0]=%f\n",w[0]);
114                 printf("matrix_w[%d][%d]=%f\n",size_matrix-1,size_matrix
                        -1,w[(size_matrix-1)*(size_matrix-1)]);
115         }
116         else{
117                 MPI_Recv(a,size_matrix*size_matrix,MPI_DOUBLE,0,10+(rank
                        -1)*2,MPI_COMM_WORLD,&status);
118                 MPI_Recv(b,size_matrix*size_matrix,MPI_DOUBLE,0,11+(rank
                        -1)*2,MPI_COMM_WORLD,&status);
119                 mult_matrix(size_matrix,a,b,c);
120                 MPI_Send(c,size_matrix*size_matrix,MPI_DOUBLE,0,9,
                        MPI_COMM_WORLD);
121         }
122         MPI_Finalize();
123         printf("FIM\n");
124         return 0;
125 }
```

E.2 Teste de execução de 4 tarefas com paralelização interna em ambiente distribuído com o uso da API MPI

```
1  //multiplicacao de matrizes N x N (alocacao dinamica), c=a x b
2  //versao 1 tarefa  - mult sequencial
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/time.h>
6  #include <mede_time.h>
7  #include <mpi.h>
8  #include <stdio.h>
9  #define NTHREADS 4
```

```
10
11 int size_matrix;
12 double *a;
13 double *b;
14 double *x;
15 double *c;
16 double *d;
17 double *y;
18 double *e;
19 double *f;
20 double *z;
21 double *g;
22 double *h;
23 double *w;
24
25 void  mult_matrix(int size_matrix, double *a,double *b,double *c)
26 {
27         int i, j, k;
28 //printf("tamanho sz=%d\n",size_matrix);
29 //fflush(stdout);
30 //        printf("a[0][0]=%f  \n",a[0]);
31         for (i = 0; i < size_matrix; i++)
32             for (j=0; j < size_matrix; j++)
33                   c[i*size_matrix+j] = 0.0;
34         #pragma omp parallel for num_threads(NTHREADS) private(i, j, k)
35                 for (i = 0; i < size_matrix; i++)
36                 {
37                           for (k=0; k < size_matrix; k++)
38                                 for (j=0; j < size_matrix; j++)
39                                       c[i*size_matrix+j] += a[i*
                                          size_matrix+k] * b[k*
                                          size_matrix+j];
40                 }
41
42 //printf("FUNCAO mat_c[%d][%d]=%f \n",size_matrix-1,size_matrix-1,c[(
       size_matrix-1)*(size_matrix-1)]);
43 //fflush(stdout);
44 //  printf("FIM mult\n");
45 //  fflush(stdout);
46 }
47 int main(int argc,char **argv)
48 {
49         int             n,n_nos, rank;
50         MPI_Status      status;
51         MPI_Request req;
52         int i,j;
53         int id_task;
54         int provided;
55         MPI_Init_thread(&argc, &argv,MPI_THREAD_MULTIPLE,&provided);
56         MPI_Comm_size(MPI_COMM_WORLD, &n_nos);
57         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
58         printf("INICIO\n");
59         fflush(stdout);
60         size_matrix= atoi(argv[1]);
61         printf ("size_matrix=%d\n",size_matrix);
62         a=(double *)calloc(size_matrix*size_matrix,sizeof(double));
63         b=(double *)calloc(size_matrix*size_matrix,sizeof(double));
64         x=(double *)calloc(size_matrix*size_matrix,sizeof(double));
65         c=(double *)calloc(size_matrix*size_matrix,sizeof(double));
66         d=(double *)calloc(size_matrix*size_matrix,sizeof(double));
67         y=(double *)calloc(size_matrix*size_matrix,sizeof(double));
68         e=(double *)calloc(size_matrix*size_matrix,sizeof(double));
69         f=(double *)calloc(size_matrix*size_matrix,sizeof(double));
70         z=(double *)calloc(size_matrix*size_matrix,sizeof(double));
```

```
71          g=(double *)calloc(size_matrix*size_matrix,sizeof(double));
72          h=(double *)calloc(size_matrix*size_matrix,sizeof(double));
73          w=(double *)calloc(size_matrix*size_matrix,sizeof(double));
74          if (rank == 0) {
75                  //TIMER_CLEAR;
76                  //TIMER_START;
77                  for (i = 0; i < size_matrix; i++)
78                          for (j=0; j < size_matrix; j++){
79                                  a[i*size_matrix+j] = 1.0;
80                                  b[i*size_matrix+j] = 1.0;
81                                  c[i*size_matrix+j] = 2.0;
82                                  d[i*size_matrix+j] = 2.0;
83                                  e[i*size_matrix+j] = 3.0;
84                                  f[i*size_matrix+j] = 3.0;
85                                  g[i*size_matrix+j] = 4.0;
86                                  h[i*size_matrix+j] = 4.0;
87                          }
88                  TIMER_CLEAR;
89                  TIMER_START;
90                  MPI_Isend(a,size_matrix*size_matrix,MPI_DOUBLE,1,10,
                        MPI_COMM_WORLD,&req);
91                  MPI_Isend(b,size_matrix*size_matrix,MPI_DOUBLE,1,11,
                        MPI_COMM_WORLD,&req);
92                  MPI_Isend(c,size_matrix*size_matrix,MPI_DOUBLE,2,12,
                        MPI_COMM_WORLD,&req);
93                  MPI_Isend(d,size_matrix*size_matrix,MPI_DOUBLE,2,13,
                        MPI_COMM_WORLD,&req);
94                  MPI_Isend(e,size_matrix*size_matrix,MPI_DOUBLE,3,14,
                        MPI_COMM_WORLD,&req);
95                  MPI_Isend(f,size_matrix*size_matrix,MPI_DOUBLE,3,15,
                        MPI_COMM_WORLD,&req);
96                  MPI_Isend(g,size_matrix*size_matrix,MPI_DOUBLE,4,16,
                        MPI_COMM_WORLD,&req);
97                  MPI_Isend(h,size_matrix*size_matrix,MPI_DOUBLE,4,17,
                        MPI_COMM_WORLD,&req);
98                  for (i=0;i<4;i++){
99                  MPI_Probe(MPI_ANY_SOURCE,9,MPI_COMM_WORLD,&status);
100                 if (status.MPI_SOURCE == 1)
101                         MPI_Recv(x,size_matrix*size_matrix,MPI_DOUBLE
                                ,1,9,MPI_COMM_WORLD,&status);
102                 else if(status.MPI_SOURCE == 2)
103                         MPI_Recv(y,size_matrix*size_matrix,MPI_DOUBLE
                                ,2,9,MPI_COMM_WORLD,&status);
104                         else if (status.MPI_SOURCE == 3)
105                                 MPI_Recv(z,size_matrix*size_matrix,
                                        MPI_DOUBLE,3,9,MPI_COMM_WORLD,&status
                                        );
106                                 else
107                                         MPI_Recv(w,size_matrix*
                                                size_matrix,MPI_DOUBLE,4,9,
                                                MPI_COMM_WORLD,&status);
108                 }
109                 TIMER_STOP;
110                 printf("Tempo=%f\n",TIMER_ELAPSED);
111                 printf("matrix_x[0][0]=%f\n",x[0]);
112                 printf("matrix_x[%d][%d]=%f\n",size_matrix-1,size_matrix
                        -1,x[(size_matrix-1)*(size_matrix-1)]);
113                 printf("matrix_y[0][0]=%f\n",y[0]);
114                 printf("matrix_y[%d][%d]=%f\n",size_matrix-1,size_matrix
                        -1,y[(size_matrix-1)*(size_matrix-1)]);
115                 printf("matrix_z[0][0]=%f\n",z[0]);
116                 printf("matrix_z[%d][%d]=%f\n",size_matrix-1,size_matrix
                        -1,z[(size_matrix-1)*(size_matrix-1)]);
117                 printf("matrix_w[0][0]=%f\n",w[0]);
118                 printf("matrix_w[%d][%d]=%f\n",size_matrix-1,size_matrix
```

```
                    -1,w[(size_matrix-1)*(size_matrix-1)]);
119         }
120         else{
121                 MPI_Recv(a,size_matrix*size_matrix,MPI_DOUBLE,0,10+(rank
                    -1)*2,MPI_COMM_WORLD,&status);
122                 MPI_Recv(b,size_matrix*size_matrix,MPI_DOUBLE,0,11+(rank
                    -1)*2,MPI_COMM_WORLD,&status);
123                 mult_matrix(size_matrix,a,b,c);
124                 MPI_Send(c,size_matrix*size_matrix,MPI_DOUBLE,0,9,
                    MPI_COMM_WORLD);
125         }
126         MPI_Finalize();
127         printf("FIM\n");
128         return 0;
129 }
```

APÊNDICE F – TESTES EM UMA APLICAÇÃO REAL

F.1 Testes de análise de desempenho em uma aplicação real

F.1.1 Teste de execução de 4 tarefas no uso da API RTE para contagem de uma única cadeia de DNAs de tamanho fixo em genomas distintos

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <error.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <sys/time.h>
8
9  #include <mede_time.h>
10
11 #include <rte.h>
12
13 #include <mede_time.h>
14
15 #define SIZE 128
16
17 int main(int argc, char *argv[])
18 {
19         int id_task2,
20                 id_task3,
21                 id_task4,
22                 id_task5;
23
24         int n_ocorrencias2 = 0,
25                 n_ocorrencias3 = 0,
26                 n_ocorrencias4 = 0,
27                 n_ocorrencias5 = 0;
28
29         /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
               >>>>>>> */
30         // Node 2
31         char file2[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCA_008795835.1_PanLeo1.0_genomic.fna.txt\0";
32         char word2[SIZE] = "TCGATTCC\0";
33
34         // Node 3
35         char file3[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000331955.2_Oorc_1.1_genomic.fna.txt\0";
36         char word3[SIZE] = "TCGATTCC\0";
37
38         // Node 4
```

```
39          char file4[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000464555.1_PanTig1.0_genomic.fna.txt\0";
40          char word4[SIZE] = "TCGATTCC\0";
41
42          // Node 5
43          char file5[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_002863925.1_EquCab3.0_genomic.fna.txt\0";
44          char word5[SIZE] = "TCGATTCC\0";
45
46          /* <<<<<< F I M      D A      I N I C I A L I Z A   D O S   A R G
                U M E N T O S >>>>>>> */
47
48          if ( rte_init("busca.cfg\0") != 0 ) {
49                  error (1 , 0 , "Nao inicializar o Processador Remoto .\n
                       " );
50                  return 1;
51          }
52
53          rte_arg_list_t argin2, argin3, argin4 , argin5;
54          rte_ret_list_t argret2, argret3, argret4, argret5;
55
56          rte_init_args(2,&argin2, 1,&argret2);
57          argin2.arg[0].size = SIZE;
58          argin2.arg[0].arg =(char *)file2;
59          argin2.arg[1].size = SIZE;
60          argin2.arg[1].arg = (char *)word2;
61          argret2.ret[0].size = sizeof(int);
62          argret2.ret[0].ret=(char *)&n_ocorrencias2;
63
64          rte_init_args(2,&argin3, 1,&argret3);
65          argin3.arg[0].size = SIZE;
66          argin3.arg[0].arg =(char *)file3;
67          argin3.arg[1].size = SIZE;
68          argin3.arg[1].arg = (char *)word3;
69          argret3.ret[0].size = sizeof(int);
70          argret3.ret[0].ret=(char *)&n_ocorrencias3;
71
72          rte_init_args(2,&argin4, 1,&argret4);
73          argin4.arg[0].size = SIZE;
74          argin4.arg[0].arg =(char *)file4;
75          argin4.arg[1].size = SIZE;
76          argin4.arg[1].arg = (char *)word4;
77          argret4.ret[0].size = sizeof(int);
78          argret4.ret[0].ret=(char *)&n_ocorrencias4;
79
80          rte_init_args(2,&argin5, 1,&argret5);
81          argin5.arg[0].size = SIZE;
82          argin5.arg[0].arg =(char *)file5;
83          argin5.arg[1].size = SIZE;
84          argin5.arg[1].arg = (char *)word5;
85          argret5.ret[0].size = sizeof(int);
86          argret5.ret[0].ret=(char *)&n_ocorrencias5;
87
88          TIMER_CLEAR;
89          TIMER_START;
90
91          id_task2 = rte_tsk_call(0, 0, "find_word\0", "/home/dbelo/
               Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin2, &
               argret2);
92          id_task3 = rte_tsk_call(1, 0, "find_word\0", "/home/dbelo/
               Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin3, &
               argret3);
93          id_task4 = rte_tsk_call(2, 0, "find_word\0", "/home/dbelo/
               Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin4, &
```

```
            argret4);
 94         id_task5 = rte_tsk_call(3, 0, "find_word\0", "/home/dbelo/
            Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin5, &
            argret5);
 95
 96         rte_sync( id_task2 );
 97         printf("Node 2 >> OCORRENCIAS=%d\n", n_ocorrencias2);
 98         fflush( stdout );
 99
100         rte_sync( id_task3 );
101         printf("Node 3 >> OCORRENCIAS=%d\n", n_ocorrencias3);
102         fflush( stdout );
103
104         rte_sync( id_task4 );
105         printf("Node 4 >> OCORRENCIAS=%d\n", n_ocorrencias4);
106         fflush( stdout );
107
108         rte_sync( id_task5 );
109         printf("Node 5 >> OCORRENCIAS=%d\n", n_ocorrencias5);
110         fflush( stdout );
111
112         TIMER_STOP;
113
114         printf("Tempo: %f \n\n", TIMER_ELAPSED);
115         fflush( stdout );
116
117         return 0;
118 }
```

