XVIII Simpósio em Sistemas Computacionais de Alto Desempenho
De 17 a 20 de Outubro de 2017
Campinas – SP

Minicursos do WSCAD 2017

Sociedade Brasileira de Computação – SBC

Organizador
Alexandro Baldassin

Realização
Sociedade Brasileira de Computação – SBC
Universidade de Campinas – UNICAMP
Universidade Estadual Paulista – UNESP

ISBN: 978-85-7669-418-2

Neste livro estão compilados os seis minicursos apresentados durante o XVIII Simpósio em Sistemas Computacionais de Alto Desempenho, realizado entre os dias 17 e 20 de outubro de 2017 em Campinas, São Paulo. Os minicursos apresentados abrangem desde aspectos mais introdutórios de programação paralela com memória compartilhada até técnicas avançadas para tolerância de falhas em sistemas distribuídos.

O minicurso *Sistemas de Computação Paralelos com a Linguagem Python* apresenta os fundamentos para desenvolvimento de código paralelo usando a linguagem e módulos Python. Além de cobrir máquinas multi/many-cores, o autor Luciano Silva também aborda o desenvolvimento para plataformas baseadas em clusters e FPGAs.

Em *Intel Modern Code: Programação Paralela e Vetorial AVX para o Processador Intel Xeon Phi Knights Landing*, Eduardo Cruz e colegas apresentam os paradigmas de programação paralela e vetorial utilizando o processador Intel Xeon Phi Knights Landing como plataforma.

Rogério Gonçalves e colegas apresentam OpenMP do ponto de vista de código gerado em *Introdução à Programação Paralela com OpenMP: Além das Diretivas de Compilação.* Nesse minicurso é mostrado como as diretivas OpenMP são expandidas durante a compilação do código e suas respectivas relações com conceitos empregados em computação paralela.

O suporte para programação paralela em Erlang é discutido no minicurso *Programação Concorrente em Erlang.* Alexandre Oliveira e colegas apresentam nesse minicurso as principais características de Erlang, assim como o processo de compilação e execução de códigos sequenciais e paralelos.

Já em *Técnicas para a Construção de Sistemas MPI Tolerantes a Falhas*, Edson Camargo e Elias Duarte Jr. apresentam métodos para tolerância de falhas em programas distribuídos baseados em MPI. Além de técnicas tradicionais como a *rollback-recovery*, os autores também discutem a mais recente proposta para tolerância a falhas em MPI, a ULFM (*User Level Failure Mitigation*).

Finalmente, o minicurso *Introdução à Otimização de Desempenho para Arquitetura Intel Xeon Phi Knights Landing (KNL)* discute formas para otimização de desempenho na plataforma KNL. Silvio Stanzani e colegas apresentam a arquitetura KNL e mostram como melhorar o desempenho dessa arquitetura através de seu sistema de memória heterogêneo, vetorização (AVX-512) e *prefetching.*

Sumário

1. **Sistemas de Computação Paralelos com a Linguagem Python**
   Luciano Silva (Universidade Mackenzie)

2. **Intel Modern Code: Programação Paralela e Vetorial AVX para o Processador Intel Xeon Phi Knights Landing**
   Eduardo H. M. Cruz, Arthur M. Krause, Emmanuell D. Carreno, Matheus S. Serpa, Philippe O. A. Navaux (UFRGS), Marco A. Z. Alves (UFPR), Igor J. F. Freitas (Intel Brasil)

3. **Introdução à Programação Paralela com OpenMP: Além das Diretivas de Compilação**
   Rogério A. Gonçalves, João M. de Queiroz Filho (UTFPR), Alfredo Goldman (IME, USP)

4. **Programação Concorrente em Erlang**
   Alexandre P. de Oliveira (Faculdade de Tecnologia de Lins), Paulo S. L. de Souza, Simone do R. S. de Souza (ICMC, USP)

5. **Técnicas para a Construção de Sistemas MPI Tolerantes a Falhas**
   Edson T. de Camargo (UTFPR), Elias P. Duarte Jr. (UFPR)

6. **Introdução à Otimização de Desempenho para Arquitetura Intel Xeon Phi Knights Landing (KNL)**
   Silvio Stanzani, Jefferson Fialho, Raphael Cóbe, Rogério Iope (NCC, UNESP), Igor J. F. Freitas (Intel Brasil)

Capítulo 1

Sistemas de Computação Paralelos com a Linguagem Python

Luciano Silva
Faculdade de Computação e Informática, Universidade Mackenzie
luciano.silva@mackenzie.br

Resumo

Python é uma linguagem de programação de alto nível, com algumas características bastante interessantes para programas paralelos: legibilidade de código, curva de aprendizagem muito curta e alcance em várias plataformas como processadores *multicore* e *manycore*, *clusters* e FPGAs. Dentro deste contexto, o objetivo deste capítulo é apresentar os fundamentos de desenvolvimento de programas paralelos utilizando a linguagem e módulos Python como suporte. Serão abordadas quatro plataformas paralelas: processadores *multicore*, processadores *manycore* (coprocessadores e GPUs), *clusters* e FPGAs. Além dos fundamentos de arquitetura destas plataformas, serão construídos e discutidos programas em Python para cada uma delas.

Palavras-chave*: Python Multicore, Python Manycore, Python em Cluster, Python FPGA, Sistemas Paralelos.*

1.1 Introdução

Python é uma linguagem de programação de alto nível, interpretada, de script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte (PALACH, 2014). Foi lançada por Guido van Rossum em 1991. Atualmente, possui um modelo de desenvolvimento comunitário, aberto e gerenciado pela organização sem fins lucrativos Python Software Foundation. Apesar de várias partes da linguagem possuírem padrões e especificações formais, a linguagem como um todo não é formalmente especificada.

A linguagem foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional, prioriza a legibilidade do código sobre a velocidade ou expressividade, além de combinar uma sintaxe concisa e clara com os recursos poderosos de sua biblioteca padrão e por módulos e frameworks desenvolvidos por terceiros.

Python é uma linguagem de propósito geral de alto nível, multi-paradigma, suporta o paradigma orientado a objetos, imperativo, funcional e procedimental. Possui tipagem dinâmica e uma de suas principais características é permitir a fácil leitura do código e exigir poucas linhas de código se comparado ao mesmo programa em outras linguagens. Devido às suas características, ela é principalmente utilizada para análise de dados em geral, tendo como origem primordial os algoritmos de Bioinformática.

Recentemente, houve um interesse bastante acentuado pelo uso de Python no desenvolvimento de programas paralelos para os mais diversos ambientes de alto desempenho, como processadores *multicore*, processadores *manycore*, *clusters* e FPGA. Dentro deste contexto, o objetivo deste pequeno texto é introduzir as técnicas fundamentais de programação em Python para estes ambientes, evidenciando a simplicidade das construções.

Este texto está organizado a seguinte forma:

- a Seção 1.2 apresenta os fundamentos da organização de programas em Python
- a Seção 1.3 apresenta a organização dos ambientes paralelos que serão tratados no texto
- a Seção 1.4 mostra os fundamentos de programação em Python para processadores multicore
- a Seção 1.5 mostra os fundamentos de programação em Python para processadores manycore
- a Seção 1.6 mostra os fundamentos de programação em Python para clusters
- finalmente, a Seção 1.7 mostra os fundamentos de programação em Python para FPGA.

No final do texto, encontram-se algumas referências adicionais para o leitor interessado em aprofundar as técnicas de programação para os ambientes tratados no texto.

1.2 Linguagem de Programação Python

Python é uma linguagem de programação de alto nível, interpretada, de script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte (PALACH, 2014).

O comando condicional é mostrado nas construções abaixo, juntamente com os respectivos operadores lógicos:

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

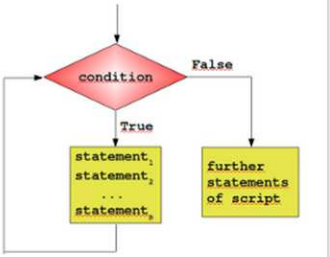
operator	function
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	not equal

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	
TRUE	NULL	NULL	TRUE	
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	
FALSE	NULL	FALSE	NULL	
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	NULL	
NULL	NULL	NULL	NULL	

```
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

if x > y and x > z:
    maximum = x
elif y > x and y > z:
    maximum = y
else:
    maximum = z

print("The maximal value is: " + str(maximum))
```

Figura 1: Comando condicional em Python.

O comando de repetição é mostrado nas construções abaixo, juntamente com os respectivos operadores lógicos:




```
n = 100

sum = 0
counter = 1
while counter <= n:
    sum = sum + counter
    counter += 1

print("Sum of 1 until %d: %d" % (n,sum))
```

Figura 2: Comando de repetição em Python.

Uma variação do comando de repetição é o comando for, que itera sobre uma lista de elementos:

```
for <variable> in <sequence>:
        <statements>
```

```
range(begin,end, step)
```

```
n = 100

sum = 0
for counter in range(1,n+1):
    sum = sum + counter

print("Sum of 1 until %d: %d" % (n,sum))
```

Figura 3: Comando de repetição for em Python.

Funções são construções bastante comuns em Python e podem ser declaradas da seguinte forma:

```
def function-name(Parameter list):
    statements, i.e. the function body
```

```
def fahrenheit(T_in_celsius):
    """ returns the temperature in degrees Fahrenheit """
    return (T_in_celsius * 9 / 5) + 32

for t in (22.6, 25.8, 27.3, 29.8):
    print(t, ": ", fahrenheit(t))
```

Figura 4: Declaração de funções em Python.

Funções em Python também suportam construções no estilo de programação funcional, onde se pode usar funções anônimas com abstrações lambda.

Vetores em Python são construídos através de listas lineares indexadas. Os elementos nos vetores podem ser heterogêneos, conforme mostrado no exemplo abaixo:

```
>>> languages = ["Python", "C", "C++", "Java", "Perl"]
>>> print(languages[0] + " and " + languages[1] + " are quite different!")
Python and C are quite different!
>>> print("Accessing the last element of the list: " + languages[-1])
Accessing the last element of the list: Perl
>>>
```

```
group = ["Bob", 23, "George", 72, "Myriam", 29]
```

Figura 5: Declaração de vetores em Python.

Em Python também existe uma construção bastante útil, chamada conjunto (set), que permite construir subconjuntos de elementos de um vetor:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)               # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # fast membership testing
True
>>> 'crabgrass' in fruit
False
```

Figura 6: Declaração de conjuntos (sets) em Python.

Dicionários, por sua vez, permitem que elementos sejam indexados por uma chave, conforme mostrado no exemplo abaixo:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

Figura 7: Declaração de dicionários em Python.

A linguagem Python também é bastante conhecida pela quantidade de módulos disponíveis para as mais diversas utilidades. Abaixo, tem-se um panorama de alguns destes módulos:

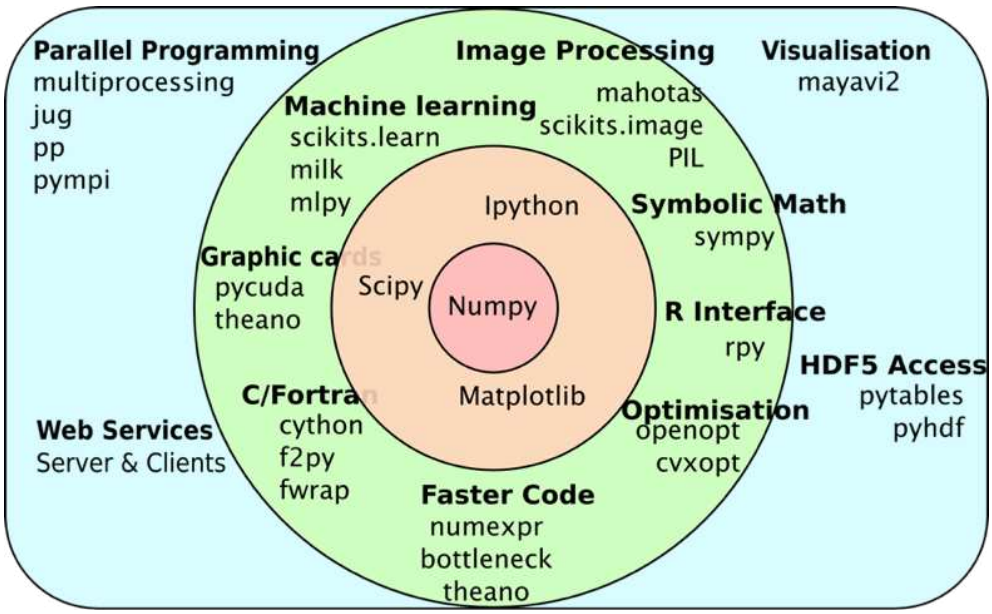


Figura 8: Alguns exemplos de módulos em Python.

Como exemplo, o módulo numpy possui diversas rotinas para manipulação de vetores uni e n-dimensionais. Várias operações com vetores declarados como tipos do numpy, conforme mostrado no exemplo abaixo:

```
import numpy as np
x = np.array([42,47,11], int)
x
array([42, 47, 11])
```

```
>>> x = np.array([1,5,2])
>>> y = np.array([7,4,1])
>>> x + y
array([8, 9, 3])
>>> x * y
array([ 7, 20,  2])
>>> x - y
array([-6,  1,  1])
>>> x / y
array([0, 1, 2])
>>> x % y
array([1, 1, 0])
```

Figura 9: Exemplos de operações envolvendo vetores em numpy.

1.3 Arquiteturas Paralelas em Python

Os ambientes de alto desempenho que foram, inicialmente, suportados em Python encontram-se na figura abaixo:

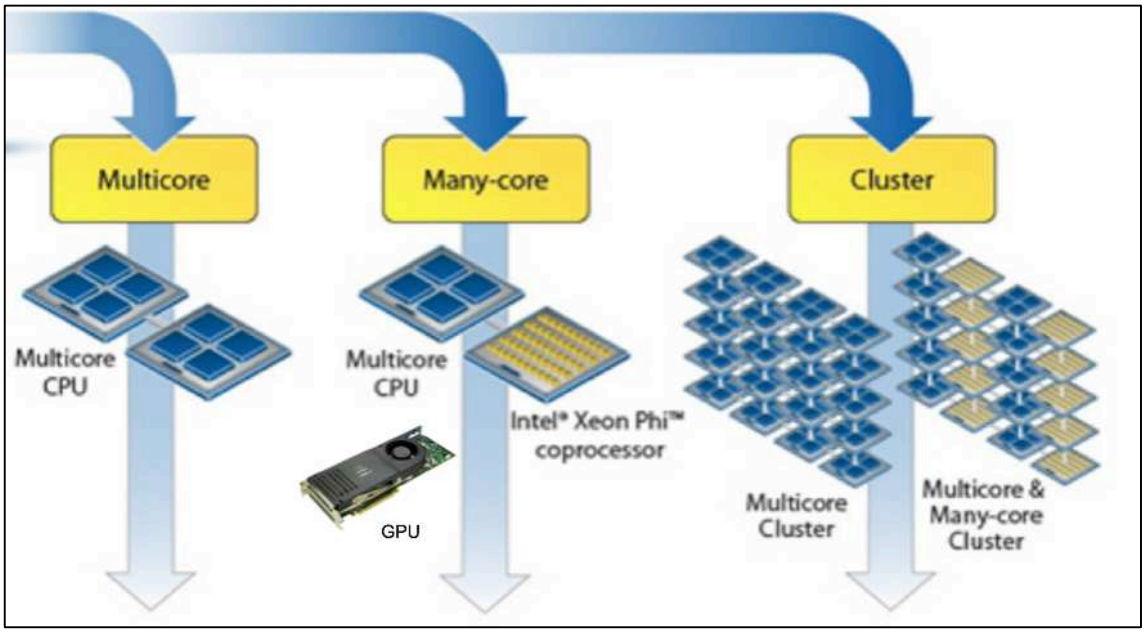


Figura 10: Arquiteturas paralelas suportadas em Python.
Fonte: adaptada de Intel (2016).

Um processador multinúcleo (múltiplos núcleos, do inglês multicore) (BARLAS, 2014) é o que tem dois ou mais núcleos de processamento (cores) no interior de um único chip. Estes núcleos são responsáveis por dividir as tarefas entre si, ou seja, permitem trabalhar em um ambiente multitarefa. Em processadores de um só núcleo, as funções de multitarefa podem ultrapassar a capacidade da CPU, o que resulta em queda no desempenho enquanto as operações aguardam para serem processadas. Em processadores de múltiplos núcleos o sistema operacional trata cada um desses núcleos como um processador diferente. Na maioria dos casos, cada unidade possui seu próprio cache e pode processar várias instruções quase simultaneamente. Adicionar novos núcleos de processamento a um processador (único encapsulamento) possibilita que as instruções das aplicações sejam executadas em paralelo, como se fossem 2 ou mais processadores distintos.
Um processador *manycore*, por sua vez, possui uma quantidade muito maior de cores independentes (centenas a milhares), quando comparados aos processadores *multicore*. Exemplos típicos de processadores manycore incluem as aceleradoras (com a Intel Xeon Phi) e Unidades de Processamento Gráfico (GPU).
Um cluster (do inglês cluster : 'grupo, aglomerado') consiste em computadores fracamente ou fortemente ligados que trabalham em conjunto, de modo que, em muitos aspectos, podem ser considerados como um único sistema. Diferentemente

dos computadores em grade, computadores em cluster têm cada conjunto de nós, para executar a mesma tarefa, controlado e programado por software.

Além do suporte às arquiteturas mostradas anteriormente, a linguagem Python ainda dá suporte a FPGA (Field Programmable Gate Array). Um FPGA é um circuito integrado projetado para ser configurado por um consumidor ou projetista após a fabricação – de onde advém "programável em campo". A grande maioria dos chips que encontramos em nosso dia-a-dia, circuitos que acompanham as televisões, celulares, etc., já vêm todos pré-programados (ASIC), isto é, com as suas funcionalidades todas definidas no ato de fabricação. Surgiu então uma categoria nova de hardware reconfigurável, o qual têm as suas funcionalidades definidas exclusivamente pelos usuários e não pelos fabricantes. A figura abaixo mostra um exemplo de FPGA:

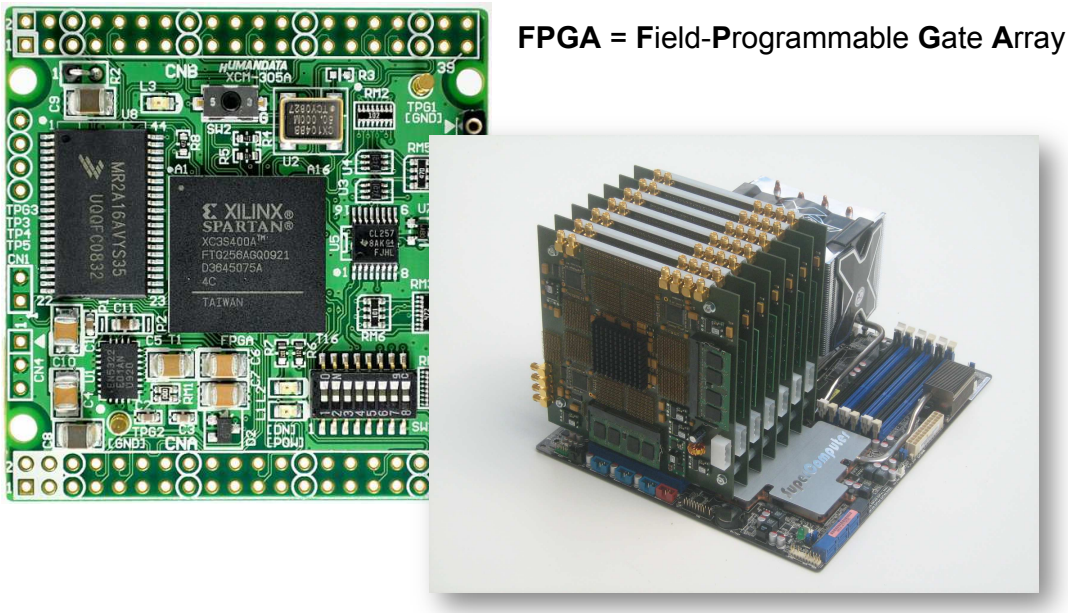


Figura 11: Um exemplo de FPGA isolada e em cluster.
Fonte: www.altera.com.

Um FPGA é um dispositivo semicondutor que é largamente utilizado para o processamento de informações digitais. Foi criado pela Xilinx Inc., e teve o seu lançamento no ano de 1985 como um dispositivo que poderia ser programado de acordo com as aplicações do usuário (programador). O FPGA é composto basicamente por três tipos de componentes: blocos de entrada e saída (IOB), blocos lógicos configuráveis (CLB) e chaves de interconexão (Switch Matrix). Os blocos lógicos são dispostos de forma bidimensional, as chaves de interconexão são dispostas em formas de trilhas verticais e horizontais entre as linhas e as colunas dos blocos lógicos.

1.4 Programação Multicore em Python

A maneira mais simples de se programar para processadores multicore utiliza o módulo multiprocessing. Neste módulo, existe uma classe chamada Pool que pode dividir o processamento entre os cores, conforme mostrado no exemplo abaixo:

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    p = Pool(5)
    print(p.map(f, [1, 2, 3]))
```

Figura 12: Um programa paralelo com o módulo multicore.

Neste exemplo, foi criado um pool de cinco processos para aplicar a função f sobre um determinado vetor. Ao invés de deixar o gerenciamento dos processos pelo multiprocessing, pode-se importar a classe Process e gerenciar o processo por código, conforme mostrado no exemplo abaixo:

```
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Figura 13: Gerenciamento de processos com a classe Process.

Processos em multiprocessing podem se comunicar através de uma fila, representada pela classe Queue:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print q.get()    # prints "[42, None, 'hello']"
    p.join()
```

Figura 14: Comunicação entre processos por fila em Python.

Como exemplo de uso de programação multicore em Python, será descrita a seguir uma aplicação para alto desempenho, aplicada a processamento de dados de Astrofísica.

O Método dos Trânsitos Planetários é um dos métodos possíveis para detecção de planetas orbitando uma estrela (PERRYMAN, 2014). Este método consiste em realizar medidas fotométricas do fluxo luminoso da estrela, gerando um conjunto de dados chamado curva de luz. Quando um planeta passa em frente à estrela, ocorre uma alteração no fluxo luminoso da estrela, percebida através da deformação de sua curva de luz.

A Figura 15 ilustra os principais elementos da geometria da curva de luz de uma estrela.

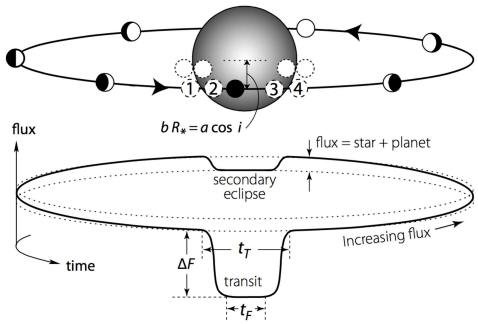


Figura 15. Geometria das curvas de luz (Perryman, 2014).

A deformação começa a ocorrer no primeiro ponto de contato (número 1 da Figura 1) e termina no último ponto de contato (número 4 da Figura 1). Se a curva de luz for conhecida, é possível se estimar diversos parâmetros como profundidade do trânsito (ΔF) e duração do trânsito (t_T) e, a partir deles, outros parâmetros do sistema estrela-planeta como densidade da estrela, raio do planeta, inclinação da órbita (i) e fator de impacto (b).

Porém, devido a diversos fatores como ruído, os trânsitos em curvas de luz reais não são tão explícitos como a geometria mostrada na Figura 15. Na Figura 16, tem-se um exemplo típico de uma curva de luz real (estrela Kepler 36).

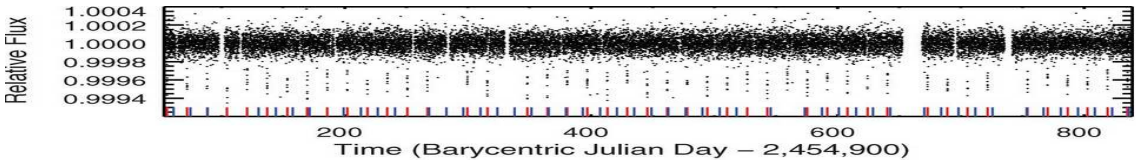


Figura 16. Curva de luz da estrela Kepler 36 (Kepler, 2014).

A simples observação visual desta curva de luz não permite a detecção imediata de trânsitos planetários. Para isto, podem ser necessários vários procedimentos. Um deles é a redução de ruído através de processos de filtragem. Outro, dependendo do tempo considerado na observação da estrela, refere-se ao fato de que uma curva de luz pode conter repetições periódicas do trânsito. Neste caso, a curva precisa ser segmentada em trechos que tenham probabilidade de conter

trânsitos e, a partir da repetição das rotinas de detecção de trânsitos em cada um destes segmentos, constrói-se uma curva de luz média. Tanto a detecção do trânsito em si, quanto as rotinas de filtragem e segmentação dos sinais têm custo computacional alto e podem ter desempenho melhorado com uso de processamento paralelo.

Um segmento de curva de luz pode ser representado matematicamente como uma sequência finita $\{x_i\}_{i=1,\ldots,n}$ de valores de fluxo luminoso. Adicionalmente, a cada valor x_i, será associado um ruído Gaussiano aditivo de média zero, com desvio-padrão σ_i. Para se detectar os índices desta sequência que indiquem o início e o final de um trânsito planetário, pode-se utilizar o Algoritmo de Mínimos Quadrados Box (BLS – *Box Least Squares*), proposto por Kovács *et al*. (2002).

O algoritmo BLS consiste em encontrar dois índices i_1e i_2 no intervalo discreto [1,...,n], com $i_1 < i_2$, chamados, respectivamente, de índices de início e final do trânsito e que minimizem a expressão quadrática

$$D = \left[\sum_{i=1}^{n} w_i x_i^2\right] - \frac{s^2}{r(1-r)}$$

com os valores w_i, s e r dados por:

$$w_i = \frac{1}{\sigma_i{}^2 \sum_{j=1}^{n} \sigma_j} \qquad s = \sum_{i=i_1}^{i_2} w_i x_i \qquad r = \sum_{i=i_1}^{i_2} w_i$$

Em notação algorítmica, o BLS sequencial consiste em busca por força bruta por todos os pares i_1e i_2, com $i_1 < i_2$:

```
1. c ← ∑_{i=1}^{n} w_i x_i^2
2. D_min = +∞
3. para i_1 de 1 até n-1 faça
4.     para i_2 de (i_1 + 1) até n faça
5.         f ← s^2 / (r(1−r))
6.         se (D_min > c − f)
7.             então D_min = c − f
8.                   (i, j) = (i_1 , i_2 )
9. retorne (i, j)
```

Algoritmo Sequencial. Pseudocódigo da versão sequencial do algoritmo BLS.

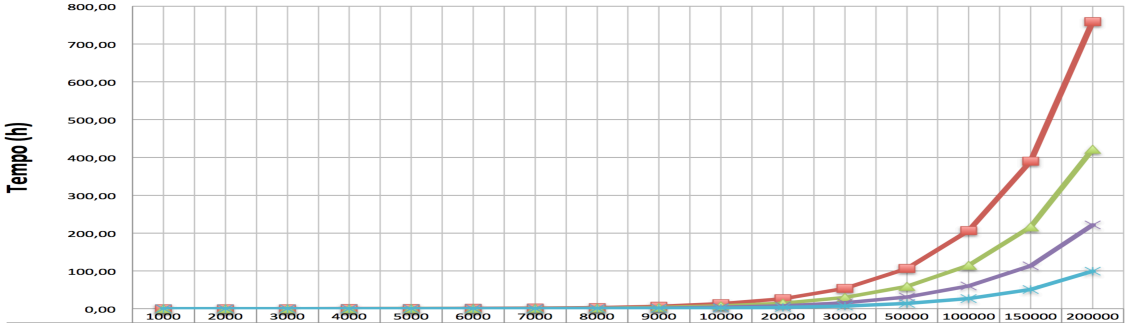
O Algoritmo BLS descrito acima é um algoritmo assintoticamente cúbico O(n^3) (dois laços encadeados com dois laços internos sequenciais para cálculo dos fatores r e s). Embora o BLS sequencial tenha complexidade polinomial, sua aplicação prática não revela a eficiência esperada para algoritmos polinomiais em função dos tamanhos típicos de curvas de luz e quantidade de estrelas monitoradas por alguns observatórios. Para o observatório Kepler (Kepler, 2014), por exemplo, os tamanhos típicos das curvas de luz são de 200.000 pontos/curva e uma quantidade da ordem 145.000 estrelas monitoradas.

A paralelização do BLS foi feita no passo 3 do pseudocódigo mostrado no Algoritmo Sequencial, utilizando o padrão de programação paralela *map-reduce* (McCool *et al*., 2012):

- utilizou-se como mapa a função $f(c,i) = (i, D_{min_i}, j)$, que recebe o valor c (como no Algoritmo 1, acima) e um índice i, devolvendo a tripla (i, D_{min_i}, j), onde j é índice que produziu o valor mínimo (D_{min_i}) para o índice i.
- Como redução, utilizou-se a função mínimo na segunda coordenada (D_{min_i}) de todas as triplas (i, D_{min_i}, j), produzindo o par (i,j) desejado.

A aplicação paralela do mapa necessita de um particionamento do conjunto de índices $i = \{1, 2, \ldots, n-1\}$. Porém, este particionamento não pode ser contínuo, pois as tarefas que forem alocadas para os primeiros índices trabalharão mais que aquelas que forem alocadas aos índices finais, devido ao passo 4 do Algoritmo Sequencial. Para balancear melhor as tarefas, utilizou-se a seguinte regra: as tarefas são numeradas de 1 a p; os índices de processamento da tarefa 1 são $\{1, n-1, (p+1), n-(p+1), (2p+1), n-(2p+1), \ldots\}$; para a tarefa 2 $\{2, n-2, (p+2), n-(p+2), \ldots\}$ e assim por diante, ou seja, se uma tarefa foi alocada para um índice muito baixo (laço do passo 4 longo), então ela irá executar um índice muito alto (laço do passo 4 curto).

As duas estratégias foram implementadas em Python, com a versão paralela utilizando map-reduce com *threads* e a estratégia de particionamento acima, sendo testadas com curvas de luz sintéticas e obtidas do observatório Kepler, utilizando um processador Intel i7 i7-5960X (8 cores, 16 threads). Para cada curva de luz, foram efetuadas 10 execuções, sendo calculadas as respectivas médias de tempo de execução e desvio-padrão. O gráfico mostrado na Figura 17 compara os resultados obtidos para 2, 4 e 8 *threads* e com números crescentes de pontos nas curvas de luz.



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	20000	30000	50000	100000	150000	200000
Sequencial	0,00	0,03	0,12	0,29	0,52	0,90	1,69	3,15	6,59	13,29	26,52	53,98	106,24	206,20	389,80	759,29
2 Threads	0,00	0,02	0,07	0,16	0,29	0,50	0,94	1,75	3,66	7,39	14,73	29,99	59,02	114,56	216,56	421,83
4 threads	0,00	0,01	0,03	0,08	0,15	0,26	0,49	0,92	1,92	3,88	7,73	15,74	30,97	60,12	113,64	221,37
8 threads	0,00	0,00	0,02	0,04	0,07	0,12	0,22	0,41	0,86	1,74	3,47	7,06	13,89	26,95	50,95	99,25

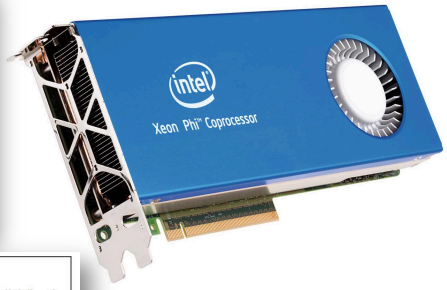
Figura 17. Comparações de tempos (em horas) entre as versões sequencial e paralela.

A diferença de desempenho começou a ocorrer para um número de pontos maior ou igual a 20000 pontos, mesmo com os custos de lançamentos de *threads*. No tamanho mais elevado de número de pontos, reduziu-se o tempo de 31.6 dias de processamento para 4.13 dias com 8 *threads*.

1.5 Programação Multicore em Python

Através do módulo pyMIC, a linguagem Python permite o carregamento e execução de núcleos dentro da aceleradora Intel Xeon Phi. O código abaixo mostra um código bastante simples (função nop), cuja versão compilada como biblioteca compartilhada (so) foi carregada e invocada numa Xeon Phi:

```
import pyMIC as mic

# acquire handle of offload target
dev = mic.devices[0]
offl_a = dev.associate(a)

# load library w/ kernel
dev.load_library("libnop.so")

# invoke kernel
dev.invoke_kernel("nop")
```




```
/* compile with:
 icc -mmic -fPIC -shared -olibnop.so nop.c
*/

#include <pymic_kernel.h>

PYMIC_KERNEL
void nop(int argc, uintptr_t argptr[],
         size_t sizes[]) {
    /* do nothing */
}
```

Figura 18: Núcleo em Xeon Phi e carga/invocação em Python.

Para GPU, pode-se usar o módulo pyCUDA, conforme exemplificado pelo código abaixo:

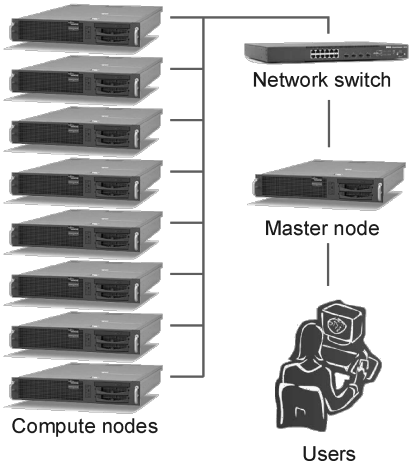
```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a = numpy.random.randn(4,4).astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a)
```


```
mod = cuda.SourceModule("""
    __global__ void twice(float *a)
    {
      int idx = threadIdx.x + threadIdx.y*4;
      a[idx] *= 2;
    }
    """)

func = mod.get_function("twice")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
```

Figura 19: Núcleo em CUDA e carga/invocação em Python.

1.6 Programação para clusters em Python

A linguagem Python disponibiliza uma interface bastante simples para acesso a MPI, através do módulo mpi4py. O código abaixo ilustra um exemplo de uso de dois processos em usando a mpi4py:




```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   data = {'a': 7, 'b': 3.14}
   comm.send(data, dest=1, tag=11)
elif rank == 1:
   data = comm.recv(source=0, tag=11)
```

Figura 20: Exemplo de código com mpi4py.

Neste exemplo, um dicionário identificado pela variável data possui as chaves a e b, com os valores 7 e 3.14 vinculados. Este dicionário é enviado pelo processo-mestre (rank=0) ao processo-escravo (rank=1).

Uma arquitetura bastante simples e interessante para testar os recursos de mpi4py consiste no uso de diversas placas de Raspberry PI, interligadas em rede, conforme mostrado abaixo:

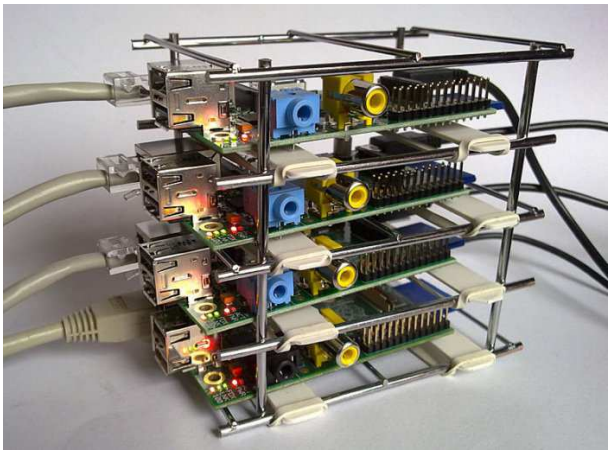


Figura 21: Um exemplo de cluster de placas Raspberry PI, para teste de mpi4py.

1.7 Programação para FPGA em Python

Geralmente, uma FPGA pode ser programada em uma linguagem de descrição de hardware chamada VHDL. O exemplo abaixo mostra um exemplo bastante simples de programa que pode ser mapeada em FPGA, via Python:

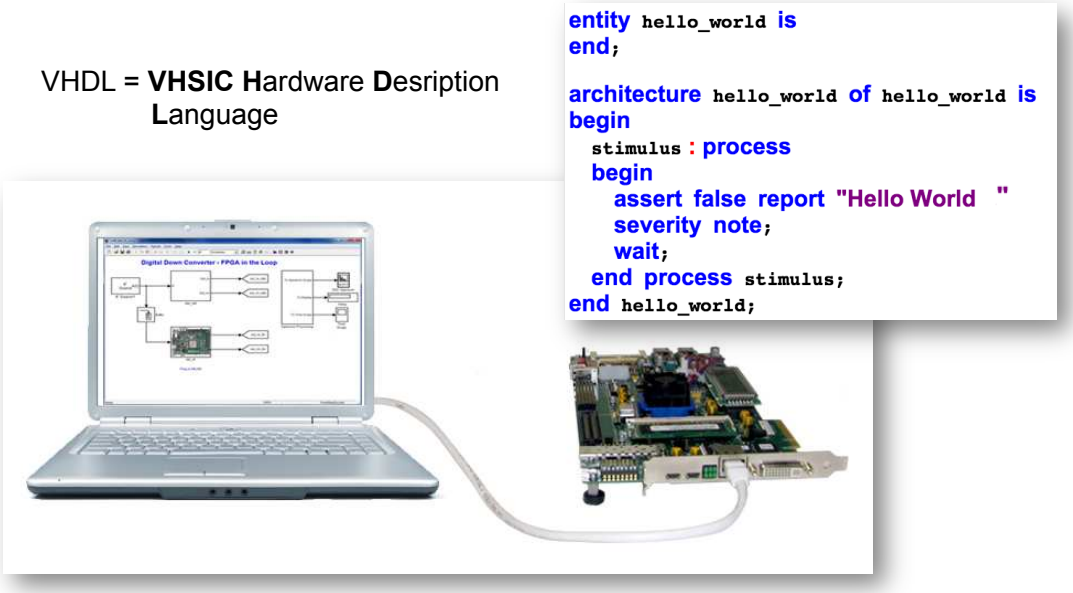


Figura 22: Um programa simples para FPGA, especificado em VHDL.

Com o módulo myhdl, programas VHDL podem ser especificados em mais alto nível e simulados antes da gravação na plava FPGA. Abaixo, tem-se um exemplo de um programa escrito em myhdl:

```
from myhdl import Signal, delay, always, now, Simulation

def HelloWorld():

    interval = delay(10)

    @always(interval)
    def sayHello():
        print "%s Hello World!" % now()

    return sayHello


inst = HelloWorld()
sim = Simulation(inst)
sim.run(30)
```

```
% python hello1.py
10 Hello World!
20 Hello World!
30 Hello World!
_SuspendSimulation: Simulated 30 timesteps
```

Figura 23: Um exemplo de temporizador especificado em myhdl.