F.1.2 Teste de execução de 4 tarefas com paralelização interna no uso da API RTE para contagem de uma única cadeia de DNAs de tamanho fixo em genomas distintos

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <error.h>
 5 #include <unistd.h>
 6 #include <stdlib.h>
 7 #include <sys/time.h>
 8
 9 #include <mede_time.h>
10
11 #include <rte.h>
12
13 #include <mede_time.h>
14
15 #define SIZE 128
16
17 int main(int argc, char *argv[])
18 {
19         int id_task2,
20                 id_task3,
21                 id_task4,
22                 id_task5;
23
24         int n_ocorrencias2 = 0,
25                 n_ocorrencias3 = 0,
26                 n_ocorrencias4 = 0,
27                 n_ocorrencias5 = 0;
28
29         /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
            >>>>>>> */
```

```
30          // Node 2
31          char file2[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCA_008795835.1_PanLeo1.0_genomic.fna.txt\0";
32          char word2[SIZE] = "TCGATTCC\0";
33
34          // Node 3
35          char file3[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000331955.2_Oorc_1.1_genomic.fna.txt\0";
36          char word3[SIZE] = "TCGATTCC\0";
37
38          // Node 4
39          char file4[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000464555.1_PanTig1.0_genomic.fna.txt\0";
40          char word4[SIZE] = "TCGATTCC\0";
41
42          // Node 5
43          char file5[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_002863925.1_EquCab3.0_genomic.fna.txt\0";
44          char word5[SIZE] = "TCGATTCC\0";
45
46          /* <<<<<< F I M      D A      I N I C I A L I Z A   D O S   A R G
                U M E N T O S >>>>>>> */
47
48          if ( rte_init("busca-par.cfg\0") != 0 ) {
49                  error (1 , 0 , "Nao inicializar o Processador Remoto .\n
                       " );
50                  return 1;
51          }
52
53          rte_arg_list_t argin2, argin3, argin4 , argin5;
54          rte_ret_list_t argret2, argret3, argret4, argret5;
55
56          rte_init_args(2,&argin2, 1,&argret2);
57          argin2.arg[0].size = SIZE;
58          argin2.arg[0].arg =(char *)file2;
59          argin2.arg[1].size = SIZE;
60          argin2.arg[1].arg = (char *)word2;
61          argret2.ret[0].size = sizeof(int);
62          argret2.ret[0].ret=(char *)&n_ocorrencias2;
63
64          rte_init_args(2,&argin3, 1,&argret3);
65          argin3.arg[0].size = SIZE;
66          argin3.arg[0].arg =(char *)file3;
67          argin3.arg[1].size = SIZE;
68          argin3.arg[1].arg = (char *)word3;
69          argret3.ret[0].size = sizeof(int);
70          argret3.ret[0].ret=(char *)&n_ocorrencias3;
71
72          rte_init_args(2,&argin4, 1,&argret4);
73          argin4.arg[0].size = SIZE;
74          argin4.arg[0].arg =(char *)file4;
75          argin4.arg[1].size = SIZE;
76          argin4.arg[1].arg = (char *)word4;
77          argret4.ret[0].size = sizeof(int);
78          argret4.ret[0].ret=(char *)&n_ocorrencias4;
79
80          rte_init_args(2,&argin5, 1,&argret5);
81          argin5.arg[0].size = SIZE;
82          argin5.arg[0].arg =(char *)file5;
83          argin5.arg[1].size = SIZE;
84          argin5.arg[1].arg = (char *)word5;
85          argret5.ret[0].size = sizeof(int);
86          argret5.ret[0].ret=(char *)&n_ocorrencias5;
87
```

```
88          TIMER_CLEAR;
89          TIMER_START;
90
91          id_task2 = rte_tsk_call(0, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin2, &
                argret2);
92          id_task3 = rte_tsk_call(1, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin3, &
                argret3);
93          id_task4 = rte_tsk_call(2, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin4, &
                argret4);
94          id_task5 = rte_tsk_call(3, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin5, &
                argret5);
95
96          rte_sync( id_task2 );
97          printf("Node 2 >> OCORRENCIAS=%d\n", n_ocorrencias2);
98          fflush( stdout );
99
100         rte_sync( id_task3 );
101         printf("Node 3 >> OCORRENCIAS=%d\n", n_ocorrencias3);
102         fflush( stdout );
103
104         rte_sync( id_task4 );
105         printf("Node 4 >> OCORRENCIAS=%d\n", n_ocorrencias4);
106         fflush( stdout );
107
108         rte_sync( id_task5 );
109         printf("Node 5 >> OCORRENCIAS=%d\n", n_ocorrencias5);
110         fflush( stdout );
111
112         TIMER_STOP;
113
114         printf("Tempo: %f \n\n", TIMER_ELAPSED);
115         fflush( stdout );
116
117         return 0;
118 }
```

F.1.3 Teste com paralização de 4 tarefas para a contagem de DNAs distintos de mesmo tamanho em um genoma específico com o uso da API RTE

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <error.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <sys/time.h>
8
9  #include <mede_time.h>
10
11 #include <rte.h>
12
13 #include <mede_time.h>
14
15 #define SIZE 128
16
17 int main(int argc, char *argv[])
18 {
19         int id_task2,
20                 id_task3,
```

```
21                 id_task4,
22                 id_task5;
23
24         int n_ocorrencias2 = 0,
25                 n_ocorrencias3 = 0,
26                 n_ocorrencias4 = 0,
27                 n_ocorrencias5 = 0;
28
29         /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
               >>>>>>> */
30         // Node 2
31         char file2[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
32         char word2[SIZE] = "TCGATTCC\0";
33
34         // Node 3
35         char file3[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
36         char word3[SIZE] = "TCGATTGG\0";
37
38         // Node 4
39         char file4[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
40         char word4[SIZE] = "TCGATTAA\0";
41
42         // Node 5
43         char file5[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
44         char word5[SIZE] = "TCGATTTT\0";
45
46         /* <<<<<< F I M      D A      I N I C I A L I Z A   D O S   A R G
                U M E N T O S >>>>>>> */
47
48         if ( rte_init("busca-par.cfg\0") != 0 ) {
49                 error (1 , 0 , "Nao inicializar o Processador Remoto .\n
                       " );
50                 return 1;
51         }
52
53         rte_arg_list_t argin2, argin3, argin4 , argin5;
54         rte_ret_list_t argret2, argret3, argret4, argret5;
55
56         rte_init_args(2,&argin2, 1,&argret2);
57         argin2.arg[0].size = SIZE;
58         argin2.arg[0].arg =(char *)file2;
59         argin2.arg[1].size = SIZE;
60         argin2.arg[1].arg = (char *)word2;
61         argret2.ret[0].size = sizeof(int);
62         argret2.ret[0].ret=(char *)&n_ocorrencias2;
63
64         rte_init_args(2,&argin3, 1,&argret3);
65         argin3.arg[0].size = SIZE;
66         argin3.arg[0].arg =(char *)file3;
67         argin3.arg[1].size = SIZE;
68         argin3.arg[1].arg = (char *)word3;
69         argret3.ret[0].size = sizeof(int);
70         argret3.ret[0].ret=(char *)&n_ocorrencias3;
71
72         rte_init_args(2,&argin4, 1,&argret4);
73         argin4.arg[0].size = SIZE;
74         argin4.arg[0].arg =(char *)file4;
75         argin4.arg[1].size = SIZE;
76         argin4.arg[1].arg = (char *)word4;
77         argret4.ret[0].size = sizeof(int);
```

```
78          argret4.ret[0].ret=(char *)&n_ocorrencias4;
79
80          rte_init_args(2,&argin5, 1,&argret5);
81          argin5.arg[0].size = SIZE;
82          argin5.arg[0].arg =(char *)file5;
83          argin5.arg[1].size = SIZE;
84          argin5.arg[1].arg = (char *)word5;
85          argret5.ret[0].size = sizeof(int);
86          argret5.ret[0].ret=(char *)&n_ocorrencias5;
87
88          TIMER_CLEAR;
89          TIMER_START;
90
91          id_task2 = rte_tsk_call(0, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin2, &
                argret2);
92          id_task3 = rte_tsk_call(1, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin3, &
                argret3);
93          id_task4 = rte_tsk_call(2, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin4, &
                argret4);
94          id_task5 = rte_tsk_call(3, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin5, &
                argret5);
95
96          rte_sync( id_task2 );
97          printf("Node 2 >> OCORRENCIAS=%d\n", n_ocorrencias2);
98          fflush( stdout );
99
100         rte_sync( id_task3 );
101         printf("Node 3 >> OCORRENCIAS=%d\n", n_ocorrencias3);
102         fflush( stdout );
103
104         rte_sync( id_task4 );
105         printf("Node 4 >> OCORRENCIAS=%d\n", n_ocorrencias4);
106         fflush( stdout );
107
108         rte_sync( id_task5 );
109         printf("Node 5 >> OCORRENCIAS=%d\n", n_ocorrencias5);
110         fflush( stdout );
111
112         TIMER_STOP;
113
114         printf("Tempo: %f \n\n", TIMER_ELAPSED);
115         fflush( stdout );
116
117         return 0;
118 }
```

F.1.4 Teste com paralização de 4 tarefas, com paralelização interna, para a contagem de DNAs distintos com o mesmo tamanho em um genoma específico com o uso da API RTE

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <error.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <sys/time.h>
8
9  #include <mede_time.h>
```

```
10
11 #include <rte.h>
12
13 #include <mede_time.h>
14
15 #define SIZE 128
16
17 int main(int argc, char *argv[])
18 {
19         int id_task2,
20                 id_task3,
21                 id_task4,
22                 id_task5;
23
24         int n_ocorrencias2 = 0,
25                 n_ocorrencias3 = 0,
26                 n_ocorrencias4 = 0,
27                 n_ocorrencias5 = 0;
28
29         /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
               >>>>>>> */
30         // Node 2
31         char file2[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
32         char word2[SIZE] = "TCGATTCC\0";
33
34         // Node 3
35         char file3[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
36         char word3[SIZE] = "TCGATTGG\0";
37
38         // Node 4
39         char file4[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
40         char word4[SIZE] = "TCGATTAA\0";
41
42         // Node 5
43         char file5[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
44         char word5[SIZE] = "TCGATTTT\0";
45
46         /* <<<<<< F I M     D A     I N I C I A L I Z A   D O S   A R G
               U M E N T O S >>>>>>> */
47
48         if ( rte_init("busca-par.cfg\0") != 0 ) {
49                 error (1 , 0 , "Nao inicializar o Processador Remoto .\n
                       " );
50                 return 1;
51         }
52
53         rte_arg_list_t argin2, argin3, argin4 , argin5;
54         rte_ret_list_t argret2, argret3, argret4, argret5;
55
56         rte_init_args(2,&argin2, 1,&argret2);
57         argin2.arg[0].size = SIZE;
58         argin2.arg[0].arg =(char *)file2;
59         argin2.arg[1].size = SIZE;
60         argin2.arg[1].arg = (char *)word2;
61         argret2.ret[0].size = sizeof(int);
62         argret2.ret[0].ret=(char *)&n_ocorrencias2;
63
64         rte_init_args(2,&argin3, 1,&argret3);
65         argin3.arg[0].size = SIZE;
66         argin3.arg[0].arg =(char *)file3;
```

```
67          argin3.arg[1].size = SIZE;
68          argin3.arg[1].arg = (char *)word3;
69          argret3.ret[0].size = sizeof(int);
70          argret3.ret[0].ret=(char *)&n_ocorrencias3;
71
72          rte_init_args(2,&argin4, 1,&argret4);
73          argin4.arg[0].size = SIZE;
74          argin4.arg[0].arg =(char *)file4;
75          argin4.arg[1].size = SIZE;
76          argin4.arg[1].arg = (char *)word4;
77          argret4.ret[0].size = sizeof(int);
78          argret4.ret[0].ret=(char *)&n_ocorrencias4;
79
80          rte_init_args(2,&argin5, 1,&argret5);
81          argin5.arg[0].size = SIZE;
82          argin5.arg[0].arg =(char *)file5;
83          argin5.arg[1].size = SIZE;
84          argin5.arg[1].arg = (char *)word5;
85          argret5.ret[0].size = sizeof(int);
86          argret5.ret[0].ret=(char *)&n_ocorrencias5;
87
88          TIMER_CLEAR;
89          TIMER_START;
90
91          id_task2 = rte_tsk_call(0, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin2, &
                argret2);
92          id_task3 = rte_tsk_call(1, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin3, &
                argret3);
93          id_task4 = rte_tsk_call(2, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin4, &
                argret4);
94          id_task5 = rte_tsk_call(3, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin5, &
                argret5);
95
96          rte_sync( id_task2 );
97          printf("Node 2 >> OCORRENCIAS=%d\n", n_ocorrencias2);
98          fflush( stdout );
99
100         rte_sync( id_task3 );
101         printf("Node 3 >> OCORRENCIAS=%d\n", n_ocorrencias3);
102         fflush( stdout );
103
104         rte_sync( id_task4 );
105         printf("Node 4 >> OCORRENCIAS=%d\n", n_ocorrencias4);
106         fflush( stdout );
107
108         rte_sync( id_task5 );
109         printf("Node 5 >> OCORRENCIAS=%d\n", n_ocorrencias5);
110         fflush( stdout );
111
112         TIMER_STOP;
113
114         printf("Tempo: %f \n\n", TIMER_ELAPSED);
115         fflush( stdout );
116
117         return 0;
118 }
```

F.1.5 Teste da API RTE para realizar a contagem de ocorrências de uma cadeia de DNAs em 4 genomas distintos com tamanhos aproximados

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <error.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <sys/time.h>
8
9  #include <mede_time.h>
10
11 #include <rte.h>
12
13 #include <mede_time.h>
14
15 #define SIZE 256
16
17 int main(int argc, char *argv[])
18 {
19         int id_task2,
20                 id_task3,
21                 id_task4,
22                 id_task5;
23
24         int n_ocorrencias2 = 0,
25                 n_ocorrencias3 = 0,
26                 n_ocorrencias4 = 0,
27                 n_ocorrencias5 = 0;
28
29         /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
               >>>>>>> */
30         // Node 2
31         char file2[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000331955.2_Oorc_1.1_genomic.fna.txt\0";
32         char word2[SIZE] = "
               ACAGTTGTTAAGGCAGCAGGCCCTGGAGGGGACACGTCACCGGTGTCTGGCTCAGCTACCACTTGGTGGGT
               \0";
33
34         // Node 3
35         char file3[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000464555.1_PanTig1.0_genomic.fna.txt\0";
36         char word3[SIZE] = "
               ACAGTTGTTAAGGCAGCAGGCCCTGGAGGGGACACGTCACCGGTGTCTGGCTCAGCTACCACTTGGTGGGT
               \0";
37
38         // Node 4
39         char file4[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_002863925.1_EquCab3.0_genomic.fna.txt\0";
40         char word4[SIZE] = "
               ACAGTTGTTAAGGCAGCAGGCCCTGGAGGGGACACGTCACCGGTGTCTGGCTCAGCTACCACTTGGTGGGT
               \0";
41
42         // Node 5
43         char file5[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCA_008795835.1_PanLeo1.0_genomic.fna.txt\0";
44         char word5[SIZE] = "
               ACAGTTGTTAAGGCAGCAGGCCCTGGAGGGGACACGTCACCGGTGTCTGGCTCAGCTACCACTTGGTGGGT
               \0";
45
46         /* <<<<<< F I M     D A     I N I C I A L I Z A   D O S   A R G
               U M E N T O S >>>>>>> */
47
```

```
48          if ( rte_init("busca.cfg\0") != 0 ) {
49                  error (1 , 0 , "Nao inicializar o Processador Remoto .\n
                        " );
50                  return 1;
51          }
52
53          rte_arg_list_t argin2, argin3, argin4 , argin5;
54          rte_ret_list_t argret2, argret3, argret4, argret5;
55
56          rte_init_args(2,&argin2, 1,&argret2);
57          argin2.arg[0].size = SIZE;
58          argin2.arg[0].arg =(char *)file2;
59          argin2.arg[1].size = SIZE;
60          argin2.arg[1].arg = (char *)word2;
61          argret2.ret[0].size = sizeof(int);
62          argret2.ret[0].ret=(char *)&n_ocorrencias2;
63
64          rte_init_args(2,&argin3, 1,&argret3);
65          argin3.arg[0].size = SIZE;
66          argin3.arg[0].arg =(char *)file3;
67          argin3.arg[1].size = SIZE;
68          argin3.arg[1].arg = (char *)word3;
69          argret3.ret[0].size = sizeof(int);
70          argret3.ret[0].ret=(char *)&n_ocorrencias3;
71
72          rte_init_args(2,&argin4, 1,&argret4);
73          argin4.arg[0].size = SIZE;
74          argin4.arg[0].arg =(char *)file4;
75          argin4.arg[1].size = SIZE;
76          argin4.arg[1].arg = (char *)word4;
77          argret4.ret[0].size = sizeof(int);
78          argret4.ret[0].ret=(char *)&n_ocorrencias4;
79
80          rte_init_args(2,&argin5, 1,&argret5);
81          argin5.arg[0].size = SIZE;
82          argin5.arg[0].arg =(char *)file5;
83          argin5.arg[1].size = SIZE;
84          argin5.arg[1].arg = (char *)word5;
85          argret5.ret[0].size = sizeof(int);
86          argret5.ret[0].ret=(char *)&n_ocorrencias5;
87
88          TIMER_CLEAR;
89          TIMER_START;
90
91          id_task2 = rte_tsk_call(0, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin2, &
                argret2);
92          id_task3 = rte_tsk_call(1, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin3, &
                argret3);
93          id_task4 = rte_tsk_call(2, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin4, &
                argret4);
94          id_task5 = rte_tsk_call(3, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin5, &
                argret5);
95
96          rte_sync( id_task2 );
97          printf("Node 2 >> OCORRENCIAS=%d\n", n_ocorrencias2);
98          fflush( stdout );
99
100         rte_sync( id_task3 );
101         printf("Node 3 >> OCORRENCIAS=%d\n", n_ocorrencias3);
102         fflush( stdout );
```

```
103
104         rte_sync( id_task4 );
105         printf("Node 4 >> OCORRENCIAS=%d\n", n_ocorrencias4);
106         fflush( stdout );
107
108         rte_sync( id_task5 );
109         printf("Node 5 >> OCORRENCIAS=%d\n", n_ocorrencias5);
110         fflush( stdout );
111
112         TIMER_STOP;
113
114         printf("Tempo: %f \n\n", TIMER_ELAPSED);
115         fflush( stdout );
116
117         return 0;
118 }
```

F.1.6 Teste da API RTE com paralelização interna de tarefas para realizar a contagem de ocorrências de uma cadeia de DNAs em 4 genomas distintos com tamanhos aproximados

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <error.h>
 5 #include <unistd.h>
 6 #include <stdlib.h>
 7 #include <sys/time.h>
 8
 9 #include <mede_time.h>
10
11 #include <rte.h>
12
13 #include <mede_time.h>
14
15 #define SIZE 256
16
17 int main(int argc, char *argv[])
18 {
19         int id_task2,
20                 id_task3,
21                 id_task4,
22                 id_task5;
23
24         int n_ocorrencias2 = 0,
25                 n_ocorrencias3 = 0,
26                 n_ocorrencias4 = 0,
27                 n_ocorrencias5 = 0;
28
29         /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
               >>>>>>> */
30         // Node 2
31         char file2[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000331955.2_Oorc_1.1_genomic.fna.txt\0";
32         char word2[SIZE] = "
               ACAGTTGTTAAGGCAGCAGGCCCTGGAGGGGACACGTCACCGGTGTCTGGCTCAGCTACCACTTGGTGGGT
               \0";
33
34         // Node 3
35         char file3[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000464555.1_PanTig1.0_genomic.fna.txt\0";
36         char word3[SIZE] = "
               ACAGTTGTTAAGGCAGCAGGCCCTGGAGGGGACACGTCACCGGTGTCTGGCTCAGCTACCACTTGGTGGGT
```