Referências Bibliográficas

BARLAS, G. Multicore and GPU Programming: An Integrated Approach. New York: Morgan Kaufmann, 2014.

CHENG, J., GROSSMAN, M. Professional CUDA C Programming. New York: Wrox, 2014.

GORELICK, M., OZSVALD, I. High Performance Python: Practical Performant Programming for Humans. New York: O'Reilly, 2014.

HERLIHY, M., SHAVIT, N. The Art of Multiprocessor Programming. New York: Morgan Kaufmann, 2012.

JEFFERS, J., REINDERS, J. Intel Xeon Phi Coprocessor High-Performance Programming. New York: Morgan Kaufmann, 2013.

KEPLER. Kepler Public Light Curves. Disponível em: https://archive.stsci.edu/kepler/ publiclightcurves.html. Acesso em: 02/05/2015.

KOVÁCS, G., ZUCKER, S., MAZEH, T. (2002) A Box-Fitting Algorithm in Search for Periodic Transits. In: Astronomy and Astrophysics, 391, 369–377, 2002.

McCOOL, M., REINDERS, J., ROBISON, A. Structured Parallel Programming: Patterns for Efficient Computation. New York: Morgan Kaufmann, 2012.

PALACH, J. Parallel Programming with Python. New York: CreateSpace Publishing, 2014.

PERRYMAN, M. The Exoplanet Handbook. Cambridge: Cambrige University Press, 2014.

RAUBER, T., RÜNGER, G. Parallel Programming for Multicore and Cluster Systems. New York: Springer, 2014.

SMITH, K.W. Cython: A Guide for Python Programmers. New York: O'Reilly, 2015.

VANDERBAUWHEDE, W., BENKRID, K. High-Performance Computing using FPGA. New York: Springer, 2014.

Capítulo 2

Intel Modern Code: Programação Paralela e Vetorial AVX para o Processador Intel Xeon Phi Knights Landing[1]

Eduardo H. M. Cruz – UFRGS – ehmcruz@inf.ufrgs.br[2,3,8]
Arthur M. Krause – UFRGS – amkrause@inf.ufrgs.br[2,4,8]
Emmanuell D. Carreño – UFRGS – edcarreno@inf.ufrgs.br[5,8]
Matheus S. Serpa – UFRGS – msserpa@inf.ufrgs.br[6,8]
Philippe O. A. Navaux – UFRGS – navaux@inf.ufrgs.br[7,8]
Marco A. Z. Alves – UFPR – mazalves@inf.ufpr.br[9,10]
Igor J. F. Freitas – Intel Brasil – igor.freitas@intel.com[11,12]

Resumo

Tradicionalmente, o aumento de desempenho das aplicações se dava de forma transparente aos programadores devido ao aumento do paralelismo a nível de instruções e aumento de frequência dos processadores. Entretanto, este modelo não se sustenta mais. Para se ganhar desempenho nas arquiteturas modernas, são necessários conhecimentos sobre programação paralela e programação vetorial. Ambos paradigmas são tratados de forma lateral em cursos de computação, sendo que muitas vezes nem são abordados.

[1] Este trabalho está sendo desenvolvido em uma parceria com a Intel através do projeto Modern Code.

[2] Apresentadores: Eduardo H. M. Cruz e Arthur M. Krause

[3] http://lattes.cnpq.br/1342155623515902 – Telefone: +55 (51) 98220-0536

[4] http://lattes.cnpq.br/6221622726544557 – Telefone: +55 (51) 99752-1572

[5] http://lattes.cnpq.br/2746136604360215 – Telefone: +55 (51) 98308-7136

[6] http://lattes.cnpq.br/3041231438928613 – Telefone: +55 (51) 99900-1846

[7] http://lattes.cnpq.br/5554254760869075 – Telefone: +55 (51) 3308-9495

[8] Campus do Vale, Universidade Federal do Rio Grande do Sul (UFRGS), Instituto de Informática, Bloco IV, Lab 201-67, Postal Code 91501-970, Porto Alegre, RS, Brasil

[9] http://lattes.cnpq.br/7011183305034415 – Telefone: +55 (41) 3361-3684

[10] Rua Cel. Francisco Heráclito dos Santos, 100 – Centro Politécnico Jardim das Américas, Caixa Postal 19081, CEP 81531-980, Curitiba, PR, Brasil

[11] http://lattes.cnpq.br/2348884127321139 – Telefone: +55 (11) 3365-5634

[12] Av. Doutor Chucri Zaida, 940, 10º andar, Torre II, São Paulo, SP, Brasil

Neste contexto, este minicurso objetiva propiciar um maior entendimento sobre os paradigmas de programação paralela e vetorial, de forma que os participantes aprendam a otimizar adequadamente suas aplicações para arquiteturas modernas. Como plataforma de desenvolvimento, será utilizado o processador Intel Xeon Phi Knights Landing.

2.1. Intel Xeon Phi Knights Landing

O Intel Xeon Phi representa um grande evolução para arquiteturas *many-core*. Ele permite a execução de instruções x86, e a execução do mesmos códigos utilizados em arquiteturas x86 tradicionais. Por exemplo, uma aplicação escrita para o Xeon Phi pode ser facilmente paralelizada com OpenMP, como qualquer máquina x86 traditional [Jeffers and Reinders 2013]. Por outro lado, outras arquiteturas *many-core*, como a GPU, requerem complexas APIs de programação, como CUDA [Cook 2012, Sanders and Kandrot 2010] e OpenCL [Stone et al. 2010]. Devido a isto, o Intel Xeon Phi pode ser adotado mais facilmente que dispositivos baseados nestas outras arquiteturas.

O primeiro processador, utilizando a arquitetura *Knights Corner* [Chrysos 2012], compreende um co-processador com vários núcleos e uma memória dedicada, e é conectado à máquina através da interface PCI express. O co-processador executa o sistema operacional Linux, separadamente do sistema operacional do hospedeiro. Programas necessitavam de recompilação para executar no Knights Corner. Pode haver vários co-processadores Xeon Phi instalados na mesma máquina. Uma aplicação pode ser executada inteiramente no Xeon Phi, or ser executada parcialmente no Xeon Phi e parcialmente no processador hospedeiro.

A arquitetura Intel Knights Landing (KNL) [Sodani et al. 2016], introduzida no ano de 2016, representa uma evolução substancial do Xen Phi. Diferente do antecessor, a KNL consiste de um processador hospedeiro capaz de executar sistemas operacionais x86 nativos. Além disso, os programas podem ser executados na KNL sem necessidade

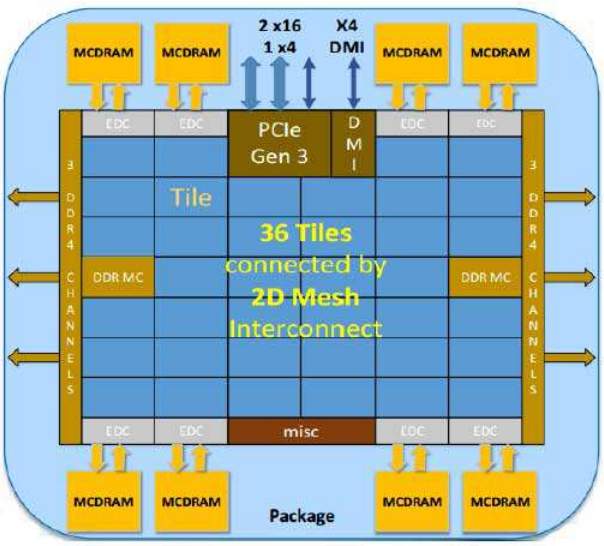


Figura 2.1: Arquitetura Knights Landing [Sodani 2015].

de recompilação. Muitas outras melhorias arquiteturais foram implantadas, tais como:

- Pipeline com execução fora-de-ordem.
- Maior banda de memória.
- Rede de interconexão provê maior banda e menor latência (MESH vs Ring Bi-direcional).
- Melhor protocolo de coerência de cache (MESIF vs MESI).

A arquitetura *Knights Landing* está ilustrada na Figura 2.1. É composto por núcleos, memórias cache, controladores de memória, um diretório de *tags* distribuído e uma rede *mesh*. A arquitetura é organizada em *tiles* e tem um diretório de *tags* distribuído, e uma interconexão *mesh*. Cada *tile* contém dois núcleos, com cache L1 privadas, uma cache L2 compartilhada e um diretório de *tags*. A arquitetura, além de controladores de memória para acessar a memória externa DDR4, inclui uma memória MCDRAM, que pode funcionar como uma memória cache (modo cache) ou como uma memória separada (modo flat), porém dentro do mesmo espaço de endereçamento. A memória cache é mantida coerente pelo protocolo MESIF, e os núcleos implementam uma pipeline com execução fora-de-ordem capaz de executar 4 *threads* utilizando a tecnologia de *multithreading* simultâneo (SMT).

Quando um acesso à cache L2 resulta numa falta na cache, um diretório de *tag* correspondente ao endereço da falta é acessado. Se a mesma linha de cache está presente em outra cache L2, os dados são redirecionados da outra cache. Caso o dado não esteja em alguma cache, o dado é buscado na memória DDR4 externa ou da MCDRAM interna, dependendo da configuração do processador. Os endereços de memória são distribuídos de maneira uniforme entre os diretórios de *tag* e controladores de memória de forma a distribuir o tráfego. Essa distribuição também depende do modo que está configurado o processador.

Uma das principais vantagens da arquitetura é sua unidade de processamento vetorial, que permite que a mesma operação seja realizada em vários operandos simultaneamente. O desempenho pode ser aumentado de forma acentuada se a aplicação usa instruções de processamento vetorial (AVX) de forma adequada. Até 16 operações de ponto flutuante de precisão simples, ou 8 operações de precisão dupla, podem ser realizadas na mesma instrução vetorial.

Nas seções seguintes, é explicado como programar o Xeon Phi.

2.2. Resumo de Programação em OpenMP

Open Multi-Processing (OpenMP) consiste em um padrão de programação paralela para arquiteturas de memória compartilhada [Chapman et al. 2008]. OpenMP utiliza a diretiva *#pragma*, definida no padrão da linguagem C/C++. Nesta seção, detalharemos todas as construções básicas definidas pelo OpenMP.

2.2.1. Inclusão das funções da biblioteca

O padrão OpenMP, além das diretivas interpretadas diretamente pelo compilador, define uma série de funções através da biblioteca `omp.h`, que pode ser incluída através do seguinte código:

```
1 #ifdef _OPENMP
2 #include <omp.h>
3 #endif
```

A macro `_OPENMP` é utilizada para identificar se o compilador suporta o OpenMP. Desta forma, a biblioteca `omp.h` só é incluída caso haja suporte. As principais funções da biblioteca são:

int omp_get_num_threads() Retorna o número de *threads* ativas naquele momento da execução.

int omp_get_thread_num() Retorna o identificador da *thread*, também conhecido como *id*.

2.2.2. Iniciando um bloco de execução paralela

Para iniciar um bloco de execução paralela, o seguinte código deve ser utilizado:

```
1 #pragma omp parallel
2 {
3 }
```

O ambiente OpenMP irá alocar um determinado número de *threads*, e todas elas executarão as linhas de comando contidas entre as chaves . O número de *threads* varia, sendo responsabilidade do programador garantir que o resultado esperado seja atingido independente do número de *threads*.

A diretiva permite também compartilhar ou replicar variáveis conforme o código a seguir:

```
1 #pragma omp parallel shared(variables) private(variables)
2 {
3 }
```

A construção `shared` define que as variáveis são compartilhadas. A construção `private` define que as variáveis são privadas e que portanto devem ser replicadas na memória. Por padrão, as variáveis são do tipo `shared`.

Para exemplificar o que foi ensinado até agora, considere o seguinte código:

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main (int argc, char **argv)
5 {
6       int myid, nthreads;
7       #pragma omp parallel private(myid) shared(nthreads)
8       {
9             myid = omp_get_thread_num();
10            printf("myid: %i\n", myid);
11            if (myid == 0)
12                  nthreads = omp_get_num_threads();
13      }
14      printf("Havia %i threads na regiÃ£o paralela\n",
           nthreads);
15      return 0;
16 }
```

Nomeamos o arquivo como `hello.c`, e para compilar com o compilador Intel utilizamos a seguinte linha de comando:

```
1 icc -o hello hello.c -fopenmp
```

Para solicitar o número de *threads* à biblioteca, pode ser definida a variável de ambiente `OMP_NUM_THREADS` como exemplifica o código a seguir:

```
1 export OMP_NUM_THREADS=4
```

Neste exemplo, é solicitado que o ambiente de execução do OpenMP crie 4 *threads*. Ao executar este código, a seguinte saída é retornada no terminal:

```
1 ./hello
2 myid: 0
3 myid: 3
4 myid: 2
5 myid: 1
6 Havia 4 threads na região paralela
```

2.2.3. Sincronizando as threads

No exemplo anterior, utilizamos um comando condicional de forma explícita para selecionar que apenas a *thread* 0 atualizasse o conteúdo da variável *nthreads*. O OpenMP prevê a seguinte construção para realizar isto:

```
1 #pragma omp master
2 {
3 }
```

Tudo que estiver entre as chaves será executado apenas pela *thread* 0. Caso não seja necessário que a *thread* 0 que execute, mas apenas uma das *threads*, pode ser utilizada a seguinte construção:

```
1 #pragma omp single
2 {
3 }
```

Outra diferença entre o `single` e o `master` é que o `single` adiciona uma barreira implícita após seu término. Isto é, apesar de apenas uma *thread* executar o bloco `single`, todas as outras *threads* ficam aguardando a execução finalizar para prosseguir. Caso não seja necessária a barreira, deve-se adicionar a diretiva `nowait` ao comando:

```
1 #pragma omp single nowait
2 {
3 }
```

Barreiras podem ser incluídas de forma explícita com a seguinte linha de comando:

```
1 #pragma omp barrier
```

Como OpenMP foca em ambientes de memória compartilhada, muitas vezes é necessário acessar e atualizar variáveis compartilhadas. Por exemplo, considere um algoritmo que soma todos os elementos de um vetor. O seguinte código foi elaborado contendo as diretivas já apresentadas:

```
1  int sum(int *v, int n)
2  {
3        int i, sum, nthreads, id;
4        sum = 0;
5        #pragma omp parallel private(i, id)
6        {
7              #pragma omp single
8              nthreads = omp_get_num_threads();
9              id = omp_get_thread_num();
10             for (i=id; i<n; i+=nthreads)
11                   sum += v[i];
12       }
13       return sum;
14 }
```

Entretanto, este código não funciona como esperado devido às *condições de corrida*. Não há uma sincronização adequada entre as *threads*, fazendo que as sucessivas operações de leitura e escrita se sobreponham. A Tabela 2.1 demonstra um exemplo de tais sobreposições, considerando que há 2 *threads* e que todos os elementos do vetor são iguais a 1. Entre os tempos 1 e 4, tudo ocorre como esperado. Porém, nos tempos 5 e 6, as operações de ambas as *threads* se sobrepõe, fazendo literalmente que uma das operações de soma seja perdida, causando um resultado errado.

O OpenMP provê mecanismos para controlar acessos a tais regiões críticas, que acessam dados compartilhados. A primeira diretiva é a `critical`:

```
1 #pragma omp critical
2 {
3 }
```

Tabela 2.1: Exemplo de execução do código de soma dos elementos de um vetor com o problema das condições de corrida.

Tempo	Thread 0	Thread 1
1	Ler sum=0	
2	Escrever sum=1	
3		Ler sum=1
4		Escrever sum=2
5	Ler sum=2	Ler sum=2
6	Escrever sum=3	Escrever sum=3

Todo conteúdo colocado entre as chaves {} ocorre de forma atômica. Se alguma *thread* tenta entrar em uma região crítica enquanto outra *thread* já se encontra na região crítica, esta *thread* fica bloqueada aguardando a outra sair da região crítica. Caso haja diferentes recursos compartilhados, para que não haja interferência nas regiões críticas, é possível estabelecer um identificador de regiões críticas:

```
1 #pragma omp critical(id)
2 {
3 }
```

O principal problema das regiões críticas é seu alto custo, pois faz com que as *threads* bloqueiem. Uma alternativa, para operações simples de lógica e aritmética, é o uso do `atomic`:

```
1 #pragma omp atomic
2 var1 += var2;
```

Neste exemplo, a operação de soma ocorre atomicamente. O `atomic` faz uso de suporte de *hardware* para realizar a operação de forma atômica, sendo mais eficiente que o `critical`.

O `critical` e `atomic` controlam o acesso à regiões críticas. Uma outra forma de sincronizar as *threads* é distribuir a carga de trabalho entre as as mesmas. Um dos principais focos do OpenMP é a paralelização de laços que não possuem dependências entre suas iterações. Anteriormente, isto foi feito de forma manual utilizando-se o número de *threads* e seus idenficadores. Isso pode ser feito de forma automática:

```
1 #pragma omp for
2 for (i=0; i<n; i++)
```

Em nosso exemplo de soma dos elementos de um vetor, o OpenMP provê a possibilidade de redução de um laço através de uma operação:

```
1 int sum(int *v, int n)
2 {
3       int i, sum;
4       sum = 0;
5       #pragma omp parallel for reduction(+:sum)
6       for (i=0; i<n; i++)
7             sum += v[i];
8       return sum;
9 }
```

2.3. Programação Paralela Vetorial (Intel AVX)

O paralelismo com execução vetorial se dá de forma diferente do explicado anteriormente. Enquanto na execução normal cada instrução opera em apenas um dado, na instrução vetorial a mesma operação é executada em vários dados de forma independente [Satish et al. 2012]. Considere o seguinte laço, que soma dois vetores e põe o resultado em um terceiro laço:

```
1 for (i=0; i<n i++)
2       a[i] = b[i] + c[i]
```

Como pode-se perceber, as iterações do laço são independentes. Supondo que haja instruções para ler e escrever 8 operandos na memória, e somar 8 operandos, pode-se visualizar o mesmo laço sendo operado vetorialmente da seguinte maneira (supondo n múltiplos de 8):

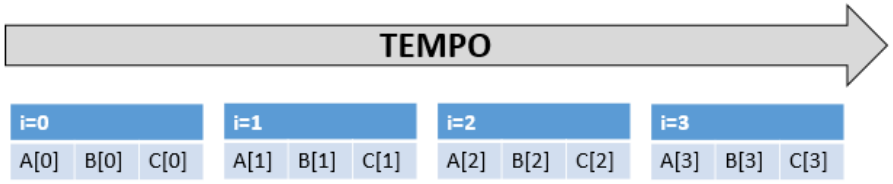
```
1 for (i=0; i<n i+=8)
2       a[i:8] = b[i:8] + c[i:8]
```

Em cada iteração do laço, carregam-se 8 operandos a partir da posição i dos vetores b e c, soma-se cada par $(b[i], c[i])$ de forma independente, e depois o bloco de 8 operandos é escrito no vetor a a partir da posição i. Este comportamento pode ser visualizado na Figura 2.2 (com 4 elementos por iteração para fins de visualização).

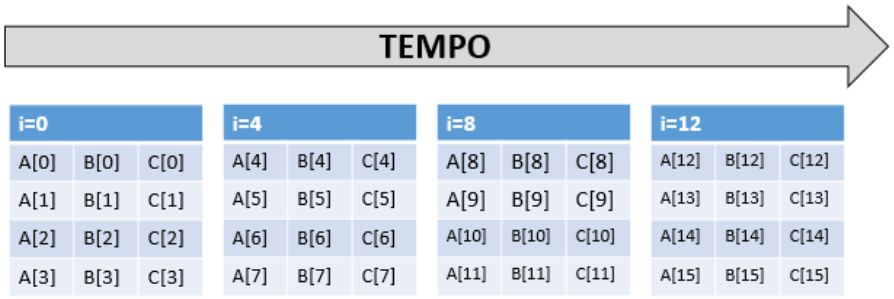
As instruções vetoriais já estão presentes há muitos anos nos processadores x86. A Figura 2.3 contém a evolução das instruções vetoriais x86. A cada etapa da evolução, aumenta-se a quantidade de dados processados por instrução, bem como o número de instruções vetoriais disponíveis. Neste documento, o foco são as instruções AVX. É importante ressaltar que, para maior eficiência, **os endereços acessados no laço em iterações sucessivas devem ser consecutivos**.

A fim de se entender como funciona, considere um código que realiza a soma da multiplicação dos elementos de um vetor:

```
1 double vsum (double *a, double *b, int n)
2 {
3       int i;
4       double r = 0;
5       for (i=0; i<n; i++)
6             r += a[i] * b[i];
7       return r;
8 }
```



(a) Execução normal.



(b) Execução vetorial com 4 elementos processados simultaneamente.

Figura 2.2: Diferença entre a execução normal de um laço e sua execução vetorial.

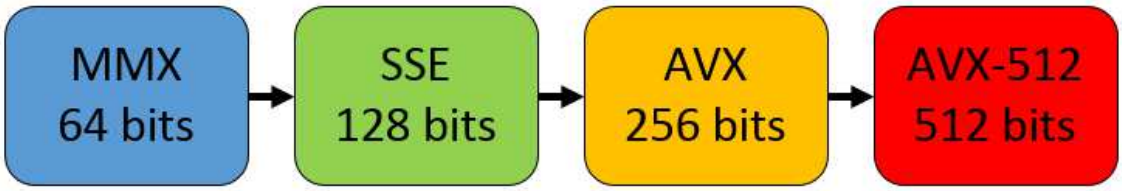


Figura 2.3: Evolução das instruções vetoriais na arquitetura x86.

Nas subseções a seguir, veremos algumas possibilidades de implementação.

2.3.1. Intrínsecas

A primeira implementação AVX a ser demonstrada é a manual, utilizando as intrínsecas do compilador da Intel. Para fins de simplificação, consideramos que o número de elementos nos vetores seja múltiplo de 8:

```
 1 #include <immintrin.h>
 2
 3 double vsum(double *a, double *b, int n)
 4 {
 5       int i;
 6       double r, partial[8];
 7       __m512d ac, va, vb, mul;
 8       ac = _mm512_set_pd(0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0);
 9       for (i=0; i<n; i+=8) {
10             va = _mm512_load_pd(&a[i]);
11             vb = _mm512_load_pd(&b[i]);
12             mul = _mm512_mul_pd(va, vb);
13             ac = _mm512_add_pd(ac, mul);
14       }
15       _mm512_storeu_pd(partial, ac);
16       r = partial[0] + partial[1] + partial[2] + partial
            [3] + partial[4] + partial[5] + partial[6] +
            partial[7];
17       return r;
18 }
```

A biblioteca `immintrin.h` inclui os tipos de dados e funções necessários. A descrição dos tipos e funções utilizados:

__m512d 8 pontos flutuantes to tipo `double` empacotados em uma variável de 512 bits.

__m512d _mm512_set_pd (double e3, double e2, double e1, double e0) Inicializa uma variável de 512 bits com 8 `doubles`.

__m512d _mm512_load_pd (double const * mem_addr) Carrega 64 bytes de dados a partir do endereço `mem_addr` e coloca na variável do tipo `__m512d`.

__m512d _mm512_mul_pd (__m512d a, __m512d b) Multiplica de forma independente os 8 pares de ponto flutuante contidos nas variáveis `a` e `b`.

__m512d _mm512_add_pd (__m512d a, __m512d b) Soma de forma independente os 8 pares de ponto flutuante contidos nas variáveis `a` e `b`.

void _mm512_storeu_pd (double * mem_addr, __m512d a) Salva os 64 bytes de dados da variável `a` a partir do endereço `mem_addr`.

A ideia desta implementação é, em cada iteração do laço, explicitamente processar 8 elementos do vetor. Caso não tivéssemos considerado o número de elementos no vetor (n) um múltiplo de 8, seria necessário processar o vetor vetorialmente até um valor k, tal que $k < n$ e k é múltiplo de 8. Para os elementos x, tal que $k < x < n$, deveria ser processado da forma tradicional, apenas 1 elemento por iteração.

Entretanto, a necessidade de conhecer as diretivas de baixo nível e ter que explicitamente codificar o código AVX desencoraja este tipo de programação. Além disso, caso seja alterado o tipo de dados de `double` para `float`, ou se o conjunto de instruções evoluir e suportar mais operandos por variáveis SIMD, é necessário recodificar o código. Nenhuma destas situações são desejáveis.

2.3.2. PRAGMA SIMD

Para que a situação anterior não seja necessária, foi introduzido o seguinte `pragma`:

```
1 #pragma simd
```

A lógica deste comando é semelhante ao `pragma omp for`, com a diferença que agora o paralelismo se dá vetorialmente. Ele também aceita a cláusula `reduction`. É responsabilidade do programador assegurar que as iterações são independentes.

O código anterior usando este `pragma` fica da seguinte maneira:

```
1 double vsum (double *a, double *b, int n)
2 {
3       int i;
4       double r = 0;
5       #pragma simd reduction(+:r)
6       for (i=0; i<n; i++) {
7             r += a[i] * b[i];
8       }
9       return r;
10 }
```

O `pragma` automaticamente considera o caso de *n* não ser múltiplo de 4.

2.3.3. PRAGMA SIMD com alinhamento de memória

Um dos principais pontos sobre vetorização é o alinhamento de memória. Alinhar memória significa fazer com que um determinado endereço de memória seja múltiplo de um determinado valor. Com o alinhamento correto, o *hardware* pode otimizar o acesso à memória [Lee et al. 2010]. Por exemplo, uma variável alinhada ao tamanho do seu tipo (exemplo um `double` alinhado em 8 bytes) sempre poderá ser armazenado na mesma linha de cache, jamais acarretando mais que uma falta por cache.

O compilador provê então diretivas para alinhamento de endereço. Por exemplo:

```
1 int partial[1024] __attribute__ ((aligned (64)));
```

Este código fará que o endereço inicial do vetor `partial` seja múltiplo de 64. Para alocação dinâmica, o compilador da Intel provê a seguinte função:

```
1 void* _mm_malloc (size_t size, size_t align )
```

O primeiro parâmetro é o tamanho em bytes, e o segundo parâmetro é o alinhamento, que deve ser uma potência de 2. Para desalocar a memória, deve-se utilizar a seguinte função:

```
1 void _mm_free (void *p)
```

Para instruções vetoriais, é recomendado alinhar os vetores em 64 bytes. Com os endereços de memória alinhados, um novo `pragma` pode ser utilizado para indicar ao compilador que os endereços de memória acessados estão alinhados:

```
1 #pragma vector aligned
```

O código então que soluciona o problema aqui tratado pode ser escrito da seguinte forma:

```
1 double vsum (double *a, double *b, int n)
2 {
3       int i;
4       double r = 0;
5       #pragma vector aligned
6       #pragma simd reduction(+:r)
7       for (i=0; i<n; i++)
8             r += a[i] * b[i];
9       return r;
10 }
```

2.3.4. Estudo de caso: multiplicação de matrizes

Como estudo de caso, será utilizado um algoritmo de multiplicação de matrizes, que é um dos exemplos mais comuns de aplicação de paralelismo. O código base de multiplicação de matrizes é o seguinte:

```
1 void matrix_mult (double *first, double *second, double *
    multiply, int first_rows, int first_cols, int
    second_cols)
2 {
3       int i, j, k;
4       double sum;
5       for (i=0; i<first_rows; i++) {
6             for (j=0; j<second_cols; j++) {
7                   sum = 0;
8                   for (k=0; k<first_cols; k++)
9                         sum += first[i*first_cols+k] *
                              second[k*second_cols+j];
10                  multiply[i*second_cols+j] = sum;
11            }
12      }
13 }
```

A primeira etapa é paralelizar com instruções vetoriais:

```
1 void matrix_mult (double *first, double *second, double *
    multiply, int first_rows, int first_cols, int
    second_cols)
2 {
3       int i, j, k;
4       double sum;
5       for (i=0; i<first_rows; i++) {
6             for (j=0; j<second_cols; j++) {
7                   sum = 0;
8                   #pragma simd reduction(+:sum)
9                   for (k=0; k<first_cols; k++)
10                        sum += first[i*first_cols+k] *
                             second[k*second_cols+j];
11                  multiply[i*second_cols+j] = sum;
12            }
13      }
14 }
```

Entretanto, esta implementação tem um problema, os endereços acessados dentro do laço pela expressão `second[k*second_cols+j]` não são consecutivos em sucessivas iterações, diminuindo o desempenho. Para solucionar isto, uma solução é modificar o algoritmo, de forma a inverter os laços das variáveis `j` e `k`:

```
1 void matrix_mult (double *first, double *second, double *
    multiply, int first_rows, int first_cols, int
    second_cols)
2 {
3       int i, j, k;
4       for (i=0; i<first_rows; i++) {
5             for (j=0; j<second_cols; j++)
6                   multiply[i*second_cols+j] = 0.0;
7       }
8       for (i=0; i<first_rows; i++) {
9             for (k=0; k<first_cols; k++) {
10                  #pragma simd
11                  for (j=0; j<second_cols; j++)
12                        multiply[i*second_cols+j] +=
                             first[i*first_cols+k] * second
                             [k*second_cols+j];
13            }
14      }
15 }
```

Agora, todos os endereços acessados no laço vetorizado são consecutivos ou, no caso de `first[i*first_cols+k]`, não se alteram. Com isto, o laço interno é paralelizado com instruções vetoriais.

O próximo passo é o alinhamento de memória. Além, para poder incluir as instruções de alinhamento de memória, é necessário não apenas alocar os dados

com a função _mm_malloc, mas também que o valor das variáveis first_cols e second_cols seja múltiplo de 4. Isto deve ocorrer para que os endereços iniciais dos vetores acessados dentro do laço, multiply[i*second_cols+j] e second[k*second_cols+j], sejam múltiplos de 4. É importante mencionar que, caso o tipo de dados fosse float, deveria ser múltiplo de 8.

Além de paralelizar o laço interno com instruções vetoriais, deve-se paralelizar o laço externo com OpenMP. Dessa forma, é possível aproveitar tanto o paralelismo a nível de *threads* quanto vetorização. A solução final do exercício é:

```
1 void matrix_mult (double *first, double *second, double *
    multiply, int first_rows, int first_cols, int
    second_cols)
2 {
3       int i, j, k;
4       for (i=0; i<first_rows; i++) {
5             for (j=0; j<second_cols; j++)
6                   multiply[i*second_cols+j] = 0.0;
7       }
8       #pragma omp parallel for private(i, j, k)
9       for (i=0; i<first_rows; i++) {
10            for (k=0; k<first_cols; k++) {
11                  #pragma vector aligned
12                  #pragma simd
13                  for (j=0; j<second_cols; j++)
14                        multiply[i*second_cols+j] +=
                             first[i*first_cols+k] * second
                             [k*second_cols+j];
15            }
16      }
17 }
```

2.4. Conclusão

Neste curso, foram apresentadas novas técnicas de programação a serem aplicadas nas novas arquiteturas. Tais técnicas são primordiais para extrair desempenho dos novos sistemas. As técnicas apresentadas permitem fazer uso dos múltiplos núcleos de arquiteturas *multicore* e *manycore*. Permitem também que cada núcleo, individualmente, possa aproveitar o paralelismo vetorial.

Referências

[Chapman et al. 2008] Chapman, B., Jost, G., and Van Der Pas, R. (2008). *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press.

[Chrysos 2012] Chrysos, G. (2012). Intel Xeon Phi X100 Family Coprocessor - the Architecture.

[Cook 2012] Cook, S. (2012). *CUDA programming: a developer's guide to parallel*

computing with GPUs. Newnes.

[Jeffers and Reinders 2013] Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi coprocessor high-performance programming*. Newnes.

[Lee et al. 2010] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. (2010). Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38:451–460.

[Sanders and Kandrot 2010] Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[Satish et al. 2012] Satish, N., Kim, C., Chhugani, J., Saito, H., Krishnaiyer, R., Smelyanskiy, M., Girkar, M., and Dubey, P. (2012). Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ACM SIGARCH Computer Architecture News*, volume 40, pages 440–451. IEEE Computer Society.

[Sodani 2015] Sodani, A. (2015). Knights landing: 2nd generation intel xeon phi processor. In *Hot Chips: A Symposium on High Performance Chips*.

[Sodani et al. 2016] Sodani, A., Gramunt, R., Corbal, J., Kim, H. S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y. C. (2016). Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2).

[Stone et al. 2010] Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73.

Capítulo 3

Introdução à Programação Paralela com OpenMP: Além das Diretivas de Compilação

Rogério A. Gonçalves[1,3], João M. de Queiroz Filho[1,4] e Alfredo Goldman[2,5]

Abstract

Compilation directives have been widely used for adapting legacy code applications to the available resources on platforms that evolve and become increasingly heterogeneous. `OpenMP` *has followed this evolution and has provided directives for these new application contexts. This text introduces* `OpenMP` *showing the main compilation directives that cover the parallel regions, loops, sections, tasks, and device accelerators. We want to show the structure of the code generated by expansion of the directives that are formed by its constructors and their clauses. The goal is to introduce* `OpenMP` *presenting a different view of commons tutorials, showing* `OpenMP` *from the generated code point of view, the code with the directives expansion and their relations with parallel programming concepts.*

Resumo

Diretivas de compilação tem sido amplamente utilizadas para a adaptação de aplicações de código legado aos recursos disponíveis em plataformas que evoluem e tornam-se cada vez mais heterogêneas. O `OpenMP` *tem acompanhado essa evolução fornecendo diretivas para esses novos contextos. Este material apresenta uma introdução ao* `OpenMP` *mostrando as principais diretivas de compilação para a cobertura de regiões paralelas, laços, seções, tasks e dispositivos aceleradores. Queremos mostrar a estrutura do código gerado pela expansão das diretivas, formadas por seus construtores e suas cláusulas. O objetivo é apresentar o* `OpenMP` *dando uma visão diferente dos tutoriais convencionais, mostrando o* `OpenMP` *do ponto de vista do código gerado com a expansão das diretivas e suas relações com conceitos de programação paralela.*

[1]Universidade Tecnológica Federal do Paraná (UTFPR) – Departamento de Computação (DACOM)
[2]Universidade de São Paulo (USP) – Instituto de Matemática e Estatística (IME)
[3]rogerioag@utfpr.edu.br
[4]joaomfilho1995@gmail.com
[5]gold@ime.usp.br

3.1. Introdução

O `OpenMP` [Dagum and Menon 1998] [OpenMP-ARB 2015] [OpenMP Site 2017] é um padrão bem conhecido e amplamente utilizado no desenvolvimento de aplicações paralelas para plataformas *multicore* e *manycores*. E desde a versão 4.0 da sua especificação [OpenMP-ARB 2013] define o suporte a dispositivos aceleradores.

O uso de *diretivas de compilação* é uma das abordagens para paralelização de código que tem se destacado no contexto de Computação Paralela, pois anotar código é usualmente mais fácil do que reescrevê-lo.

As diretivas de compilação são como anotações que fornecem dicas sobre o código original e guiam o compilador no processo de paralelização das regiões anotadas. Estas diretivas são comumente implementadas usando-se as diretivas de pré-processamento `#pragma`, em `C/C++`, e sentinelas `!$`, no `Fortran`.

Durante o pré-processamento e a compilação, se o código anotado é analisado por um compilador com suporte ao `OpenMP` as diretivas tem seus construtores substituídos por um formato de código específico. O código gerado dessa expansão das diretivas é composto por chamadas às funções do *runtime* do `OpenMP` [Dagum and Menon 1998] [OpenMP-ARB 2015] [OpenMP Site 2017].

O `OpenMP` trabalha em sistemas com memória compartilhada e implementa o modelo `fork-join`, no qual múltiplas *threads* são criadas em regiões paralelas e executam tarefas definidas implicitamente ou explicitamente usando-se as diretivas do `OpenMP` [OpenMP-ARB 2011] [OpenMP-ARB 2013] [OpenMP-ARB 2015]. A Figura 3.1 apresenta a ideia do modelo `fork-join` implementado com regiões paralelas.

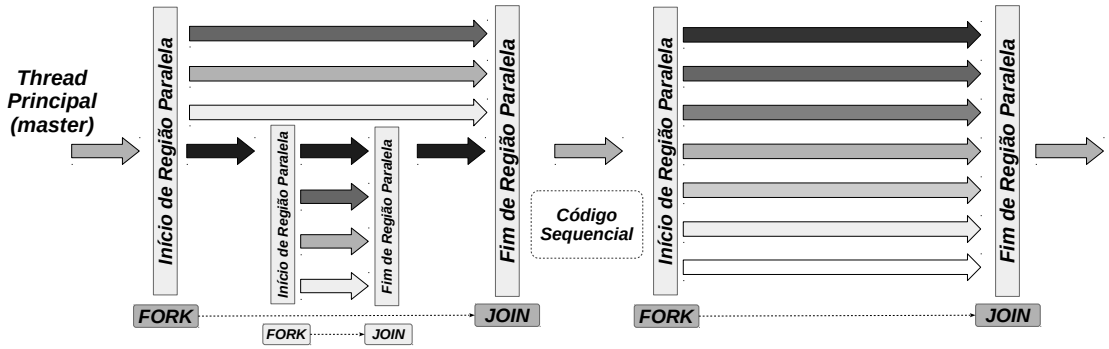


Figura 3.1. Modelo `Fork-join` baseado em regiões paralelas

A partir dos blocos anotados como regiões paralelas, o *runtime* cria implicitamente um time de *threads* que executarão o código da região paralela. A *thread* principal fará parte do time de *threads* e no final da região paralela, quando as outras *threads* forem destruídas, ela seguirá em execução.

Além da criação implícita de *threads*, é possível atribuir a alguma dessas *threads* a execução de tarefas explícitas definidas pelo programador usando a diretiva *task*.

O `OpenMP` traz aos usuários a possibilidade de aplicar conceitos de paralelismo em alto nível por meio das diretivas de compilação que permitem anotações no código.

Além disso, em grande parte os tutoriais apresentam a ideia do uso de diretivas de compilação para gerar código para aplicações paralelas como algo extremamente trivial.

Utilizar `OpenMP` não é apenas colocar as diretivas no código e esperar que magicamente o *runtime* paralelize o código, é necessário que o código seja paralelizável e sabermos quais transformações serão aplicadas ao código para gerar a versão paralela. Antes de tudo é importante conhecer os conceitos sobre paralelismo, concorrência, sincronização e sobre como o código irá ser executado [Gonçalves et al. 2016].

Grande parte das linguagens de programação apresentam mecanismos para a criação de *threads* (`spawn`) e para a sincronização do trabalho entre as tarefas (`join`, `sync`) em comandos nativos da linguagem ou por extensões, como as diretivas de compilação fornecidas pelo `OpenMP`. Alguns desses conceitos comuns são apresentados ou implementados de maneiras diferentes, mas a ideia e o significado permanecem os mesmo.

As implementações de bibliotecas ou extensão para paralelismo podem ser de mais baixo nível ou de mais alto nível. A biblioteca `pthread` [Nichols et al. 1996] disponível em distribuições do GNU/Linux, fornece ao programador funções para a criação de *threads* (`pthread_create(...)`) e para a sincronização do trabalho compartilhado entre as *threads* (`pthread_join()`).

O `Cilk` [Blumofe et al. 1995] fornece também funções com as atribuições de criar *threads* (`cilk_spawn`) e de sincronização (`cilk_sync`), e também tem suporte à paralelização de laços com `cilk_for`.

A abordagem proposta pelo `OpenMP` é ser de mais alto nível, que o código seja anotado e não reescrito. Porém em muitos casos não se trata de somente uma forma simples de anotar o código, é necessário programar via diretivas de compilação, o que torna-se complexo.

A motivação inicial desse estudo foi a necessidade de interceptar código de aplicações `OpenMP` para fazer *offloading* de código para aceleradores. Nosso estudo foi baseado nas diretivas de compilação da biblioteca `libgomp` do `GCC` [GNU Libgomp 2016]. Outras aplicações como bibliotecas para a construção de *traces* e *logs* são possíveis via interceptação de chamadas de funções *(hooking)* das aplicações. O código que trata as chamadas de funções, mensagens ou intercepta eventos, fazendo uma espécie de *proxy* é chamado de *hook* (gancho).

O restante do texto está organizado da seguinte forma. Na Seção 3.2 apresentamos o padrão `OpenMP` e algumas de suas implementações. Algumas ds principais diretivas de compilação e o estudo sobre a expansão de código dessas diretivas está na Seção 3.3. A Seção 3.4 trata sobre possíveis aplicações e criação de uma biblioteca de interceptação de código. Por fim serão apresentadas as considerações finais na Seção 3.5.

3.2. Implementações do `OpenMP`

O `OpenMP` [OpenMP-ARB 2011] [OpenMP-ARB 2013] [OpenMP-ARB 2015] implementa o modelo `fork-join`, no qual a *master thread* executa sequencialmente até encontrar uma região paralela. Nesta região executa uma operação similar a um `fork` e cria um time de *threads* que irão executar na região paralela (regiões paralelas aninhadas são

permitidas). Quando todas as *threads* alcançam a barreira no final da região paralela, o time de *threads* é destruído e a *master thread* continua sozinha a execução até encontrar uma nova região paralela.

O padrão `OpenMP` é suportado por praticamente todos os compiladores atuais. Compiladores como `GCC` [GCC 2015] [GNU Libgomp 2015a], Intel `icc` [Intel 2016b] e `LLVM clang` [Lattner and Adve 2004] [LLVM OpenMP 2015] tem implementações do `OpenMP` [OpenMP 2017]. Entre as implementações que são bem conhecidas atualmente estão a `GNU GCC libgomp` [GNU Libgomp 2016] e a `Intel OpenMP* Runtime Library (libomp)` [Intel 2016a] que trabalham com os compiladores `GCC` e `clang`.

A especificação do `OpenMP` foi expandida para dar suporte a *offloading* de código para dispositivos aceleradores, o que é um tópico de grande importância considerando que as plataformas tornam-se cada vez mais heterogêneas. A biblioteca `libgomp` [GNU Libgomp 2015a] [GNU Libgomp 2015b] [GNU Libgomp 2016] teve seu nome trocado recentemente de `GNU OpenMP Runtime Library` para `GNU Offloading and Multi Processing Runtime Library` sendo capaz de fazer offloading de código usando o padrão `OpenACC` [OpenACC 2015] [OpenACC 2017].

3.3. Expansão das Diretivas de Compilação

As *diretivas de compilação* são formadas por construtores e cláusulas que são substituídos por uma versão de código expandido durante a fase de pré-processamento. Nesta seção serão apresentados algumas das diretivas e os respectivos formatos de código pós expansão. O formato de código é estruturado e é composto de chamadas às funções do *runtime* do `OpenMP`.

Verificamos o formato de código gerado pelo `GCC` com a `libgomp` para os construtores de regiões paralelas *(parallel region)*, compartilhamento de trabalho em laços *(for)*, construtores para a declaração de tarefas explícitas *(task)* e algumas combinações com outros recursos, como *taskloop* para tarefas com compartilhamento de trabalho de laços e suporte a laços e tarefas com vetorização *(simd)*. Além disso apresentamos exemplos com o construtor `target` para *offloading* para dispositivos aceleradores.

3.3.1. Regiões paralelas: construtor `parallel`

Esta é uma das mais importantes diretivas, pois ela é responsável pela demarcação de regiões paralelas, indicando a região de código que será executada em paralelo. Se esse construtor não for especificado o programa será executado de forma sequencial. Regiões paralelas são criadas em `OpenMP` usando-se o construtor `#pragma omp parallel`.

Quando uma região paralela é encontrada pela *thread* principal, é criado um time de *threads* que irão executar o código da região paralela. A *thread* principal torna-se a *thread* mestre desse grupo. Porém, esse construtor não divide o trabalho entre as *threads*, apenas cria a região paralela e o grupo de *threads*. O formato de código para o construtor de região paralela é mostrado no Código 3.1.

Código 3.1. Formato do construtor *parallel*

```
#pragma omp parallel [clause[ [,] clause] ... ] new-line
{
  /* Bloco estruturado. */
}
```

A diretiva *parallel* é implementada com a criação de uma nova função *(outlined function)* usando o código contido no bloco estruturado delimitado pelo construtor. A `libgomp` [GNU Libgomp 2015a] [GNU Libgomp 2015b] usa funções para delimitar a região de código. As duas funções relacionadas com a construção do formato de regiões paralelas na `ABI` da `libgomp` estão listadas no Quadro 3.1.

Quadro 3.1: ABI `libgomp` – Funções relacionadas com a diretiva *parallel*

```
void GOMP_parallel_start(void (*fn)(void *), void *data, unsigned num_threads)
void GOMP_parallel_end(void)
```

De acordo com a documentação da `libgomp` [GNU Libgomp 2015a], o código gerado pós expansão assume o formato apresentado no Código 3.2. São inseridas chamadas às funções do *runtime* do `OpenMP` que demarcam o início e o fim da região paralela, entre essas chamadas a *thread* principal faz uma chamada à função que implementa o código extraído da região paralela.

Código 3.2. Formato de Código Expandido para o construtor *parallel*

```
/* Uma nova função é criada. */
void subfunction (void *data){
  use data;
  body;
}

/* A diretiva é substituída por chamadas ao runtime para criar a região paralela */
setup data;

GOMP_parallel_start(subfunction, &data, num threads);
subfunction(&data);
GOMP_parallel_end();
```

O Código 3.3 mostra o formato do código expandido gerado pelo `GCC` para a diretiva *parallel*. O código está escrito em `GIMPLE`, a representação intermediária utilizada pelo `GCC`.

Código 3.3. Código expandido gerado pelo `GCC` para *parallel*

```
/* Uma nova função é criada. */
main._omp_fn.0 (struct .omp_data_s.0 * .omp_data_i) {
  return;
}

main () {
  int i;
  int D.1804;
  struct .omp_data_s.0 .omp_data_o.1;

<bb 2>:
  .omp_data_o.1.i = i;
  __builtin_GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, 0);
  main._omp_fn.0 (&.omp_data_o.1);
  __builtin_GOMP_parallel_end ();
  i = .omp_data_o.1.i;
```

```
17    D.1804 = 0;
18
19  <L0>:
20    return D.1804;
21  }
```

Uma estrutura `omp_data` é declarada para passar argumentos para a função que irá executar o código da região paralela. A *thread* principal fará uma chamada à função `GOMP_parallel_start(...)` passando como parâmetro o ponteiro da função extraída (main._omp_fn.0) para a criação das *threads* pelo *runtime* do `OpenMP` e também fará uma chamada para a mesma função garantindo sua participação no time de *threads*. Todas as *threads* que terminarem a execução ficarão aguardando na barreira declarada implicitamente no fim da região paralela.

O construtor `parallel` apresenta algumas cláusulas para a definição do número de *threads* a serem criadas no time `(num_threads)`, para um teste condicional se a região paralela deve ou não ser criada `(if)`, e para definições de compartilhamento de dados `(shared e private)`. Pode ser utilizado em conjunto com outros contrutores como `single` e `master` para as situações nas quais seja necessário especificar qual das *threads* deve executar partes do código de uma região paralela. Outros construtores para sincronização entre as *threads* com uma barreira explícita como o `barrier` e para evitar condições de corrida em regiões críticas `(critical)`. O Código 3.4 apresenta um exemplo do uso de algumas cláusulas e desses construtores de compartilhamento de trabalho e sincronização.

Código 3.4. Exemplo de código com o construtor `parallel` e algumas cláusulas

```
1  int main(int argc, char *argv[]) {
2    int n = atoi(argv[1]);
3    int id, valor = 0;
4    printf("Thread[%d][%lu]: Antes da Região Paralela.\n", omp_get_thread_num(), (long int
         ) pthread_self());
5
6    #pragma omp parallel if(n>1024) num_threads(4) default(none) shared(valor) private(id)
7    {
8      id = omp_get_thread_num();
9      long int id_sys = (long int) pthread_self();
10     printf("Thread[%d][%lu]: Código Executado por todas as threads.\n", id, id_sys);
11
12     #pragma omp master
13     {
14       printf("Thread[%d][%lu]: Código Executado pela thread master.\n", id, (long int)
             pthread_self());
15     }
16
17     #pragma omp single
18     {
19       printf("Thread[%d][%lu]: Código Executado por uma das threads.\n", id, (long int)
             pthread_self());
20     }
21
22     if(omp_get_thread_num() == 3){
23       printf("Thread[%d][%lu]: Código Executado pela thread de id: 3.\n", id, (long int)
              pthread_self());
24     }
25
26     #pragma omp critical
27     {
28       printf("Thread[%d][%lu]: Executando a região crítica.\n", id, (long int)
             pthread_self());
```

```
29          printf("Thread[%d][%lu]: Antes... valor: %d\n", id, (long int) pthread_self(),
                valor);
30          valor = valor + id;
31          printf("Thread[%d][%lu]: Depois.. valor: %d\n", id, (long int) pthread_self(),
                valor);
32      }
33
34      printf("Thread[%d][%lu]: Barreira.\n", id, (long int) pthread_self());
35
36      #pragma omp barrier
37
38      printf("Thread[%d][%lu]: Depois da barreira.\n", id, (long int) pthread_self());
39    }
40
41    printf("Thread[%d][%lu]: Depois da Região Paralela.\n", omp_get_thread_num(), (long
          int) pthread_self());
42
43    return 0;
44  }
```

A saída da execução do Código 3.4 é apresentada no Terminal 3.1.

Terminal 3.1

```
rogerio@chamonix:/src/example-parallel-with-clauses$ ./example-parallel-with-clauses.
    exe 4096
Thread[0][18446744072495294336]: Antes da Região Paralela.
Thread[0][18446744072495294336]: Código Executado por todas as threads.
Thread[0][18446744072495294336]: Código Executado pela thread master.
Thread[0][18446744072495294336]: Código Executado por uma das threads.
Thread[1][18446744072482789120]: Código Executado por todas as threads.
Thread[3][18446744072466003712]: Código Executado por todas as threads.
Thread[2][18446744072474396416]: Código Executado por todas as threads.
Thread[3][18446744072466003712]: Código Executado pela thread de id: 3.
Thread[0][18446744072495294336]: Executando a região crítica.
Thread[0][18446744072495294336]: Antes... valor: 0
Thread[0][18446744072495294336]: Depois.. valor: 0
Thread[0][18446744072495294336]: Barreira.
Thread[2][18446744072474396416]: Executando a região crítica.
Thread[2][18446744072474396416]: Antes... valor: 0
Thread[2][18446744072474396416]: Depois.. valor: 2
Thread[2][18446744072474396416]: Barreira.
Thread[1][18446744072482789120]: Executando a região crítica.
Thread[1][18446744072482789120]: Antes... valor: 2
Thread[1][18446744072482789120]: Depois.. valor: 3
Thread[1][18446744072482789120]: Barreira.
Thread[3][18446744072466003712]: Executando a região crítica.
Thread[3][18446744072466003712]: Antes... valor: 3
Thread[3][18446744072466003712]: Depois.. valor: 6
Thread[3][18446744072466003712]: Barreira.
Thread[0][18446744072495294336]: Depois da barreira.
Thread[2][18446744072474396416]: Depois da barreira.
Thread[3][18446744072466003712]: Depois da barreira.
Thread[1][18446744072482789120]: Depois da barreira.
Thread[0][18446744072495294336]: Depois da Região Paralela.
rogerio@chamonix:/src/example-parallel-with-clauses$
```

3.3.2. Loops: construtor `for`

Um time de *threads* é criado quando uma região paralela é alcançada, porém com apenas o construtor de região paralela todas as *threads* irão executar o mesmo código que compõe o corpo da *outlined function*.

O construtor `for` é usado para distribuir o trabalho e coordenar a execução paralela entre as *threads* do time. Especifica que as iterações de um ou mais laços irão ser

executadas em paralelo pelas *threads* no contexto de tarefas implícitas. As iterações são distribuídas entre as *threads* que estão em execução dentro da região paralela.

Os construtores para a especificação de região paralela e de laços podem ser utilizados separadamente ou combinados. O Código 3.5 mostra uma região paralela com um construtor *for*, o que é equivalente ao uso dos construtores em modo combinado apresentado no Código 3.6.

Código 3.5. Construtor *for* dentro de uma região paralela

```
1 #pragma omp parallel
2 {
3   #pragma omp for
4   for (i = lb; i <= ub; i++){
5     body;
6   }
7 }
```

Código 3.6. Construtores *parallel* e *for* combinados

```
1   #pragma omp parallel for
2   for (i = lb; i <= ub; i++){
3     body;
4   }
```

Semelhante ao que ocorre no processamento do construtor *parallel* individualmente, um construtor `parallel` com um construtor `for` ou o formato combinado deles `parallel for` também é implementado com a criação de uma nova função *(outlined function)*.

O código expandido que substitui a declaração do construtor `parallel` e do *for* associado ao laço paralelo é composto pelas chamadas para criar e finalizar a região paralela que são feitas ao *runtime* do `OpenMP` e pela *outlined function*. O que muda é que neste caso o corpo da nova função terá o código para controlar a distribuição das iterações do laço em *(chunks)* que são executados pelas *threads*. A Figura 3.2 apresenta como os *chunks* de iterações de um laço são executados pelas *threads*.

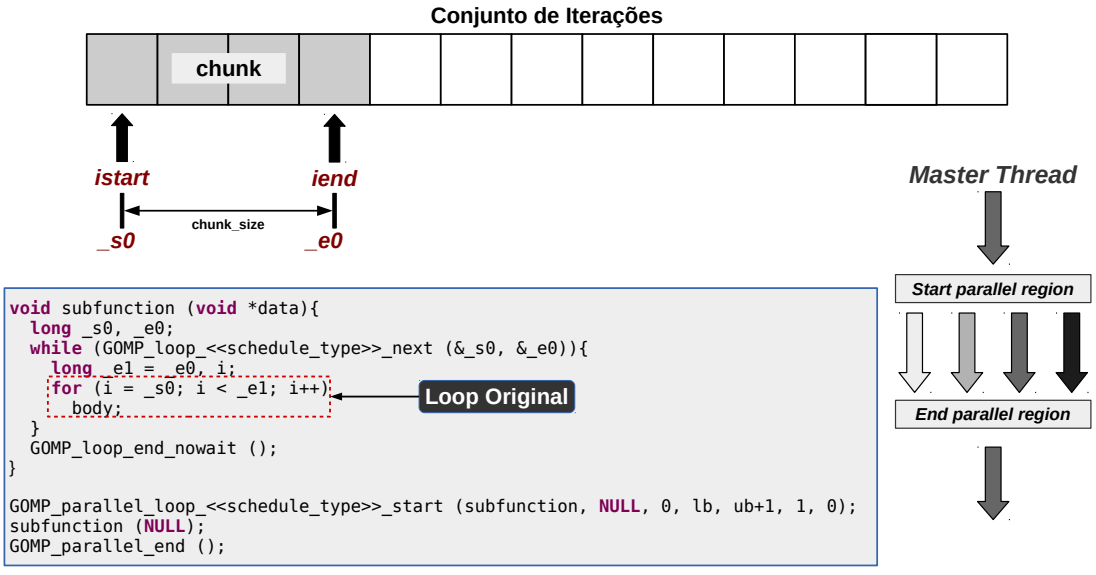


Figura 3.2. Esquema de execução das iterações do laço

O trabalho é dividido entre as *threads* de acordo com o algoritmo de escalonamento de iterações adotado. O escalonamento é definido usando-se a cláusula `schedule` e os tipos que estão disponíveis no `OpenMP` são: *static, auto, runtime, dynamic e guided*.

O algoritmo de escalonamento define como as iterações do laço associado serão divididas em subconjuntos contíguos e não vazios, chamados de *chunks*, e como são distribuídos entre as *threads* pertencentes à região paralela [OpenMP-ARB 2015]. Os tipos de escalonamento de iterações de laços que podem ser utilizados no `OpenMP`:

1. Estático – `schedule(static, chunk_size)`: Baseia-se na ideia de que cada *thread* irá executar a mesma quantidade de iterações, se um *chunk_size* não for especificado irá dividir o número de iterações pelo número de *threads* formando *chunks* de tamanhos iguais e pelo menos um *chunk* é distribuído para cada *thread*, caso contrário seguirá no esquema *round-robin* pela ordem dos *ids* das *threads*, atribuindo *chunks* para cada uma delas até que todo o conjunto de iterações tenha sido executado.

2. Dinâmico - `schedule(dynamic, chunk_size)`: As iterações são distribuídas para as *threads* do time em *chunks*, conforme as *threads* requisitam mais trabalho. Cada *thread* executa um *chunk* de iterações e então requisita outro *chunk* até que não restem mais *chunks* para serem distribuídos. Cada *chunk* contém *chunk_size* iterações, exceto o último *chunk* a ser distribuído que pode ter um número menor de iterações. Quando a variável *chunk_size* não está definida, o valor padrão é 1.

3. Guiado – `schedule(guided, chunk_size)`: As iterações são atribuídas para as *threads* do time em *chunks* também conforme as *threads* requisitam mais trabalho. Cada *thread* executa um *chunk* de iterações e então requisita outro, até que não existam mais *chunks* a serem atribuídos. Para um *chunk_size* especificado como 1, o tamanho de cada *chunk* é proporcional ao número de iterações não atribuídas dividido pelo número de *threads* no time, decrescendo até 1. Para um *chunk_size* com um valor *k* (maior que 1), o tamanho de cada *chunk* é determinado da mesma forma, com a restrição de que os *chunks* não contenham menos que *k* iterações, exceto o último.

4. Auto – `schedule(auto)`: A decisão do escalonamento é delegada para o compilador ou para o *runtime*.

5. Runtime – `schedule(runtime)`: A decisão do escalonamento é adiada até o momento de execução, só é conhecida em tempo de execução. Tanto o *schedule* quando o *chunk_size* são obtidos do `run-sched-var ICV`. Se o `ICV` é definido para *auto*, o escalonamento é definido pela implementação. Quando o tipo especificado for *runtime* ou *auto* o valor de *chunk_size* não deve ser definido.

Quando não é especificado qual algoritmo de escalonamento a ser utilizado pelo *runtime* ou ele é do tipo *auto*, o `GCC` gera o código usando as funções da `libgomp` para o formato de escalonamento *static*, que por padrão faz uma divisão estática das iterações do laço pelo número de *threads*. O Código 3.7 apresenta um laço que terá suas iterações distribuídas entre as *threads* estaticamente.

Código 3.7. Laço sem escalonamento definido

```
1  int main() {
2    int id, i;
3
4    printf("Thread[%d][%lu]: Antes da Região Paralela.\n", omp_get_thread_num(), (long int
         ) pthread_self());
5
6    #pragma omp parallel num_threads(4) default(none) private(id)
7    {
8      // Todas as threads executam esse código.
9      id = omp_get_thread_num();
10
11     #pragma omp for
12     for(i=0; i<16; i++){
13       printf("Thread[%d][%lu]: Trabalhando na iteração %lu.\n", id, (long int)
             pthread_self(), i);
14     }
15   }
16   printf("Thread[%d][%lu]: Depois da Região Paralela.\n", omp_get_thread_num(), (long
         int) pthread_self());
17
18   return 0;
19 }
```

A saída produzida pela execução do Código 3.7 é apresentada no Terminal 3.2.

Terminal 3.2

```
rogerio@chamonix:/src/example-for$ ./example-for-constructor-static.exe
Thread[0][1476638592]: Antes da Região Paralela.
Thread[1][1464133376]: Trabalhando na iteração 4.
Thread[1][1464133376]: Trabalhando na iteração 5.
Thread[1][1464133376]: Trabalhando na iteração 6.
Thread[1][1464133376]: Trabalhando na iteração 7.
Thread[0][1476638592]: Trabalhando na iteração 0.
Thread[0][1476638592]: Trabalhando na iteração 1.
Thread[0][1476638592]: Trabalhando na iteração 2.
Thread[0][1476638592]: Trabalhando na iteração 3.
Thread[3][1447347968]: Trabalhando na iteração 12.
Thread[3][1447347968]: Trabalhando na iteração 13.
Thread[3][1447347968]: Trabalhando na iteração 14.
Thread[3][1447347968]: Trabalhando na iteração 15.
Thread[2][1455740672]: Trabalhando na iteração 8.
Thread[2][1455740672]: Trabalhando na iteração 9.
Thread[2][1455740672]: Trabalhando na iteração 10.
Thread[2][1455740672]: Trabalhando na iteração 11.
Thread[0][1476638592]: Depois da Região Paralela.
rogerio@chamonix:/src/example-for$
```

O Código 3.8 apresenta um laço anotado com o construtor `for` e com a cláusula `schedule(dynamic)`. Nesse tipo de escalonamento as *threads* ficam solicitando mais trabalho para o *runtime* até que todas as iterações tenham sido executadas. Desta maneira a execução depende de quais *threads* ficaram disponíveis, podendo uma *thread* receber mais *chunks* de iterações que outras.

Código 3.8. Laço com `schedule(dynamic)`

```
1  int main() {
2    int id, i;
3
4    printf("Thread[%d][%lu]: Antes da Região Paralela.\n", omp_get_thread_num(), (long int
         ) pthread_self());
5
6    #pragma omp parallel num_threads(4) default(none) private(id)
7    {
```

```
8       // All threads executes this code.
9       id = omp_get_thread_num();
10
11      #pragma omp for schedule(dynamic,2)
12      for(i=0; i<16; i++){
13        printf("Thread[%d][%lu]: Trabalhando na iteração %lu.\n", id, (long int)
              pthread_self(), i);
14      }
15    }
16    printf("Thread[%d][%lu]: Depois da Região Paralela.\n", omp_get_thread_num(), (long
          int) pthread_self());
17
18    return 0;
19  }
```

A saída produzida pela execução do Código 3.8 é apresentada no Terminal 3.3.

Terminal 3.3

```
rogerio@chamonix:/src/example-for$ ./example-for-constructor-dynamic.exe
Thread[0][18446744073366411136]: Antes da Região Paralela.
Thread[0][18446744073366411136]: Trabalhando na iteração 0.
Thread[0][18446744073366411136]: Trabalhando na iteração 1.
Thread[0][18446744073366411136]: Trabalhando na iteração 8.
Thread[0][18446744073366411136]: Trabalhando na iteração 9.
Thread[0][18446744073366411136]: Trabalhando na iteração 10.
Thread[0][18446744073366411136]: Trabalhando na iteração 11.
Thread[2][18446744073345513216]: Trabalhando na iteração 4.
Thread[2][18446744073345513216]: Trabalhando na iteração 5.
Thread[2][18446744073345513216]: Trabalhando na iteração 14.
Thread[2][18446744073345513216]: Trabalhando na iteração 15.
Thread[0][18446744073366411136]: Trabalhando na iteração 12.
Thread[0][18446744073366411136]: Trabalhando na iteração 13.
Thread[1][18446744073353905920]: Trabalhando na iteração 2.
Thread[1][18446744073353905920]: Trabalhando na iteração 3.
Thread[3][18446744073337120512]: Trabalhando na iteração 6.
Thread[3][18446744073337120512]: Trabalhando na iteração 7.
Thread[0][18446744073366411136]: Depois da Região Paralela.
rogerio@chamonix:/src/example-for$
```

Quando os tipos de escalonamento *runtime, dynamic* ou *guided* são usados, o formato do código gerado é o mesmo, mas ainda apresentam dois formatos distintos dependendo de como estão definidos o limite superior do laço e o *chunk_size*. Se o código utiliza valores numéricos para essas definições o código gerado é de um formato. Caso contrário, se as definições são feitas com base em variáveis ou expressões que precisam ser avaliadas, então o formato de código é outro.

Código 3.9. Laço com limite superior usando valor

```
1 #pragma omp parallel for schedule(<<
      schedule_type>>)
2 for (i = 0; i < 1024; i++){
3   // body.
4 }
```

Código 3.10. Laço com limite superior usando variável

```
1 n = 1024;
2 #pragma omp parallel for schedule(<<
      schedule_type>>)
3 for (i = 0; i < n; i++){
4   // body.
5 }
```

Foram identificados dois formatos de código para a execução de laços, como em cada formato a estrutura é a mesma para os tipos de escalonamentos *dynamic, runtime* ou *guided*, estão representados nos códigos pela marcação «`schedule_type`». Desta forma, o `GCC` e a biblioteca `libgomp` usam as funções listadas no Quadro 3.2 para

delimitar a região paralela e criar o *primeiro formato* de laço.

Quadro 3.2: ABI `libgomp` – Funções usadas para *parallel for* no primeiro formato

```
void GOMP_parallel_loop_<<schedule_type>>_start (void (*fn) (void *), void *data,
    unsigned num_threads, long start, long end, long incr);
void GOMP_parallel_end (void);
bool GOMP_loop_<<schedule_type>>_next (long *istart, long *iend);
void GOMP_loop_end_nowait (void);
```

Como no *segundo formato* é necessário avaliar a expressão ou variável que define o valor assumido pelo limite superior do laço ou do *chunk_size*, somente é criada a região paralela e a inicialização do laço é feita dentro da função criada para tratar o laço. As funções utilizadas no segundo formato são apresentadas no Quadro 3.3.

Quadro 3.3: ABI `libgomp` – Funções usadas para *parallel for* no segundo formato

```
void GOMP_parallel_start (void (*fn) (void *), void *data, unsigned num_threads);
void GOMP_parallel_end (void);
void GOMP_parallel_loop_<<schedule_type>>_start (void (*fn) (void *), void *data,
    unsigned num_threads, long start, long end, long incr);
bool GOMP_loop_<<schedule_type>>_next (long *istart, long *iend);
void GOMP_loop_end_nowait (void);
```

O código gerado para executar laços que se enquadram no *primeiro formato* é mostrado no Código 3.11.

Código 3.11. Código expandido para laços no primeiro formato

```
1  void subfunction (void *data){
2    long _s0, _e0;
3    while (GOMP_loop_<<schedule_type>>_next (&_s0, &_e0)){
4      long _e1 = _e0, i;
5      for (i = _s0; i < _e1; i++){
6        body;
7      }
8    }
9    GOMP_loop_end_nowait ();
10 }
11
12 /* O laço anotado é substituído. */
13 setup data;
14
15 GOMP_parallel_loop_<<schedule_type>>_start (subfunction, &data,
16 num_threads, start, end, incr, chunk_size,...);
17 subfunction (&data);
18 GOMP_parallel_end ();
```

O Código 3.12 apresenta a estrutura do código gerado para o *segundo formato*.

Código 3.12. Código expandido para laços no segundo formato

```
1  void subfunction (void *data){
2    long i, _s0, _e0;
3    if (GOMP_loop_<<schedule_type>>_start (0, n, 1, &_s0, &_e0)){
4      do {
5        long _e1 = _e0;
6        for (i = _s0; i < _e0; i++) {
7          body;
8        }
9      } while (GOMP_loop_<<schedule_type>>_next (&_s0, &_e0));
10   }
```

```
11    GOMP_loop_end ();
12  }
13
14  /* O laço anotado é substituído. */
15  setup data;
16
17  GOMP_parallel_start (subfunction, &data, num_threads);
18  subfunction (&data);
19  GOMP_parallel_end ();
```

O GCC utiliza o GIMPLE como formato de código intermediário, a visualização do código intermediário gerado para o *primeiro formato* é apresentada na Figura 3.3.

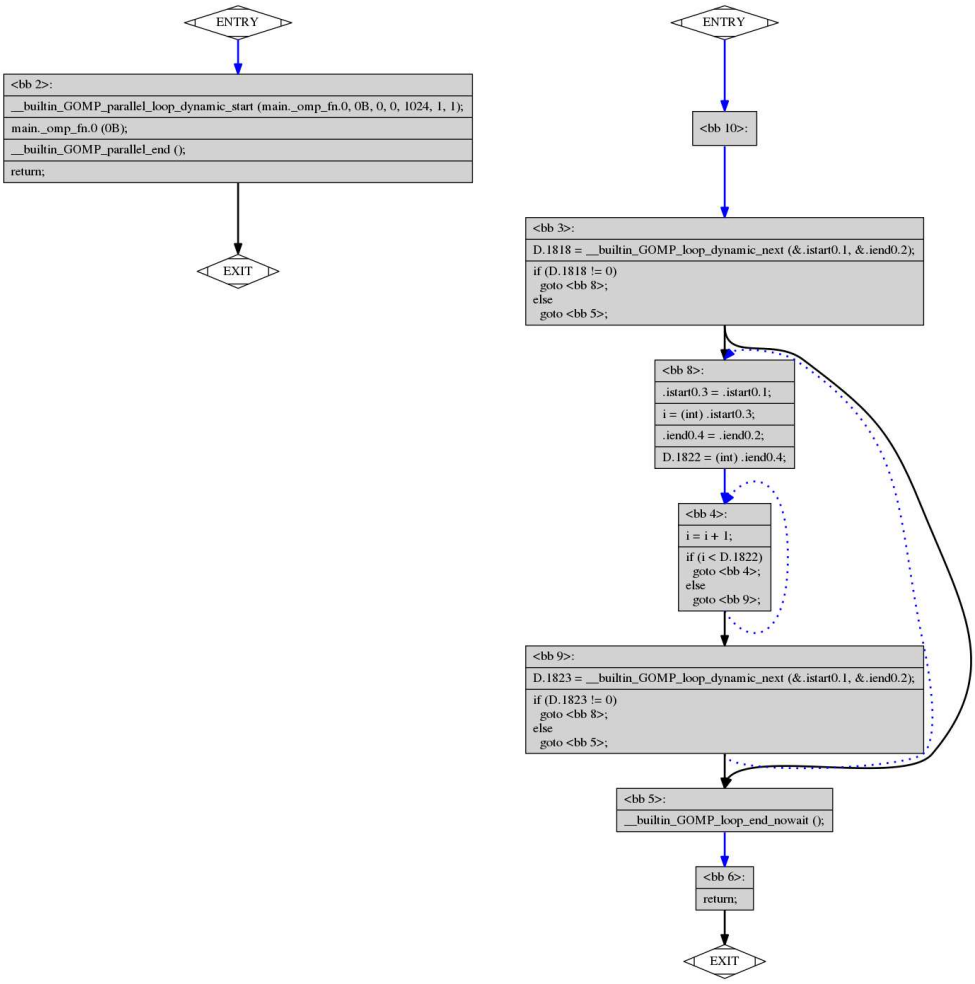


Figura 3.3. Visualização do primeiro formato laço utilizando `schedule(dynamic)`

No *primeiro formato* o início da região paralela é marcado com a chamada à função `GOMP_parallel_loop_«schedule_type»_start()`, que além de criar o time de *threads* também inicializa os controles da execução do laço. Dentro da *outlined function* a chamada à função `GOMP_loop_«schedule_type»_next(...)` é usada pela *thread* para recuperar o primeiro *chunk*. Cada *thread* executa este primeiro trabalho e depois entra em *loop* recuperando e executando novos *chunks* até que não tenha mais trabalho a ser feito. Quando as *threads* terminam a execução finalizam a

execução do laço chamando `GOMP_loop_end_nowait()` e o compartilhamento de trabalho do laço é também finalizado. Então a região paralela é finalizada com a chamada `GOMP_parallel_end()` que desaloca o time de *threads*.

A Figura 3.4 mostra a visualização do código para o *segundo formato*. A mesma semântica é aplicada ao *segundo formato*, mesmo que utilize diferentes funções.

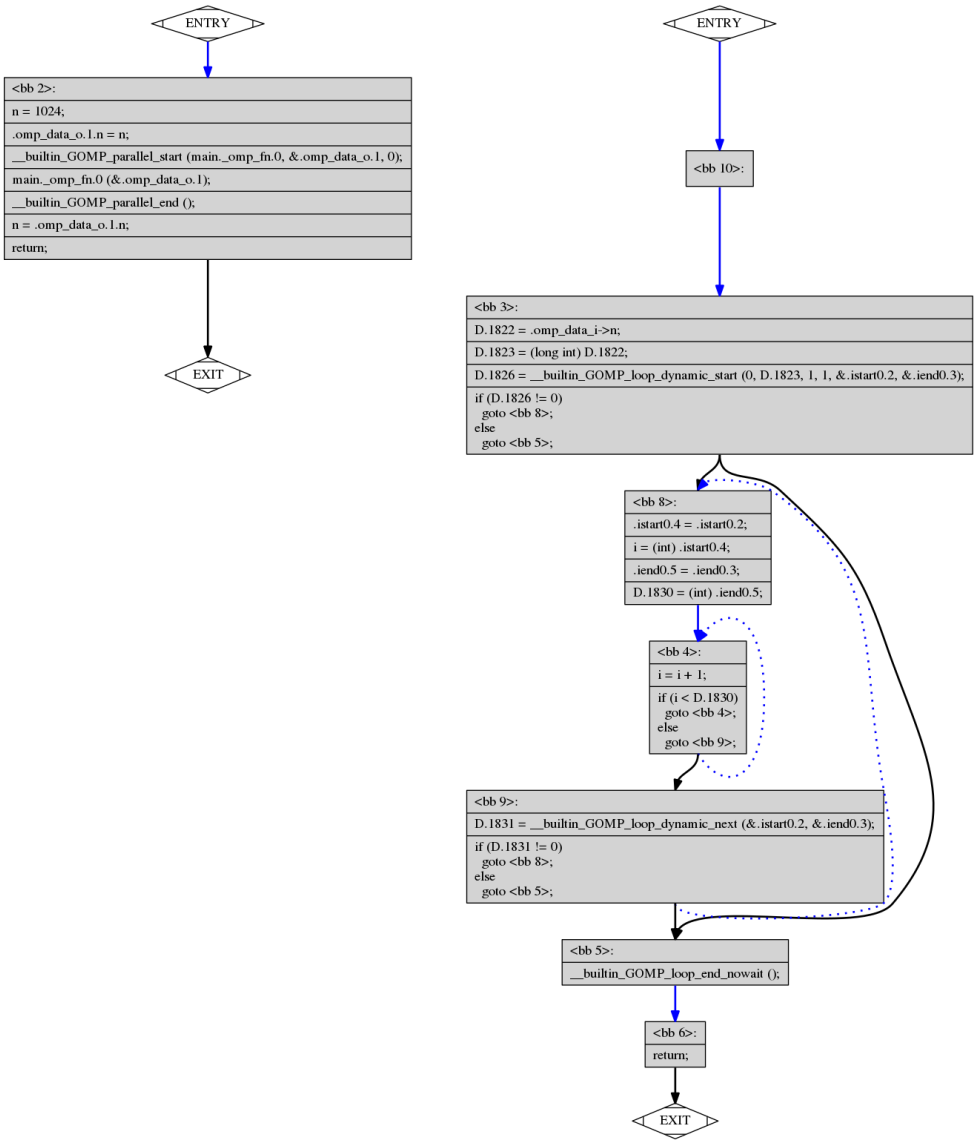


Figura 3.4. Visualização do segundo formato laço utilizando `schedule(dynamic)`

No *segundo formato* a chamada à função `GOMP_parallel_start(...)` inicia a região paralela e nesta chamada somente é criado o time de *threads*. A inicialização do compartilhamento de trabalho do laço é feito dentro da *outlined function* e a chamada `GOMP_loop_«schedule_type»_start(...)` é usada para recuperar o primeiro *chunk*. As *threads* que conseguem obter o seu primeiro *chunk* pode executá-lo e usam a função `GOMP_loop_«schedule_type»_next(...)` para recuperar os próximos *chunks* até terminarem as iterações do laço e então finalizarem com a chamada à

`GOMP_loop_end_nowait()`. A região paralela também é finalizada usando a mesma chamada à função `GOMP_parallel_end()` que desaloca o time de *threads*.

A Figura 3.5 resume os dois formatos de código que são gerados para laços.

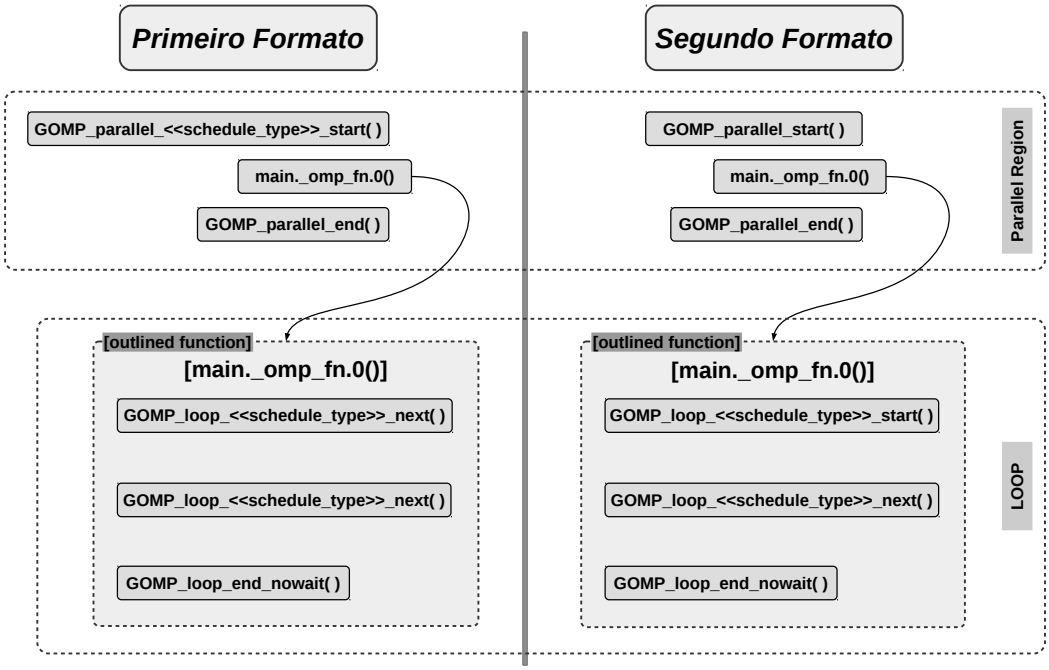


Figura 3.5. Comparativo dos dois formatos de laços

O Código 3.13 apresenta dois laços com diferentes escalonamentos e definições de limite superior e *chunk_size*.

Código 3.13. Código de uma região paralela com dois laços