```
            \0";
37
38          // Node 4
39          char file4[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_002863925.1_EquCab3.0_genomic.fna.txt\0";
40          char word4[SIZE] = "
               ACAGTTGTTAAGGCAGCAGGCCCTGGAGGGGACACGTCACCGGTGTCTGGCTCAGCTACCACTTGGTGGGT
               \0";
41
42          // Node 5
43          char file5[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCA_008795835.1_PanLeo1.0_genomic.fna.txt\0";
44          char word5[SIZE] = "
               ACAGTTGTTAAGGCAGCAGGCCCTGGAGGGGACACGTCACCGGTGTCTGGCTCAGCTACCACTTGGTGGGT
               \0";
45
46          /* <<<<<< F I M      D A      I N I C I A L I Z A   D O S   A R G
                U M E N T O S >>>>>>> */
47
48          if ( rte_init("busca-par.cfg\0") != 0 ) {
49                  error (1 , 0 , "Nao inicializar o Processador Remoto .\n
                       " );
50                  return 1;
51          }
52
53          rte_arg_list_t argin2, argin3, argin4 , argin5;
54          rte_ret_list_t argret2, argret3, argret4, argret5;
55
56          rte_init_args(2,&argin2, 1,&argret2);
57          argin2.arg[0].size = SIZE;
58          argin2.arg[0].arg =(char *)file2;
59          argin2.arg[1].size = SIZE;
60          argin2.arg[1].arg = (char *)word2;
61          argret2.ret[0].size = sizeof(int);
62          argret2.ret[0].ret=(char *)&n_ocorrencias2;
63
64          rte_init_args(2,&argin3, 1,&argret3);
65          argin3.arg[0].size = SIZE;
66          argin3.arg[0].arg =(char *)file3;
67          argin3.arg[1].size = SIZE;
68          argin3.arg[1].arg = (char *)word3;
69          argret3.ret[0].size = sizeof(int);
70          argret3.ret[0].ret=(char *)&n_ocorrencias3;
71
72          rte_init_args(2,&argin4, 1,&argret4);
73          argin4.arg[0].size = SIZE;
74          argin4.arg[0].arg =(char *)file4;
75          argin4.arg[1].size = SIZE;
76          argin4.arg[1].arg = (char *)word4;
77          argret4.ret[0].size = sizeof(int);
78          argret4.ret[0].ret=(char *)&n_ocorrencias4;
79
80          rte_init_args(2,&argin5, 1,&argret5);
81          argin5.arg[0].size = SIZE;
82          argin5.arg[0].arg =(char *)file5;
83          argin5.arg[1].size = SIZE;
84          argin5.arg[1].arg = (char *)word5;
85          argret5.ret[0].size = sizeof(int);
86          argret5.ret[0].ret=(char *)&n_ocorrencias5;
87
88          TIMER_CLEAR;
89          TIMER_START;
90
91          id_task2 = rte_tsk_call(0, 0, "find_word\0", "/home/dbelo/
```

```
            Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin2, &
            argret2);
92          id_task3 = rte_tsk_call(1, 0, "find_word\0", "/home/dbelo/
            Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin3, &
            argret3);
93          id_task4 = rte_tsk_call(2, 0, "find_word\0", "/home/dbelo/
            Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin4, &
            argret4);
94          id_task5 = rte_tsk_call(3, 0, "find_word\0", "/home/dbelo/
            Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin5, &
            argret5);
95
96          rte_sync( id_task2 );
97          printf("Node 2 >> OCORRENCIAS=%d\n", n_ocorrencias2);
98          fflush( stdout );
99
100         rte_sync( id_task3 );
101         printf("Node 3 >> OCORRENCIAS=%d\n", n_ocorrencias3);
102         fflush( stdout );
103
104         rte_sync( id_task4 );
105         printf("Node 4 >> OCORRENCIAS=%d\n", n_ocorrencias4);
106         fflush( stdout );
107
108         rte_sync( id_task5 );
109         printf("Node 5 >> OCORRENCIAS=%d\n", n_ocorrencias5);
110         fflush( stdout );
111
112         TIMER_STOP;
113
114         printf("Tempo: %f \n\n", TIMER_ELAPSED);
115         fflush( stdout );
116
117         return 0;
118 }
```

F.1.7 Teste de escalabilidade da API RTE com tarefas sequenciais

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <error.h>
 5 #include <unistd.h>
 6 #include <stdlib.h>
 7 #include <sys/time.h>
 8
 9 #include <mede_time.h>
10
11 #include <rte.h>
12
13 #include <mede_time.h>
14
15 #define SIZE 128
16
17 int main(int argc, char *argv[])
18 {
19         int id_task2,
20                 id_task3,
21                 id_task4,
22                 id_task5;
23
24         int n_ocorrencias2 = 0,
25                 n_ocorrencias3 = 0,
26                 n_ocorrencias4 = 0,
```

```
27              n_ocorrencias5 = 0;
28
29          /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
                >>>>>>> */
30          // Node 2
31          char file2[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
                /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
32          char word2[SIZE] = "TCGATTCCGG\0";
33
34          // Node 3
35          char file3[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
                /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
36          char word3[SIZE] = "TCGATTCCAA\0";
37
38          // Node 4
39          char file4[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
                /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
40          char word4[SIZE] = "TCGATTCCTT\0";
41
42          // Node 5
43          char file5[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
                /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
44          char word5[SIZE] = "TCGACCGGAA\0";
45
46          /* <<<<<< F I M     D A     I N I C I A L I Z A   D O S   A R G
                U M E N T O S >>>>>>> */
47
48          if ( rte_init("busca.cfg\0") != 0 ) {
49                  error (1 , 0 , "Nao inicializar o Processador Remoto .\n
                        " );
50                  return 1;
51          }
52
53          rte_arg_list_t argin2, argin3, argin4 , argin5;
54          rte_ret_list_t argret2, argret3, argret4, argret5;
55
56          rte_init_args(2,&argin2, 1,&argret2);
57          argin2.arg[0].size = SIZE;
58          argin2.arg[0].arg =(char *)file2;
59          argin2.arg[1].size = SIZE;
60          argin2.arg[1].arg = (char *)word2;
61          argret2.ret[0].size = sizeof(int);
62          argret2.ret[0].ret=(char *)&n_ocorrencias2;
63
64          rte_init_args(2,&argin3, 1,&argret3);
65          argin3.arg[0].size = SIZE;
66          argin3.arg[0].arg =(char *)file3;
67          argin3.arg[1].size = SIZE;
68          argin3.arg[1].arg = (char *)word3;
69          argret3.ret[0].size = sizeof(int);
70          argret3.ret[0].ret=(char *)&n_ocorrencias3;
71
72          rte_init_args(2,&argin4, 1,&argret4);
73          argin4.arg[0].size = SIZE;
74          argin4.arg[0].arg =(char *)file4;
75          argin4.arg[1].size = SIZE;
76          argin4.arg[1].arg = (char *)word4;
77          argret4.ret[0].size = sizeof(int);
78          argret4.ret[0].ret=(char *)&n_ocorrencias4;
79
80          rte_init_args(2,&argin5, 1,&argret5);
81          argin5.arg[0].size = SIZE;
82          argin5.arg[0].arg =(char *)file5;
83          argin5.arg[1].size = SIZE;
```

```
84          argin5.arg[1].arg = (char *)word5;
85          argret5.ret[0].size = sizeof(int);
86          argret5.ret[0].ret=(char *)&n_ocorrencias5;
87
88          TIMER_CLEAR;
89          TIMER_START;
90
91          id_task2 = rte_tsk_call(0, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin2, &
                argret2);
92          id_task3 = rte_tsk_call(1, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin3, &
                argret3);
93          id_task4 = rte_tsk_call(2, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin4, &
                argret4);
94          id_task5 = rte_tsk_call(3, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin5, &
                argret5);
95
96          rte_sync( id_task2 );
97          printf("Node 2 >> OCORRENCIAS=%d\n", n_ocorrencias2);
98          fflush( stdout );
99
100         rte_sync( id_task3 );
101         printf("Node 3 >> OCORRENCIAS=%d\n", n_ocorrencias3);
102         fflush( stdout );
103
104         rte_sync( id_task4 );
105         printf("Node 4 >> OCORRENCIAS=%d\n", n_ocorrencias4);
106         fflush( stdout );
107
108         rte_sync( id_task5 );
109         printf("Node 5 >> OCORRENCIAS=%d\n", n_ocorrencias5);
110         fflush( stdout );
111
112         TIMER_STOP;
113
114         printf("Tempo: %f \n\n", TIMER_ELAPSED);
115         fflush( stdout );
116
117         return 0;
118 }
```

F.1.8 Teste de escalabilidade da API RTE com tarefas com paralelização interna

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <error.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <sys/time.h>
8
9  #include <mede_time.h>
10
11 #include <rte.h>
12
13 #include <mede_time.h>
14
15 #define SIZE 128
16
17 int main(int argc, char *argv[])
18 {
```

```
19          int id_task2,
20                  id_task3,
21                  id_task4,
22                  id_task5;
23
24          int n_ocorrencias2 = 0,
25                  n_ocorrencias3 = 0,
26                  n_ocorrencias4 = 0,
27                  n_ocorrencias5 = 0;
28
29          /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
                >>>>>>> */
30          // Node 2
31          char file2[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
32          char word2[SIZE] = "TCGATTCCGG\0";
33
34          // Node 3
35          char file3[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
36          char word3[SIZE] = "TCGATTCCAA\0";
37
38          // Node 4
39          char file4[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
40          char word4[SIZE] = "TCGATTCCTT\0";
41
42          // Node 5
43          char file5[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
44          char word5[SIZE] = "TCGACCGGAA\0";
45
46          /* <<<<<< F I M      D A      I N I C I A L I Z A   D O S   A R G
                U M E N T O S >>>>>>> */
47
48          if ( rte_init("busca-par.cfg\0") != 0 ) {
49                  error (1 , 0 , "Nao inicializar o Processador Remoto .\n
                       " );
50                  return 1;
51          }
52
53          rte_arg_list_t argin2, argin3, argin4 , argin5;
54          rte_ret_list_t argret2, argret3, argret4, argret5;
55
56          rte_init_args(2,&argin2, 1,&argret2);
57          argin2.arg[0].size = SIZE;
58          argin2.arg[0].arg =(char *)file2;
59          argin2.arg[1].size = SIZE;
60          argin2.arg[1].arg = (char *)word2;
61          argret2.ret[0].size = sizeof(int);
62          argret2.ret[0].ret=(char *)&n_ocorrencias2;
63
64          rte_init_args(2,&argin3, 1,&argret3);
65          argin3.arg[0].size = SIZE;
66          argin3.arg[0].arg =(char *)file3;
67          argin3.arg[1].size = SIZE;
68          argin3.arg[1].arg = (char *)word3;
69          argret3.ret[0].size = sizeof(int);
70          argret3.ret[0].ret=(char *)&n_ocorrencias3;
71
72          rte_init_args(2,&argin4, 1,&argret4);
73          argin4.arg[0].size = SIZE;
74          argin4.arg[0].arg =(char *)file4;
75          argin4.arg[1].size = SIZE;
```

```
76          argin4.arg[1].arg = (char *)word4;
77          argret4.ret[0].size = sizeof(int);
78          argret4.ret[0].ret=(char *)&n_ocorrencias4;
79
80          rte_init_args(2,&argin5, 1,&argret5);
81          argin5.arg[0].size = SIZE;
82          argin5.arg[0].arg =(char *)file5;
83          argin5.arg[1].size = SIZE;
84          argin5.arg[1].arg = (char *)word5;
85          argret5.ret[0].size = sizeof(int);
86          argret5.ret[0].ret=(char *)&n_ocorrencias5;
87
88          TIMER_CLEAR;
89          TIMER_START;
90
91          id_task2 = rte_tsk_call(0, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin2, &
                argret2);
92          id_task3 = rte_tsk_call(1, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin3, &
                argret3);
93          id_task4 = rte_tsk_call(2, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin4, &
                argret4);
94          id_task5 = rte_tsk_call(3, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin5, &
                argret5);
95
96          rte_sync( id_task2 );
97          printf("Node 2 >> OCORRENCIAS=%d\n", n_ocorrencias2);
98          fflush( stdout );
99
100         rte_sync( id_task3 );
101         printf("Node 3 >> OCORRENCIAS=%d\n", n_ocorrencias3);
102         fflush( stdout );
103
104         rte_sync( id_task4 );
105         printf("Node 4 >> OCORRENCIAS=%d\n", n_ocorrencias4);
106         fflush( stdout );
107
108         rte_sync( id_task5 );
109         printf("Node 5 >> OCORRENCIAS=%d\n", n_ocorrencias5);
110         fflush( stdout );
111
112         TIMER_STOP;
113
114         printf("Tempo: %f \n\n", TIMER_ELAPSED);
115         fflush( stdout );
116
117         return 0;
118 }
```

F.2 Testes de comparação entre RTE e MPI em uma aplicação real

F.2.1 Teste de uma aplicação real com tarefas sequenciais em quatro *Guests* RTE

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <error.h>
5  #include <unistd.h>
```

```
 6 #include <stdlib.h>
 7 #include <sys/time.h>
 8
 9 #include <mede_time.h>
10
11 #include <rte.h>
12
13 #include <mede_time.h>
14
15 #define SIZE 128
16
17 int main(int argc, char *argv[])
18 {
19         int id_task2,
20                 id_task3,
21                 id_task4,
22                 id_task5;
23
24         int n_ocorrencias2 = 0,
25                 n_ocorrencias3 = 0,
26                 n_ocorrencias4 = 0,
27                 n_ocorrencias5 = 0;
28
29         /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
               >>>>>>> */
30         // Node 2
31         char file2[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
32         char word2[SIZE] = "TCGATTCC\0";
33
34         // Node 3
35         char file3[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
36         char word3[SIZE] = "TCGATTCC\0";
37
38         // Node 4
39         char file4[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
40         char word4[SIZE] = "TCGATTCC\0";
41
42         // Node 5
43         char file5[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
44         char word5[SIZE] = "TCGATTCC\0";
45
46         /* <<<<<< F I M     D A     I N I C I A L I Z A   D O S   A R G
               U M E N T O S >>>>>>> */
47
48         if ( rte_init("busca.cfg\0") != 0 ) {
49                 error (1 , 0 , "Nao inicializar o Processador Remoto .\n
                       " );
50                 return 1;
51         }
52
53         rte_arg_list_t argin2, argin3, argin4 , argin5;
54         rte_ret_list_t argret2, argret3, argret4, argret5;
55
56         rte_init_args(2,&argin2, 1,&argret2);
57         argin2.arg[0].size = SIZE;
58         argin2.arg[0].arg =(char *)file2;
59         argin2.arg[1].size = SIZE;
60         argin2.arg[1].arg = (char *)word2;
61         argret2.ret[0].size = sizeof(int);
62         argret2.ret[0].ret=(char *)&n_ocorrencias2;
```

```
63
64          rte_init_args(2,&argin3, 1,&argret3);
65          argin3.arg[0].size = SIZE;
66          argin3.arg[0].arg =(char *)file3;
67          argin3.arg[1].size = SIZE;
68          argin3.arg[1].arg = (char *)word3;
69          argret3.ret[0].size = sizeof(int);
70          argret3.ret[0].ret=(char *)&n_ocorrencias3;
71
72          rte_init_args(2,&argin4, 1,&argret4);
73          argin4.arg[0].size = SIZE;
74          argin4.arg[0].arg =(char *)file4;
75          argin4.arg[1].size = SIZE;
76          argin4.arg[1].arg = (char *)word4;
77          argret4.ret[0].size = sizeof(int);
78          argret4.ret[0].ret=(char *)&n_ocorrencias4;
79
80          rte_init_args(2,&argin5, 1,&argret5);
81          argin5.arg[0].size = SIZE;
82          argin5.arg[0].arg =(char *)file5;
83          argin5.arg[1].size = SIZE;
84          argin5.arg[1].arg = (char *)word5;
85          argret5.ret[0].size = sizeof(int);
86          argret5.ret[0].ret=(char *)&n_ocorrencias5;
87
88          TIMER_CLEAR;
89          TIMER_START;
90
91          id_task2 = rte_tsk_call(0, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin2, &
                argret2);
92          id_task3 = rte_tsk_call(1, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin3, &
                argret3);
93          id_task4 = rte_tsk_call(2, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin4, &
                argret4);
94          id_task5 = rte_tsk_call(3, 0, "find_word\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libbusca.so\0", &argin5, &
                argret5);
95
96          rte_sync( id_task2 );
97          printf("Node 2 >> OCORRENCIAS=%d\n", n_ocorrencias2);
98          fflush( stdout );
99
100         rte_sync( id_task3 );
101         printf("Node 3 >> OCORRENCIAS=%d\n", n_ocorrencias3);
102         fflush( stdout );
103
104         rte_sync( id_task4 );
105         printf("Node 4 >> OCORRENCIAS=%d\n", n_ocorrencias4);
106         fflush( stdout );
107
108         rte_sync( id_task5 );
109         printf("Node 5 >> OCORRENCIAS=%d\n", n_ocorrencias5);
110         fflush( stdout );
111
112         TIMER_STOP;
113
114         printf("Tempo: %f \n\n", TIMER_ELAPSED);
115         fflush( stdout );
116
117         return 0;
118 }
```

F.2.2 Teste de uma aplicação real com tarefas sequenciais em quatro nós MPI

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <string.h>
5  #include <mpi.h>
6  #include <sys/mman.h>
7  #include <fcntl.h>
8  #include <sys/stat.h>
9
10 #include <mede_time.h>
11
12 typedef struct type_argT {char nome_arq[128];char palavra[20];} type_arg
       ;
13
14 int busca_palavra(char arquivo[128],char palavra[20])
15 {
16     unsigned long int file_size, i, j;
17     char *vet;
18
19     int ocorrencias = 0;
20
21     bool achou;
22
23     FILE *f;
24
25     printf("palavra = %s\n",palavra);
26     printf("len = %ld\n\n", strlen(palavra));
27
28     printf("file = %s\n", arquivo);
29     fflush(stdout);
30
31     f = fopen(arquivo, "r");
32
33     if( f != NULL ) {
34         fseek(f,0L,SEEK_END);
35         file_size = ftell(f);
36         fseek(f,0L,SEEK_SET);
37         vet=(char *) malloc( file_size * sizeof(char));
38         if( fread(vet, 1, file_size,f) > 0 ) {
39             for(i = 0; i < file_size - strlen(palavra); i++) {
40                 achou = true;
41
42                 // Comparison of line position with word
43                 for(j = 0; j < strlen(palavra); j++){
44                     if(palavra[j] != vet[i+j]){
45                         achou = false;
46                         break;
47                     }
48                 }
49
50                 if (achou == true){
51                     ocorrencias++;
52                 }
53             }
54         } else {
55             printf("Falha ao tentar ler o arquivo = %s\n", arquivo);
56             fflush(stdout);
57         }
58         fclose(f);
59
60     } else {
61         printf("Falha ao abrir o arquivo = %s\n", arquivo);
```

```
62          fflush(stdout);
63      }
64      printf("=======================================\n");
65      printf("Total de ocorrencias = %d\n",ocorrencias);
66      printf("=======================================\n");
67
68      return ocorrencias;
69  }
70
71  int main(int argc, char *argv[])
72  {
73      int             i,n,n_nos, rank;
74      MPI_Status      status;
75      MPI_Request req;
76      int n_ocorrencias;
77
78      MPI_Init(&argc, &argv);
79      MPI_Comm_size(MPI_COMM_WORLD, &n_nos);
80      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
81
82      if (rank==0) {
83          type_arg arg1;
84          bzero(&arg1, sizeof(type_arg));
85          strncpy(arg1.nome_arq, "/home/dbelo/Workspace/epusp/lahpc/testes
                /rte/busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt
                \0", sizeof(arg1.nome_arq));
86          strncpy(arg1.palavra, "TCGATTCC\0", sizeof(arg1.palavra));
87
88
89          type_arg arg2;
90          bzero(&arg2, sizeof(type_arg));
91          strncpy(arg2.nome_arq, "/home/dbelo/Workspace/epusp/lahpc/testes
                /rte/busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt
                \0", sizeof(arg2.nome_arq));
92          strncpy(arg2.palavra, "TCGATTCC\0", sizeof(arg2.palavra));
93
94          type_arg arg3;
95          bzero(&arg3, sizeof(type_arg));
96          strncpy(arg3.nome_arq, "/home/dbelo/Workspace/epusp/lahpc/testes
                /rte/busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt
                \0", sizeof(arg3.nome_arq));
97          strncpy(arg3.palavra, "TCGATTCC\0", sizeof(arg3.palavra));
98
99          type_arg arg4;
100         bzero(&arg4, sizeof(type_arg));
101         strncpy(arg4.nome_arq, "/home/dbelo/Workspace/epusp/lahpc/testes
                /rte/busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt
                \0", sizeof(arg4.nome_arq));
102         strncpy(arg4.palavra, "TCGATTCC\0", sizeof(arg4.palavra));
103
104         TIMER_CLEAR;
105         TIMER_START;
106         MPI_Isend(&arg1, sizeof(type_arg),MPI_CHAR,1,10,MPI_COMM_WORLD,&
                req);
107         MPI_Isend(&arg2, sizeof(type_arg),MPI_CHAR,2,10,MPI_COMM_WORLD,&
                req);
108         MPI_Isend(&arg3, sizeof(type_arg),MPI_CHAR,3,10,MPI_COMM_WORLD,&
                req);
109         MPI_Isend(&arg4, sizeof(type_arg),MPI_CHAR,4,10,MPI_COMM_WORLD,&
                req);
110
111         for(i=1;i<n_nos;i++) {
112             MPI_Recv(&n_ocorrencias,1,MPI_INT,MPI_ANY_SOURCE,20,
                    MPI_COMM_WORLD,&status);
```