```
1  num_t = 8;
2  #pragma omp parallel num_threads(num_t)
3  {
4    #pragma omp for schedule(runtime)
5    for (i = 0; i < 1024; i++){
6      body_1;
7    }
8    #pragma omp for schedule(dynamic, 32)
9    for (j = 0; j < n; j++){
10     body_2;
11   }
12 }
```

A Figura 3.6 mostra uma representação gráfica do código gerado para a região paralela com dois laços. No caso de códigos com múltiplos laços são geradas barreiras implíticas entre os laços no código final. Além as *threads* que terminarem seu trabalho antes das outras, aguardarão a conclusão na barreira implícita gerada pelo final da região paralela.

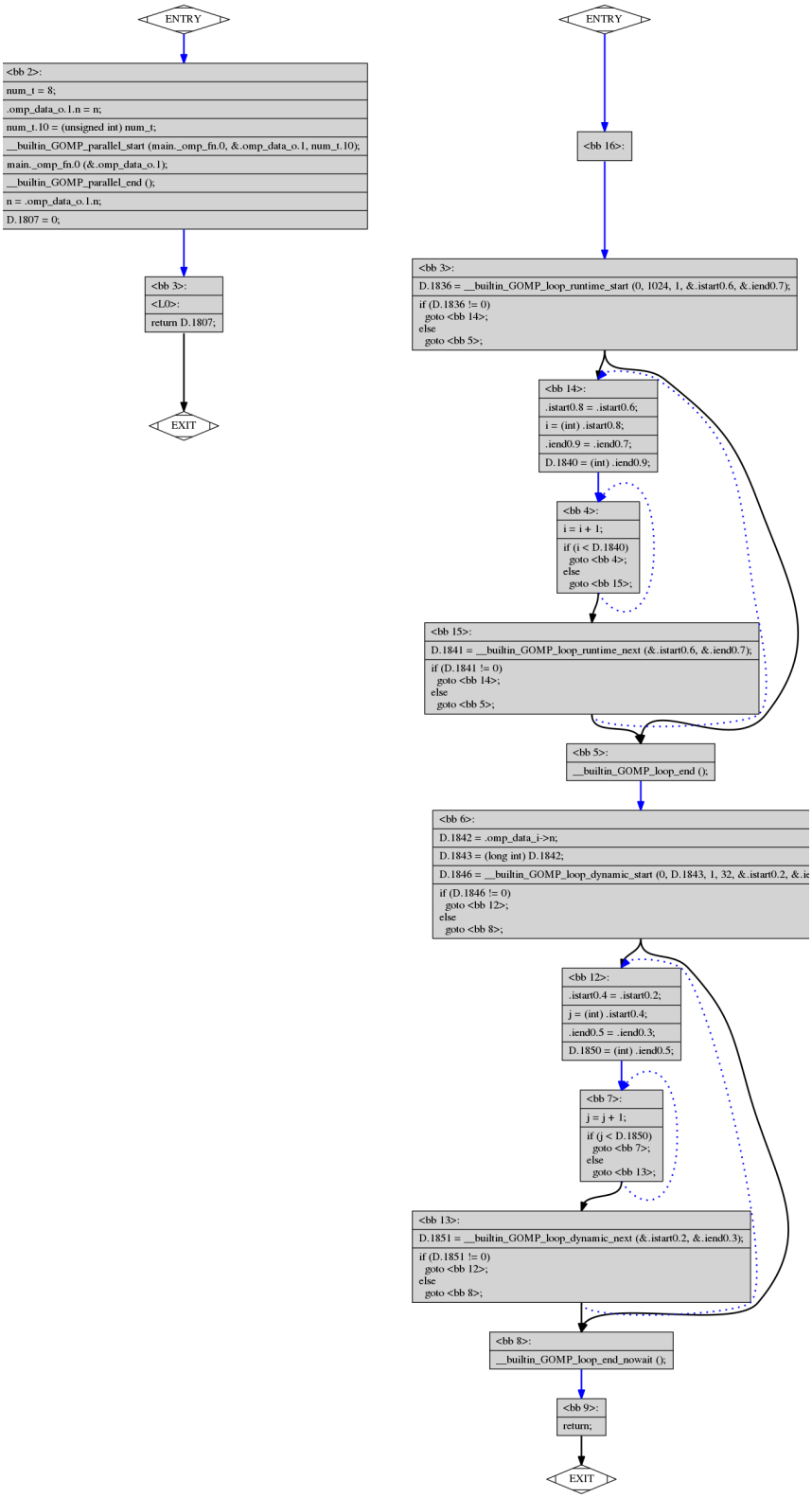


Figura 3.6. Representação gráfica de dois laços dentro de uma mesma região paralela

Podemos perceber que é seguido o mesmo processo, com o código da região paralela é criada uma nova função que agora terá o código dos dois laços, seguindo o formato

de laço. A Figura 3.7 mostra o código da função com dois laços separados por uma barreira que é gerada com a chamada à função `GOMP_loop_end()`.

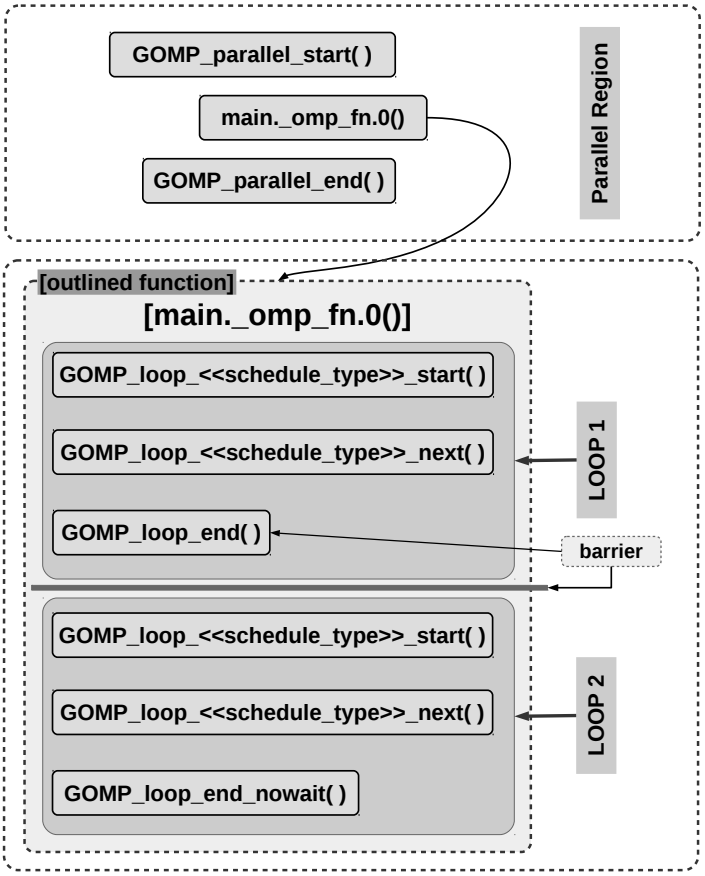


Figura 3.7. Formato para dois laços dentro de uma mesma região paralela

3.3.3. Seções: construtor `sections`

O *sections* é um construtor de compartilhamento de trabalho não iterativo que permite a definição de um conjunto de blocos estruturados utilizando a diretiva `#pragma omp sections` para indicar a criação de seções de código e a diretiva `#pragma omp section` para especificar cada bloco que será associado a uma seção. Os blocos são distribuídos para serem executados pelas *threads* do time criado pela região paralela, isto é, cada bloco é executado por uma das *threads* no contexto de uma tarefa implícita.

A sintaxe para uso dos construtores de seções é apresentado no Código 3.14. O conceito é que cada um dos blocos `(bloco_1, bloco_2 e bloco_3)` seja executado por alguma das *threads* do time criado pela região paralela.

Código 3.14. Uso dos construtores de seções

```
#pragma omp sections
{
  #pragma omp section
  bloco_1;
  #pragma omp section
  bloco_2;
  #pragma omp section
  bloco_3;
}
```

A estrutura de código que é gerada para a execução das seções é apresentada no Código 3.15. Este código estará dentro da função extraída para a execução da região paralela e todas as *threads* pertencentes ao time criado por essa região paralela irão executá-la. O bloco de seções é iniciado com a chamada `GOMP_sections_start(3)` (o argumento 3 indica o número de seções definidas no código) e as *threads* que atingirem o código do laço que itera sobre o conjunto de seções primeiro obterão uma das seções para executarem com a chamada à função `GOMP_sections_next()`, até que todas as seções definidas tenham sido executadas.

Código 3.15. Código dos construtores de seções expandido

```
for (i = GOMP_sections_start (3); i != 0; i = GOMP_sections_next ())
  switch (i) {
    case 1:
      bloco_1;
      break;
    case 2:
      bloco_2;
      break;
    case 3:
      bloco_3;
      break;
  }
  GOMP_barrier ();
```

O Código 3.16 apresenta um exemplo do uso de seções com a cláusula de redução `(reduction)`. Cada uma das *threads* irá trabalhar sobre o código de uma das seções produzindo um valor para sua cópia de *sum*. A cláusula indica que ao final da execução deve ser feita uma redução de soma `(reduction(+:sum))` nas cópias de *sum* que pertencem a cada uma das seções, gerando um único valor para *sum*.

Código 3.16. Exempo do uso dos construtores de seções

```
int main(int argc, char *argv[]) {
  int i, id;
  int sum = 0;

  fprintf(stdout, "Thread[%d][%lu]: Antes da Região Paralela.\n", omp_get_thread_num(),
      (long int)pthread_self());

  #pragma omp parallel num_threads(8) private(id)
  {
    id = omp_get_thread_num();
    #pragma omp sections reduction(+:sum)
    {
      #pragma omp section
      {
        fprintf(stdout, "   Thread[%lu,%lu]: Trabalhando na seção 1.\n", id, (long int)
            pthread_self());
        for(i=0; i<1024;i++){
          sum += i;
```

```
17          }
18        }
19
20        #pragma omp section
21        {
22          fprintf(stdout, "   Thread[%lu,%lu]: Trabalhando na seção 2.\n", id, (long int)
                pthread_self());
23          for(i=0; i<1024;i++){
24            sum += i;
25          }
26        }
27      }
28    }
29    fprintf(stdout, "Thread[%d][%lu]: Depois da Região Paralela.\n", omp_get_thread_num(),
          (long int)pthread_self());
30    fprintf(stdout, "Thread[%d][%lu]: sum: %d\n", omp_get_thread_num(), (long int)
          pthread_self(), sum);
31
32    return 0;
33  }
```

A execução das seções ocorre de maneira independente, cada uma das seções é atribuída a uma das *threads*. As *threads* que terminam a execução de sua parte do trabalho ficam aguardando em uma barreia implicíta adicionada no final do bloco de seções. Na saída da execução do Código 3.16 que é apresentada no Terminal 3.4 é possível visualizar que a seção 2 foi executada antes da seção 1.

Terminal 3.4

```
rogerio@chamonix:/src/example-sections-reduction$ ./example-sections-reduction.exe
Thread[0][18446744073314326400]: Antes da Região Paralela...
Thread[0,18446744073314326400]: Trabalhando na seção 2.
Thread[4,18446744073276643072]: Trabalhando na seção 1.
Thread[0][18446744073314326400]: Depois da Região Paralela.
Thread[0][18446744073314326400]: sum: 1047552
rogerio@chamonix:/src/example-sections-reduction$
```

O construtor `sections` foi a primeira forma de execução de blocos de código independentes e não iterativos, mesmo que a execução ainda ocorra dentro de uma região paralela com a criação de *threads* de maneira implícita.

A Figura 3.8 apresenta a visualização do código intermediário gerado para o Código 3.16.

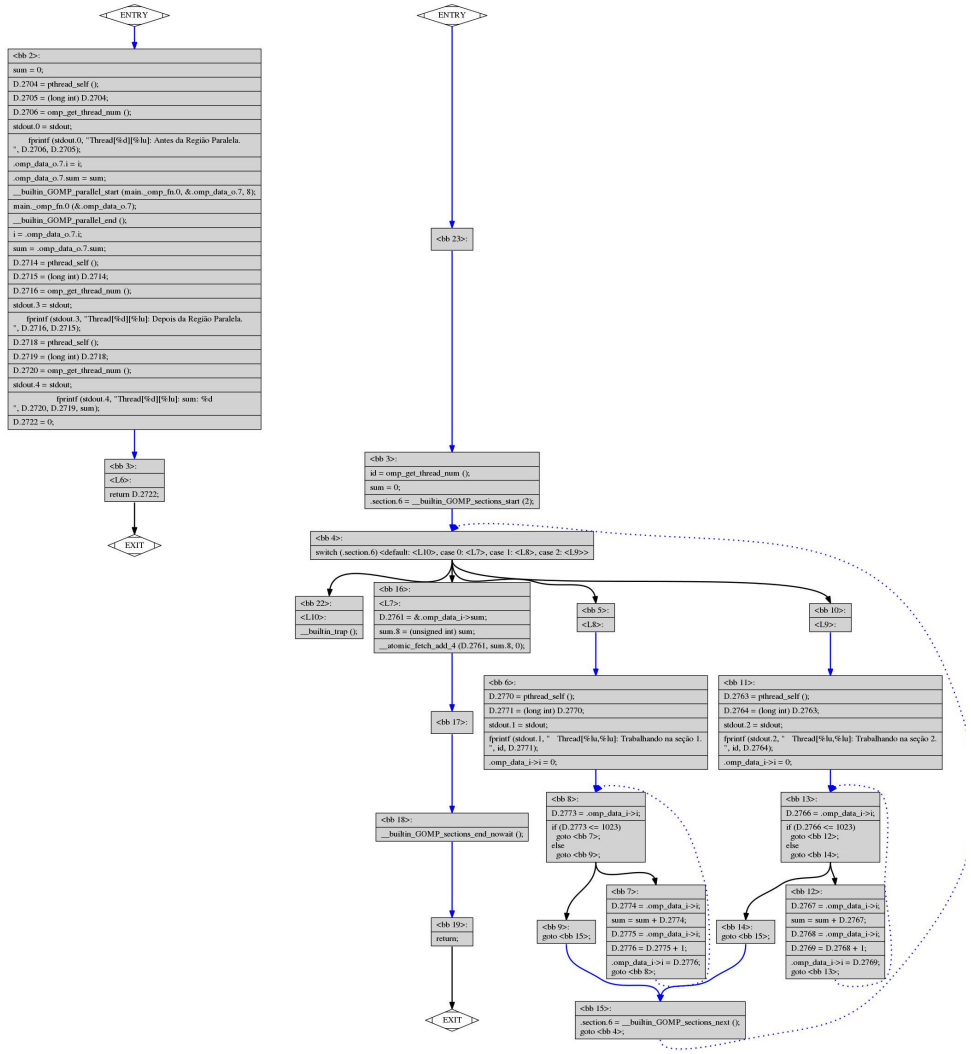


Figura 3.8. Código gerado para os construtores de seções

3.3.4. Tarefas: construtor `task`

O construtor *task* permite a criação de tarefas explícitas. O construtor `task` está disponível a partir das especificações 3.0 e 3.1 e é implementado pela `libgomp` do `GCC 4.4` e `GCC 4.7`, respectivamente. Quando uma *thread* encontra um construtor *task*, uma nova tarefa é gerada para executar o bloco associado ao construtor. A sintaxe do construtor `task` é apresentada no Código 3.17.

Código 3.17. Formato do construtor *task*

```
1 #pragma omp task [clause[ [,] clause] ... ] new-line
2 /* Bloco estruturado. */
```

O Código 3.18 apresenta como o construtor `task` é usado dentro de uma região paralela. Se for necessário criar apenas uma nova tarefa, o construtor `single` pode ser

utilizado para garantir esse comportamento, caso contrário todas a *threads* do time irão cria uma nova *task*.

Código 3.18. Formato do construtor *task*

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    {
      /* Bloco estruturado. */
    }
  }
}
```

As funções da biblioteca `libgomp` que são utilizadas para gerar o código relacionado com o construtor *task* são listadas no Quadro 3.4.

Quadro 3.4: ABI `libgomp` – Funções usadas para a implementação do construtor *task*

```
void GOMP_parallel_start (void (*fn) (void *), void *data, unsigned num_threads);
void GOMP_parallel_end (void);
void GOMP_task (void (*fn) (void *), void *data, void (*cpyfn) (void *, void *),
  long arg_size, long arg_align, bool if_clause, unsigned flags,
void **depend);
void GOMP_taskwait (void);
```

O Código 3.19 apresenta um exemplo do uso da diretiva *task*. Neste exemplo três *tasks* são criadas dentro de uma região paralela. O construtor *single* é utilizado para garantir que o código seja executado apenas uma vez por uma das *threads* do time. Caso contário, as 8 *threads* criadas executariam o mesmo código criando cada uma delas três *tasks*.

Código 3.19. Exemplo de uso do construtor *task*

```
int main(int argc, char *argv[]) {
  int id = 0;
  int x = atoi(argv[1]);

  fprintf(stdout, "Thread[%lu,%lu]: Antes da região paralela.\n", omp_get_thread_num(),
      (long int) pthread_self());

  #pragma omp parallel num_threads(8) firstprivate(x) private(id)
  {
    id = omp_get_thread_num();
    fprintf(stdout, "  Thread[%lu,%lu]: Todas as threads executam.\n", id, (long int)
        pthread_self());

    #pragma omp single
    {
      fprintf(stdout, "  Thread[%lu,%lu]: Antes de criar tasks.\n", id, (long int)
          pthread_self());
      #pragma omp task if(x > 10)
      {
        fprintf(stdout, "    Thread[%lu,%lu]: Trabalhando na task 1.\n",
            omp_get_thread_num(), (long int) pthread_self());
      }

      #pragma omp task if(x > 20)
      {
        fprintf(stdout, "    Thread[%lu,%lu]: Trabalhando na task 2.\n",
            omp_get_thread_num(), (long int) pthread_self());
```

```
23        }
24
25        fprintf(stdout, "    Thread[%lu,%lu]: Antes do taskwait.\n", id, (long int)
              pthread_self());
26        #pragma omp taskwait
27        fprintf(stdout, "    Thread[%lu,%lu]: Depois do taskwait.\n", id, (long int)
              pthread_self());
28
29        #pragma omp task
30        {
31          fprintf(stdout, "    Thread[%lu,%lu]: Trabalhando na task 3.\n",
                omp_get_thread_num(), (long int) pthread_self());
32        }
33      }
34    }
35    fprintf(stdout, "Thread[%lu,%lu]: Depois da região paralela.\n", omp_get_thread_num(),
          (long int) pthread_self());
36
37    return 0;
38  }
```

Ainda no Código 3.19, pode ser visto o uso da cláusula `if` que também pode ser aplicada ao construtor `task` indicando uma condição para a criação da nova tarefa. No exemplo a `task 1` será criada somente se o valor da variável *x* recebido por parâmetro for maior que 10 e a `task 2` será criada se esse valor for maior que 20, já a `task 3` será criada sem nenhuma condição. A *thread* que entra no bloco do construtor `single` criará as duas primeiras *threads* e ficará aguardando o término da execução delas na diretiva `#pragma omp taskwait`. A saída da execução do Código 3.19 é apresentada no Terminal 3.5.

Terminal 3.5

```
rogerio@chamonix:/src/example-tasks$ ./example-tasks.exe 1024
Thread[0,140369357629312]: Antes da região paralela.
Thread[0,140369357629312]: Todas as threads executam.
Thread[0,140369357629312]: Antes de criar tasks.
Thread[7,140369294767872]: Todas as threads executam.
Thread[0,140369357629312]: Antes do taskwait.
Thread[3,140369328338688]: Todas as threads executam.
Thread[1,140369345124096]: Todas as threads executam.
Thread[6,140369303160576]: Todas as threads executam.
Thread[5,140369311553280]: Todas as threads executam.
Thread[4,140369319945984]: Todas as threads executam.
Thread[7,140369294767872]: Trabalhando na task 1.
Thread[0,140369357629312]: Trabalhando na task 2.
Thread[0,140369357629312]: Depois do taskwait.
Thread[5,140369311553280]: Trabalhando na task 3.
Thread[2,140369336731392]: Todas as threads executam.
Thread[0,140369357629312]: Depois da região paralela.
rogerio@chamonix:/src/example-tasks$
```

O formato de código gerado pelo `GCC` é apresentado no Código 3.20. Podemos perceber que são criadas novas funções para tratar a região paralela e o código das tarefas declaradas. Na função criada para tratar o código da região paralela as novas tarefas são criadas com as chamadas para a função `GOMP_task(...)`. Em cada uma das chamadas ao *runtime* do `OpenMP` é passado o ponteiro da função que deve ser executada pela nova *task*.

Código 3.20. Formato de código expandido para a diretiva *task*

```
main._omp_fn.3 (void * .omp_data_i)
{
  /* Declaração de variáveis suprimida. */

<bb 20>:

<bb 14>:
  D.3626 = pthread_self ();
  D.3627 = (long int) D.3626;
  D.3628 = omp_get_thread_num ();
  stdout.8 = stdout;
  fprintf (stdout.8, "    Thread[%lu,%lu]: Trabalhando na task 3.\n", D.3628, D.3627);
  return;
}

main._omp_fn.2 (void * .omp_data_i)
{
  /* Declaração de variáveis suprimida. */

<bb 22>:

<bb 11>:
  D.3630 = pthread_self ();
  D.3631 = (long int) D.3630;
  D.3632 = omp_get_thread_num ();
  stdout.5 = stdout;
  fprintf (stdout.5, "    Thread[%lu,%lu]: Trabalhando na task 2.\n", D.3632, D.3631);
  return;
}

main._omp_fn.1 (void * .omp_data_i)
{
  /* Declaração de variáveis suprimida. */

<bb 24>:

<bb 8>:
  D.3634 = pthread_self ();
  D.3635 = (long int) D.3634;
  D.3636 = omp_get_thread_num ();
  stdout.4 = stdout;
  fprintf (stdout.4, "    Thread[%lu,%lu]: Trabalhando na task 1.\n", D.3636, D.3635);
  return;
}

main._omp_fn.0 (struct .omp_data_s.10 & restrict .omp_data_i)
{
  /* Declaração de variáveis suprimida. */

<bb 26>:

<bb 5>:
  x = .omp_data_i->x;
  id = omp_get_thread_num ();
  D.3640 = pthread_self ();
  D.3641 = (long int) D.3640;
  stdout.2 = stdout;
  fprintf (stdout.2, "  Thread[%lu,%lu]: Todas as threads executam.\n", id, D.3641);

<bb 6>:
  D.3643 = __builtin_GOMP_single_start ();
  if (D.3643 == 1)
    goto <bb 7>;
  else
   goto <bb 16>;

<bb 16>:

```

```
<bb 17>:
  return;

<bb 7>:
  D.3644 = pthread_self ();
  D.3645 = (long int) D.3644;
  stdout.3 = stdout;
  fprintf (stdout.3, "  Thread[%lu,%lu]: Antes de criar tasks.\n", id, D.3645);
  D.3647 = x > 10;

<bb 25>:
  __builtin_GOMP_task (main._omp_fn.1, 0B, 0B, 0, 1, D.3647, 0, 0B, 0);

<bb 9>:

<bb 10>:
  D.3648 = x > 20;

<bb 23>:
  __builtin_GOMP_task (main._omp_fn.2, 0B, 0B, 0, 1, D.3648, 0, 0B, 0);

<bb 12>:

<bb 13>:
  D.3649 = pthread_self ();
  D.3650 = (long int) D.3649;
  stdout.6 = stdout;
  fprintf (stdout.6, "    Thread[%lu,%lu]: Antes do taskwait.\n", id, D.3650);
  __builtin_GOMP_taskwait ();
  D.3652 = pthread_self ();
  D.3653 = (long int) D.3652;
  stdout.7 = stdout;
  fprintf (stdout.7, "    Thread[%lu,%lu]: Depois do taskwait.\n", id, D.3653);

<bb 21>:
  __builtin_GOMP_task (main._omp_fn.3, 0B, 0B, 0, 1, 1, 0, 0B, 0);

<bb 15>:
  goto <bb 16>;
}

main (int argc, char * * argv)
{
  /* Declaração de variáveis suprimida. */

<bb 2>:
  if (argc <= 1)
    goto <bb 3>;
else
    goto <bb 4>;

<bb 3>:
  D.3562 = *argv;
  stderr.0 = stderr;
  fprintf (stderr.0, "Uso: %s <x>\n", D.3562);
  exit (0);

<bb 4>:
  id = 0;
  D.3564 = argv + 8;
  D.3565 = *D.3564;
  x = atoi (D.3565);
  D.3566 = pthread_self ();
  D.3567 = (long int) D.3566;
  D.3568 = omp_get_thread_num ();
  stdout.1 = stdout;
  fprintf (stdout.1, "Thread[%lu,%lu]: Antes da região paralela.\n", D.3568, D.3567);
  .omp_data_o.15.x = x;
  __builtin_GOMP_parallel (main._omp_fn.0, &.omp_data_o.15, 8, 0);
  .omp_data_o.15 = {CLOBBER};
```

```
139   D.3596 = pthread_self ();
140   D.3597 = (long int) D.3596;
141   D.3598 = omp_get_thread_num ();
142   stdout.9 = stdout;
143   fprintf (stdout.9, "Thread[%lu,%lu]: Depois da região paralela.\n", D.3598, D.3597);
144   D.3600 = 0;
145
146 <L4>:
147   return D.3600;
148 }
```

A Figura 3.9 mostra a representação gráfica do Código 3.20 com as quatro novas funções, uma para tratar a região paralela e as outras para as três *tasks*.

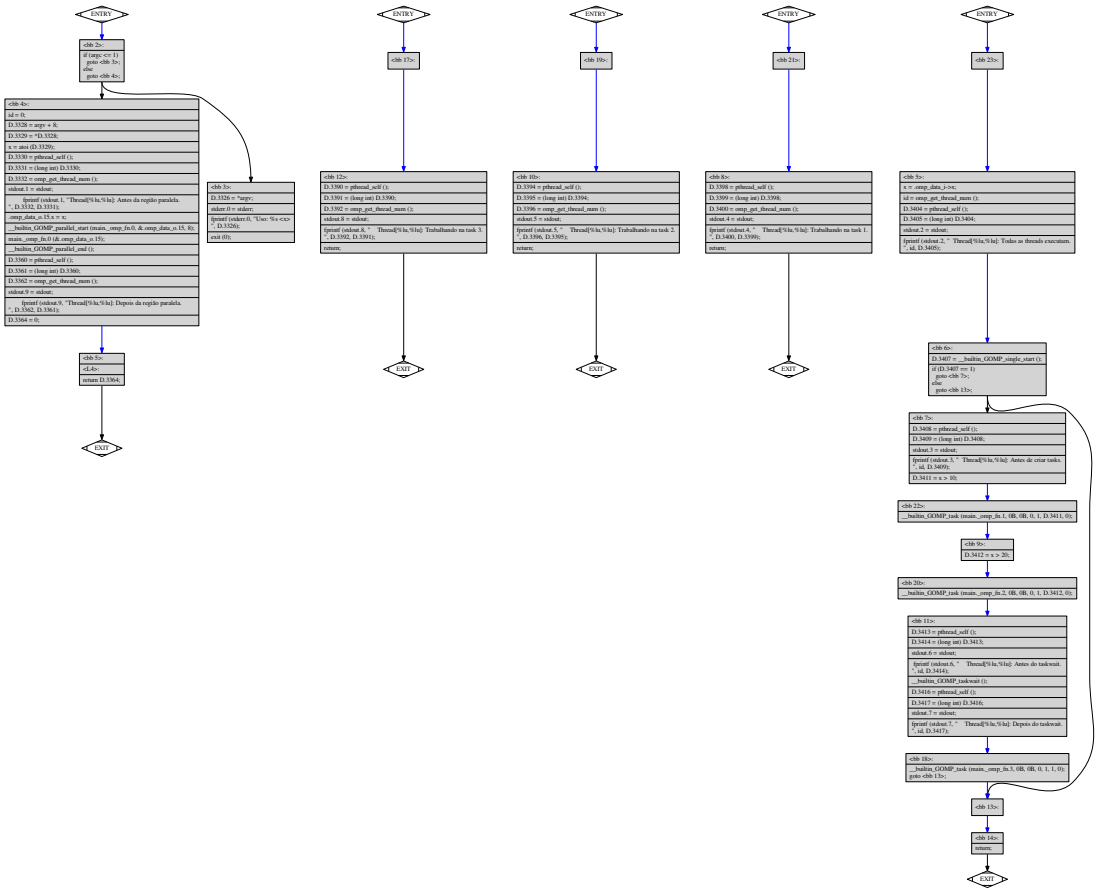


Figura 3.9. Visualização do código gerado para o construtor `task`

3.3.5. Tarefas com *loops*: construtor `taskloop`

O construtor *taskloop* é utilizado para distribuir as iterações de um ou mais laços aninhados para tarefas. Quando um construtor `taskloop` é encontrado uma nova função é criada e durante a execução subconjuntos de iterações do laço serão associados às tarefas criadas pelo construtor. O construtor `taskloop` está disponível a partir da especificação 4.5 e da `libgomp` do `GCC 6.x`. A sintaxe para uso do construtor `taskloop` é apresentada no Código 3.21.

Código 3.21. Formato do construtor *taskloop*

```
#pragma omp taskloop
{
  // for-loops.
}
```

O Código 3.22 apresenta um exemplo do uso do construtor `taskloop` aplicado a um laço. Com esse construtor é possível determinar o número de *threads* que serão criadas para a execução do laço com a cláusula `num_tasks()` e determinar o tamanho do subconjunto de iterações que cada *thread* irá executar através da cláusula `grainsize()`.

Código 3.22. Exemplo de uso do construtor *taskloop*

```
void func(){
  int i, j;
  fprintf(stdout, "Thread[%lu,%lu]: taskloop.\n", omp_get_thread_num(), (long int)
      pthread_self());
  #pragma omp taskloop num_tasks(8) private(j) grainsize(2)
  for (i = 0; i < 16; i++) {
    for (j = 0; j < i; j++) {
      fprintf(stdout, "Thread[%lu,%lu]: Trabalhando na iteração (%d,%d).\n",
          omp_get_thread_num(), (long int) pthread_self(),i,j);
    }
  }
}

int main(int argc, char *argv[]) {

  fprintf(stdout, "Thread[%lu,%lu]: Antes da Região Paralela.\n", (long int)
      omp_get_thread_num(), (long int) pthread_self());

  #pragma omp parallel num_threads(4)
  {
    #pragma omp single
    {
      fprintf(stdout, "  Thread[%lu,%lu]: Antes das tasks.\n", (long int)
          omp_get_thread_num(), (long int) pthread_self());
      #pragma omp taskgroup
      {
        #pragma omp task
        {
          fprintf(stdout, "Thread[%lu,%lu]: Trabalhando na task avulsa.\n",
              omp_get_thread_num(), (long int) pthread_self());
        }

        #pragma omp task
        {
          fprintf(stdout, "Thread[%lu,%lu]: Trabalhando na task func().\n",
              omp_get_thread_num(), (long int) pthread_self());
          func();
        }
      }
    }
  }

  fprintf(stdout, "Thread[%lu,%lu]: Depois da Região Paralela.\n", (long int)
      omp_get_thread_num(), (long int) pthread_self());

  return 0;
}
```

As funções da biblioteca `libgomp` que são utilizadas para gerar o código relacionado com o construtor *taskloop* são listadas no Quadro 3.5. As funções para a implementação de região paralela e tarefas que são utilizadas no exemplo já foram apresentadas.

Quadro 3.5: ABI `libgomp` – Funções usadas para a implementação do construtor *taskloop*

```
void GOMP_taskloop (void (*fn) (void *), void *data, void (*cpyfn) (void *, void *),
    long arg_size, long arg_align, unsigned flags, unsigned long num_tasks, int
    priority, TYPE start, TYPE end, TYPE step)
```

O Código 3.23 apresenta o código intermediário gerado pelo `GCC`. Foram criadas quatro funções, uma para tratar a região paralela (`main._omp_fn.1`) que cria um grupo de tarefas com o construtor `taskgroup` e duas *tasks* são criadas. Uma das tarefas executará a função (`main._omp_fn.2`) e outra que executará a função `main._omp_fn.2` que tem o construtor `taskloop`. A chamada à `GOMP_taskloop (func._omp_fn.0, ...)` irá executar a função `func._omp_fn.0` que tem o código do laço.

Código 3.23. Formato de código expandido para o construtor *taskloop*

```
1  func._omp_fn.0 (struct  & restrict .omp_data_i)
2  {
3    /* Declaração de variáveis suprimida. */
4
5  <bb 15>:
6
7  <bb 4>:
8    D.3591 = .omp_data_i->D.3579;
9    D.3592 = .omp_data_i->D.3581;
10   D.3593 = (int) D.3591;
11   D.3594 = (int) D.3592;
12   i = D.3593;
13
14 <bb 5>:
15   j = 0;
16
17 <bb 7>:
18   if (j < i)
19     goto <bb 6>;
20   else
21     goto <bb 8>;
22
23 <bb 8>:
24   i = i + 1;
25   if (i < D.3594)
26     goto <bb 5>;
27   else
28     goto <bb 9>;
29
30 <bb 9>:
31
32 <bb 10>:
33   return;
34
35 <bb 6>:
36   D.3597 = pthread_self ();
37   D.3598 = (long int) D.3597;
38   D.3599 = omp_get_thread_num ();
39   stdout.1 = stdout;
40   fprintf (stdout.1, "Thread[%lu,%lu]: Trabalhando na iteração (%d,%d).\n", D.3599, D
         .3598, i, j);
41   j = j + 1;
42   goto <bb 7>;
43 }
44
45 func ()
46 {
47   /* Declaração de variáveis suprimida. */
```

```
48
49  <bb 2>:
50    D.3565 = pthread_self ();
51    D.3566 = (long int) D.3565;
52    D.3567 = omp_get_thread_num ();
53    stdout.0 = stdout;
54    fprintf (stdout.0, "Thread[%lu,%lu]: taskloop.\n", D.3567, D.3566);
55    D.3574 = 0;
56    D.3573 = 16;
57    __builtin_GOMP_taskloop (func._omp_fn.0, &.omp_data_o.3, 0B, 16, 8, 1280, 8, 0, D
          .3574, D.3573, 1);
58    .omp_data_o.3 = {CLOBBER};
59    return;
60  }
61
62  main._omp_fn.3 (void * .omp_data_i)
63  {
64    /* Declaração de variáveis suprimida. */
65
66  <bb 17>:
67
68  <bb 10>:
69    D.3642 = pthread_self ();
70    D.3643 = (long int) D.3642;
71    D.3644 = omp_get_thread_num ();
72    stdout.7 = stdout;
73    fprintf (stdout.7, "Thread[%lu,%lu]: Trabalhando na task func().\n", D.3644, D.3643);
74    func ();
75    return;
76  }
77
78  main._omp_fn.2 (void * .omp_data_i)
79  {
80    /* Declaração de variáveis suprimida. */
81
82  <bb 19>:
83
84  <bb 7>:
85    D.3646 = pthread_self ();
86    D.3647 = (long int) D.3646;
87    D.3648 = omp_get_thread_num ();
88    stdout.6 = stdout;
89    fprintf (stdout.6, "Thread[%lu,%lu]: Trabalhando na task avulsa.\n", D.3648, D.3647);
90    return;
91  }
92
93  main._omp_fn.1 (void * .omp_data_i)
94  {
95    /* Declaração de variáveis suprimida. */
96
97  <bb 21>:
98
99  <bb 3>:
100
101 <bb 4>:
102   D.3650 = __builtin_GOMP_single_start ();
103   if (D.3650 == 1)
104     goto <bb 5>;
105   else
106     goto <bb 13>;
107
108 <bb 13>:
109
110 <bb 14>:
111   return;
112
113 <bb 5>:
114   D.3651 = pthread_self ();
115   D.3652 = (long int) D.3651;
116   D.3653 = omp_get_thread_num ();
```

```
117   D.3654 = (long int) D.3653;
118   stdout.5 = stdout;
119   fprintf (stdout.5, "  Thread[%lu,%lu]: Antes das tasks.\n", D.3654, D.3652);
120
121 <bb 6>:
122   __builtin_GOMP_taskgroup_start ();
123
124 <bb 20>:
125   __builtin_GOMP_task (main._omp_fn.2, 0B, 0B, 0, 1, 1, 0, 0B, 0);
126
127 <bb 8>:
128
129 <bb 9>:
130
131 <bb 18>:
132   __builtin_GOMP_task (main._omp_fn.3, 0B, 0B, 0, 1, 1, 0, 0B, 0);
133
134 <bb 11>:
135
136 <bb 12>:
137   __builtin_GOMP_taskgroup_end ();
138   goto <bb 13>;
139 }
140
141 main (int argc, char * * argv)
142 {
143   /* Declaração de variáveis suprimida. */
144
145 <bb 2>:
146   D.3601 = pthread_self ();
147   D.3602 = (long int) D.3601;
148   D.3603 = omp_get_thread_num ();
149   D.3604 = (long int) D.3603;
150   stdout.4 = stdout;
151   fprintf (stdout.4, "Thread[%lu,%lu]: Antes da Região Paralela.\n", D.3604, D.3602);
152   __builtin_GOMP_parallel (main._omp_fn.1, 0B, 4, 0);
153   D.3619 = pthread_self ();
154   D.3620 = (long int) D.3619;
155   D.3621 = omp_get_thread_num ();
156   D.3622 = (long int) D.3621;
157   stdout.8 = stdout;
158   fprintf (stdout.8, "Thread[%lu,%lu]: Depois da Região Paralela.\n", D.3622, D.3620);
159   D.3624 = 0;
160
161 <L2>:
162   return D.3624;
163 }
```

A Figura 3.10 apresenta a visualização gráfica do código gerado para o exemplo com o construtor `taskloop`.

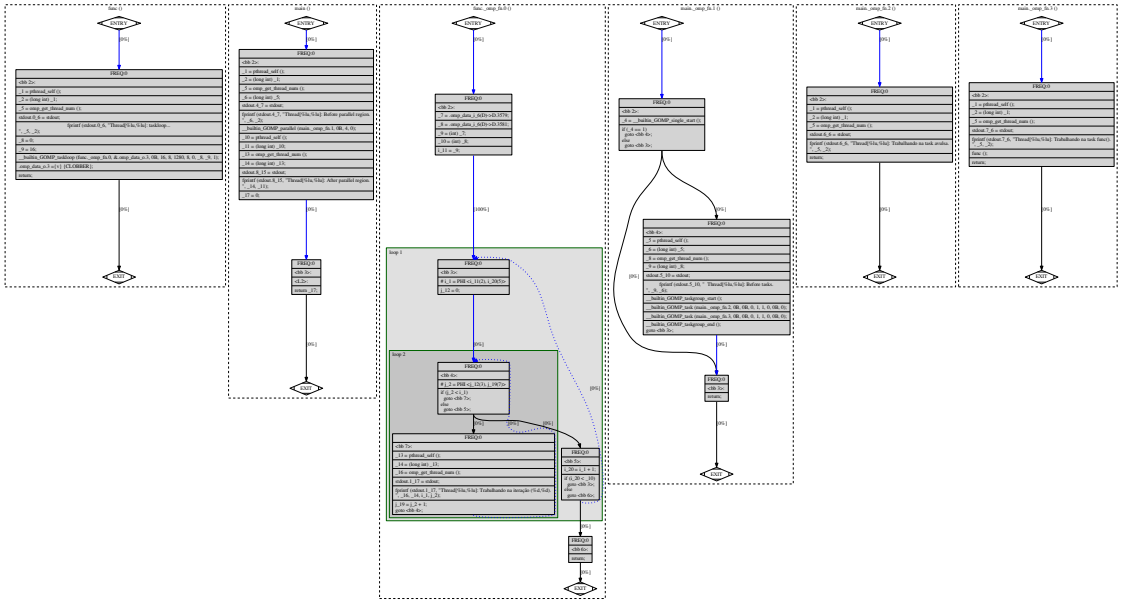


Figura 3.10. Visualização do exemplo com o construtor `taskloop`

3.3.6. Suporte à Vetorização: `construtor simd`

O construtor `simd` pode ser aplicado a um laço diretamente, indicando que múltiplas iterações do laço podem ser executadas concorrentemente usando instruções `SIMD`. Também pode ser combinado com construtores como o `for` e `taskloop` para que o conjunto de iterações seja dividido entre as *threads* e essas iterações possam ser executadas usando instruções `SIMD`. A sintaxe para uso do construtor `simd` é apresentada no Código 3.24.

Código 3.24. Formato do construtor *simd*

```
1 #pragma omp simd [clause[[,] clause] ...]new-line
2 for-loops
```

O Código 3.25 apresenta um exemplo de código que utiliza o construtor `simd` em um laço que faz a multiplicação de dois *arrays*.

Código 3.25. Exemplo de Código usando o construtor *simd*

```
1  int main(int argc, char **argv) {
2    int i;
3    double res;
4    init_array();
5
6    #pragma omp simd
7    for (i = 0; i < N; i++) {
8      h_c[i] += h_a[i] * h_b[i];
9    }
10
11   return 0;
12 }
```

O corpo do laço com instruções `SIMD` é apresentado no Código 3.26, que apresenta instruções `SIMD` como `mulsd` do `SSE2` *(Streaming SIMD Extensions)*. É importante destacar que as instruções `SIMD` utilizadas irão variar conforme as extensões disponíveis na arquitetura alvo.

Código 3.26. Código do corpo do laço com instruções `SIMD`

```
.L6:
  movl   -20(%rbp), %eax
  cltq
  movsd  h_c(,%rax,8), %xmm1
  movl   -20(%rbp), %eax
  cltq
  movsd  h_a(,%rax,8), %xmm2
  movl   -20(%rbp), %eax
  cltq
  movsd  h_b(,%rax,8), %xmm0
  mulsd  %xmm2, %xmm0
  addsd  %xmm0, %xmm1
  movq   %xmm1, %rax
  movl   -20(%rbp), %edx
  movslq  %edx, %rdx
  movq   %rax, h_c(,%rdx,8)
  addl   $1, -20(%rbp)
```

O construtor `simd` pode ser combinado com o `for`. O Código 3.27 apresenta um exemplo que utiliza os construtores `for` e `simd` combinados. Neste exemplo o mesmo formato de código visto nos exemplos anteteriores será gerado, uma função é extraída para tratar a região paralela e dentro desta função teremos o formato de código para o laço. A distribuição das iterações do laço ocorrerá normalmente conforme o algoritmo de escalonamento e o que mudará com a adição do construtor `simd` é que cada *thread* do time irá receber um *chunk* de iterações para executar e então cada uma dessas partições de iterações do laço utilizarão instruções `SIMD`.

Código 3.27. Exemplo de Código usando os construtores `for` e `simd`

```
int main(int argc, char *argv[]) {
  int i;
  /* Inicialização dos vetores. */
  init_array();

  #pragma omp parallel for simd schedule(dynamic, 32) num_threads(4)
  for (i = 0; i < N; i++) {
    h_c[i] = h_a[i] * h_b[i];
  }

  /* Resultados. */
  print_array();
  check_result();

  return 0;
}
```

Os Códigos 3.28 e 3.29 apresentam o código *assembly* gerado para a função *main* e a função para execução do laço com instruções `SIMD` em seu corpo.

Código 3.28. Código da função main

```
main:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $48, %rsp
  movl  %edi, -36(%rbp)
  movq  %rsi, -48(%rbp)
  movl  $0, %eax
  call  init_array
  leaq  -32(%rbp), %rax
  pushq $0
  pushq $32
  movl  $1, %r9d
  movl  $1048576, %r8d
  movl  $0, %ecx
  movl  $4, %edx
  movq  %rax, %rsi
  movl  $main._omp_fn.0, %edi
  call  GOMP_parallel_loop_dynamic
  addq  $16, %rsp
  movl  -32(%rbp), %eax
  movl  %eax, -4(%rbp)
  movl  $0, %eax
  call  check_result
  movl  $0, %eax
  leave
  ret
  .size main, .-main
```

Código 3.29. Código da função extraída para tratar o laço com instruções `SIMD`

```
  .type main._omp_fn.0, @function
main._omp_fn.0:
  /* Código Suprimido. */
  call  GOMP_loop_dynamic_next
  testb %al, %al
  je  .L13
.L17:
  /* Código Suprimido. */
.L15:
  cmpl  %edx, -20(%rbp)
  jge .L14
  movl  -20(%rbp), %eax
  cltq
  movsd h_a(,%rax,8), %xmm1
  movl  -20(%rbp), %eax
  cltq
  movsd h_b(,%rax,8), %xmm0
  mulsd %xmm1, %xmm0
  movl  -20(%rbp), %eax
  cltq
  movsd %xmm0, h_c(,%rax,8)
  addl  $1, -20(%rbp)
  jmp .L15
.L14:
  cmpl  $1048576, -20(%rbp)
  je  .L16
.L18:
  /* Código Suprimido. */
  call  GOMP_loop_dynamic_next
  testb %al, %al
  jne .L17
  jmp .L13
.L16:
  /* Código Suprimido. */
  jmp .L18
.L13:
  cmpl  $1048576, %ebx
  je  .L19
.L20:
  call  GOMP_loop_end_nowait
  jmp .L21
.L19:
  /* Código Suprimido. */
```

O construtor `simd` pode também ser combinado com o `taskloop`. O Código 3.30 apresenta um exemplo que utiliza os construtores `taskloop` e `simd` combinados. As iterações do laço serão executadas em paralelo por *tasks* e as iterações que cada *thread* executa podem ser transformadas em instruções `SIMD`.

Código 3.30. Exemplo de Código usando os construtores `taskloop` e `simd`

```
void func(){
  int i;

  #pragma omp taskloop simd num_tasks(4)
  for (i = 0; i < N; i++) {
    h_c[i] = h_a[i] * h_b[i];
  }
}
```

A Figura 3.11 apresenta a visualização do código para o construtor `taskloop`. Pode ser visto que toda a estrutura de execução do construtor `taskloop` e dos construtores utilizados na região paralela é criada. É em `func()` que o construtor `taskloop` foi declarado e então quando `GOMP_taskloop (func._omp_fn.0,...)` é chamada, como parâmetro é passado a função que executa o laço. O efeito que o construtor `simd` causa é perceptível somente na geração do código final.

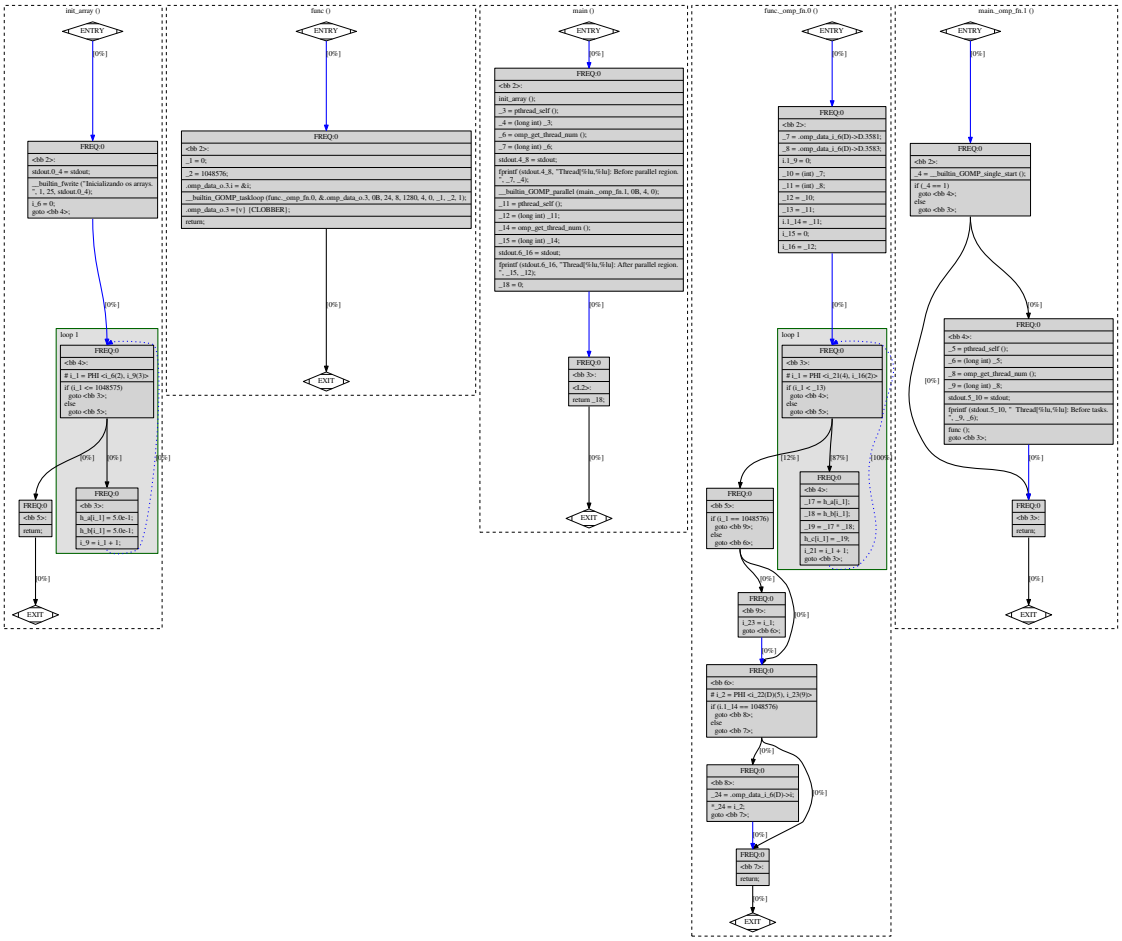


Figura 3.11. Visualização do código gerado para o construtor `taskloop` combinado com `simd`

3.3.7. *Offloading* para Aceleradores: construtor `target`

Para falarmos sobre *diretivas de compilação* para aceleradores temos que introduzir o modelo de programação para aceleradores como as `GPUs`. Para esse tipo de dispositivo acelerador é necessário definir uma função *kernel* que terá sua execução lançada no dispositivo. Um *kernel* para a soma de vetores escrito em `CUDA` [NVIDIA 2017] pode ser visto no Código 3.31.

Código 3.31. Função *kernel* em CUDA para soma de vetores

```
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  if (id < n)
    c[id] = a[id] + b[id];
}
```

O Código 3.32 mostra como os dados são declarados. Como o exemplo é de soma de vetores, temos a declaração de três *arrays* (h_a, h_b e h_c) que são alocados e representam os dados na memória principal do *host* e mais três ponteiros que são alocados na memória da GPU que irão representar os três vetores do lado do dispositivo *(device)* (d_a, d_b e d_c. Para a alocação de memória do lado *device* existe uma função cudaMalloc(...) equivalente à função malloc(...).

Código 3.32. Declaração e alocação de dados do lado *host* e do lado *device*

```
int main( int argc, char* argv[] ){
float *h_a;
float *h_b;
float *h_c;

// Declaracao dos vetores de entrada na memoria da GPU.
float *d_a;
float *d_b;
// Declaracao do vetor de saida do dispositivo.
float *d_c;

// Tamanho em bytes de cada vetor.
size_t bytes = n * sizeof(float);

// Alocacao de memoria para os vetores do host.
h_a = (float*) malloc(bytes);
h_b = (float*) malloc(bytes);
h_c = (float*) malloc(bytes);

// Alocacao de memoria para cada vetor na GPU.
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);

// Inicializacao dos arrays.
```

CUDA fornece função para realizar transferências de dados entre a memória principal e a memória do dispositivo, o Código 3.33 apresenta a cópia dos dados dos *arrays*.

Código 3.33. Transferência dos dados para a memória do dispositivo

```
// Copia dos vetores do host para o dispositivo.
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
```

A chamada à função *kernel* pode ser vista no Código 3.34. Na ativação do *kernel* a configuração da estrutura do arranjo de *threads* (grid e bloco) precisa ser definida explicitamente pelo programador. Essa configuração determina quantas *threads* serão criadas e como estarão organizadas em blocos dentro do *grid* mapeado para o dispositivo.

Código 3.34. Ativação da função *kernel*

```
1  int blockSize, gridSize;
2
3  // Numero de threads em cada bloco de threads.
4  blockSize = 1024;
5
6  // Numero de blocos de threads no grid.
7  gridSize = (int)ceil((float)n/blockSize);
8
9  // Chamada a funcao kernel.
10 vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
```

O Código 3.35 apresenta a cópia do resultado (d_c) da soma de vetores realizada no dispositivo para (h_c) na memória do *host*.

Código 3.35. Cópia do resultado e liberação da memória alocada

```
1  // Copia do vetor resultado da GPU para o host.
2  cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost );
3
4  // Liberacao da memoria da GPU.
5  cudaFree(d_a);
6  cudaFree(d_b);
7  cudaFree(d_c);
8
9  // Liberacao da memoria do host.
10 free(h_a);
11 free(h_b);
12 free(h_c);
13
14 return 0;
15 }
```

Com o exemplo de soma de vetores escrito em CUDA, no modelo clássico de execução, no qual as transferências são declaradas explicitamente, é possível ter uma ideia das operações envolvidas na execução de código em dispositivo aceleradores.

No contexto de *diretivas de compilação* o padrão OpenACC [OpenACC 2015] [OpenACC 2017] fornece um conjunto de diretivas para que da mesma maneira que podemos anotar código em OpenMP, possamos anotar código de laços e regiões paralelizáveis que podem ser transformados em *kernels* e ter sua execução acelerada por uma GPU.

Como no OpenMP as diretivas em C/C++ são especificadas usando #pragma e se o compilador não tiver suporte as anotações são ignoradas na compilação. Cada diretiva em C/C++ inicia com #pragma acc e existem construtores e cláusulas para a criação de *kernels* com base em laços, por exemplo.

O modelo de execução do OpenACC tem três níveis: *gang*, *worker* e *vector*. É um mapeamento dos elementos presentes no contexto de GPUs que utilizam CUDA, sendo gang=bloco, worker=warp, vector=threads, sendo um *warp* é um conjunto de *threads* escalonáveis num multiprocessador (SM) [Denise Stringhini 2012]. O Código 3.36 apresenta o formato das diretivas do OpenACC .

Código 3.36. Formato das diretivas do OpenACC

```
1  #pragma acc directive-name [clause [[,] clause]...] new-line
```

O Código 3.37 apresenta o exemplo soma de vetores escrito com as diretivas do OpenACC. Na função main podemos ver o construtor data que especifica através da

cláusulas `copyin` e `copyout` os dados que devem ser copiados da memória do *host* para a memória do dispositivo e vice-versa. Nesse exemplo, os *arrays* `a` e `b` serão copiados como entrada para a execução do *kernel* e o `c` será copiado após a execução do *kernel* como resultado.

Código 3.37. Exemplo de Soma de Vetores anotado com diretivas `OpenACC`

```
1  void vecaddgpu(float *restrict c, float *a, float *b, int n){
2    #pragma acc kernels for present(c,a,b)
3    for( int i = 0; i < n; ++i )
4      c[i] = a[i] + b[i];
5  }
6
7  int main( int argc, char* argv[] ){
8
9    #pragma acc data copyin(a[0:n],b[0:n]) copyout(c[0:n])
10   {
11     vecaddgpu(c, a, b, n);
12   }
13
14   return 0;
15 }
```

A saída gerada pelo compilador `pgcc` [PGROUP 2015] é apresentada no Teminal 3.6. As mensagens indicam que o laço anotado com o construtor `kernels for` foi detectado como paralelizável e no lançamento da execução do *kernel* cada bloco será criado com 256 *threads*. Também foram geradas as operações de transferências de dados.

Terminal 3.6

```
rogerio@chamonix:/src/example-openacc$ pgcc -acc -ta=nvidia,time -Minfo=accel -fast
    vectoradd.c -o vectoradd-acc-gpu
  vecaddgpu:
    12, Generating present(b[0:])
        Generating present(a[0:])
        Generating present(c[0:])
        Generating compute capability 1.0 binary
        Generating compute capability 2.0 binary
    13, Loop is parallelizable
        Accelerator kernel generated
        13, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
        CC 1.0 : 5 registers; 36 shared, 4 constant, 0 local memory bytes; 100%
            occupancy
        CC 2.0 : 5 registers; 4 shared, 48 constant, 0 local memory bytes; 100%
            occupancy
  main:
    44, Generating copyout(c[0:n])
        Generating copyin(b[0:n])
        Generating copyin(a[0:n])
rogerio@chamonix:/src/example-openacc$
```

Para *offloading* de código para dispositivos aceleradores, no `OpenMP` temos o construtor `target`. A sintaxe de uso desse construtor é apresentada no Código 3.38.

Código 3.38. Sintax do Construtor `target` do OpenMP

```
1 #pragma omp target [clause[ [ , ] clause] ... ] new-line
2 bloco-estruturado
```

O Código 3.39 apresenta os construtores `target` e `parallel for` combinados. O construtor `target` faz o mapeamento de variáveis para a memória do dispositivo e lança a execução do código no dispositivo. Uma função com o código associado ao construtor `target` é criada para ser executada no dispositivo alvo. O dispositivo alvo *(device target)* pode ser definido chamando a função `omp_set_default_device(int device_num)` com o número do dispositivo sendo passado como argumento ou definindo-se a variável de ambiente `OMP_DEFAULT_DEVICE` ou ainda usando a cláusula `device (device_num)`.

Código 3.39. Exemplo com construtor `target` combinado com laço paralelo

```
1 void vecaddgpu(float *restrict c, float *a, float *b, int n){
2   #pragma omp target device(0)
3   #pragma omp parallel for private(i)
4   for( int i = 0; i < n; ++i ){
5     c[i] = a[i] + b[i];
6   }
7 }
```

O mapeamento de dados para o dispositivo pode ser feito usando-se a cláusula `map` admitida pelo construtor `target`.

As variáveis a, b e c são mapeadas explicitamente para o dispositivo alvo. A variável n é mapeada implicitamente, pois é referenciada no código. Os tipos de mapeamento aceitos pela cláusula `map` indicam o sentido da transferência de dados a ser realizada: `map(to:vars)` *(host→device)* e `map(from:vars)` *(device→host)*. O Código 3.40 apresenta o uso da cláusula `map`. A declaração `map(to:a[0:n],b[:n])` indica que os arranjos a, b devem ser copiados para a memória do dispositivo e `map(from: c[0:n])` que o arranjo c será copiado de volta para a memória do *host*, como um resultado da execução do *kernel*.

Código 3.40. Mapeando dados para o disposito com a cláusula `map`

```
1 void vecaddgpu(float *restrict c, float *a, float *b, int n){
2   #pragma omp target map(to: a[0:n], b[:n]) map(from: c[0:n])
3   #pragma omp parallel for private(i)
4   for( int i = 0; i < n; ++i ) {
5     c[i] = a[i] + b[i];
6   }
7 }
```

O construtor `target` também permite a escolha de fazer o *offloading* do código para o dispositivo ou não, com base no tamanho dos dados, por exemplo. Isso pode ser feito utilizando a cláusula `if`, que possui o comportamento semelhante ao que vimos para região paralela. Um exemplo com a cláusula `if` para que a execução do *kernel* será lançada no dispositivo somente para tamanho de n que ultrapasse um limiar de valores é apresentado no Código 3.41.

Código 3.41. Decidindo sobre *offloading* utilizando a cláusula `if`

```
1  #define THRESHOLD 1024
2
3  void vecaddgpu(float *restrict c, float *a, float *b, int n){
4    #pragma omp target data map(to: a[0:n], b[:n]) map(from: c[0:n]) if(n>THRESHOLD)
5    {
6      #pragma omp target if(n>THRESHOLD)
7      #pragma omp parallel for if(n>THRESHOLD)
8      for( int i = 0; i < n; ++i )
9        c[i] = a[i] + b[i];
10   }
11 }
```

Ainda no Código 3.41 é possível percebermos que as transferências de dados também podem ser declaradas com o construtor `target data` que cria um novo ambiente de dados que será utilizado pelo *kernel*. A cópia dos dados também pode ser condicionada a um tamanho dos dados utilizando a cláusula `if`, e as transferências somente devem ser feitas para a memória do dispositivo se o objetivo for lançar a execução do *kernel*.

Especificar uma região de dados pode ser útil quando múltiplos *kernels* irão executar sobre os mesmos dados. O Código 3.42 apresenta uma região de dados definida com o construtor `target data` que especifica somente a cópia de volta do *array c*, pois entre as execuções das *target regions* há uma atualização dos elementos de *a* e *b* que são copiados da memória do *host* para a memória do dispositivo antes da execução de cada *kernel*.

Código 3.42. Declarando dois *kernels* para mesma região de dados

```
1  #define THRESHOLD 1048576
2
3  void vecaddgpu(float *restrict c, float *a, float *b, int n){
4    #pragma omp target data map(from: c[0:n])
5    {
6      #pragma omp target if(n>THRESHOLD) map(to: a[0:n], b[:n])
7      #pragma omp parallel for
8      for( int i = 0; i < n; ++i )
9        c[i] = a[i] + b[i];
10
11     // Reinicialização dos dados.
12     init(a,b);
13
14     #pragma omp target if(n>THRESHOLD) map(to: a[0:n], b[:n])
15     #pragma omp parallel for
16     for( int i = 0; i < n; ++i )
17       c[i] = c[i] + (a[i] * b[i]);
18     }
19   }
20 }
```

Uma outra maneira de se fazer a atualização dos dados entre as execuções dos *kernels* é utilizando o construtor `target update` que atualiza os dados de uma seção de mapeamento para o ambiente de dados do dispositivo. O Código 3.43 apresenta o código do exemplo anterior modificado para usar o construtor `target update`, que também admite a cláusula `if` que pode ser utilizada para atualizar os dados entre as execuções dos *kernels* se esses foram modificados.

Código 3.43. Atualizando os dados entre as execuções dos *kernels*

```
void vecaddgpu(float *restrict c, float *a, float *b, int n){
  int changed = 0;
  #pragma omp target data map(to: a[0:n], b[:n]) map(from: c[0:n])
  {
    #pragma omp target
    #pragma omp parallel for
    for( int i = 0; i < n; ++i )
      c[i] = a[i] + b[i];

    changed = init(a,b);

    #pragma omp target update if (changed) to(a[0:n], b[:n])

    #pragma omp target
    #pragma omp parallel for
    for( int i = 0; i < n; ++i )
      c[i] = c[i] + (a[i] * b[i]);
  }
}
```

O Código 3.44 apresenta o mesmo exemplo de soma de vetores feito em `CUDA` e em `OpenACC` no `OpenMP` utilizando o construtor `target` e suas combinações vistas nos exemplos anteriores.

Código 3.44. Atualizando os dados entre as execuções dos *kernels*

```
#define THRESHOLD 1024

float *h_a;
float *h_b;
float *h_c;
int n = 0;

/* Código Suprimido. */

void vecaddgpu(float *restrict c, float *a, float *b){
  #pragma omp target data map(to: a[0:n], b[:n]) map(from: c[0:n]) if(n>THRESHOLD)
  {
    #pragma omp target if(n>THRESHOLD)
    #pragma omp parallel for if(n>THRESHOLD)
    for( int i = 0; i < n; ++i ){
      c[i] = a[i] + b[i];
    }
  }
}

int main(int argc, char *argv[]) {
  int i;
  n = atoi(argv[1]);

  h_a = (float*) malloc(n*sizeof(float));
  h_b = (float*) malloc(n*sizeof(float));
  h_c = (float*) malloc(n*sizeof(float));

  init_array();

  vecaddgpu(h_c, h_a, h_b);

  return 0;
}
```

A Figura 3.12 apresenta a estrutura do código gerado para o exemplo do Código 3.44.

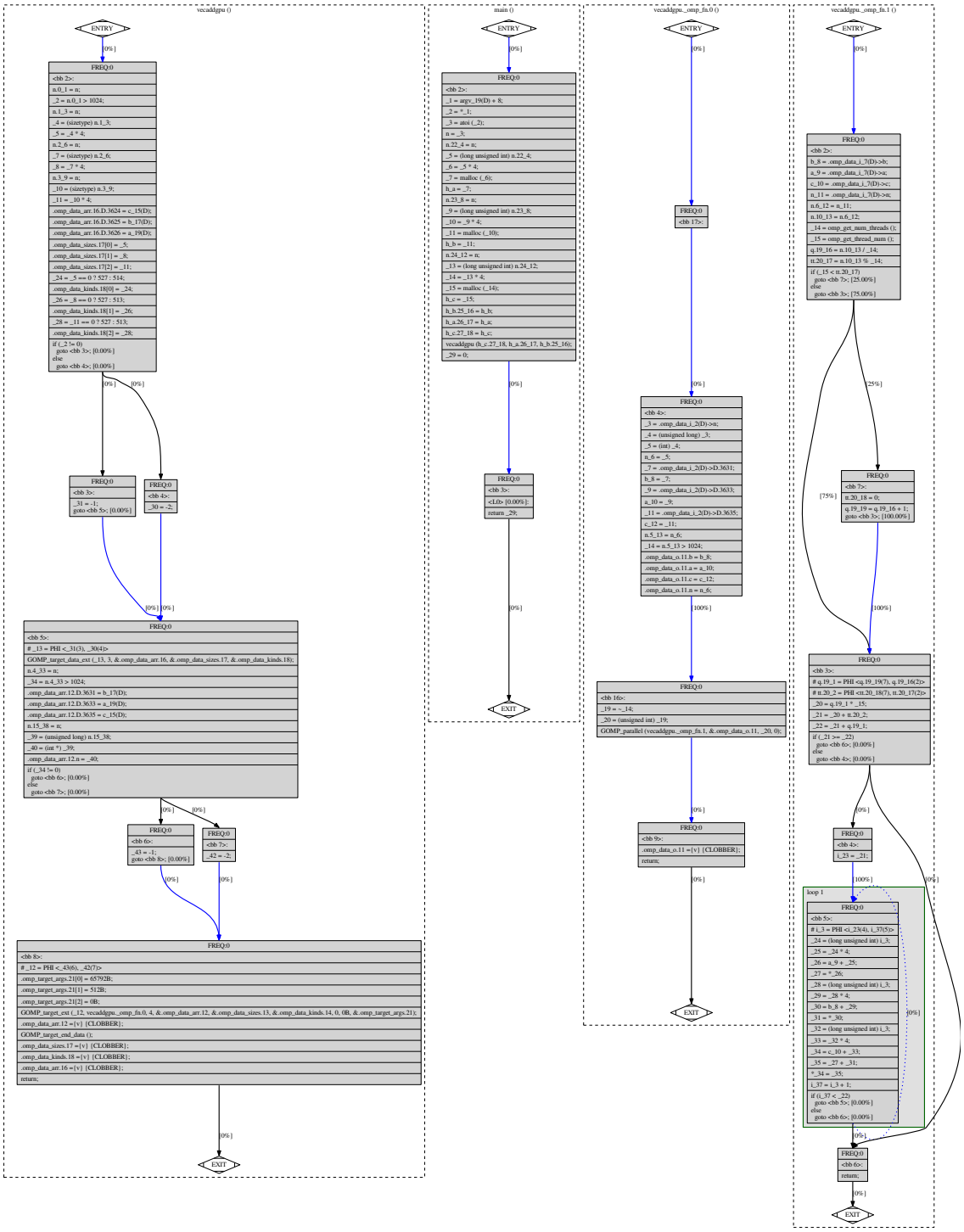


Figura 3.12. Visualização do Código intermediário gerado para o construtor `target`

As funções relacionadas com a geração de código para o construtor `target` que identificamos na `ABI` da `libgomp` estão listadas no Quadro 3.6.

Quadro 3.6: ABI `libgomp` – Funções relacionadas com o construtor *target*

```
void GOMP_parallel (void (*fn) (void *), void *data, unsigned num_threads, unsigned
    int flags)
void GOMP_target_data_ext (int device, size_t mapnum, void **hostaddrs, size_t *sizes,
     unsigned short *kinds)
void GOMP_target_end_data (void)
void GOMP_target_update (int device, const void *unused, size_t mapnum, void **
    hostaddrs, size_t *sizes, unsigned char *kinds)
void GOMP_target_ext (int device, void (*fn) (void *), size_t mapnum, void **hostaddrs
    , size_t *sizes, unsigned short *kinds, unsigned int flags, void **depend, void **
    args)
```

Como o Código 3.44 utiliza a cláusula `if` para decidir se deve ou não fazer o *offloading* para o dispositivo com base no tamanho dos dados. Executamos o exemplo soma de vetores com tamanho de dados $n = 16384$ e utilizamos a ferramenta de perfilamento, o `nvprof` para nos certificarmos que as transferências de dados e o *offloading* de código para o dispositivo seria feito. A saída da execução é apresentada no Terminal 3.7.

Terminal 3.7

```
rogerio@ragserver:~/example-target$ nvprof ./example-target.exe 16384
Inicializando os arrays.
==2381== NVPROF is profiling process 2381, command: ./example-target.exe 16384
Verificando o resultado.
Resultado Final: (16384.000000, 1.000000)
==2381== Profiling application: ./example-target.exe 16384
==2381== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
97.56%  2.6697ms         1  2.6697ms  2.6697ms  2.6697ms  vecaddgpu$_omp_fn$0
1.61%  44.160us         6  7.3600us  1.0560us  21.408us  [CUDA memcpy HtoD]
0.82%  22.496us         1  22.496us  22.496us  22.496us  [CUDA memcpy DtoH]

==2381== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
60.98%  131.59ms         1  131.59ms  131.59ms  131.59ms  cuCtxCreate
34.12%  73.631ms         1  73.631ms  73.631ms  73.631ms  cuCtxDestroy
1.24%  2.6735ms         1  2.6735ms  2.6735ms  2.6735ms  cuCtxSynchronize
1.17%  2.5168ms        22  114.40us  32.989us  999.11us  cuLinkAddData
1.07%  2.3177ms         1  2.3177ms  2.3177ms  2.3177ms  cuModuleLoadData
0.45%  961.91us         1  961.91us  961.91us  961.91us  cuLinkComplete
0.25%  544.36us         1  544.36us  544.36us  544.36us  cuLaunchKernel
0.18%  388.84us         3  129.61us  125.77us  135.86us  cuMemAlloc
0.18%  387.49us         1  387.49us  387.49us  387.49us  cuMemAllocHost
0.11%  239.10us         3  79.698us  74.062us  85.600us  cuMemFree
0.09%  186.99us         1  186.99us  186.99us  186.99us  cuMemFreeHost
0.05%  116.62us        11  10.601us     114ns  114.64us  cuDeviceGetAttribute
0.05%  105.62us         6  17.604us  7.6740us  41.930us  cuMemcpyHtoD
0.03%  69.121us         1  69.121us  69.121us  69.121us  cuMemcpyDtoH
0.02%  38.732us         1  38.732us  38.732us  38.732us  cuLinkCreate
0.00%  3.9020us         9     433ns     315ns     605ns  cuMemGetAddressRange
0.00%  2.9470us        14     210ns     144ns     398ns  cuCtxGetDevice
0.00%  1.6580us         1  1.6580us  1.6580us  1.6580us  cuModuleGetFunction
0.00%  1.4500us         3     483ns     127ns     906ns  cuDeviceGetCount
0.00%     756ns         2     378ns     310ns     446ns  cuFuncGetAttribute
0.00%     751ns         1     751ns     751ns     751ns  cuMemHostGetDevicePointer
0.00%     698ns         1     698ns     698ns     698ns  cuLinkDestroy
0.00%     639ns         1     639ns     639ns     639ns  cuModuleGetGlobal
0.00%     592ns         1     592ns     592ns     592ns  cuInit
0.00%     553ns         2     276ns     223ns     330ns  cuDeviceGet
0.00%     155ns         1     155ns     155ns     155ns  cuCtxGetCurrent
rogerio@ragserver:~/example-target$
```

Da mesma forma o exemplo foi executado com $n = 512$ e podemos verificar com o `nvprof` que nenhuma operação relacionada ao dispositivo (transferências de dados e

lançamento da execução de *kernels*) que caracterizaria o *offloading* de código foi realizada. A saída da execução é apresentada no Terminal 3.8.

Terminal 3.8

```
rogerio@ragserver:~/example-target$ nvprof ./example-target.exe 512
Inicializando os arrays.
Verificando o resultado.
Resultado Final: (512.000000, 1.000000)
======== Warning: No CUDA application was profiled, exiting
```

3.4. Aplicações

Conhecer como é o formato de código gerado e as funções da `ABI` do *runtime* do `OpenMP` pode ser útil para a construção de bibliotecas de interceptação de código via *hooking*.

Essas bibliotecas podem ser pré-carregadas para alterarem o comportamento da execução de aplicações `OpenMP`. Essa técnica pode ser utilizada para a execução de código pré ou pós chamada ao *runtime* do `OpenMP`. O que pode cobrir desde *logging*, criação de *traces* [Trahay et al. 2011], monitoramento [Mohr et al. 2002] e avaliação de desempenho ou *offloading* de código para dispositivos aceleradores.

Para criar *hooks* para funções da `libgomp` é necessário criar uma biblioteca que tenha funções com o mesmo nome das funções disponibilizadas em sua `ABI`. Uma vez que a biblioteca de *hooking* seja carregada antes da biblioteca `libgomp`, os símbolos como as chamadas para as funções do *runtime* do `OpenMP` serão ligados aos símbolos da biblioteca de interceptação. A ideia é recuperar do *linker* via *dlsym* um ponteiro para a função original para que a chamada original possa ser feita de dentro da função *proxy*. Um *hook* para a função `GOMP_parallel_start()` é apresentado no Código 3.45.

Código 3.45. Exemplo de criação de uma hook para a função *GOMP_parallel_start*

```
1 void GOMP_parallel_start (void (*fn) (void *), void *data, unsigned num_threads){
2   PRINT_FUNC_NAME;
3
4   /* Retrieve the OpenMP runtime function. */
5   typedef void (*func_t) (void (*fn) (void *), void *, unsigned);
6   func_t lib_GOMP_parallel_start = (func_t) dlsym(RTLD_NEXT, "GOMP_parallel_start");
7
8   lib_GOMP_parallel_start(fn, data, num_threads);
9 }
```

No Código 3.46 é definida uma macro para a recuperação do ponteiro para a função original, no caso ponteiros para funções do *runtime* `OpenMP`.

Código 3.46. Definição de macro para recuperar o ponteiro para a função original

```
1 #define GET_RUNTIME_FUNCTION(hook_func_pointer,func_name)    \
2   do {                                                       \
3     if (hook_func_pointer) break;                            \
4     void *__handle = RTLD_NEXT;                              \
5     hook_func_pointer = (typeof(hook_func_pointer)) (uintptr_t) dlsym(__handle,
          func_name);                                  \
6     PRINT_ERROR();                                           \
7   } while(0)
```

O Código 3.47 apresenta a mesma função proxy usando a macro para recuperar o ponteiro para a função original. Além disso, apresenta a ideia de chamadas de funções

para executar algum código antes (`PRE_`) ou algum código depois (`POST_`).

Código 3.47. Definição de macro para recuperar o ponteiro para a função original

```
void GOMP_parallel_start (void fn)(void, void *data, unsigned num_threads){
  PRINT_FUNC_NAME;

  /* Retrieve the OpenMP runtime function. */
  GET_RUNTIME_FUNCTION(lib_GOMP_parallel_start, "GOMP_parallel_start");

  /* Código a ser executado antes. */
  PRE_GOMP_parallel_start();

  /* Chamada à função original. */
  lib_GOMP_parallel_start(fn, data, num_threads);

  /* Código a ser executado depois. */
  POST_GOMP_parallel_start();
}
```

O Código 3.48 apresenta a função *proxy* para a função de inicialização de laço com escalonamento do tipo *dynamic*.

Código 3.48. Definição de macro para recuperar o ponteiro para a função original

```
void GOMP_parallel_loop_dynamic_start (void fn)(void, void *data,
unsigned num_threads, long start, long end,
long incr, long chunk_size){
  PRINT_FUNC_NAME;

  /* Retrieve the OpenMP runtime function. */
  GET_RUNTIME_FUNCTION(lib_GOMP_parallel_loop_dynamic_start, "
      GOMP_parallel_loop_dynamic_start");

  /* Código a ser executado antes. */
  PRE_GOMP_parallel_loop_dynamic_start();

  /* Chamada à função original. */
  lib_GOMP_parallel_loop_dynamic_start(fn, data, num_threads, start, end, incr,
      chunk_size);

  /* Código a ser executado depois. */
  POST_GOMP_parallel_loop_dynamic_start();
}
```

A função *proxy* para a função de término de laços de repetição é apresentada no Código 3.49.