```
113                 printf("rank=%d  OCORRENCIAS=%d\n",status.MPI_SOURCE,
                        n_ocorrencias);
114             }
115             TIMER_STOP;
116             printf("Tempo: %f \n", TIMER_ELAPSED);
117             fflush(stdout);
118          }
119         else{
120             type_arg x;
121             MPI_Recv(&x,sizeof(type_arg),MPI_CHAR,0,10,MPI_COMM_WORLD,&
                    status);
122             int n_ocorrencias=busca_palavra(x.nome_arq,x.palavra);
123             MPI_Send(&n_ocorrencias,1,MPI_INT,0,20,MPI_COMM_WORLD);
124         }
125         MPI_Finalize();
126         return 0;
127     }
```

F.2.3 Teste de uma aplicação real com paralelização interna em quatro *Guests* RTE

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4  #include <error.h>
 5  #include <unistd.h>
 6  #include <stdlib.h>
 7  #include <sys/time.h>
 8
 9  #include <mede_time.h>
10
11  #include <rte.h>
12
13  #include <mede_time.h>
14
15  #define SIZE 128
16
17  int main(int argc, char *argv[])
18  {
19          int id_task2,
20                  id_task3,
21                  id_task4,
22                  id_task5;
23
24          int n_ocorrencias2 = 0,
25                  n_ocorrencias3 = 0,
26                  n_ocorrencias4 = 0,
27                  n_ocorrencias5 = 0;
28
29          /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
                >>>>>>> */
30          // Node 2
31          char file2[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
                /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
32          char word2[SIZE] = "TCGATTCC\0";
33
34          // Node 3
35          char file3[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
                /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
36          char word3[SIZE] = "TCGATTCC\0";
37
38          // Node 4
39          char file4[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
```

```
            /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
40          char word4[SIZE] = "TCGATTCC\0";
41
42          // Node 5
43          char file5[SIZE] = "/home/dbelo/Workspace/epusp/lahpc/testes/rte
               /busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt\0";
44          char word5[SIZE] = "TCGATTCC\0";
45
46          /* <<<<<< F I M      D A      I N I C I A L I Z A   D O S   A R G
                U M E N T O S >>>>>>> */
47
48          if ( rte_init("busca-par.cfg\0") != 0 ) {
49                  error (1 , 0 , "Nao inicializar o Processador Remoto .\n
                       " );
50                  return 1;
51          }
52
53          rte_arg_list_t argin2, argin3, argin4 , argin5;
54          rte_ret_list_t argret2, argret3, argret4, argret5;
55
56          rte_init_args(2,&argin2, 1,&argret2);
57          argin2.arg[0].size = SIZE;
58          argin2.arg[0].arg =(char *)file2;
59          argin2.arg[1].size = SIZE;
60          argin2.arg[1].arg = (char *)word2;
61          argret2.ret[0].size = sizeof(int);
62          argret2.ret[0].ret=(char *)&n_ocorrencias2;
63
64          rte_init_args(2,&argin3, 1,&argret3);
65          argin3.arg[0].size = SIZE;
66          argin3.arg[0].arg =(char *)file3;
67          argin3.arg[1].size = SIZE;
68          argin3.arg[1].arg = (char *)word3;
69          argret3.ret[0].size = sizeof(int);
70          argret3.ret[0].ret=(char *)&n_ocorrencias3;
71
72          rte_init_args(2,&argin4, 1,&argret4);
73          argin4.arg[0].size = SIZE;
74          argin4.arg[0].arg =(char *)file4;
75          argin4.arg[1].size = SIZE;
76          argin4.arg[1].arg = (char *)word4;
77          argret4.ret[0].size = sizeof(int);
78          argret4.ret[0].ret=(char *)&n_ocorrencias4;
79
80          rte_init_args(2,&argin5, 1,&argret5);
81          argin5.arg[0].size = SIZE;
82          argin5.arg[0].arg =(char *)file5;
83          argin5.arg[1].size = SIZE;
84          argin5.arg[1].arg = (char *)word5;
85          argret5.ret[0].size = sizeof(int);
86          argret5.ret[0].ret=(char *)&n_ocorrencias5;
87
88          TIMER_CLEAR;
89          TIMER_START;
90
91          id_task2 = rte_tsk_call(0, 0, "find_word\0", "/home/dbelo/
               Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin2, &
               argret2);
92          id_task3 = rte_tsk_call(1, 0, "find_word\0", "/home/dbelo/
               Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin3, &
               argret3);
93          id_task4 = rte_tsk_call(2, 0, "find_word\0", "/home/dbelo/
               Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin4, &
               argret4);
```

```
94          id_task5 = rte_tsk_call(3, 0, "find_word\0", "/home/dbelo/
               Workspace/epusp/lahpc/rte/libs/libbusca-par.so\0", &argin5, &
               argret5);
95
96          rte_sync( id_task2 );
97          printf("Node 2 >> OCORRENCIAS=%d\n", n_ocorrencias2);
98          fflush( stdout );
99
100         rte_sync( id_task3 );
101         printf("Node 3 >> OCORRENCIAS=%d\n", n_ocorrencias3);
102         fflush( stdout );
103
104         rte_sync( id_task4 );
105         printf("Node 4 >> OCORRENCIAS=%d\n", n_ocorrencias4);
106         fflush( stdout );
107
108         rte_sync( id_task5 );
109         printf("Node 5 >> OCORRENCIAS=%d\n", n_ocorrencias5);
110         fflush( stdout );
111
112         TIMER_STOP;
113
114         printf("Tempo: %f \n\n", TIMER_ELAPSED);
115         fflush( stdout );
116
117         return 0;
118 }
```

F.2.4 Teste de uma aplicação real com paralelização interna em quatro nós MPI

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <string.h>
5  #include <mpi.h>
6  #include <sys/mman.h>
7  #include <sys/stat.h>
8  #include <fcntl.h>
9  #include <omp.h>
10
11 #include <mede_time.h>
12
13 #define NTHREADS 4
14
15 typedef struct type_argT {char nome_arq[128];char palavra[20];} type_arg
       ;
16
17 int busca_palavra(char arquivo[128],char palavra[20])
18 {
19     unsigned long int file_size,
20                       i, j;
21     char *vet;
22     int ocorrencias = 0;
23     bool achou;
24
25     FILE *f;
26
27     printf("palavra = %s\n", palavra);
28     printf("len = %ld\n\n", strlen(palavra));
29     fflush(stdout);
30
31     printf("arquivo = %s\n", arquivo);
32     fflush(stdout);
33
```

```
34      f = fopen(arquivo, "r");
35      if( f != NULL ) {
36          fseek(f,0L,SEEK_END);
37          file_size=ftell(f);
38          fseek(f,0L,SEEK_SET);
39          vet=(char *) malloc(file_size * sizeof(char));
40          if(fread(vet,1,file_size,f) > 0){
41
42              printf("NTHREADS %d, tamanho %lu\n", NTHREADS, file_size);
43
44               // Busca pela palavra
45              #pragma omp parallel num_threads(NTHREADS) private (achou,i,
                    j) reduction(+:ocorrencias)
46              {
47                  #pragma omp for
48                  for(i = 0; i < (file_size - strlen(palavra)); i++){
49                      achou = true;
50
51                      // Comparacao da posicao da linha com a palavra
52                      for(j = 0; ((j < strlen(palavra))&& (achou == true))
                            ; j++){
53                          if(palavra[j] != vet[i+j]){
54                              achou = false;
55                              break;
56                          }
57                      }
58                      if (achou == true){
59                          ocorrencias++;
60                      }
61                  }
62                  printf(">>>>>>> i = %ld\n", i);
63                  fflush(stdout);
64              }
65          } else {
66              printf("Falha ao tentar ler o arquivo = %s\n", arquivo);
67              fflush(stdout);
68          }
69          // Fecha o arquivo
70          fclose(f);
71      } else {
72          printf("Falha ao abrir o arquivo = %s\n", arquivo);
73          fflush(stdout);
74      }
75
76      printf("=======================================\n");
77      printf("Total de ocorrencias = %d\n",ocorrencias);
78      printf("=======================================\n");
79
80      return ocorrencias;
81  }
82
83  int main(int argc, char *argv[])
84  {
85      int i,n,n_nos, rank;
86
87      MPI_Status status;
88      MPI_Request req;
89
90      int n_ocorrencias;
91      int num_ocorrencias[5];
92
93      MPI_Init(&argc, &argv);
94      MPI_Comm_size(MPI_COMM_WORLD, &n_nos);
95      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
 96      if (rank==0) {
 97          type_arg arg1;
 98          bzero(&arg1, sizeof(type_arg));
 99          strncpy(arg1.nome_arq, "/home/dbelo/Workspace/epusp/lahpc/testes
                /rte/busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt
                \0", sizeof(arg1.nome_arq));
100          strncpy(arg1.palavra, "TCGATTCC\0", sizeof(arg1.palavra));
101          type_arg arg2;
102          bzero(&arg2, sizeof(type_arg));
103          strncpy(arg2.nome_arq, "/home/dbelo/Workspace/epusp/lahpc/testes
                /rte/busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt
                \0", sizeof(arg2.nome_arq));
104          strncpy(arg2.palavra, "TCGATTCC\0", sizeof(arg2.palavra));
105          type_arg arg3;
106          bzero(&arg3, sizeof(type_arg));
107          strncpy(arg3.nome_arq, "/home/dbelo/Workspace/epusp/lahpc/testes
                /rte/busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt
                \0", sizeof(arg3.nome_arq));
108          strncpy(arg3.palavra, "TCGATTCC\0", sizeof(arg3.palavra));
109          type_arg arg4;
110          bzero(&arg4, sizeof(type_arg));
111          strncpy(arg4.nome_arq, "/home/dbelo/Workspace/epusp/lahpc/testes
                /rte/busca/files/GCF_000243295.1_TriManLat1.0_genomic.fna.txt
                \0", sizeof(arg4.nome_arq));
112          strncpy(arg4.palavra, "TCGATTCC\0", sizeof(arg4.palavra));
113
114          TIMER_CLEAR;
115          TIMER_START;
116          MPI_Isend(&arg1, sizeof(type_arg),MPI_CHAR,1,10,MPI_COMM_WORLD,&
                req);
117          MPI_Isend(&arg2, sizeof(type_arg),MPI_CHAR,2,10,MPI_COMM_WORLD,&
                req);
118          MPI_Isend(&arg3, sizeof(type_arg),MPI_CHAR,3,10,MPI_COMM_WORLD,&
                req);
119          MPI_Isend(&arg4, sizeof(type_arg),MPI_CHAR,4,10,MPI_COMM_WORLD,&
                req);
120
121          for(i=1;i<n_nos;i++) {
122                  MPI_Recv(&n_ocorrencias,1,MPI_INT,MPI_ANY_SOURCE,20,
                        MPI_COMM_WORLD,&status);
123                  num_ocorrencias[status.MPI_SOURCE]=n_ocorrencias;
124          }
125          TIMER_STOP;
126          for (i=1;i<n_nos;i++)
127                  printf("OCORRENCIAS[%5d]=%d\n",i,num_ocorrencias[i]);
128
129          printf("Tempo: %f \n", TIMER_ELAPSED);
130          fflush(stdout);
131      }
132      else{
133          type_arg x;
134          MPI_Recv(&x,sizeof(type_arg),MPI_CHAR,0,10,MPI_COMM_WORLD,&
                status);
135          int n_ocorrencias=busca_palavra(x.nome_arq,x.palavra);
136          MPI_Send(&n_ocorrencias,1,MPI_INT,0,20,MPI_COMM_WORLD);
137      }
138      MPI_Finalize();
139      return 0;
140  }
```

APÊNDICE G – EXEMPLOS DE TESTES COM ARQUITETURAS HETEROGÊNEAS

G.1 Execução distribuída em arquiteturas heterogêneas com a API RTE

```
1  //soma escalar, c=a + b
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/time.h>
5
6  #include <rte.h>
7
8  #include <mede_time.h>
9
10 rte_dvcs_types_t list_type_dvc[ 128 ];
11 int a, b, c, x, y;
12
13 float z;
14
15 int main()
16 {
17         int id_tsk1,
18             id_tsk2;
19
20         list_type_dvc[0] = rte_dvc_cpu;  // CPU
21
22         rte_init_dvcs_lst( list_type_dvc, 1 );
23
24         printf("INICIO SOMA & MULTIPLICACAO ESCALAR EM AMBINTES
               HETEROGENEOS\n");
25         fflush(stdout);
26
27         if ( rte_init( "soma-mult_escalar_htr.cfg" ) != 0 ) {
28                 printf( "Nao inicializar o Processador Remoto .\n" );
29                 return 1;
30         }
31
32         a = 2;
33         b = 4;
34
35         x = 3;
36         y = 5;
37
38         printf( "a = %d  b = %d\nx = %d  y = %d\n", a, b, x, y );
39         fflush( stdout );
40
41         rte_arg_list_t argin1;
42         rte_ret_list_t argret1;
```

```
43
44          rte_arg_list_t argin2;
45          rte_ret_list_t argret2;
46
47          rte_init_args( 2, &argin_sum, 1, &argret );  //2=numero de
                parametros  argin=entrada
48
49          argin1.arg[0].size= sizeof(int);
50          argin1.arg[0].arg=(char *)&a;
51          argin1.arg[1].size= sizeof(int);
52          argin1.arg[1].arg=(char *)&b;
53          argret1.ret[0].size= sizeof(int);
54          argret1.ret[0].ret=(char *)&c;
55
56          rte_init_args( 2, &argin_sum, 1, &argret );  //2=numero de
                parametros  argin=entrada
57
58          argin2.arg[0].size= sizeof(int);
59          argin2.arg[0].arg=(char *)&x;
60          argin2.arg[1].size= sizeof(int);
61          argin2.arg[1].arg=(char *)&y;
62          argret2.ret[0].size= sizeof(float);
63          argret2.ret[0].ret=(float *)&z;
64
65          TIMER_CLEAR;
66          TIMER_START;
67
68          id_tsk1=rte_tsk_call( RTE_LOCAL_GUEST, 0, "sum\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/librte-math.so\0", &argin4, &
                argret4 );
69          printf( "id_task1 = %d\n", id_task1 );
70          fflush( stdout );
71
72          id_tsk2=rte_tsk_call( 0, 0, "mult\0", "/home/dbelo/Workspace/
                epusp/lahpc/rte/libs/librte-math.so\0", &argin2,  &argret2 );
73          printf( "id_task2 = %d\n", id_task2 );
74          fflush( stdout );
75
76          rte_sync(id_tsk1); // Synk Task 1
77          printf( "a * b = c >>> %d + %d = %f \n", a, b, b );
78          fflush(stdout);
79
80          rte_sync(id_tsk2); // Synk Task 2
81          printf( "x * y = z >>> %d * %d = %f \n", x, y, z);
82          fflush(stdout);
83
84          TIMER_STOP;
85
86          printf("Tempo: %f \n", TIMER_ELAPSED);
87          fflush(stdout);
88
89          rte_finalize();
90
91          printf("FIM\n");
92          fflush(stdout);
93          return 0;
94  }
```

G.2 Conjunto de tarefas RTE com funções de operações matemáticas básicas

```
1  #include <math.h>
2  #include <stdio.h>
3
```

```
 4 #include <rte.h>
 5
 6 void __attribute__((visibility ("default"))) sum( void *arg ) // soma(
       int a;int b;int* result)  resp: no endereco result
 7 {
 8         rte_args_t* arg_task = ( rte_args_t* )arg;
 9         int *a = ( int* )arg_task->args[ 0 ];
10         int *b = ( int* )arg_task->args[ 1 ];
11         int *c = ( int* )arg_task->rets[ 0 ];
12         printf("run sum\n");
13         printf("a=%d\n", a);
14         printf("b=%d\tadd res[%p]\n", b, result);
15         fflush(stdout);
16         *c = a + b;
17 }
18
19 void __attribute__((visibility ("default"))) subtract( void *arg )
20 {
21         rte_args_t* arg_task = ( rte_args_t* )arg;
22         int *a = ( int* )arg_task->args[ 0 ];
23         int *b = ( int* )arg_task->args[ 1 ];
24         int *c = ( int* )arg_task->rets[ 0 ];
25         printf("run subtract\n");
26         printf("a=%d\n", a);
27         printf("b=%d\tadd res[%p]\n", b, result);
28         fflush(stdout);
29         *c = a - b;
30 }
31
32 void __attribute__((visibility ("default"))) mult( void *arg )
33 {
34         rte_args_t* arg_task = ( rte_args_t* )arg;
35         int *a = ( int* )arg_task->args[ 0 ];
36         int *b = ( int* )arg_task->args[ 1 ];
37         float *c = ( float* )arg_task->rets[ 0 ];
38         printf("run mult\n");
39         printf("a=%d\n", a);
40         printf("b=%d\tadd res[%p]\n", b, result);
41         fflush(stdout);
42         *c = a * b;
43 }
44
45 void __attribute__((visibility ("default"))) divide( void *arg )
46 {
47         int *a = ( int* )arg_task->args[ 0 ];
48         int *b = ( int* )arg_task->args[ 1 ];
49         float *c = ( float* )arg_task->rets[ 0 ];
50
51         printf("run divide\n");
52         printf("a=%d\n", a);
53         printf("b=%d\tadd res[%p]\n", b, result);
54
55         fflush(stdout);
56         *c = (b != 0)? (a / b): (float)INFINITY;
57 }
```

G.3 Teste da API RTE com chamada e execução de uma tarefa numa GPGPU Local(*Host*)

```
1 //multiplicacao de matrizes N x N (alocacao dinamica), c=a x b
2 //versao 1 tarefa  - mult sequencial
3 #include <stdio.h>
4 #include <stdlib.h>
```

```
 5 #include <sys/time.h>
 6
 7 #include <rte.h>
 8
 9 #include <mede_time.h>
10
11 #define NTHREADS 8
12 #define MAX_DEVICES 100
13
14 int list_type_dvc[MAX_DEVICES];
15
16 int size_matrix;
17 double *a;
18 double *b;
19 double *c;
20
21 int main(int argc,char **argv)
22 {
23         int i,j;
24         int id_task;
25         list_type_dvc[0]=0;  // CPU
26         printf("INICIO\n");
27         fflush(stdout);
28         if (argc==1) {
29                 printf("FALTOU ESPECIFICAR N: TAMANHO DAS MATRIZES\n");
30                 return 1;
31         }
32
33         size_matrix= atoi(argv[1]);
34         printf ("size_matrix=%d\n",size_matrix);
35
36         a=(double *)calloc(size_matrix*size_matrix,sizeof(double));
37         b=(double *)calloc(size_matrix*size_matrix,sizeof(double));
38         c=(double *)calloc(size_matrix*size_matrix,sizeof(double));
39         printf("INICIO\n");
40         fflush(stdout);
41         for (i = 0; i < size_matrix; i++)
42                 for (j=0; j < size_matrix; j++){
43                         a[i*size_matrix+j] = 1.0;
44                         b[i*size_matrix+j] = 1.0;
45         }
46         rte_arg_list_t  argin1;
47         rte_ret_list_t argret1;
48         rte_init_args(3,&argin1, 1,&argret1);
49         argin1.arg[0].size= sizeof(int);
50         argin1.arg[0].arg=(char *)&size_matrix;;
51         argin1.arg[1].size= size_matrix*size_matrix*sizeof(double);
52         argin1.arg[1].arg=(char *)a;
53         argin1.arg[2].size= size_matrix*size_matrix*sizeof(double);
54         argin1.arg[2].arg=(char *)b;
55         argret1.ret[0].size= size_matrix*size_matrix*sizeof(double);
56         argret1.ret[0].ret=(char *)c;
57
58         TIMER_CLEAR;
59         TIMER_START;
60
61         id_task=rte_tsk_call(RTE_LOCAL_GUEST, 0, "mult_matrix\0", "/home
               /dbelo/Workspace/epusp/lahpc/rte/libs/libmult-1-gpu.so\0", &
               argin1,  &argret1);
62         printf("id_task=%d\n",id_task);
63         fflush(stdout);
64         rte_sync(id_task); // Synk Task
65
66         TIMER_STOP;
```

```
67
68          printf("Tempo: %f \n", TIMER_ELAPSED);
69          printf("Aps rte_sync\n");
70          printf("matrix_c[0][0]=%f\n",c[0]);
71          printf("TERMINOU task1\n");
72          printf("matrix_c[%d][%d]=%f\n",size_matrix-1,size_matrix-1,c[(
                size_matrix-1)*(size_matrix-1)]);
73          fflush(stdout);
74          rte_finalize();
75
76          printf("FIM\n");
77          return 0;
78  }
```

G.4 Conjunto de tarefas RTE de multiplicação de matrizes dinâmicas na GPGPUs

```
1   //multiplicacao de matrizes - versao sequencial
2   // matrizes a e b
3   // matriz resultante c
4   #include <math.h>
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include <string.h>
8
9   #include <rte.h>
10
11  #define TILE_WIDTH 32
12
13  __global__ void matrixMulKernel(double* A, double* B, double* C, int
        width)
14  {
15          __shared__ double sA[TILE_WIDTH][TILE_WIDTH];
16          __shared__ double sB[TILE_WIDTH][TILE_WIDTH];
17          int m,k;
18          int bx = blockIdx.x; int by = blockIdx.y;
19          int tx = threadIdx.x; int ty = threadIdx.y;
20
21          int row = by * TILE_WIDTH + ty;
22          int col = bx * TILE_WIDTH + tx;
23
24          double p_value = 0.0;
25          for ( m = 0; m < width / TILE_WIDTH; ++m)
26          {
27                  sA[ty][tx] = A[row * width + (m * TILE_WIDTH + tx)];
28                  sB[ty][tx] = B[(m * TILE_WIDTH + ty) * width + col];
29
30                  __syncthreads();
31
32                  for ( k = 0; k < TILE_WIDTH; ++k)
33                          p_value += sA[ty][k] * sB[k][tx];
34
35                  __syncthreads();
36          }
37
38          C[row * width + col] = p_value;
39  }
40
41
42  #ifdef __cplusplus
43    extern "C" {
44  #endif
45  void __attribute__((visibility ("default"))) mult_matrix(void *arg)
```