Código 3.49. Definição de macro para recuperar o ponteiro para a função original

```
void GOMP_loop_end (void){
  PRINT_FUNC_NAME;

  /* Retrieve the OpenMP runtime function. */
  GET_RUNTIME_FUNCTION(lib_GOMP_loop_end, "GOMP_loop_end");

  /* Código a ser executado antes. */
  PRE_GOMP_loop_end();

  /* Chamada à função original. */
  lib_GOMP_loop_end();

  /* Código a ser executado depois. */
  POST_GOMP_loop_end();
}
```

O Código 3.50 apresenta a função *proxy* capaz de interceptar a função de criação de *tasks*. Podendo da mesma forma que outras funções de interceptação, executar um código antes e outro depois da chamada à função original.

Código 3.50. Definição de macro para recuperar o ponteiro para a função original

```
1  void GOMP_task (void fn)(void), void *data, void cpyfn)(void *, void),
2  long arg_size, long arg_align, bool if_clause, unsigned flags,
3  void **depend){
4    PRINT_FUNC_NAME;
5
6    /* Retrieve the OpenMP runtime function. */
7    GET_RUNTIME_FUNCTION(lib_GOMP_task, "GOMP_task");
8
9    /* Código a ser executado antes. */
10   PRE_GOMP_task();
11
12   /* Chamada à função original. */
13   lib_GOMP_task(fn, data, cpyfn, arg_size, arg_align, if_clause, flags, depend);
14
15   /* Código a ser executado depois. */
16   POST_GOMP_task();
17 }
```

Parte da saída da execução do exemplo do uso do construtor `task` com a biblioteca de interceptação é apresentada no Código 3.9.

Terminal 3.9

```
rogerio@chamonix:/src/simple-omp-hook/tests/parallel-region-with-tasks$ LD_PRELOAD=./
    libhookomp.so ./parallel-region-with-tasks.exe 1024
Thread[0,139859015989184]: Antes da região paralela.
TRACE: [ hookomp.c:0000753] Calling [GOMP_parallel_start()]
TRACE: [prepostfunctions.c:0000024] Calling [PRE_GOMP_parallel_start()]
TRACE: [prepostfunctions.c:0000033] Calling [POST_GOMP_parallel_start()]
Thread[1,139858992330496]: Todas as threads executam.
Thread[2,139858983937792]: Todas as threads executam.
TRACE: [ hookomp.c:0000987] Calling [GOMP_single_start()]
TRACE: [ hookomp.c:0000987] Calling [GOMP_single_start()]
...
Thread[1,139858992330496]: Antes de criar tasks.
TRACE: [ hookomp.c:0000830] Calling [GOMP_task()]
TRACE: [ hookomp.c:0000771] Calling [GOMP_parallel_end()]
TRACE: [prepostfunctions.c:0000029] Calling [PRE_GOMP_parallel_end()]
TRACE: [prepostfunctions.c:0000062] Calling [PRE_GOMP_task()]
TRACE: [prepostfunctions.c:0000067] Calling [POST_GOMP_task()]
Thread[3,139858975545088]: Trabalhando na task 1.
TRACE: [ hookomp.c:0000830] Calling [GOMP_task()]
TRACE: [prepostfunctions.c:0000062] Calling [PRE_GOMP_task()]
TRACE: [prepostfunctions.c:0000067] Calling [POST_GOMP_task()]
Thread[2,139858983937792]: Trabalhando na task 2.
Thread[1,139858992330496]: Antes do taskwait.
TRACE: [ hookomp.c:0000848] Calling [GOMP_taskwait()]
Thread[1,139858992330496]: Depois do taskwait.
TRACE: [ hookomp.c:0000830] Calling [GOMP_task()]
TRACE: [prepostfunctions.c:0000062] Calling [PRE_GOMP_task()]
TRACE: [prepostfunctions.c:0000067] Calling [POST_GOMP_task()]
Thread[0,139859015989184]: Trabalhando na task 3.
TRACE: [prepostfunctions.c:0000037] Calling [POST_GOMP_parallel_end()]
Number of parallel regions: 1
Number of tasks: 3
Number of Finished tasks: 3
Thread[0,139859015989184]: Depois da região paralela.
rogerio@chamonix:/src/simple-omp-hook/tests/parallel-region-with-tasks$
```

3.5. Considerações Finais

Pelo fato do `OpenMP` ser um padrão amplamente utilizado em aplicações paralelas para sistemas *multicore* e com aceleradores, é importante conhecer sobre o seu funcionamento. É fundamental ter conhecimentos que vão além do uso das diretivas, ter ideia de como o código final é gerado, do seu formato e de como é executado. Pois em alguns casos não é simplesmente anotar o código, é necessário saber se o mesmo é paralelizável, um laço de repetição é um bom exemplo disso.

Mas ainda assim o uso de diretivas de compilação tem uma grande vantagem com relação ao uso de bibliotecas para criação de aplicações *multithreading* como a `pthreads`. A quantidade de código a ser escrito inserindo anotações nos devidos lugares é muito menor. Sem a preocupação de alterar o código de maneira que não seja mais compilado pelas ferramentas originais, pois se o compilador não reconhecer as diretivas elas são simplesmente ignoradas.

Existem diversas outras diretivas de compilação do `OpenMP` que não foram abordadas neste texto, mas que podem ser consultadas na documentção do `OpenMP` e utilizadas com outras implementações e ferramentas de compilação [OpenMP Site 2017] [OpenMP-ARB 2015].

Agradecimentos

O material desse minicurso foi preparado no âmbito dos projetos de Extensão *"Escola de Computação Paralela"* (UTFPR DIREC Nº 028/2017) e de Pesquisa *"Estudo Exploratório sobre Técnicas e Mecanismos para Paralelização Automática e Offloading de Código em Sistemas Heterogêneos"* (UTFPR PDTI Nº 916/2017).

Referências

[Blumofe et al. 1995] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216.

[Dagum and Menon 1998] Dagum, L. and Menon, R. (1998). OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

[Denise Stringhini 2012] Denise Stringhini, Rogério Aparecido Gonçalves, A. G. (2012). Introdução à Computação Heterogênea. In de Souza; Renata Galante; Roberto Cesar Junior; Aurora Pozo, L. C. A. A. F., editor, *XXXI Jornadas de Atualização em Informática (JAI)*, volume 21 of *1*, chapter 7, pages 262–309. SBC, 1 edition. http://www.lbd.dcc.ufmg.br/bdbcomp/servlet/Trabalho?id=12580.

[GCC 2015] GCC (2015). GCC, the GNU Compiler Collection.

[GNU Libgomp 2015a] GNU Libgomp (2015a). GNU libgomp, GNU Offloading and Multi Processing Runtime Library documentation (Online manual).

[GNU Libgomp 2015b] GNU Libgomp (2015b). GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation. Technical report, GNU.

[GNU Libgomp 2016] GNU Libgomp (2016). GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation. Technical report, GNU libgomp.

[Gonçalves et al. 2016] Gonçalves, R., Amaris, M., Okada, T., Bruel, P., and Goldman, A. (2016). Openmp is not as easy as it appears. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5742–5751.

[Intel 2016a] Intel (2016a). Intel® OpenMP* Runtime Library Interface. Technical report, Intel. OpenMP* 4.5, https://www.openmprtl.org.

[Intel 2016b] Intel (2016b). OpenMP* Support. https://software.intel.com/pt-br/node/522678.

[Lattner and Adve 2004] Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, number c in CGO '04, pages 75–86, Palo Alto, California. IEEE Computer Society.

[LLVM OpenMP 2015] LLVM OpenMP (2015). OpenMP®: Support for the OpenMP language.

[Mohr et al. 2002] Mohr, B., Malony, A. D., Shende, S., and Wolf, F. (2002). Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*, 23(1):105–128.

[Nichols et al. 1996] Nichols, B., Buttlar, D., and Farrell, J. P. (1996). *Pthreads programming - a POSIX standard for better multiprocessing.* O'Reilly.

[NVIDIA 2017] NVIDIA (2017). CUDA C Best Practices Guide. Technical report, NVIDIA. DG-05603-001_v9.0, Version v9.0.176, http://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[OpenACC 2015] OpenACC (2015). OpenACC Application Programming Interface. Version 2.5. http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf.

[OpenACC 2017] OpenACC (2017). OpenACC – More Science, Less Programming. http://www.openacc.org/.

[OpenMP 2017] OpenMP (2017). OpenMP Compilers. http://www.openmp.org/resources/openmp-compilers/.

[OpenMP-ARB 2011] OpenMP-ARB (2011). OpenMP Application Program Interface Version 3.1. Technical report, OpenMP Architecture Review Board (ARB).

[OpenMP-ARB 2013] OpenMP-ARB (2013). OpenMP Application Program Interface Version 4.0. Technical report, OpenMP Architecture Review Board (ARB).

[OpenMP-ARB 2015] OpenMP-ARB (2015). OpenMP Application Program Interface Version 4.5. Technical report, OpenMP Architecture Review Board (ARB). Version 4.5.

[OpenMP Site 2017] OpenMP Site (2017). OpenMP® – Enabling HPC since 1997: The OpenMP API specification for parallel programming.

[PGROUP 2015] PGROUP (2015). PGI Accelerator Compilers with OpenACC Directives.

[Trahay et al. 2011] Trahay, F., Rue, F., Faverge, M., Ishikawa, Y., Namyst, R., and Dongarra, J. (2011). EZTrace: a generic framework for performance analysis. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Newport Beach, CA, United States. Poster Session.

Capítulo 4

Programação Concorrente em Erlang

Alexandre Ponce de Oliveira, Paulo Sérgio Lopes de Souza e Simone do Rocio Senger de Souza

Abstract

Most programs written are sequential, in which the instructions are sequentially executed. In certain applications, this programming underutilizes hardware resources. In contrast to sequential programs, concurrent programming considers the use of instructions that can be executed in parallel in order to improve application performance and increase the utilization of the available computational resources. Erlang is a functional language that has support for concurrent programming. This chapter presents the main characteristics of the language, the compiling and executing process the codes and examples of sequential and concurrent programs.

Resumo

A maioria dos programas escritos são sequenciais, no qual as instruções são executadas sequencialmente. Em determinadas aplicações essa programação subutiliza os recursos de hardware. Em contrapartida aos programa sequenciais, a programação concorrente considera o uso de instruções que podem ser executadas ao mesmo tempo, de modo a melhorar o desempenho da aplicação e aumentar a utilização dos recursos computacionais disponíveis. Erlang é uma linguagem funcional que possui suporte para programação concorrente. Este capítulo apresenta as principais características da linguagem, o processo de compilação e execução dos códigos e exemplos de programas sequenciais e concorrentes.

4.1. Introdução

As aplicações computacionais atuam em diversas áreas do mercado. Essa diversidade estimulou a criação de vários paradigmas de linguagens de programação com características distintas [9]. Tal diversidade permite a categorização das linguagens de programação em imperativas, orientadas a objetos, lógicas e funcionais.

As linguagens imperativas foram projetadas em função da arquitetura de Von Neumann, onde dados e instruções são armazenados em memória e posteriormente são requisitados pela Central Processing Unit (CPU) para execução. As linguagens imperativas podem ser caracterizadas pelo estado de um programa, o qual é mantido por variáveis. Os valores das variáveis são modificados através de instruções de atribuição e tais mudanças causam mudança no estado de um programa [9]. Alguns exemplos de linguagens imperativas são: C, Pascal, Fortran, Basic, Algol, Cobol, Ada, Python e Assembly.

Linguagens de programação orientadas a objetos têm como foco o uso de abstração de dados, onde o processamento com objetos de dados é encapsulado e o acesso a esses objetos é protegido. Tais linguagens utilizam também o conceito de herança que potencializa a reutilização de softwares e a vinculação dinâmica de métodos, flexibilizando o uso da herança [9]. Exemplos dessas linguagens são: JAVA, C++, C#, Python, Ruby e Smalltalk.

Linguagens de programação lógicas são sistemas de notação simbólica para escrever instruções lógicas com algoritmos específicos e utilizar regras de inferência lógica para produzir resultados. Os objetos em proposições lógicas de programação são representados por constantes ou variáveis [9] [8]. Prolog é a linguagem lógica mais conhecida e utilizada [11].

Linguagens funcionais são baseadas em funções matemáticas que são a correspondência entre dois conjuntos. Uma definição de função é descrita por uma expressão ou por uma tabela. As funções normalmente são aplicadas para um elemento específico do conjunto. Uma característica importante das funções matemáticas é o uso de recursão e expressões condicionais para controlar a ordem em que as expressões de correspondência são avaliadas [9]. Outras características igualmente importantes são: dados imutáveis, *higher-order functions* (funções de ordem superior), *lazy evaluation* e *pattern matching* (correspondência de padrão) [1]. Alguns exemplos de linguagens funcionais são: Lisp, Scheme, Scala, ML, Miranda, Haskell, Elixir e Erlang.

Independente do paradigma de programação utilizado, a grande maioria dos programas escritos são sequenciais, no qual, obviamente, utilizam-se primitivas que são executadas sequencialmente. Para determinadas aplicações essa programação implica em processamento lento e subutilização dos recursos de hardware [31, 65, 70]. A programação concorrente quebra esse paradigma sequencial e considera o uso de primitivas executadas ao mesmo tempo (em tese - a depender dos recursos de processamento para isso), de modo a melhorar o desempenho da aplicação e aumentar a utilização dos recursos computacionais disponíveis.

O desenvolvimento de programas concorrentes requer ferramentas eficazes para iniciar/finalizar processos concorrentes e para permitir a interação entre tais processos. Para isso, podem ser utilizadas extensões para linguagens sequenciais, as quais normalmente são implementadas por meio de bibliotecas que permitem o uso de novas primitivas, voltadas à geração de processos e à interação entre eles. Exemplos dessas extensões são as implementações do padrão MPI, PThreads e OpenMP. As linguagens de programação concorrente possuem os comandos necessários aos algoritmos concorrentes já embutidos na própria linguagem. CSP, ADA, Java, Occam, Haskell, Scala, Go Land, Elixir e Erlang são exemplos dessas linguagens. [31, 65, 70, 79, 88].

Considerando o contexto de programação concorrente, este capítulo aborda a linguagem de programação funcional Erlang. Erlang foi projetada para programação concorrente, em tempo real e sistemas distribuídos tolerantes a falhas [3]. O objetivo principal na criação da linguagem Erlang foi melhorar a programação das aplicações telefônicas que são atualizadas em tempo de execução e, consequentemente, não permitiam a perda do serviço durante a atualização do código [6].

Na próxima seção (Seção 1.2) são descritas as principais características da linguagem Erlang, o ambiente de programação e as formas de compilação e execução dos programas Erlang. Alguns conceitos são exemplificados por meio de exemplos de códigos para uma melhor ilustração. Na Seção 1.3 são ilustrados diversos exemplos de códigos Erlang que exploram os aspectos sequenciais, e principalmente, programas concorrentes. A Seção 1.4 apresenta as considerações finais do capítulo.

4.2. Linguagem funcional Erlang

Erlang é uma linguagem funcional desenvolvida pela Ericsson Computer Science Laboratory, a partir de 1986. Erlang teve como objetivo principal o desenvolvimento de aplicações na área de telecomunicações com características de distribuição, em grande escala e em softwares de controle de tempo real [5]. A comunicação entre processos não utiliza memória compartilha, é realizada através de trocas de mensagens [3].

4.2.1. História de Erlang

De acordo com Armstrong [2], a linguagem Erlang começou como um meta-interpretador escrito em Prolog que adicionou o recurso de execução paralela de processos, habilidades para concorrência e melhorou o tratamento de erros.

Esses novos recursos fizeram a linguagem crescer e ser nomeada como Erlang, cujo nome é uma homenagem ao matemático dinamarquês Agner Krarup Erlang (1878-1929), criador da teoria de processos estocásticos em equilíbrio estatístico que foi muito utilizada na indústria de telecomunicações [5].

A linguagem Erlang foi influenciada pela combinação de linguagens de sistemas concorrentes como Ada, Modula e Chill, linguagens funcionais como ML e Miranda e a linguagem lógica Prolog Armstrong:2007:HE:1238844.1238850 [6].

Por volta de 1988, foi construída uma máquina abstrata chamada de Joe's Abstract Machine (JAM), a qual contava com primitivas para concorrência e manipulação de exceções. Nesta época o grupo cresceu para três pessoas, Mike Williams escreveu a máquina virtual para executar os códigos gerados pela JAM, Joe ARMSTRONG escreveu o compilador e Robert Virding escreveu as bibliotecas de suporte. Em 1990 Claes Vikström se juntou ao grupo e melhorou o desempenho da distribuição de processos para a linguagem [2].

Em 1992, em função de um problema de desempenho, Bogumil Husman desenvolveu a Bogdans Erlang Abstract Machine (BEAM), a qual substituiu a JAM em 1997 e tornou-se o sistema base para desenvolvimento de novos produtos em Erlang [**?**] [2].

Para o desenvolvimento de aplicações completas era necessário um ambiente de desenvolvimento completo. Em 1996 foi lançado um framework que contemplava todo o

ambiente de execução, desde a codificação dos programas até sua execução, chamado de Open Telecom Platform (OTP). Em 1998 este framework tornou-se open source [2].

4.2.2. Características de Erlang

A linguagem Erlang, desde a sua construção, tem como foco o desenvolvimento de aplicações concorrentes, de tempo real com características de alta disponibilidade e tolerantes a falhas [5].

A seguir são detalhadas as principais características de Erlang encontradas em [4] [6]:

Construção de alto nível: Erlang é uma linguagem declarativa e, por princípio, tenta descrever o que deve ser calculado, ao invés de dizer como o valor é calculado. A definição de uma função usa a correspondência de padrão (pattern matching) para selecionar um caso entre diferentes casos e obter componentes a partir de estruturas de dados complexas. Erlang não só combina padrão sobre os dados de alto nível, mas também uma sequência de bits que permitem funções de manipulação de protocolos. O código descrito na Figura 4.1 tem a definição de uma forma (no caso, um quadrado ou um círculo) e, de acordo com o tipo de forma que a função recebe, corresponde à cláusula correta da função e retorna a área correspondente.

```
area({quadrado, Lado})  ->   Lado * Lado;
area({circulo, Raio})   ->   math:pi() * Raio * Raio.
```

Figura 4.1. Exemplo de Função que utiliza Pattern Matching

Guards são muito utilizados em Erlang e considerados como uma extensão de pattern matching. Guards são condições avaliadas e que precisam ser satisfeitas para continuar a executar o conteúdo de uma cláusula. A Figura 4.2 mostra a aplicação de guards em um programa que calcula o fatorial de um número.

```
fatorial(Num) when Num == 0 -> 1;
fatorial(Num) when Num > 0 -> Num * fatorial(Num - 1).
```

Figura 4.2. Exemplo de Função que utiliza Guards

No código da Figura 4.2 a primeira condição (guard) só é executada quando o valor da variável Num for igual a zero, para valores maiores que zero é executada a segunda condição.

A funcionalidade conhecida como "Compreensões de lista"combina os geradores de lista com os filtros, retornando outra lista que contém os elementos da primeira, após a aplicação dos filtros. A Figura 4.3 mostra alguns exemplos de compreensões de lista.

O primeiro exemplo da Figura 4.3 realiza um filtro para mostrar apenas os elementos da lista que são menores que 4 (comando 2), no segundo e terceiro exemplos mostram, respectivamente, os números pares (comando 3) e números impares (comando 4).

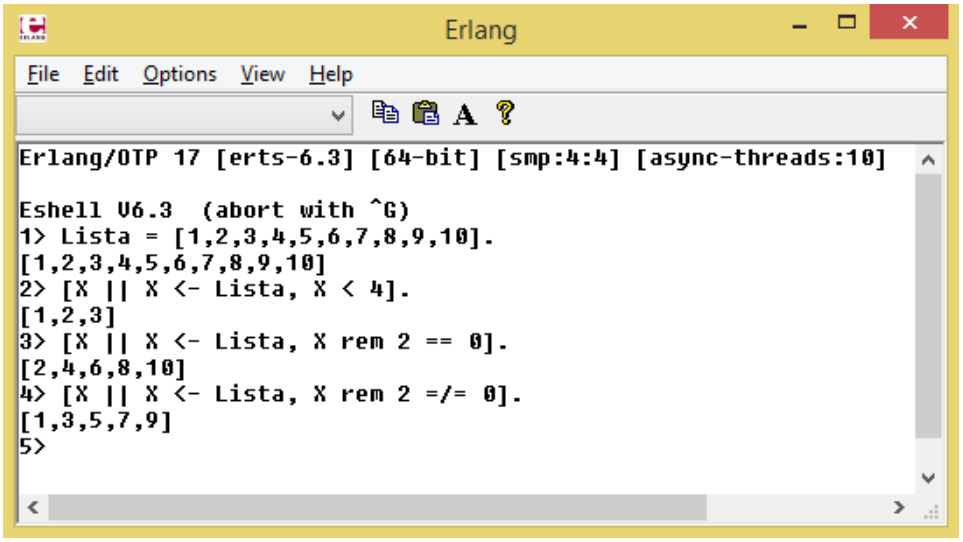


Figura 4.3. Exemplo de uso de compreensões de lista

Processos Concorrentes: A concepção de Erlang é fundamentada pelo conceito de concorrência. A linguagem Erlang não fornece threads para compartilhar memória, portanto cada processo Erlang é executado em seu próprio espaço de memória e possui sua própria heap e stack. Assim processos não interferem uns com os outros, e consequentemente não há condição de disputa. Os processos em Erlang são considerados leves, pois a Virtual Machine (VM) Erlang não cria uma thread de sistema operacional para cada processo criado. A própria VM cria, escalona e gerencia os processos, independentemente do sistema operacional. Tal procedimento resulta na criação de processos na casa dos microssegundos.

Passagem de Mensagem: Processos em Erlang comunicam-se através de passagem de mensagens. Tais mensagens podem ser qualquer valor de dados em Erlang. A operação de envio (*send*) é bloqueante, pois o processo continua o processamento apenas quando for realizada a cópia da mensagem do buffer da aplicação para um buffer do sistema operacional. Cada processo possui uma caixa de mensagens que armazena as mensagens recebidas e são obtidas de forma seletiva e bloqueantes [7]. Em ambientes distribuídos a ordem de recebimento das mensagens depende das condições do ambiente da rede. A Figura 4.4 mostra um exemplo de envio de mensagem entre dois processos.

No exemplo da Figura 4.4, o processo *cliente* envia uma mensagem ao processo *calc_area* contendo o seguinte conteúdo: *{quadrado, Lado}*. O operador ! é equivalente à primitiva send . O processo *calc_area*, por sua vez, executa a primitiva *receive* e fica aguardando o recebimento de uma mensagem com o conteúdo *{quadrado, Lado}*. Ao receber uma mensagem, o processo calc_area envia uma mensagem de volta ao processo cliente com o conteúdo *{resp, Lado*Lado}*.

Da mesma forma que na criação de processos, a troca de mensagens fica em torno dos microssegundos, pois todo o contexto envolvido resume-se a cópia de dados a partir do espaço de memória de um processo para o outro.

Desta forma, Erlang reduz o tempo para a criação de processos e para a troca de

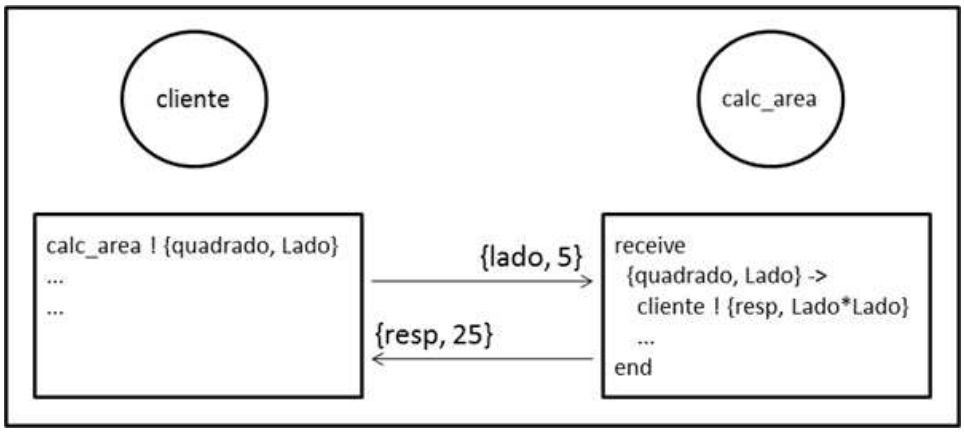


Figura 4.4. Exemplo de comunicação entre processos

mensagens, em relação às linguagens que utilizam memória compartilhada, semáforos e threads no nível de sistema operacional.

Atualização de código com o sistema em execução: Aplicações de telecomunicações necessitam de alta disponibilidade e as atualizações não podem interromper seu funcionamento. Para tanto, Erlang permite a substituição do código fonte com o sistema em execução. O sistema runtime mantém uma tabela global contendo os endereços para todos os módulos carregados. Quando uma nova versão de módulo é atualizada, os novos processos utilizam essa versão atualizada, enquanto os processos anteriores continuam a utilizar a versão antiga até terminarem. A linguagem permite a execução simultânea de duas versões do módulo em um sistema, assim torna-se possível a melhoria ou correção de bugs de um sistema sem interrupção.

Ambiente Distribuído: As aplicações Erlang podem ser executadas em um ambiente distribuído de forma transparente. Uma instância de uma VM Erlang é denominada nó. Um ou mais computadores podem executar vários nós independentemente de sua arquitetura de hardware ou sistema operacional. Processos podem ser criados em nós de outras máquinas, pois é possível o registro do processo na VM Erlang. As mensagens podem ser passadas entre os processos independentemente se eles estão no mesmo ou em diferentes nós. A Figura 4.5 contém um trecho de código que cria processos em dois nós que pertencem a hosts diferentes.

```
No1 = 'node1@urano',
No2 = 'node2@marte',
global:register_name(pid0, spawn(No1,calc,soma,[8,18])),
global:register_name(pid1, spawn(No2,calc,sub,[10,8])).
```

Figura 4.5. Exemplo de criação de processos em hosts diferentes

No código da Figura 4.5 foram criados dois processos: um registrado com o nome *pid0* e o outro com *pid1*, respectivamente nos hosts, *urano* e *marte*.

Tolerância a Falhas: Erlang possui um conjunto de bibliotecas que suportam o

conceito de processos supervisores e trabalhadores e detecção de exceções e mecanismos de recuperação. Os processos criam *links* entre si para receberem notificações, em forma de mensagens, se o processo remoto, ao qual este processo foi ligado, terminar. Essa monitoração é possível mesmo se o processo remoto estiver em execução em uma máquina diferente da rede. Caso um processo falhe, o mesmo pode ser reiniciado pelo seu processo supervisor. A Figura 4.6 mostra uma função que faz a criação de um novo processo (linha) por meio da primitiva *spawn_link*. Essa primitiva cria um *link* entre os processos.

```
1  start_pong() ->
2  No1 = 'node1@urano',
3  global:register_name(pong, spawn_link(No1, m_pong, pong, [])),
4  process_flag(trap_exit, true),
5  receive
6   {'EXIT', Pid, Reason} ->
7     io:format("Proc finish(~p). Reason: ~p~n", [Pid, Reason])
8  end.
```

Figura 4.6. Exemplo de criação de processos com links

Como pode ser observado na Figura 4.6, um processo criado por meio da primitiva *spawn_link* envia um sinal para o processo criador quando ele finaliza de forma normal ou anormal. Esse sinal pode ser convertido em forma de mensagem com o uso do flag *trap_exit* definido como *true* (linha 4). Neste caso, os sinais de saída (exit) são convertidos em mensagens e podem ser monitorados pelo processo criador, por meio da primitiva *receive* (linha 5). Para receber um sinal de *exit*, o qual significa a finalização normal do processo criado, o processo criador aguarda pela tupla *{'EXIT', From, Reason}* conforme cláusula *receive* na linha 6.

No exemplo da Figura 4.6 apenas uma mensagem foi mostrada na tela (linha 7). Entretanto, se a aplicação necessita de tolerância a falhas pode ser tomada alguma ação específica, por exemplo, criar um novo processo.

Gerência de Memória: A VM Erlang gerencia automaticamente a memória, desobrigando o programador de se preocupar com a alocação que é feita pelo sistema runtime e com a liberação de memória que é feita pelo coletor de lixo (garbage collector). A área de memória de cada processo é coletada separadamente, quando um processo termina, sua memória é simplesmente recuperada.

A coleta separada resulta em tempos de coleta de lixo menores, o que contribui para se alcançar o cumprimento de prazos que aplicações de tempo real necessitam. Caso a memória de todos os processos seja coletada ao mesmo tempo, sem um coletor de memória distribuído que possa fazer a coleta de lixo de maneira incremental, o sistema poderá parar por um longo período.

4.2.3. Tipos de Dados em Erlang

Erlang, por não ser uma linguagem puramente funcional, é uma linguagem com definição dinâmica de tipos e com semântica funcional, em grande parte, livre de efeitos colaterais (i.e. mudança de estado). As variáveis não são declaradas, o valor é vinculado na primeira ocorrência e, uma vez atribuído um valor, este não se altera mais. São oito tipos de dados

simples e dois tipos de dados compostos existentes em Erlang [3] [6]:

Inteiros: podem ser positivos ou negativos e representar números em bases diferentes de 10. Para isso a notação Base#Valor é utilizada. É possível também expressar caracteres como valores American Standard Code for Information Interchange (ASCII), quando a notação $Caracter é utilizada. Alguns exemplos de inteiros são: -18, 15, 0, 2#110, 16#CD e $A.

Floats: representam os números em ponto flutuante no formato duplo de 64 bits usando o padrão Institute of Electrical and Electronics Engineers (IEEE) 754 adotado em 1985 [10]. Podem ser representados na faixa de -10^{308} até 10^{308}.

Átomos: são usados para indicar valores distintos, chamados também de constantes literais, podem ser iniciados com letra minúscula ou delimitados entre aspas simples. Na primeira forma podem ser usadas letras, números e os caracteres arroba, ponto e *underline*; na segunda forma podem ser usados quaisquer caracteres.

Referências: as referências são objetos únicos, usados para a comparação de igualdade. As referências servem, por exemplo, para identificar ou confirmar a transmissão de mensagens entre dois processos em rede. Para fazer uso das referências é utilizada a Built-In Function (BIF) denominada make_ref(). BIF são funções internas (nativas) de Erlang.

Binários: um binário é uma sequência de 8 bits. Existem primitivas em Erlang para compor e decompor binários e também para permitir uma eficiente entrada e saída de dados binários. Pids: é a identificação de um processo em Erlang. Processos são criados pela primitiva spawn e recebem um PID. Portas - portas são usadas em Erlang para permitir a comunicação entre processos, mesmo que tais processos sejam escritos em outras linguagens de programação. As portas são criadas pela BIF open_port().

Funs: são funções anônimas, em algumas linguagens são conhecidas como expressões lambdas. Funs tem a sintaxe: Z = fun(X) -> 2*X end. As Funs podem ser usadas como funções de ordem superior (higher-order functions); podem receber funções como parâmetros e produzir uma função como resultado.

Tuplas: é um tipo de dado composto de um conjunto fixo de tipos de dados, os quais não precisam ser todos do mesmo tipo. Uma tupla é escrita por chaves, ..., e os elementos que a compõem são separados por vírgula. Exemplos de tuplas: 123, abc, abc, 123, abc, 123 e cor, 'Azul'.

Listas: representam um dos principais tipos de dados de Erlang. São semelhantes às tuplas, entretanto, são usadas para armazenar um número variável de tipos de dados. São escritos entre colchetes, [...], e seus elementos são separados por vírgulas. A sintaxe [] denota uma lista vazia. Quando se trabalha com listas é possível referir-se ao primeiro elemento da lista e o restante da lista, quando o primeiro elemento da lista foi removido. Por convenção é possível usar o primeiro elemento da lista como o cabeça da lista e o restante da lista como a cauda, [Head|Tail].

4.2.4. Interpretador de Comandos Erlang

A Figura 4.7 mostra a tela inicial do interpretador de comandos Erlang. Com este interpretador é possível declarar variáveis para os tipos de dados, compilar os programas escritos em Erlang e executar as funções dos programas compilados. Só não é possível escrever suas próprias funções. De acordo com Armstrong [5], os programas Erlang são compostos por módulos, cada módulo possui um determinado número de funções, as funções podem ser instanciadas a partir de outros módulos se forem exportadas pela diretiva export.

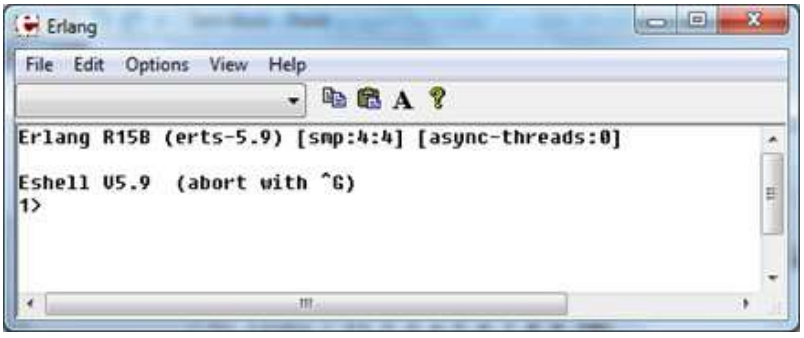


Figura 4.7. Interpretador de Comandos Erlang

O código da Figura 4.8 é um exemplo de um programa em Erlang que efetua o cálculo do dobro de um número.

```
-module(exemplo).
-export([double/1]).
double(X) -> 2*X.
```

Figura 4.8. Exemplo de um programa em Erlang

Para a escrita dos programas é necessário utilizar as diretivas module e export que são obrigatórias em todo código. Toda diretiva de compilação começa com o sinal de menos; o nome dado à diretiva module é o mesmo nome dado ao nome do arquivo.

No código mostrado da Figura 4.8, o nome do arquivo é *exemplo.erl* e é compilado pelo interpretador de comandos conforme ilustra a Figura 4.9. No código existe uma função chamada *double* que recebe um número como parâmetro. Nota-se que na diretiva *export* a função *double* é exportada, sendo possível executar pelo interpretador de comandos. O número depois da barra (/) indica a quantidade de parâmetros que precisa ser passada para a função.

A Figura 4.9 mostra o processo de compilação e execução do programa. Para compilar basta usar o comando *c* e informar o nome do programa entre parênteses. Para executar é necessário o nome do programa seguido pelo caractere dois pontos (:) e o nome da função. Depois são informados entre parênteses os parâmetros necessários para o processamento da função.

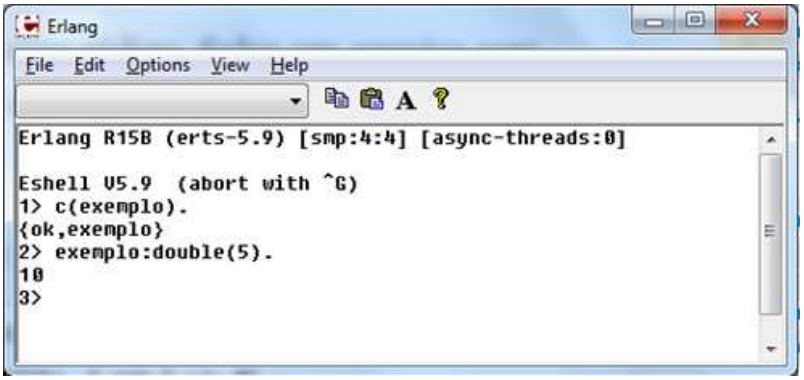


Figura 4.9. Exemplo de compilação e execução de um programa

4.2.5. Programação Concorrente em Erlang

Erlang foi concebida para permitir a programação concorrente. Cada atividade concorrente em Erlang é chamada de processo e a única forma de interação entre os processos é através de passagem de mensagens. Os processos são gerenciados pela VM Erlang, com isso, podem ser gerenciadas diretamente por Erlang e não pelo sistema operacional [5] [6].

De acordo com Armstrong [5], Erlang possui três primitivas básicas para programação concorrente. A primitiva *spawn* tem como objetivo criar novos processos; as primitivas ! (*send*) e *receive* são usadas, respectivamente para o envio e recebimento de mensagens entre os processos.

A primitiva *spawn(Módulo, Função, Argumentos)* avalia a função exportada de um módulo com uma lista de argumentos que são os seus parâmetros. A BIF *spawn/3* (existem também as BIFs *spawn/1*, *spawn/2*, *spawn/4*) retorna a identificação do processo, i.e., seu PID. A Figura 4.10 ilustra a criação de um processo.

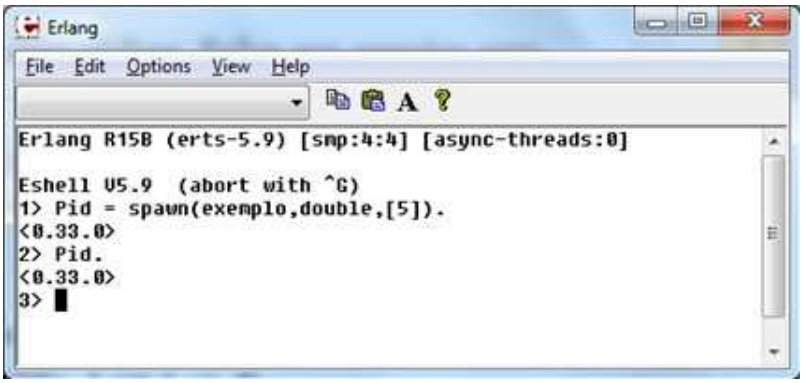


Figura 4.10. Exemplo de criação de um processo

Como pode ser observado na Figura 4.10, a variável *Pid* recebeu o número de identificação do processo criado, no caso, <0.33.0>. A primitiva ! (*send*) tem a seguinte sintaxe: *Pid ! Msg* , onde *Msg* contém a mensagem a ser enviada para o processo identificado por *Pid*. Cada processo possui uma caixa postal (um *buffer*) que armazena todas as

mensagens recebidas na ordem de chegada. A Figura 4.11 mostra um exemplo de envio de mensagem.