```
46 {
47         int i, j ;
48         printf("TESTE\n");
49         fflush(stdout);
50         int * size;
51         double *a;
52         double *b;
53         double *c;
54         double* dA;
55         double* dB;
56         double* dC;
57         rte_args_t *arg_task=(rte_args_t *)arg;  // enderecos de
               argumentos e endereco de resposta
58         size=(int *)arg_task->args[0];
59         c=(double *)arg_task->rets[0];
60         int size_matrix=*size;
61         a=(double *)arg_task->args[1];
62         b=(double *)arg_task->args[2];
63 printf("tamanho sz=%d\n",size_matrix);
64 fflush(stdout);
65         printf("a[0][0]=%f  \n",a[0]);
66         for (i = 0; i < size_matrix; i++)
67             for (j=0; j < size_matrix; j++)
68                   c[i*size_matrix+j] = 0.0;
69         // multiplica  GPU
70         //Copiando dados para device
71         int size_total=size_matrix*size_matrix*sizeof(double);
72         cudaMalloc((void**) &dA, size_total);
73         cudaMalloc((void**) &dB, size_total);
74         cudaMalloc((void**) &dC, size_total);
75         cudaMemcpy(dA, a, size_total, cudaMemcpyHostToDevice);
76         cudaMemcpy(dB, b, size_total, cudaMemcpyHostToDevice);
77         // especificando dimensoes de Grid e Bloco
78         dim3 dimGrid(size_matrix / TILE_WIDTH, size_matrix / TILE_WIDTH,
               1);
79         dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
80         // calculando multiplicacao de matrizes
81         matrixMulKernel<<<dimGrid, dimBlock>>>(dA, dB, dC, size_matrix);
82         // copiando matriz resultante para host
83         cudaMemcpy(c, dC, size_total, cudaMemcpyDeviceToHost);
84
85 //printf("FUNCAO mat_c-gpu[%d][%d]=%f \n",size_matrix-1,size_matrix-1,c
       [(size_matrix-1)*size_matrix+(size_matrix-1)]);
86 printf("FUNCAO mat_c-gpu[%d][%d]=%f \n",size_matrix-1,size_matrix-1,c[
       size_matrix*size_matrix-1]);
87 fflush(stdout);
88   printf("FIM mult\n");
89   fflush(stdout);
90 }
91 #ifdef __cplusplus
92 }
93 #endif
```

G.5 Teste da API RTE com chamadas e execuções de quatro tarefas remotas e uma local nas respectivas GPGPUs

```
1 //multiplicacao de matrizes N x N (alocacao dinamica), c=a x b
2 //versao 1 tarefa  - mult sequencial
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/time.h>
6
7 #include <rte.h>
```

```
 8
 9  #include <mede_time.h>
10
11  #define NTHREADS 8
12
13  rte_dvcs_types_t list_type_dvc[ 128 ];
14
15  int size_matrix;
16  double* a0;
17  double* b0;
18  double* c0;
19
20  double* a1;
21  double* b1;
22  double* c1;
23
24  double* a2;
25  double* b2;
26  double* c2;
27
28  double* a3;
29  double* b3;
30  double* c3;
31
32  double* a4;
33  double* b4;
34  double* c4;
35
36  int main(int argc,char **argv)
37  {
38          int i,j;
39          int id_task0, id_task1, id_task2, id_task3, id_task4;
40          list_type_dvc[0]=rte_dvc_cpu;   // CPU
41
42          rte_arg_list_t argin0;
43          rte_ret_list_t argret0;
44
45          rte_arg_list_t argin1;
46          rte_ret_list_t argret1;
47
48          rte_arg_list_t argin2;
49          rte_ret_list_t argret2;
50
51          rte_arg_list_t argin3;
52          rte_ret_list_t argret3;
53
54          rte_arg_list_t argin4;
55          rte_ret_list_t argret4;
56
57          printf("INICIO\n");
58          fflush(stdout);
59          if (argc==1) {
60                  printf("FALTOU ESPECIFICAR N: TAMANHO DAS MATRIZES\n");
61                  fflush(stdout);
62                  return 1;
63          }
64
65          if ( rte_init( "mult-1-dynamic-gpu.cfg" ) != 0 ) {
66                  printf( "Nao inicializar o Processador Remoto .\n" );
67                  fflush(stdout);
68                  return 1;
69          }
70
71          rte_init_dvcs_lst( list_type_dvc, 1 );
```

```
72
73          size_matrix= atoi(argv[1]);
74          printf ("size_matrix=%d\n",size_matrix);
75
76          a0 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
77          b0 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
78          c0 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
79
80          a1 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
81          b1 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
82          c1 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
83
84          a2 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
85          b2 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
86          c2 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
87
88          a3 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
89          b3 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
90          c3 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
91
92          a4 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
93          b4 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
94          c4 = (double *)calloc(size_matrix*size_matrix,sizeof(double));
95
96          printf("INICIO\n");
97          fflush(stdout);
98
99          /* <<<<<< I N I C I A L I Z A   O S   A R G U M E N T O S
                >>>>>>> */
100         for (i = 0; i < size_matrix; i++)
101                 for (j=0; j < size_matrix; j++){
102                         a0[i*size_matrix+j] = 1.0;
103                         b0[i*size_matrix+j] = 1.0;
104
105                         a1[i*size_matrix+j] = 2.0;
106                         b1[i*size_matrix+j] = 2.0;
107
108                         a2[i*size_matrix+j] = 3.0;
109                         b2[i*size_matrix+j] = 3.0;
110
111                         a3[i*size_matrix+j] = 4.0;
112                         b3[i*size_matrix+j] = 4.0;
113
114                         a4[i*size_matrix+j] = 5.0;
115                         b4[i*size_matrix+j] = 5.0;
116         }
117
118         rte_init_args(3,&argin0, 1,&argret0);
119
120         argin0.arg[0].size= sizeof(int);
121         argin0.arg[0].arg=(char *)&size_matrix;;
122         argin0.arg[1].size= size_matrix*size_matrix*sizeof(double);
123         argin0.arg[1].arg=(char *)a0;
124         argin0.arg[2].size= size_matrix*size_matrix*sizeof(double);
125         argin0.arg[2].arg=(char *)b0;
126         argret0.ret[0].size= size_matrix*size_matrix*sizeof(double);
127         argret0.ret[0].ret=(char *)c0;
128
129         rte_init_args(3,&argin1, 1,&argret1);
130
131         argin1.arg[0].size= sizeof(int);
132         argin1.arg[0].arg=(char *)&size_matrix;;
133         argin1.arg[1].size= size_matrix*size_matrix*sizeof(double);
134         argin1.arg[1].arg=(char *)a1;
```

```
135         argin1.arg[2].size= size_matrix*size_matrix*sizeof(double);
136         argin1.arg[2].arg=(char *)b1;
137         argret1.ret[0].size= size_matrix*size_matrix*sizeof(double);
138         argret1.ret[0].ret=(char *)c1;
139
140         rte_init_args(3,&argin2, 1,&argret2);
141
142         argin2.arg[0].size= sizeof(int);
143         argin2.arg[0].arg=(char *)&size_matrix;;
144         argin2.arg[1].size= size_matrix*size_matrix*sizeof(double);
145         argin2.arg[1].arg=(char *)a2;
146         argin2.arg[2].size= size_matrix*size_matrix*sizeof(double);
147         argin2.arg[2].arg=(char *)b2;
148         argret2.ret[0].size= size_matrix*size_matrix*sizeof(double);
149         argret2.ret[0].ret=(char *)c2;
150
151         rte_init_args(3,&argin3, 1,&argret3);
152
153         argin3.arg[0].size= sizeof(int);
154         argin3.arg[0].arg=(char *)&size_matrix;;
155         argin3.arg[1].size= size_matrix*size_matrix*sizeof(double);
156         argin3.arg[1].arg=(char *)a3;
157         argin3.arg[2].size= size_matrix*size_matrix*sizeof(double);
158         argin3.arg[2].arg=(char *)b3;
159         argret3.ret[0].size= size_matrix*size_matrix*sizeof(double);
160         argret3.ret[0].ret=(char *)c3;
161
162         rte_init_args(3,&argin4, 1,&argret4);
163
164         argin4.arg[0].size= sizeof(int);
165         argin4.arg[0].arg=(char *)&size_matrix;;
166         argin4.arg[1].size= size_matrix*size_matrix*sizeof(double);
167         argin4.arg[1].arg=(char *)a4;
168         argin4.arg[2].size= size_matrix*size_matrix*sizeof(double);
169         argin4.arg[2].arg=(char *)b4;
170         argret4.ret[0].size= size_matrix*size_matrix*sizeof(double);
171         argret4.ret[0].ret=(char *)c4;
172
173         TIMER_CLEAR;
174         TIMER_START;
175
176         id_task0=rte_tsk_call(RTE_LOCAL_GUEST, 0, "mult_matrix\0", "/
                home/dbelo/Workspace/epusp/lahpc/rte/libs/libmult-1-gpu.so\0"
                , &argin0,  &argret0);
177         printf("id_task0=%d\n",id_task1);
178         fflush(stdout);
179
180         id_task1=rte_tsk_call(0, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-1-gpu.so\0", &argin1,
                 &argret1);
181         printf("id_task1=%d\n",id_task1);
182         fflush(stdout);
183
184         id_task2=rte_tsk_call(1, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-1-gpu.so\0", &argin2,
                 &argret2);
185         printf("id_task2=%d\n",id_task2);
186         fflush(stdout);
187
188         id_task3=rte_tsk_call(2, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-1-gpu.so\0", &argin3,
                 &argret3);
189         printf("id_task3=%d\n",id_task3);
190         fflush(stdout);
```

```
191
192         id_task4=rte_tsk_call(3, 0, "mult_matrix\0", "/home/dbelo/
                Workspace/epusp/lahpc/rte/libs/libmult-1-gpu.so\0", &argin4,
                 &argret4);
193         printf("id_task4=%d\n",id_task4);
194         fflush(stdout);
195
196         rte_sync(id_task0); // Synk Task 0
197         printf("Aps rte_sync [0]\n");
198         printf("matrix_c0[0][0]=%f\n",c0[0]);
199         printf("TERMINOU task0\n");
200         printf("matrix_c0[%d][%d]=%f\n\n",size_matrix-1,size_matrix-1,c0
                [(size_matrix-1)*(size_matrix-1)]);
201         fflush(stdout);
202
203         rte_sync(id_task1); // Synk Task 1
204         printf("Aps rte_sync [1]\n");
205         printf("matrix_c1[0][0]=%f\n",c1[0]);
206         printf("TERMINOU task1\n");
207         printf("matrix_c1[%d][%d]=%f\n\n",size_matrix-1,size_matrix-1,c1
                [(size_matrix-1)*(size_matrix-1)]);
208         fflush(stdout);
209
210         rte_sync(id_task2); // Synk Task 2
211         printf("Aps rte_sync [2]\n");
212         printf("matrix_c2[0][0]=%f\n",c2[0]);
213         printf("TERMINOU task2\n");
214         printf("matrix_c2[%d][%d]=%f\n\n",size_matrix-1,size_matrix-1,c2
                [(size_matrix-1)*(size_matrix-1)]);
215         fflush(stdout);
216
217         rte_sync(id_task3); // Synk Task 3
218         printf("Aps rte_sync [3]\n");
219         printf("matrix_c3[0][0]=%f\n",c3[0]);
220         printf("TERMINOU task3\n");
221         printf("matrix_c3[%d][%d]=%f\n\n",size_matrix-1,size_matrix-1,c3
                [(size_matrix-1)*(size_matrix-1)]);
222         fflush(stdout);
223
224         rte_sync(id_task4); // Synk Task 4
225         printf("Aps rte_sync [4]\n");
226         printf("matrix_c4[0][0]=%f\n",c4[0]);
227         printf("TERMINOU task4\n");
228         printf("matrix_c4[%d][%d]=%f\n",size_matrix-1,size_matrix-1,c4[(
                size_matrix-1)*(size_matrix-1)]);
229         fflush(stdout);
230
231         TIMER_STOP;
232
233         printf("Tempo: %f \n", TIMER_ELAPSED);
234         fflush(stdout);
235
236         rte_finalize();
237
238         printf("FIM\n");
239         fflush(stdout);
240         return 0;
241 }
```