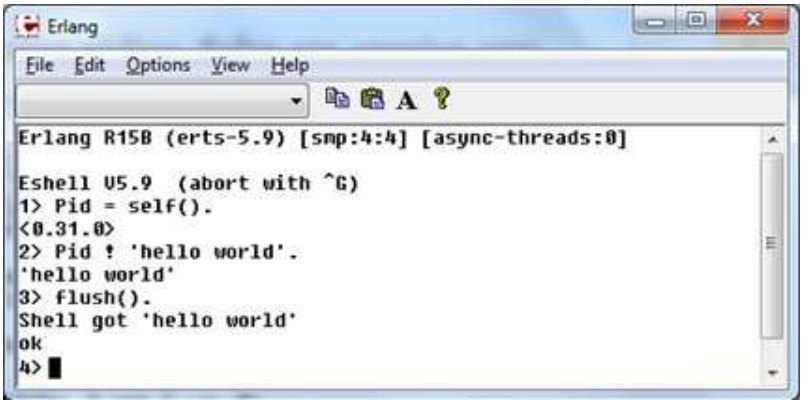


Figura 4.11. Exemplo de envio de mensagem para um processo

Na Figura 4.11 observa-se o envio da mensagem *'hello world'* para o processo *Pid* (comando 2). O interpretador de comandos Erlang também é um processo e pode ser usado para enviar ou receber mensagem. No exemplo foi utilizada a BIF *self()* que retorna o PID do processo atual. Este PID do processo atual foi armazenado na variável *Pid* para receber mensagens (comando 1). Por fim, a BIF *flush()* é usada para descarregar as mensagens da caixa postal e mostrá-las no interpretador de comandos (comando 3).

A sintaxe da primitiva receive pode ser observada na Figura 4.12.

```
receive
     Cláusula1 -> Ação1;
     Cláusula2 -> Ação2;
     ...
end
```

Figura 4.12. Exemplo de sintaxe para recebimento de mensagens

Na Figura 4.12, *Cláusula1* e *Cláusula2* são padrões que são comparados com as mensagens que estão chegando. Quando uma mensagem é recebida e uma correspondência é encontrada, a mensagem é removida da caixa postal e sua ação correspondente é executada. O processo que executou a primitiva de comunicação *receive* fica bloqueado até que uma mensagem seja recebida e esta seja comparada com alguma cláusula da primitiva *receive*. Armstrong [5] mostra um exemplo de um programa que troca mensagens entre dois processos. Este código é exibido na Figura 4.13.

O código da Figura 4.13 possui duas funções. A função *start* cria um novo processo que instancia a função *loop*. A função *loop* usa a primitiva *receive* e aguarda o recebimento de mensagens. Ao receber uma mensagem de outro processo que contém um PID e uma mensagem (tupla *{From, Message}*), o processo *loop* envia uma mensagem de volta para o processo com a mensagem *'Hello World'*. A Figura 4.14 mostra a troca de mensagens entre dois processos, de acordo com o código descrito na Figura 4.13.

```
-module(echo).
-export([start/0]).
start() ->
   spawn(echo, loop, []).
loop() ->
   receive
      {From, Message} ->
         From ! 'Hello World',
         loop()
   end.
```

Figura 4.13. Exemplo de programa com primitivas send e *receive*

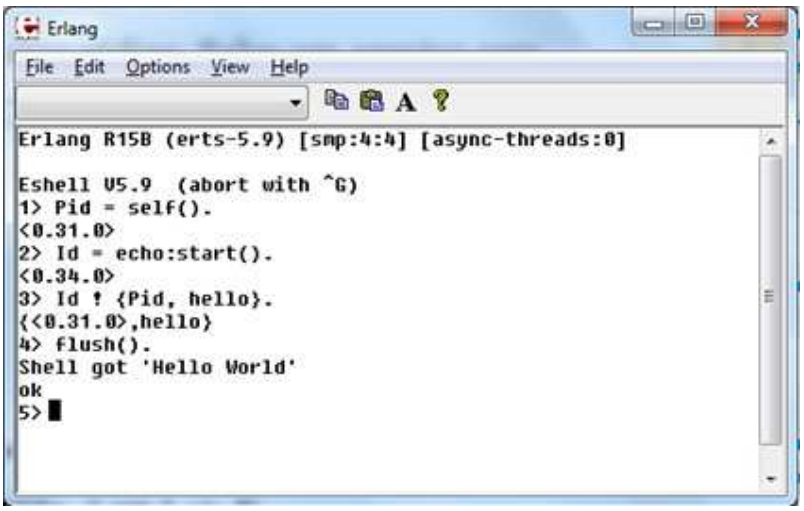


Figura 4.14. Exemplo de troca de mensagens entre dois processos

Na Figura 4.14 observa-se o processo *Pid* (comando 1) e a criação do processo *Id*, que executou a função *start* do módulo *echo* (comando 2). Em seguida, o processo *Pid* envia uma mensagem para o processo *Id* com sua identificação e a mensagem *hello* (comando 3). O processo *Id*, ao receber a mensagem, vai comparar com o padrão e retornar a mensagem *'Hello World'* para o processo *Pid*. Por fim, a BIF *flush()* é usada para checar se a mensagem de retorno enviada pelo processo *Id* foi recebida (comando 4).

4.2.6. Considerações Finais

Este capítulo apresentou as características das principais linguagens funcionais, com ênfase na linguagem Erlang. Em relação à Erlang foram descritas sua história, desde a criação pelos engenheiros da Ericsson até a linguagem tornar-se *open source*, suas principais características, detalhes de implementação e execução e exemplos de códigos com primitivas da linguagem utilizadas para a programação básica e concorrente.

Erlang apresenta diversas vantagens em relação às demais linguagens de programação. Pode-se destacar a atualização de código com o sistema em operação; manipula com eficiência um grande número de processos; apropriado para desenvolvimento de aplicações distribuídas e de tempo real; e robustez para permitir a execução contínua de um sistema.

Em contraposição às suas vantagens, Erlang também apresenta as seguintes desvantagens: as aplicações Erlang não têm bom desempenho para aplicações que efetuam cálculos numéricos; e a linguagem não é indicada para o desenvolvimento de aplicações com interfaces gráficas, os recursos que as bibliotecas da linguagem fornecem estão muito aquém das demais linguagens.

Algumas características, como por exemplo: dados imutáveis, *higher-order functions* e *pattern matching* não estão presentes nas linguagens imperativas. Além dessas, existem outras características como tolerância a falhas, chamadas de funções, chamadas recursivas e comunicação entre processos que as aplicações desenvolvidas em Erlang apresentam e que diferem em alguns aspectos das linguagens imperativas.

Referências

[1] J. Armstrong. The development of erlang. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 196–203, New York, NY, USA, 1997. ACM.

[2] J. Armstrong. A history of erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[3] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.

[4] J. Armstrong, B. Dcker, R. Virding, and M. Williams. Implementing a functional language for highly parallel real time applications. In *Eighth International Conference on Software Engineering for Telecommunication Systems and Services*, pages 157–163, New York, NY, USA, 1992. IEEE.

[5] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[6] F. Cesarini and S. Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[7] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[8] K. C. Louden. *Programming Languages: Principles and Practice*. Wadsworth Publ. Co., Belmont, CA, USA, 1993.

[9] R. W. Sebesta. *Concepts of Programming Languages*. Pearson, 10th edition, 2012.

[10] W. Stallings. *Arquitetura e organização de computadores*. PRENTICE HALL BRASIL, 2010.

[11] B. A. Tate. *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*. Pragmatic Bookshelf, 1st edition, 2010.

Capítulo 5

Técnicas para a Construção de Sistemas MPI Tolerantes a Falhas

Edson T. Camargo e Elias P. Duarte Jr.

Resumo

O MPI é um dos principais padrões para o desenvolvimento de aplicações paralelas e distribuídas baseadas no paradigma de troca de mensagens. Diversos sistemas de computação de alto desempenho são baseados em MPI. Um dos maiores desafios dos sistemas de alto desempenho diz respeito à confiabilidade, ou seja, à capacidade de oferecer serviços corretos ininterruptamente. Este minicurso tem como objetivo apresentar as principais técnicas empregadas para a construção de sistemas MPI tolerantes a falhas. São apresentadas técnicas tradicionais aplicadas a sistemas baseados em MPI, como a técnica rollback-recovery, incluindo suas variantes baseadas em checkpoints e em registro de mensagens. Além disso, também é apresentada a especificação ULFM (User Level Failure Mitigation), a mais recente proposta de tolerância a falhas para o padrão MPI.

5.1. Introdução

O MPI (*Message Passing Interface*) pode ser considerado o padrão *de facto* para o desenvolvimento de aplicações paralelas e distribuídas [MPI Forum 2015, Fagg and Dongarra 2000]. O padrão é baseado no paradigma de troca de mensagens, ou seja, é utilizado em ambientes computacionais onde os nodos acessam uma memória local e se comunicam através mensagens transmitidas em uma rede que conecta os nodos. O MPI consiste de um conjunto de bibliotecas de funções padronizadas pelo MPI-Fórum [Forum 2017b]. O padrão especifica rotinas para a comunicação entre processos, gerenciamento e criação de processos, entrada e saída de dados, gerenciamento de grupos, sincronização e o estabelecimento de topologias virtuais.

Embora o MPI seja amplamente utilizado, uma questão ainda em aberto no padrão é a tolerância a falhas. O padrão MPI parte do princípio que suas aplicações executam em uma infraestrutura computacional confiável e, desta forma, a tolerância a falhas não é contemplada em sua especificação. A falha de um único processo implica na interrupção de toda a aplicação. Mesmo as mais importantes implementações do padrão MPI, como

a Open MPI [Gabriel et al. 2004, open mpi.org 2017] e a MPICH [Gropp et al. 1996, mpich.org 2017] não são capazes de manter a aplicação em execução perante a falha de um único processo.

Os sistemas de computação de alto desempenho (*High Performance Computing* - HPC) empregam amplamente o padrão MPI [Fagg and Dongarra 2000]. Esses sistemas geralmente executam simulações científicas e industriais complexas que, potencialmente, são de longa duração, lidam com um volume de dados gigantesco e requerem uso intensivo de computação. A falha destes sistemas se traduz em prejuízos econômicos, até mesmo em termos do desperdício de energia, uma vez que o consumo desses sistemas é enorme. Além disso, falhas frequentes podem ser um impeditivo na execução de aplicações de longa duração. Estima-se que certos sistemas HPC de grande escala têm que lidar com a ocorrência de falhas em um intervalo de poucas horas [Egwutuoha et al. 2013]. Nesse contexto, projetar e construir mecanismos de tolerância a falhas efetivos e permitir que as aplicações completem adequadamente as suas execuções, apesar da ocorrência de falhas, é uma tarefa árdua. Abordagens e técnicas eficientes para tolerar falhas em aplicações HPC baseadas no padrão MPI são indispensáveis.

Este minicurso tem como objetivo apresentar algumas das principais técnicas para a construção de sistemas MPI tolerantes a falhas. O minicurso assume que o leitor tem familiaridade com o MPI; para uma introdução ao MPI recomenda-se [Pacheco 1996].

Inicialmente, é apresentada uma visão geral dos conceitos básicos de tolerância a falhas. A seguir, as principais técnicas de tolerância a falhas empregadas em sistemas MPI são apresentadas. Destaca-se a principal técnica aplicada à sistemas MPI, a *rollback-recovery* [Elnozahy et al. 2002, Egwutuoha et al. 2013]. No contexto da técnica de *rollback-recovery*, as abordagens baseadas em *checkpoints* e em registro de mensagens são descritas. Também são apresentados as técnicas de replicação máquina de estado [Charron-Bost et al. 2010] e a técnica chamada de *Algorithm-Based Fault Tolerance* (ABFT) [Du et al. 2012]. Trabalhos relacionados a cada um dessas técnicas são descritos. Além disso, a especificação ULFM (*User Level Failure Mitigation*) [Bland et al. 2013] é apresentada.

A ULFM é a mais recente proposta do MPI-Fórum para padronizar a semântica de tolerância a falhas em MPI. Através da ULFM o desenvolvedor da aplicação pode escolher a técnica de tolerância a falhas que melhor se adequa ao seu programa. Para tanto, a ULFM apresenta um conjunto de rotinas para detectar falhas, notificá-las e recuperar a capacidade dos processos de se comunicarem. Esse conjunto de rotinas é descrito juntamente com alguns exemplos de código que demonstram o seu funcionamento na prática.

O restante do minicurso está organizado da seguinte maneira. A seção 5.2 apresentada os modelos e os conceitos principais de tolerância a falhas. Ainda na seção 5.2 a técnica *rollback-recovery* é definida, bem como as suas variantes baseadas em *checkpoints* e em registro de mensagens. A seção 5.3 descreve a tolerância a falhas no padrão MPI. Essa seção inicia com a apresentação do padrão MPI e logo após as propostas de tolerância a falhas do MPI-Fórum são apresentadas. Na sequência, trabalhos que se apoiam nas abordagens de *rollback-recovery* são descritos. A seguir, são apresentadas as técnicas de replicação máquina de estado e ABFT. Por fim, a seção 5.4 apresenta a conclusão do minicurso.

5.2. Tolerância a Falhas: Definições Básicas

Um sistema confiável, ou tolerante a falhas, é aquele em que se pode ter confiança no seu funcionamento (em inglês *dependable*) [Avizienis et al. 2004]. A tolerância a falhas é a propriedade que garante a correta e eficiente operação de um sistema apesar da ocorrência de falhas em qualquer um dos seus componentes [Kshemkalyani and Singhal 2011].

Um sistema é projetado para executar corretamente e entregar serviços ao usuário de acordo com a sua especificação. Uma falha no serviço (*failure*) é caracterizada pelo não cumprimento da sua especificação ou devido à mesma não descrever adequadamente as funções do sistema. Uma falha no serviço é provocada por um erro (*error*). Um erro é ocasionado pela manifestação de uma falha (*fault*), ou defeito, em um dos componentes do sistema, que pode tanto ser interno quanto externo ao sistema. Por exemplo, estão suscetíveis a falhas os processos, os processadores, a memória ou a rede [Avizienis et al. 2004, Jalote 1994]. O tempo médio entre a ocorrência de falhas (*Mean Time Between Failures* - MTBF) é uma medida primária de confiabilidade do sistema baseada em análises estatísticas do sistema e de seus componentes [Kshemkalyani and Singhal 2011]. A medida é usualmente empregada para indicar o tempo previsto até uma falha ocorrer. Os sistemas HPC de grande escala, em especial os sistemas *petascale* e os futuros sistemas *exascale*, que podem atingir a ordem de 10^{15} e 10^{18} operações de ponto flutuante por segundo, respectivamente, apresentam um MTBF que pode chegar a poucas horas ou mesmo minutos [Di Martino et al. 2014, Egwutuoha et al. 2013, Cappello et al. 2009, El-Sayed and Schroeder 2013, Moody et al. 2010]. Esses sistemas estão sujeitos a diversos tipos de falhas [El-Sayed and Schroeder 2013, Schroeder and Gibson 2010].

Um modelo de falhas define o modo pelo qual os componentes do sistema podem falhar. Além disso, oferece uma classificação que especifica as suposições que podem ser feitas sobre o comportamento do componente quando o mesmo falha [Jalote 1994]. As falhas podem ser classificadas de acordo com as seguintes categorias: parada (*crash*), omissão, temporização e bizantinas. Laranjeira [Laranjeira et al. 1991] adiciona a essa classificação o modelo de falha por computação incorreta. Pode-se também citar dois modelos relacionados: *crash-recovery* e *fail-stop*. A seguir os modelos de falhas são descritos.

Uma falha *crash* ocasiona a parada permanente do componente e a perda do seu estado interno. Ao falhar, o componente não é submetido a qualquer transição incorreta de estado. Porém, a unidade não executa qualquer ação e tampouco responde a estímulos externos. Na falha por omissão o componente não responde tanto ocasionalmente quanto sistematicamente aos estímulos. Ou seja, na falha por omissão, o componente não envia e/ou recebe a algumas mensagens. A falha que leva o componente a responder muito cedo ou a muito tarde é chamada de falha de temporização, isto é, a falha viola uma propriedade temporal do sistema. Um componente com falha bizantina se comporta de maneira arbitrária, incluindo comportamento malicioso. A falha bizantina é a mais abrangente e a falha por parada a mais restritiva. Ou seja, a falha bizantina inclui todas as outras. A falha de temporização contém as falhas por omissão e por parada. Por sua vez, a falha por omissão envolve a falha por parada.

Conforme mencionado acima, há ainda a falha do tipo computação incorreta, que pode ser considerada como um subconjunto da falha bizantina. Esse tipo de falha ocorre

quando o componente não produz o resultado correto em resposta a uma tarefa correta recebida como entrada. Seguindo a classificação tradicional, a falha por computação incorreta inclui todas as demais exceto a bizantina.

O modelo de falha *crash* exclui a possibilidade de recuperação do componente. Se a recuperação é possível, o modelo de falhas é dito *crash-recovery*. Nesse modelo, um componente que constantemente para e se recupera é chamado de instável [Aguilera et al. 2000]. Quando os modelos de falhas são aplicados a sistemas distribuídos, os componentes correspondem a processos. Um processo geralmente guarda suas informações em um armazenamento do tipo estável ou volátil. Durante a recuperação apenas os dados armazenados em dispositivo estável são recuperados. As falhas em um sistema podem ainda ser classificadas de acordo com o modelo *fail-stop* [Guerraoui et al. 2011]. O modelo *fail-stop* inclui a falha do tipo *crash*, porém todo processo correto pode detectar a falha por parada do processo. A maioria trabalhos de tolerância a falhas em MPI assumem o modelo *fail-stop* [Bouteiller et al. 2006, Bland et al. 2013].

5.2.1. Detectores de Falhas

Um detector de falhas, a grosso modo, é um serviço que indica quais processos de um sistema distribuído estão falhos. Um detector de falhas é frequentemente implementado como um objeto local a cada processo [Chandra and Toueg 1996, Felber et al. 1999, Guerraoui et al. 2011]. Determinar entre um processo falho ou sem-falha é condição essencial para solucionar diversos problemas presentes nos sistemas distribuídos, como o consenso [Fischer et al. 1985]. O consenso, informalmente, permite que um conjunto de processos concordem sobre um valor único com base em valores propostos inicialmente, considerando que esses valores iniciais podem ser diferentes para cada processo. Um detector de falhas nem sempre detecta com precisão a falha de um processo [Guerraoui et al. 2011]. Da forma como foram propostos por Chandra e Toueg [Chandra and Toueg 1996], o detector é dito não-confiável, isto é, é possível que um processo tenha falhado mas não seja suspeito de ter falhado, bem como um processo tenha sido suspeito sem ter realmente falhado.

No modelo de Chandra e Toueg, cada processo mantém uma lista dos processos possivelmente falhos e acessa um módulo local através do qual pode consultar o estado dos demais processos. Como os detectores podem errar e eventualmente adicionar à lista um processo correto, é possível revisar o estado do processo posteriormente. Uma vez que os módulos são locais, em um mesmo instante dois módulos de detectores podem ter visões diferentes do sistema como um todo. Com base no comportamento dos detectores não-confiáveis e considerando as propriedades de completude (*completeness*) e exatidão (*accuracy*), Chandra e Toueg definiram oito diferentes classes de detectores. A completude assegura que processos falhos terminarão por ser suspeitos. A exatidão restringe os equívocos que podem ser cometidos pelo detector.

Entre as classes de detectores de falhas definidas, destacam-se a P (*perfect*), com as propriedades de completude forte e exatidão forte e o $\diamond$W que apresenta completude fraca e exatidão fraca. Chandra e Toueg provaram que detectores que possuem a completude fraca são equivalentes aos detectores com completude forte e provaram ser o detector de falhas $\diamond$W o mais fraco que permite o consenso [Chandra and Toueg 1996].

A implementação de detectores de falhas em sistemas distribuídos geralmente faz uso de mensagens de monitoramento. O monitoramento de processos é realizado através de trocas de mensagens. O envio e de recebimento das mensagens deve obedecer a um limite de tempo (*timeout*). De acordo com Felber et al. [Felber et al. 1999] existem basicamente dois modelos de monitoramento, conhecidos como *push* e *pull*. No primeiro, cada processo monitorado envia periodicamente mensagens do tipo "*I am alive*", também chamadas de *heartbeat*, para o detector que o está monitorando. No modelo *pull* a direção é oposta, isto é, o detector questiona o processo monitorado ("*Are you alive?*"). Se a troca de mensagens ocorrer dentro do intervalo de tempo definido significa que os processos monitorados estão operacionais.

É possível citar ainda outros modelos de detectores, como o modelo Dual [Felber et al. 1999], o modelo Gossip [Renesse et al. 1998] e o detector *heartbeat* proposto por Aguilera et al. [Aguilera et al. 1997]. Basicamente, o modelo Dual combina o modelo *push* e o modelo *pull*. No modelo Gossip o monitoramento é probabilístico. Os processos que recebem as informações são escolhidos aleatoriamente. Uma das vantagens é que o modelo apresenta boa escalabilidade. O modelo de Aguilera, ao invés de usar um *timeout*, somente incrementa um contador a cada mensagem de monitoramento.

5.2.2. *Rollback-Recovery*

A técnica de tolerância a falhas *rollback-recovery* é frequentemente empregada para prover tolerância a falhas em aplicações de alto desempenho [Stellner 1996, Elnozahy et al. 2002, Fagg and Dongarra 2000, Egwutuoha et al. 2013]. Desse modo, as aplicações podem reiniciar a partir de um estado salvo previamente. A técnica assume um sistema distribuído onde os processos da aplicação se comunicam através de uma rede e têm acesso a um dispositivo de armazenamento confiável que sobrevive a falhas. Periodicamente, os processos salvam informações de recuperação no dispositivo confiável durante a sua execução sem-falhas. Após a ocorrência de uma falha, a aplicação usa as informações de recuperação para reiniciar a sua computação a partir de um estado anterior. As informações de recuperação incluem os *checkpoints*, isto é, os estados dos processos participantes. Alguns protocolos também incluem informações sobre a recepção de mensagens. Basicamente, um estado global consistente é aquele em que se o estado de um processo reflete uma mensagem recebida, então o estado correspondente do emissor deve refletir o envio daquela mensagem [Kshemkalyani and Singhal 2011]. Um conjunto de *checkpoints* que corresponde a um estado consistente é chamado de *linha de recuperação*. O principal objetivo dos protocolos de *rollback-recovery* é restaurar o sistema a partir da mais recente linha de recuperação após uma falha.

O nível de integração da implementação da técnica de *rollback-recovery* a uma aplicação pode ser classificado de três formas: usuário, aplicação e sistema. No nível de aplicação (*application-level*), o programador ou algum mecanismo de pré-processamento insere o código de *checkpoint* diretamente no código da aplicação. A vantagem dessa abordagem é a independência de plataforma. Porém, há a falta de transparência. Exige-se do programador um bom conhecimento da aplicação para decidir em que momento o estado da aplicação deve ser salvo. Na abordagem em nível de usuário (*user-level*), uma biblioteca é ligada à aplicação e usada para realizar o *checkpoint*. Na abordagem em nível de sistema (*system-level*) o sistema operacional é responsável por salvar o estado da

aplicação. É uma abordagem totalmente transparente à aplicação e não há necessidade de alterações de código. A grande desvantagem é a falta de portabilidade [Egwutuoha et al. 2013].

Os protocolos de *rollback-recovery* podem ser classificados em duas categorias: baseados somente no *checkpoint* (*checkpoint-based*) ou baseados em registro de mensagens (*log-based*), descritos a seguir.

5.2.2.1. *Rollback-Recovery* Baseado em *Checkpoints*

Os protocolos de *rollback-recovery* baseados em *checkpoints* podem ser divididos em três abordagens: coordenada, não coordenada e induzida pela comunicação (CIC). Quando o *checkpoint* é realizado independentemente em cada processo, sem uma coordenação global, é chamado de não coordenado. A vantagem da abordagem não coordenada está em cada processo criar o seu *checkpoint* quando lhe é mais conveniente. Entretanto, com essa abordagem, um estado global consistente pode nunca ser atingido. Nesse caso, os *checkpoints* realizados tornam-se inúteis e devem ser descartados [Elnozahy et al. 2002, Kshemkalyani and Singhal 2011].

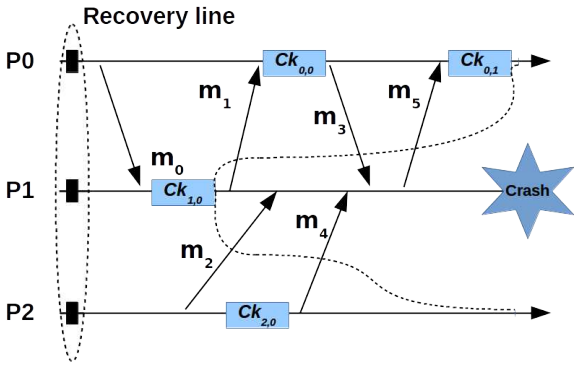


Figura 5.1. *Checkpoint* não coordenado e uma linha de recuperação.

A Figura 5.1 apresenta um cenário no qual o mais recente conjunto de *checkpoints* (isto é, $Ck_{0,1}$, $Ck_{1,0}$, e $Ck_{2,0}$) não resulta em uma linha de recuperação. Isso se deve ao fato de que a mensagem m_5 é recebida por P_0 mas não enviada por P_1 — nesse caso, m_5 é chamada de uma *mensagem órfã* e P_0 um *processo órfão*. P_0 é então obrigado a retroceder a um *checkpoint* anterior (isto é, $Ck_{0,0}$). O problema entretanto persiste, pois m_1 também foi recebida por P_0, mas não enviada por P_1. Desta forma, a única linha de recuperação corresponde ao estado inicial da aplicação. Esse fenômeno é conhecido como *efeito dominó*. O *checkpointing* não coordenado é suscetível ao efeito dominó.

O *checkpointing* coordenado evita o efeito dominó. Os processos se sincronizam para realizar os *checkpoints* e, consequentemente, criar um estado global consistente [Kshemkalyani and Singhal 2011]. Embora a abordagem coordenada seja relativamente fácil de implementar, a sua execução impõe uma sobrecarga considerável à aplicação, uma vez que os processos precisam se coordenar e salvar os seus estados simultaneamente no dispositivo de armazenamento. Além disso, mesmo que um único processo falhe, todos os processos precisam retroceder ao último *checkpoint*.

A abordagem denominada CIC (*communication-induced checkpointing*) força cada processo a realizar os *checkpoints* com base em informações inseridas nas mensagens recebidas dos outros processos. Os *checkpoints* são realizados de forma a manter a estado consistente em todo o sistema. A técnica reúne as vantagens das abordagens coordenada e não coordenada. No entanto, a abordagem relaxa a necessidade de coordenação global, o que a torna ineficiente na prática [Bouteiller et al. 2006]. Além disso, gera um grande número de *checkpoints*, resultando em sobrecarga no armazenamento e sobrecarga no canal de comunicação devido às informações inseridas nas mensagens da aplicação [Kshemkalyani and Singhal 2011, Egwutuoha et al. 2013].

O primeiro algoritmo a coordenar todos os *checkpoints* é apresentado por Chandy e Lamport [Chandy and Lamport 1985]. O algoritmo assume canais FIFO, isto é, todos os processos recebem as mensagens enviadas por um determinado processo na mesma ordem. Qualquer processo pode decidir iniciar um *checkpoint* e quando o faz envia uma mensagem especial chamada *marker* no seu canal de comunicação. Ao receber uma mensagem *marker* pela primeira vez um processo realiza o *checkpoint*. Após o *checkpoint*, todas a mensagens recebidas são armazenadas até uma nova mensagem *marker* ser recebida. Entre os trabalhos que fazem uso desse algoritmo há o CoCheck [Stellner 1996] e o LAM/MPI [Burns et al. , Sankaran et al. 2005], descritos na Seção 5.3.2.

5.2.2.2. *Rollback-Recovery* Baseado em Registro de Mensagens

Os protocolos de *rollback-recovery* baseados em registro de mensagens empregam tanto *checkpoints* quanto o registro de eventos não-determinísticos com o objetivo de evitar as desvantagens das abordagens coordenada e não coordenada. Um *evento* corresponde a um passo de comunicação ou um passo de computação de um processo. Um evento é determinístico quando a partir do estado atual existe somente um estado resultante possível para o evento. Se um evento pode resultar em estados diferentes, então é dito não-determinístico. A recepção de mensagens com uma identificação de emissor explícita é um evento determinístico e não requer o seu registro. Ao contrário, quando se aguarda uma mensagem de um emissor desconhecido então a recepção é dita não-determinística [Bouteiller et al. 2010].

Os protocolos de registro de mensagens assumem que todos os eventos não-determinísticos executados por um processo podem ser identificados e a informação necessária para reproduzir cada evento durante a recuperação pode ser codificada em tuplas chamadas de *determinantes* [Kshemkalyani and Singhal 2011]. A maioria dos protocolos de registro de mensagens assume que a recepção das mensagens é o único evento não-determinístico. O registro de mensagens evita o efeito dominó do *checkpointing* não coordenado salvando todas as mensagens recebidas. Por exemplo, na Figura 5.1, as mensagens m_2, m_4 e m_3 recebidas pelo processo P_1 devem ser salvas, assim como os determinantes que contém a ordem de recepção das mensagens. Durante a recuperação do processo P_1 somente P_1 retrocede. Assim, o estado de P_1 eventualmente será o mesmo ao anterior à falha, uma vez que as mensagens m_2, m_4 e m_3 são reaplicadas na mesma ordem.

Dependendo de como os determinantes são registrados, os protocolos de registro de mensagem podem ser classificados em pessimista, otimista ou causal [Elnozahy et al.

2002]. No registro pessimista, um processo primeiro armazena o determinante antes de entregar a mensagem. Apesar de simplificar a recuperação e a coleta de lixo, a abordagem pessimista gera uma sobrecarga durante a execução da aplicação: a aplicação precisa aguardar pelo armazenamento de cada determinante para então prosseguir. No registro de mensagens otimista, os processos armazenam os determinantes assincronamente, reduzindo assim a sobrecarga. Entretanto, a abordagem otimista pode gerar processos órfãos devido as falhas e, com isso, tornar a recuperação complexa. O registro causal busca combinar as vantagens do registro pessimista e otimista [Bouteiller et al. 2005]: ter baixa sobrecarga e evitar processos órfãos. Entretanto, os protocolos causais requerem que o determinante seja inserido em cada mensagem trocada pela aplicação até que esse seja confiavelmente armazenado.

O registro de mensagens geralmente faz uso da abordagem *sender-based* [Johnson and Zwaenepoel 1987]. Nessa abordagem, durante a operação normal cada mensagem enviada é salva no emissor. Dessa forma, o receptor da mensagem somente armazena o determinante correspondente, descrevendo o evento de entrega.

O *event logger* desempenha um papel crucial nos protocolos de registro de mensagens [Bouteiller et al. 2005]. O *event logger* recebe os determinantes dos processos da aplicação, armazena-os localmente, e notifica os processos da aplicação após armazená-los. Apesar do *event logger* exercer um grande impacto na eficiência dos protocolos de registro de mensagens, muitos protocolos o implementam como um componente centralizado e incapaz de tolerar falhas [Ropars and Morin 2009, Bouteiller et al. 2006, Ropars and Morin 2010]. Uma vez que *event logger* precisa notificar os processos da aplicação ao salvar um determinante, um *event logger* centralizado facilmente se torna em um gargalo conforme aumenta o número de processos. Além disso, a falha do *event logger* pode paralisar a aplicação ou levá-la a um estado inconsistente durante a recuperação. Em [Camargo et al. 2017] é apresentado um *event logger* distribuído e tolerante a falhas baseado em consenso para aplicações HPC.

Trabalhos que empregam o registro de mensagens como estratégia de tolerância a falhas são descritos na Seção 5.3.2. Descreve-se a seguir o padrão MPI e diversos trabalhos que abordam a tolerância a falhas em aplicações HPC baseadas em MPI.

5.3. Tolerância a Falhas em MPI

O padrão MPI (*Message Passing Interface*) oferece um dos principais modelos para o desenvolvimento de aplicações paralelas e distribuídas baseado no paradigma de troca de mensagens. O paradigma de troca de mensagens se destina a ambientes computacionais em que os nodos acessam uma memória local e estão conectados através de uma rede - que pode ser tanto um barramento de alta velocidade quanto uma rede local de computadores. Embora o MPI seja baseado no paradigma de troca de mensagens, o padrão também pode ser utilizado em sistemas que fazem uso de memória compartilhada.

O MPI consiste de um conjunto de bibliotecas de funções padronizadas pelo MPI-Fórum [Forum 2017b]. Há rotinas para comunicação direta entre dois processos, isto é, para comunicação ponto-a-ponto, e rotinas para comunicação coletiva. Além disso, há primitivas para o gerenciamento e criação de processos, entrada e saída de dados em paralelo, gerenciamento de grupos e sincronização de processos e o estabelecimento de

topologias virtuais, onde os processos são organizados de forma virtual para realizar a comunicação. O MPI-Fórum é a entidade composta por pesquisadores, desenvolvedores e organizações responsáveis por desenvolver e manter a norma MPI, atualmente na sua versão 3.1 [MPI Forum 2015]. Entre as principais implementações do MPI, destacam-se a MPICH [Gropp et al. 1996] e a Open MPI [Gabriel et al. 2004].

Um conceito fundamental em MPI é o comunicador (*communicator*), uma importante estrutura de dados que define o contexto da comunicação e o conjunto de processos pertencentes a esse contexto. No comunicador, os processos são identificados unicamente por meio de um número inteiro positivo chamado *rank*. Há um comunicador pré-definido chamado `MPI_COMM_WORLD` que reúne todos os processos disponíveis no início da execução de uma aplicação ou programa MPI. Um programa MPI pode possuir um ou mais comunicadores.

O Algoritmo 5.1 apresenta um código MPI básico em linguagem C que lança um determinado número de processos em uma máquina local ou em múltiplas máquinas interligadas por uma rede (dependendo da listagem de máquinas presente no arquivo de configuração *hostfile*). A linha 9 inicia o ambiente MPI. Repare que a biblioteca mpi.h é incluída na linha 1. Após isso, na linha 10, a função `MPI_Comm_rank()` determina o *rank* dos processos dentro do comunicador. Nesse exemplo, o comunicador usado é o `MPI_COMM_WORLD`. Na linha 11, a função `MPI_Comm_size()` informa a quantidade de processos presentes no comunicador. A função `MPI_Get_processor_name()` obtém o nome da máquina onde o processo está em execução. A linha 17 finaliza o ambiente MPI.

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6      int rank, size, name_size;
7      char processor_name[MPI_MAX_PROCESSOR_NAME];
8
9      MPI_Init(&argc,&argv);
10     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11     MPI_Comm_size(MPI_COMM_WORLD, &size);
12
13     MPI_Get_processor_name( processor_name, &name_size );
14
15     printf("Rank %d of %d running on host %s\n", rank, size, processor_name);
16
17     MPI_Finalize();
18     return 0;
19 }
```

Algoritmo 5.1. Exemplo de código MPI em linguagem C

Após compilar o código do Algoritmo 5.1 e gerar o executável `ex1`, a execução pode ser realizada da seguinte forma: $ mpiexec -np 4 ./ex1

De acordo com essa linha de código 4 processos serão lançados na máquina local.

Uma propriedade fundamental, porém ausente na especificação original MPI, é a tolerância a falhas [Gropp and Lusk 2004]. O padrão MPI assume que a infraestrutura subjacente é totalmente confiável [MPI Forum 2015]. Dessa forma, o padrão não define o comportamento preciso que as implementações MPI devem adotar perante falhas [Bland

et al. 2013, Gropp and Lusk 2004]. Basicamente, uma falha é tratada como um erro interno da aplicação como, por exemplo, a violação de um espaço de memória. Dessa forma, as falhas de processo ou de rede são repassadas à aplicação simplesmente como se fossem erros de chamadas de funções. Consequentemente, desloca-se a responsabilidade de detectar e de tratar as falhas para as implementações MPI. A norma define os manipuladores de erros (*error handlers*), que são associados ao comunicador MPI, para lidar com os erros do programa.

O manipulador de erros `MPI_ERRORS_ARE_FATAL` é associado por padrão ao comunicador `MPI_COMM_WORLD`. Esse manipulador define que a manifestação de um erro durante a chamada de uma função MPI leva os processos no comunicador a abortar a sua execução, encerrando assim toda a aplicação. Por outro lado, o manipulador `MPI_ERRORS_RETURN` retorna um código de erro que indica que a ocorrência de uma falha [MPI Forum 2015]. Mesmo que um código de erro seja retornado ao programa MPI, o padrão MPI não estabelece mecanismos para lidar com as falhas. Por essa razão, o suporte a tolerância a falhas pela norma MPI é considerado inadequado [Bland et al. 2012b, Bland et al. 2012c]. Além disso, as duas principais implementações MPI citadas anteriormente adotam o manipulador de erro `MPI_ERRORS_ARE_FATAL` por padrão e não dão suporte adequado ao manipulador de erros `MPI_ERRORS_RETURN`, impedindo a continuidade da aplicação no caso de falha.

Diversos trabalhos visam adicionar à implementação MPI rotinas específicas para lidar com as falhas. Entre esses estão o FT-MPI [Fagg and Dongarra 2000], FT/MPI [Batchu et al. 2004], Gropp e Lusk [Gropp and Lusk 2004] e recentemente o NR-MPI [Suo et al. 2013]. O MPI-Fórum também criou um grupo de trabalho específico para abordar a tolerância a falhas na norma MPI. Desse grupo surgiram as propostas chamadas de *Run-through Stabilization Proposal* (RTS) [Hursey et al. 2011] e de *User-Level Failure Mitigation* (ULFM) [Bland et al. 2012a, Bland et al. 2013], descritas a seguir.

5.3.1. RTS e ULFM

A falta de primitivas e de uma semântica de tolerância a falhas na norma MPI, que permitam às aplicações sobreviverem e se recuperarem de falhas de processos, aliada à frequente ocorrência de falhas nos sistemas HPC de grande escala, incentivaram o MPI-Fórum a criar um grupo de trabalho específico para o tema. O Grupo de Trabalho de Tolerância a Falhas (*Fault Tolerance Working Group* - FTWG) [Forum 2017a] foi estabelecido pelo MPI-Fórum, por volta do ano de 2009, com a responsabilidade de otimizar o padrão MPI para permitir o desenvolvimento de programas HPC portáveis, escaláveis e tolerantes a falhas [Hursey et al. 2011, Hursey and Graham 2011]. Os esforços do grupo de trabalho resultaram em duas propostas: a RTS (*Run-through Stabilization Proposal*) [Hursey et al. 2011, Forum 2017c] e a ULFM (*User-Level Failure Mitigation*) [Bland et al. 2012b, Bland et al. 2013, Forum b].

A RTS foi a primeira proposta do grupo de tolerância a falhas. Devido à complexidade presente na implementação das primitivas, a proposta RTS não prosseguiu o seu desenvolvimento. A especificação ULFM é o mais recente esforço do MPI-Fórum para padronizar a semântica de tolerância a falhas em MPI. A implementação da ULFM está em desenvolvimento como um subprojeto do projeto Open MPI [Gabriel et al. 2004, MPI-

Forum] e na implementação MPICH [Bland et al. 2015]. Existe a expectativa de que a adoção da ULFM pelo padrão MPI se dê a partir das próximas versões da norma MPI [Forum a].

A RTS e a ULFM possuem semelhanças. Em ambas há o mínimo de interfaces necessárias para recuperar a capacidade do MPI de continuar transportando suas mensagens após uma falha. Além disso, as propostas não definem uma estratégia de recuperação específica. Ao invés disso, disponibilizam um conjunto de funções às aplicações a fim de repararem o seu estado. As novas funções propostas tanto na RTS quanto na ULFM permitem que o desenvolvedor escolha a técnica de tolerância a falhas que melhor se adequa ao programa. A compatibilidade de código com as versões anteriores do MPI também está entre os requisitos observados.

Tanto na RTS quanto na ULFM a aplicação é notificada da falha de um processo ao tentar se comunicar diretamente (comunicação ponto-a-ponto, por exemplo através de um `MPI_Send()` e um `MPI_Receive`) ou indiretamente (operação coletiva, por exemplo através de um `MPI_Bcast()`) com o processo falho. Basicamente, as propostas se comprometem a informar quais condições específicas impedem que a entrega da mensagem ocorra com sucesso, sem que isso promova a interrupção automática da aplicação. A RTS e a ULFM adotam o modelo de falhas *fail-stop* e os manipuladores de erros propostos na norma MPI são os meios para informar a aplicação sobre as falhas de processos.

Na especificação RTS, a implementação MPI deve fornecer um detector de falhas perfeito [Chandra and Toueg 1996]. Isso significa que em algum momento todo processo falho será conhecido por todos os outros processos. Na RTS os processos podem estar em um de três estados: *OK*, *FAILED* or *NULL*. Os processos *OK* são os que executam normalmente. Os processos com o estado *FAILED* foram detectados como falhos. Os processos marcados com *NULL* são processos falhos cujos *ranks* recebem a constante `MPI_PROC_NULL`.

A RTS trata as falhas de processos de acordo com o modelo de comunicação empregado [Hursey et al. 2011]. A comunicação ponto-a-ponto recebe um tratamento diferente da comunicação coletiva: a comunicação realizada por um par de processos raramente é afetada pela falha de outro processo; na comunicação coletiva, que envolve um grupo de processos, a falha de um único processo afeta os demais. Dessa forma, a RTS fornece duas abordagens para o tratamento de falhas: local e global. Enquanto o tratamento das local das falhas se destina à comunicação ponto-a-ponto, o tratamento global é destinado à comunicação coletiva.

Na RTS, um processo usa uma função de validação para atualizar, acessar e modificar o estado dos processos no comunicador MPI. Com isso, a RTS mantém forte consistência entre os processos. A primitiva `MPI_Comm_validate` é usada para o tratamento local da falha e a primitiva `MPI_Comm_validate_all` para o tratamento global da falha. A última é também responsável por retornar a mesma lista de processos falhos para todos os processos. Um algoritmo de consenso distribuído é empregado pela primitiva `MPI_Comm_validate_all`. Essa primitiva sincroniza os detectores de falhas, reabilita as comunicações coletivas, identifica todos os processos falhos e fornece um valor de retorno uniforme às funções coletivas.

Na especificação ULFM, a falha de um processo somente é detectada se esse processo participa ativamente de uma comunicação (ponto-a-ponto ou coletiva). Isto é, somente os processos que se comunicam diretamente com o processo falho o detectam. Ao contrário da RTS, a ULFM não cita o uso de um detector de falhas. A aplicação é notificada da falha durante a execução das operações de comunicação. A Figura 5.2 apresenta um exemplo com três processos (A, B e C) que realizam uma comunicação ponto-a-ponto.

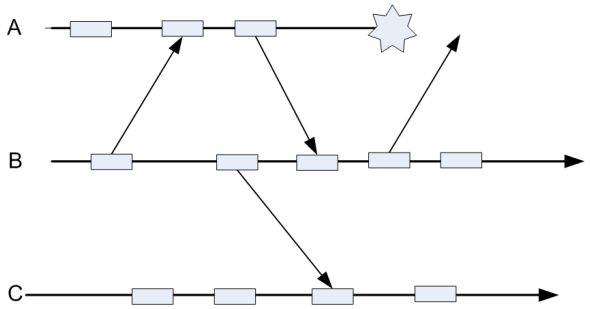


Figura 5.2. Detecção de falhas ULFM - Processo B detecta a falha em A.

Conforme apresenta a Figura 5.2, o processo B detecta a falha do processo A após enviar mensagem para A. Por sua vez, o processo C não identifica a falha em A. Essencialmente, a ocorrência de uma falha indica que a comunicação não pôde ser executada com sucesso. Por razões de desempenho, não há propagação automática sobre a ocorrência de falhas. Se durante uma operação coletiva um processo falhar, é possível que somente alguns processos identifiquem a falha. Ao todo, a ULFM disponibiliza ao usuário cinco funções para lidar com as situações de falhas. Entre essas, algumas permitem estabelecer uma visão consistente entre os processos. As primitivas da ULFM são descritas a seguir.

A operação de revogação, realizada pelo construtor `MPI_Comm_revoke`, é a mais crucial e complexa entre os construtores. Essa operação notifica todos os processos que o comunicador MPI a que pertencem está inválido. Dessa forma, evita a inconsistência entre os processos associados a um comunicador. O comunicador torna-se inválido e as comunicações futuras, ou as comunicações pendentes, são interrompidas e marcadas com um código de erro. A operação de revogação da ULFM conta com requisitos semelhantes à difusão confiável (*reliable broadcast*) [Guerraoui et al. 2011]. Nessa implementação, a ULFM usa um grafo binomial (*binomial graph*) onde o iniciador marca o comunicador como revogado e envia uma mensagem de revogação a outros $log(n)$ processos, considerando n processos. O processo, ao receber a mensagem de revogação, verifica se o comunicador foi marcado como inválido e, em caso contrário, atua como novo iniciador.

A primitiva `MPI_Comm_agree` é empregada para determinar uma visão consistente entre os processos. Essa função executa uma operação coletiva e faz com que os processos concordem com um valor booleano, mesmo se o comunicador foi revogado. Basicamente, o valor booleano pode significar o sucesso (0) ou a falha (1) na comunicação com um processo específico. Para fazer uso dessa primitiva o processo que identifica a falha deve antes revogar o comunicador. O construtor `MPI_Comm_shrink` permite criar um novo comunicador, eliminando todos os processos falhos de um comunicador inválido. Essa operação é coletiva e executa um algoritmo de consenso para assegurar

que todos os processos tenham a mesma visão no novo comunicador. Por fim, os construtores `MPI_Comm_failure_ack` e `MPI_Comm_failure_get_acked` são usados para informar quais processos dentro do comunicador se encontram falhos.

Por exemplo, na Figura 5.2, o processo B identifica a falha do processo A e então executa a função de revogação para que todos os processos corretos, no caso o processo C, tornem o seu comunicador inválido. Após isso, todos os processos executam a operação de acordo (`MPI_Comm_agree`) para garantir uma visão consistente sobre o estado do comunicador. Então, um novo comunicador válido, somente com processos corretos, é criado por meio da função `MPI_Comm_shrink`. Se houver a necessidade de identificar qual processo falhou (no caso o A), as primitivas `MPI_Comm_failure_ack` e `MPI_Comm_failure_get_acked` são empregadas.

```
1  #include <mpi.h>
2  #include <mpi-ext.h>
3  #include <stdio.h>
4  #include <signal.h>
5
6  int main(int argc, char *argv[])
7  {
8      int rank, size, name_size, result;
9      char processor_name[MPI_MAX_PROCESSOR_NAME];
10     char string_error[MPI_MAX_ERROR_STRING];
11
12     MPI_Init(&argc,&argv);
13     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14     MPI_Comm_size(MPI_COMM_WORLD, &size);
15
16     MPI_Comm_set_errhandler(MPI_COMM_WORLD,
17                             MPI_ERRORS_RETURN);
18
19     MPI_Get_processor_name( processor_name, &name_size );
20
21     if( rank == (size-1) )
22         raise(SIGKILL);
23
24     result = MPI_Barrier(MPI_COMM_WORLD);
25     MPI_Error_string(result, string_error, &name_size);
26
27     printf("Rank %d of %d running on host %s (error %s)\n", rank, size, processor_name,
       string_error);
28
29     MPI_Finalize();
30     return 0;
31 }
```

Algoritmo 5.2. Notificação de erro usando a ULFM

O Algoritmo 5.2 apresenta um exemplo de código que detecta e notifica a falha de um processo aos demais processos que não falharam. A linha 2 inclui a biblioteca ULFM. Neste exemplo, o processo com mais alto *rank* é morto, simulando uma falha. (linhas 21 e 22). Note que a linha 16 inclui a função `MPI_Comm_set_errhandler()`. Essa função associa o manipulador de erros `MPI_ERRORS_RETURN` ao comunicador `MPI_COMM_WORLD`. Através dessa função os erros detectados retornam um código. Na linha 24, se algum erro for detectado na execução da função coletiva `MPI_Barrier()`, esse erro será retornado para a variável `result`. A função `MPI_Error_string`, na linha 24, associa o código numérico a uma `string`. Ao executar o código acima com a linha de comando abaixo, a falha do processo origina um erro e a mensagem `MPI_ERR_PROC_FAILED` será exibida pelos processos que não falharam.

$ mpiexec -np 4 -am ft-enable-mpi ./ex2

```
1  #include <mpi.h>
2  #include <mpi-ext.h>
3  #include <stdio.h>
4  #include <signal.h>
5
6  static void verbose_errhandler(MPI_Comm* comm, int* err, ...) {
7      int rank, size, name_size;
8      char errstr[MPI_MAX_ERROR_STRING];
9
10     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11     MPI_Comm_size(MPI_COMM_WORLD, &size);
12
13     MPI_Error_string( *err, errstr, &name_size );
14     printf("Rank %d / %d: recebeu uma notificação de erro %s\n",
15            rank, size, errstr);
16 }
17
18 int main(int argc, char *argv[])
19 {
20     int rank, size, name_size;
21     char processor_name[MPI_MAX_PROCESSOR_NAME];
22     MPI_Errhandler errh;
23
24     MPI_Init(&argc,&argv);
25     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
26     MPI_Comm_size(MPI_COMM_WORLD, &size);
27
28     MPI_Comm_create_errhandler(verbose_errhandler, &errh);
29     MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);
30
31     MPI_Get_processor_name( processor_name, &name_size );
32
33     if( rank == (size-1) ) raise(SIGKILL);
34
35     MPI_Barrier(MPI_COMM_WORLD);
36
37     printf("Rank %d of %d running on host %s \n", rank, size, processor_name);
38
39     MPI_Finalize();
40     return 0;
41 }
```

Algoritmo 5.3. Definição de um manipulador de erros

O Algoritmo 5.3 modifica o Algoritmo 5.2 criando um manipulador de erros através da função `MPI_Comm_create_errhandler()`. Esse manipulador é posteriormente é associado ao comunicador (linha 29). A partir de então, toda a detecção de erro invoca a função `verbose_errhandler`.

A função `MPI_Comm_create_errhandler()` é modificada no Algoritmo 5.4 para identificar os processos que falharam. São empregadas as funções `failure_ack()` e `failure_get_acked()` para essa tarefa (linhas 18 e 19) juntamente com funções de manipulação de grupos em MPI. A função `failure_get_acked()` obtém o grupo de processos que tiveram suas falhas identificadas. A partir de então, através de uma comparação com o grupo original é possível identificar os processos que falharam (linha 30).

```
1
2  static void verbose_errhandler(MPI_Comm* pcomm, int* perr, ...) {
3      MPI_Comm comm = *pcomm;
4      int err = *perr;
```

```
5       char errstr[MPI_MAX_ERROR_STRING];
6       int i, rank, size, nf, len, eclass;
7       MPI_Group group_c, group_f;
8       int *ranks_gc, *ranks_gf;
9
10      MPI_Error_class(err, &eclass);
11      if( MPIX_ERR_PROC_FAILED != eclass ) {
12          MPI_Abort(comm, err);
13      }
14
15      MPI_Comm_rank(comm, &rank);
16      MPI_Comm_size(comm, &size);
17
18      MPIX_Comm_failure_ack(comm);
19      MPIX_Comm_failure_get_acked(comm, &group_f);
20      MPI_Group_size(group_f, &nf);
21      MPI_Error_string(err, errstr, &len);
22      printf("Rank %d / %d: recebeu uma notificação de erro %s. "
23          "%d processo(s) falhou(aram): { ", rank, size, errstr, nf);
24
25      ranks_gf = (int*)malloc(nf * sizeof(int));
26      ranks_gc = (int*)malloc(nf * sizeof(int));
27      MPI_Comm_group(comm, &group_c);
28      for(i = 0; i < nf; i++)
29          ranks_gf[i] = i;
30      MPI_Group_translate_ranks(group_f, nf, ranks_gf,
31                                group_c, ranks_gc);
32      for(i = 0; i < nf; i++)
33          printf("%d ", ranks_gc[i]);
34      printf("}\n");
35  }
```

Algoritmo 5.4. Identificação dos processos falhos

As falhas temporárias, tanto de rede quanto de processo, não fazem parte do escopo da ULFM, mas podem ser tratadas em nível de implementação. Uma falha temporária pode ser promovida a uma falha permanente (conforme o modelo *fail-stop*). Ou seja, se um processo sem-falha detecta que um processo deixa de responder, mesmo que temporariamente, o processo sem-falha classifica esse processo como falho e continuamente ignora e descarta qualquer comunicação com o processo falho. Nesse caso, como dito anteriormente, para evitar que os processos tenham uma visão diferente sobre o estado de algum processo, as rotinas da ULFM (`MPI_Comm_revoke`, `MPI_Comm_agree` e `MPI_Comm_shrink`) são usadas.

5.3.2. *Rollback-Recovery* em MPI

Conforme descrito na Seção 5.2.2, o *Rollback-Recovery* é a principal técnica de tolerância empregado em aplicações HPC baseadas no padrão MPI. A seguir alguns trabalhos que fazem uso dessa técnica são descritos brevemente.

O ambiente CoCheck [Stellner 1996] é o primeiro esforço para incluir a tolerância a falhas em MPI. O ambiente faz uso da técnica de *checkpoint-restart* e de migração de processos. A sua implementação é em nível de usuário. Tanto o reinício da aplicação quanto a migração de processo são transparentes à aplicação. O ambiente executa acima da biblioteca MPI e o programador precisa associá-lo à aplicação. A partir de então, as primitivas do CoCheck são usadas para comunicação ao invés das primitivas originais. Há um processo especial para coordenar os *checkpoints*. O mesmo envia uma notificação para todos os processos envolvidos na execução da aplicação realizarem o *checkpoint*.

Para tanto, cada processo envia mensagens especiais, chamadas *ready messages*, nos canais para assegurar que não há mensagens em trânsito e assim garantir a consistência global. Cada processo independentemente mantém um *checkpoint* consistente. É possível também realizar um *checkpoint* de toda a aplicação. A principal desvantagem do CoCheck está na necessidade de sincronizar toda a aplicação para efetuar o *checkpoint*, o que pode levar a problemas de escalabilidade. Outra desvantagem do ambiente é implementar a sua própria versão do MPI, chamada tuMPI [Fagg and Dongarra 2004].

O LAM/MPI é uma implementação de referência do MPI [Burns et al.]. No trabalho de Sankaran et al. [Sankaran et al. 2005] o LAM/MPI é estendido para suportar o *checkpoint* coordenado. A abordagem integra o LAM/MPI com a implementação de *checkpoint* em nível de sistema denominada BLCR (*Berkeley Lab's linux Checkpoint/-Restart*) [Duell 2003] através de interfaces definidas para *checkpoint-restart*. O BLCR é uma implementação que suporta aplicações com múltiplas *threads* no ambiente Linux. O LAM/MPI implementa o *checkpoint* em nível de usuário. Há algum tempo os esforços de desenvolvimento do LAM/MPI e dos seus mecanismos de tolerância a falhas foram portados para o desenvolvimento do Open MPI. No entanto, o Open MPI atualmente não dá suporte a qualquer estratégia de *checkpointing*.

A aplicação da técnica de *rollback-recovery* puramente baseada em *checkpoints* para sistemas HPC de larga escala vem sendo colocada em cheque devido ao MTBF cada vez menor desses sistemas [Cappello et al. 2009, Egwutuoha et al. 2013]. Embora eficiente, o custo para armazenar e recuperar os *checkpoints* pode exceder o MTBF dos futuros sistemas HPC *exascale*. Mais tempo será gasto lidando com as falhas do que realizando a computação útil [Ropars et al. 2013, Cappello et al. 2009]. Por exemplo, o Blue Waters [of Illinois], um sistema HPC *petascale* da universidade de Illinois, apresenta um MTBF médio de 4,2 horas [Di Martino et al. 2014]. Por outro lado, como apontado em Tiwari et al. [Tiwari et al. 2014] uma aplicação de astrofísica possui um volume de dados de 160TB e pode levar 360 horas para finalizar sua execução. Realizar um *checkpoint* a cada 1 hora, por exemplo, causa um grande impacto no sistema. Aumentar o intervalo de *checkpoint* pode reduzir o impacto no sistema, porém aumenta a quantidade de trabalho perdido no caso de falha: o processamento realizado entre o último *checkpoint* e a falha.

Diversos trabalhos buscam melhorar o desempenho da técnica de *rollback-recovery* baseado em *checkpoints* para sistemas HPC de larga escala através de diferentes propostas, como o Multi-level Checkpointing [Moody et al. 2010], protocolos híbrido ou hierárquicos que combinam o *checkpoint* coordenado com o registro de mensagens [Riesen et al. 2012, Ropars et al. 2013] e versões otimizadas [Tiwari et al. 2014, Bouteiller et al. 2013]. A seguir apresentamos algumas dessas abordagens, incluindo a abordagem de registro de mensagens.

O Diskless Checkpoint [Plank et al. 1998, Chen et al. 2005] é uma técnica que elimina a sobrecarga imposta pelo armazenamento estável do *checkpoint* tradicional. Nessa técnica, o estado de uma aplicação distribuída de longa duração é persistido tanto em memória quando em disco local. Adicionalmente, codificações relacionadas a esses *checkpoints* são armazenadas em processos redundantes - que podem ou não estar envolvidos na computação. Quando uma falha ocorre, os processos que não falharam recuperam o seu último *checkpoint*. O estado dos processos falhos pode ser calculado a partir do *check-*

point dos processos que não falharam e das codificações relacionadas aos *checkpoints*.

Multi-level Checkpointing [Moody et al. 2010] é uma abordagem que emprega múltiplos tipos de *checkpoints*, com diferentes níveis de resiliência e custo, em uma única execução da aplicação. O nível inferior é o mais lento e o mais resiliente. Nesse nível o *checkpoint* é escrito em um sistema de arquivos paralelo - o qual pode suportar a falha de todo o sistema. Os níveis superiores, apesar de serem mais rápidos, são os menos resilientes: os *checkpoints* são salvos em armazenamento local, tais como a memória RAM, memória *flash*, disco local ou cópias redundantes entre os nodos do sistema. Os trabalhos descritos em [Bautista-Gomez et al. 2011] e [Di et al. 2014] propõem, respectivamente, otimizações à abordagem através de códigos de proteção de dados e estudos sobre o intervalo de *checkpoint* em execuções onde o número de processadores e/ou núcleos envolvidos na computação é variável.

Fenix [Gamell et al. 2014, Gamell et al. 2015] é um arcabouço que permite a recuperação transparente e em tempo de execução de aplicações MPI. O arcabouço faz uso da especificação ULFM para sobreviver as falhas e emprega a técnica de Diskless Checkpoint: os dados da aplicação são salvos na memória dos nodos vizinhos [Zheng et al. 2012]. Também são disponibilizadas primitivas para o desenvolvedor realizar o *checkpoint* dos dados essenciais da aplicação. Fenix adota uma abordagem de chamada de *checkpoints* implícitos, onde os *checkpoints* são salvos de forma não-coordenada, porém, considerando a posição onde os *checkpoints* são inseridos no código há a garantia de que estados globais consistentes são sempre gerados pela aplicação. A avaliação do Fenix foi realizada em uma aplicação MPI executando milhares de processos. O arcabouço não está disponível para uso.

O projeto MPICH-V [Bouteiller et al. 2006] apresenta três protocolos de registros de mensagens que trabalham em conjunto com o *checkpointing* não coordenado. Dois protocolos são pessimistas e um é causal. O MPICH-V1 é um protocolo pessimista projetado para ambientes heterogêneos e com alta volatilidade, tais como grades computacionais formadas por *desktops*. O MPICH-V1 faz uso de um componente remoto e confiável, chamado de *Channel Memory* (CM) responsável por armazenar o conteúdo das mensagens e a ordem de recepção das mensagens MPI. Cada processo primeiro envia a sua mensagem para o CM do receptor. Então, o receptor solicita a mensagem do seu próprio CM. Apesar de haver um CM para cada processo, eles não suportam falhas. O MPICH-V2 [Bouteiller et al. 2003] é um protocolo pessimista destinado a grandes *clusters*. O MPICH-V2 conta com a abordagem *sender-based*. Os processos se comunicam diretamente. Ao invés de usar os CMs, o MPICH-V2 emprega *event loggers* que são usados como armazenamento remoto confiável. Quando um processo recebe uma mensagem, envia o determinante da mensagem para o *event logger*.

Em Lemarinier et al. [Lemarinier et al. 2006] uma estratégia de *checkpointing* coordenada baseada no algoritmo de Chandy e Lamport é comparada com o protocolo MPICH-V2 usando o *checkpointing* não coordenado em diferentes frequências de falhas e volume de dados da aplicação. A principal conclusão é a de que o registro de mensagens se torna relevante para *clusters* de grande escala a partir de uma taxa de falhas de uma falha a cada hora para aplicações com grande volume de dados.

Em Bouteiller et al. [Bouteiller et al. 2005], os autores investigam os benefícios

de um *event logger* no protocolo de registro de mensagens causal. Três protocolos foram implementados e comparados com e sem um *event logger*: Manetho, LogOn and Vcausal. A conclusão dos autores é a de que o *event logger* exerce um grande impacto em diversos aspectos do desempenho, incluindo o desempenho da aplicação e da recuperação de falhas. O trabalho assume um *event logger* confiável. Os autores destacam que empregar apenas um *event logger* para consistência ocasiona um gargalo conforme aumenta o número de processos. Os autores ainda afirmam que é necessário investigar como distribuir o registro de eventos entre múltiplos *event loggers*.

No trabalho em [Bouteiller et al. 2009], Bouteiller et al. comparam experimentalmente um protocolo pessimista e otimista de registro de mensagens considerando o refinamento proposto em um trabalho anterior envolvendo os mesmos autores [Bouteiller et al. 2010] que distingue eventos determinísticos dos não-determinísticos em MPI. Esse refinamento é codificado em um protocolo chamado de `Vprotocol` na implementação Open MPI. Como consequência, o número de mensagens enviadas ao *event logger* diminui consideravelmente. No `Vprotocol` o *event logger* não é tolerante a falhas e é implementado como um processo especial disponível à aplicação em um grupo externo ao grupo MPI principal, ou seja, o `MPI_COMM_WORLD`. Atualmente, o `Vprotocol` não está disponível na biblioteca Open MPI.

Geralmente, os protocolos de registro de mensagens criam determinantes para todas as mensagens recebidas. No entanto, é possível reduzir o número total de determinantes armazenados distinguindo os eventos determinísticos dos não-determinísticos [Bouteiller et al. 2010]. Por exemplo, um evento não-determinístico ocorre em MPI quando o processo receptor usa uma marcação `MPI_ANY_SOURCE` na primitiva `MPI_Recv`. Conforme definido em Cappello et al. [Cappello et al. 2010], muitas aplicações MPI contêm somente eventos de comunicação determinísticos. Alguns protocolos de tolerância a falhas foram propostos para essa classe de aplicação [Guermouche et al. 2012, Lefray et al. 2013, Ropars et al. 2013]. Porém, importantes aplicações MPI são não-determinísticas. Além disso, os desenvolvedores geralmente incluem o não-determinismo na codificação para melhorar o desempenho da aplicação.

O trabalho de Ropars e Morin [Ropars and Morin 2009] propõe o O2P, um protocolo ativo de registro de mensagens otimista. Nesse protocolo o *event logger* é implementado como um processo MPI capaz de manipular comunicações assíncronas. O *event logger* é inicializado separadamente da aplicação MPI. Os processos da aplicação se conectam ao *event logger* ao iniciar o registro dos determinantes. O O2P assume um *event logger* confiável. Os experimentos com um grande número de processos e uma alta taxa de comunicação mostram que o *event logger* é um gargalo para o desempenho do sistema.

Um *event logger* distribuído para o protocolo O2P também é proposto por Ropars e Morin [Ropars and Morin 2010]. O *event logger* aproveita a arquitetura multi-core dos processadores para ser executado em paralelo com os processos da aplicação. Cada nodo executa um *event logger*. Por exemplo, um nodo que possua um processador com quatro núcleos de processamento pode ter três processos da aplicação e um *event logger*. Os determinantes são salvos na memória volátil do *event logger* e são replicados entre os *event loggers*. Há um parâmetro chamado `replicationdegree` que informa ao *event logger* original quantos processos devem receber a cópia do determinante. Por exemplo,

se o `replicationdegree` é dois, um processo envia seus determinantes para outros dois *event loggers*. Quando um *event logger* recebe duas respostas de confirmações o determinante é considerado estável. O autor ainda propõe um protocolo de disseminação epidêmica (*gossip*) para espalhar os determinantes estáveis para todos os *event loggers*. Apesar de esse protocolo oferecer uma forma distribuída de salvar os determinantes, a solução falha se uma única resposta de confirmação não for recebida pelo emissor. Isto é, a tolerância a falhas da solução não é garantida.

O trabalho proposto por Camargo et al. [Camargo et al. 2017] propõe e implementa um *event logger* distribuído e tolerante a falhas baseado em consenso para os protocolos de registro de mensagens. O protocolo se apoia no algoritmo de consenso Paxos [Lamport 2001] para replicar o *event logger*. Nesse trabalho, duas abordagens são propostas. A primeira abordagem é baseada no algoritmo Paxos tradicional e é chamada de Paxos Clássico. A segunda abordagem propõe um configuração onde cada processo da aplicação tem a sua própria instância de consenso e é chamada de Paxos Paralelo. Os resultados apresentados demonstram que a abordagem Paxos Paralelo tem desempenho superior a um *event logger* centralizado e é capaz de suportar um número configurável de falhas.

Um protocolo híbrido que combina o *checkpointing* coordenado e o registro de mensagem otimista é proposto por Riesen et al. [Riesen et al. 2012]. O protocolo faz uso de nodos adicionais que agem como *event loggers* ou nodos extras no caso da falha de um nodo. O *checkpointing* coordenado auxilia o registro de mensagens limitando o tamanho dos registros e evitando retornar à aplicação ao seu estado inicial no caso de um estado inconsistente se fazer presente. Por sua vez, o registro de mensagens evita reiniciar todos os processos no caso de falhas na maioria das vezes. O trabalho assume a presença de um serviço para detectar processos falhos e reiniciar processos em nodos extras. Se o *event logger* falha, a aplicação continua sua execução. Ao atingir um *checkpoint* global, o serviço usa um nodo extra para lançar um novo *event logger* e informar cada processo sobre o novo *event logger*. No entanto, se um nodo precisa recuperar determinantes que foram perdidos devido à falha do *event logger*, então a aplicação reinicia a partir do último *checkpoint* coordenado. O *event logger* não é distribuído. Apesar de o protocolo de registro de mensagens otimista diminuir a sobrecarga, processos órfãos podem ser criados. Outro protocolo híbrido é proposto em [Bouteiller et al. 2013]: *checkpoint* coordenado é usado dentro dos nodos, onde os processo são relacionados, e *checkpoint* não coordenado com registro de mensagens pessimista é empregado entre os nodos.

5.3.3. Replicação Máquina de Estado Aplicada para MPI

A replicação máquina de estados (*State-Machine Replication* - SMR) é uma das mais importantes técnicas de tolerância a falhas [Charron-Bost et al. 2010]. Nessa técnica, frequentemente empregada para fornecer serviços com alta disponibilidade, o estado de um processo é replicado de tal forma que, se um processo falhar, a sua réplica, ou cópia, mantém o serviço disponível. O serviço é definido por uma máquina de estados, que consiste de variáveis e de operações. Uma operação pode tanto ler o estado das variáveis quanto modificá-las. A execução das operações são determinísticas, isto é, se duas réplicas executam a mesma sequência de operações na mesma ordem, o mesmo estado deve ser produzido em ambas [Schneider 1990].

O trabalho de Ferreira et al. [Ferreira et al. 2011] avalia a viabilidade da técnica de replicação máquina de estados como principal mecanismo de tolerância a falhas para os sistemas HPC *exascale*. A justificativa dos autores para usar a técnica de replicação é a diminuição drástica do tempo médio de interrupção devido a uma falha (*Mean Time To Interrupt* - MTTI) e o fato de estudos apresentarem que os sistemas *exascale* podem levar mais do que 50% do seu tempo lendo e escrevendo *checkpoints*. O *checkpoint-restart* seria aplicado em algumas situações como, por exemplo, quando todas as réplicas falhassem. As aplicações MPI são o objeto de estudo do autor: as cópias redundantes dos processos MPI permitem que perante uma falha do processo original a aplicação continue a sua execução de forma transparente, sem a necessidade de *rollback-recovery*. Segundo o autor, a técnica de replicação também poderia ser usada para detectar um leque maior de falhas, potencialmente incluindo as falhas maliciosas (bizantinas).

Ferreira et al. argumenta que não são todos os processos que precisariam ser duplicados. Por exemplo, considerando o modelo mestre-escravo somente o processo mestre seria replicado. Para realizar a avaliação, o trabalho combina modelagem, análise empírica e simulação para estudar os custos e benefícios da replicação em comparação com *checkpoint-restart*. Para estudar a sobrecarga imposta pela replicação dos processos, a ferramenta rMPI é projetada e implementada. A rMPI é implementada acima das implementações MPI existentes e fornece redundância de computação transparentemente às aplicações MPI determinísticas. O autor conclui que a técnica de replicação máquinas de estados tem potencial para atender as demandas de HPC.

A Ferramenta rMPI adota o modelo de falhas *fail-stop*: um processo falha por parada e então a sua réplica assume. O trabalho de Fiala et al. [Fiala et al. 2012] propõe a ferramenta redMPI com o objetivo de detectar e corrigir erros do tipo computação incorreta através da replicação da computação. Basicamente, a ferramenta compara as tarefas executadas pela réplica principal com as executas pelas suas cópias.

O trabalho de Bougeret et al. [Bougeret et al. 2014] adota uma estratégia de replicação de grupo ao invés de replicação de processos proposto em Ferreira et al. [Ferreira et al. 2011]. A replicação de grupo consiste em executar múltiplas instâncias da aplicação concorrentemente. Ao contrário do trabalho de Ferreira et al., a estratégia pode ser utilizada em qualquer modelo de programação de sistemas HPC. Um estratégia de *checkpoint* também é usada pelos autores. Os resultados obtidos no trabalho demonstram que a replicação de grupo pode apresentar vantagens em sistemas HPC de grande escala.

5.3.4. Tolerância a Falhas Codificada no Algoritmo da Aplicação

A técnica de ABFT (*Algorithm-Based Fault Tolerance*) faz uso das propriedades do algoritmo da aplicação para recuperá-la de falhas durante a sua execução, como se ignorasse a existência de falhas [Davies et al. 2011, Du et al. 2012, Hursey and Graham 2011, Wang et al. 2011]. A técnica não é transparente à aplicação. Os requisitos mínimos para utilizar a técnica são a detecção, notificação e propagação de falhas, assim como o suporte do ambiente de execução, que deve ser resiliente. Dessa forma, um dos empecilhos para a ampla adoção da técnica em aplicações MPI é falta de primitivas e de uma semântica padronizada de tolerância a falhas. Conforme apresentado na Seção 5.3, a especificação MPI e suas implementações de referência não fornecem meios para detectar e sobreviver

as falhas de rede e de processos. A maioria dos trabalhos que aplicam a técnica ABFT em MPI usam as implementações FT-MPI, a RTS ou a ULFM (Seção 5.3.1).

A técnica foi originalmente proposta por Huang e Abraham para detectar e corrigir erros em algumas operações em matrizes causados por falhas transientes ou permanentes no hardware [Huang and Abraham 1984]. De acordo com os autores, para algumas operações em matrizes há uma relação entre o *checksum* de entrada e o *checksum* apresentado nos resultados finais. Com base nessa relação, a técnica é desenvolvida para detectar, localizar e corrigir certos erros de cálculo do processador nas operações de matrizes.

A técnica ABFT é adaptada para sistemas HPC por Chen e Dongarra para suportar falhas de acordo com o modelo *fail-stop* durante a execução de programas que envolvam operações em matrizes [Chen and Dongarra 2006, Chen and Dongarra 2008]. Assim como Huang e Abraham, Chen e Dongarra fazem uso da relação do *checksum*, mencionado acima. A técnica é aplicada sem a necessidade de qualquer mecanismos de *checkpoint-restart*. Um estado global consistente é mantido em memória por meio da relação do *checksum*. Então, perante uma falha, a computação pode ser recuperada. No entanto, os processos corretos precisam aguardar a recuperação para continuar a execução da aplicação. A implementação FT-MPI é usada pelos autores [Chen and Dongarra 2008].

O trabalho de Wang et al. [Wang et al. 2011] propõe uma estratégia, chamada de ABFT-*hot-replacement*, para evitar que os processos corretos tenham que parar e aguardar pela recuperação dos dados do processo falho. Quando as falhas ocorrem durante a execução da aplicação, o trabalho atualiza o processo falho com um processo redundante correspondente. O trabalho também se apoia na relação do *checksum* e também é aplicado em operação de matrizes envolvendo transformações lineares. A implementação MPICH é adaptada para lidar com as falhas em nível de aplicação. Um trabalho semelhante é o de Bosilca et al. [Bosilca et al. 2009] onde os nodos redundantes são usados juntamente com uma abordagem de *diskless checkpoint* (Seção 5.2.2.1). Em Davies et al. [Davies et al. 2011] a técnica é usada para fatoração ou decomposição LU em matrizes. Os trabalhos em [Chen and Wu 2015, Jia et al. 2013] adotam abordagens semelhantes.

Hursey e Graham [Hursey and Graham 2011] apresentam como as primitivas da especificação RTS podem ser empregadas para tornar uma aplicação MPI que se comunica em uma topologia de anel tolerante a falhas. O objetivo é apresentar as primitivas da RTS aos desenvolvedores de aplicações MPI interessados em aplicar a técnica ABFT. A preocupação do autor é apresentar um exemplo de como é possível desenvolver uma aplicação tolerante a falhas e não especificamente aplicar a técnica ABFT. A recuperação dos processos não é abordada no trabalho. Duas versões do código da aplicação de comunicação em anel são apresentadas: uma que não tolera falhas e outra mantém a comunicação mesmo perante falhas. Hursey e Graham discutem ainda questões como duplicação de mensagens, detecção das falhas, terminação e eleição de um novo líder.

O *Checkpoint-on-Failure* (CoF) [Bland et al. 2012c] propõe uma estratégia de *checkpoint-restart* em nível de aplicação para ser usada juntamente com a técnica ABFT. A estratégia se apoia na possibilidade de as aplicações MPI serem notificadas de falhas de processos através da constante `MPI_ERRORS_RETURN` e também na técnica ABFT para restaurar os dados dos processos falhos através de métodos matemáticos. No CoF não há um *checkpoint* periódico, o mesmo é acionado quando ocorre uma falha em um

processo, a partir de então: 1) os processos corretos não mais executam chamadas MPI e realizam o *checkpoint* do seu estado atual; 2) processos corretos finalizam sua execução; 3) inicia-se a execução de uma nova aplicação; 4) o *checkpoint* é recuperado nessa nova aplicação; 5) através da técnica ABFT os dados do processo falho que não puderam ser salvos são restaurados e; 6) um estado global consistente é obtido e a aplicação retoma a sua execução. O CoF possui a vantagem de não exigir *checkpoints* periódicos, porém está restrito a algoritmos que suportam a técnica ABFT. No trabalho o CoF usado em uma aplicação matemática de fatoração linear.

Outras técnicas de tolerância a falhas aplicadas a sistemas baseados em MPI incluem a migração de processos [Wang et al. 2012, Stellner 1996], a predição de falhas [Rajachandrasekar et al. 2012, Gainaru et al. 2013], a exploração do determinismo na comunicação dos sistemas HPC [Cappello et al. 2010], a detecção de falhas [Kharbas et al. 2012, Genaud et al. 2009, Bosilca et al. 2016] e a redundância como técnica para detectar e corrigir erros do tipo de computação incorreta [Fiala et al. 2012].

5.4. Conclusão

O MPI é o padrão de *facto* para o desenvolvimento de aplicações paralelas e distribuídas baseado no paradigma de troca de mensagens. Este minicurso apresentou algumas das técnicas para a construção de sistemas MPI tolerantes a falhas. Foram apresentadas uma visão geral do conceito de tolerância a falhas, as técnicas *rollback-recovery*, replicação máquina de estados e ABFT. Além disso, foram apresentadas as recentes propostas de padronização da semântica de tolerância a falhas no padrão MPI por meio das especificações RTS e ULFM.

As especificações RTS e ULFM oferecem um conjunto de primitivas para o desenvolvedor da aplicação adequar a sua aplicação à uma técnica de tolerância a falhas de sua preferência. A ULFM é a proposta vigente e, ao contrário da RTS, não exige um detector de falhas explícito. De fato, na ULFM a falha de um processo somente é detectada se esse processo está diretamente envolvido em uma comunicação. A partir de então é possível usar as primitivas da ULFM para recuperar o estado do comunicador MPI e continuar a execução.

Dentre as técnicas de tolerância a falhas para sistemas MPI, o mecanismo de *rollback-recovery* é o mais tradicional e o mais empregado. A técnica pode ser baseada em *checkpoints* ou em registro de mensagens. A primeira exige a sincronização dos processos para garantir um estado global consistente. Os protocolos de *rollback-recovery* baseados em registro de mensagens empregam tanto *checkpoints* quanto o registro de eventos não-determinísticos com o objetivo de evitar as desvantagens das abordagens coordenada e não coordenada. Basicamente, a técnica consiste em forçar a reexecução dos processos falhos a partir de determinantes armazenados em um *event logger*.

Entre os trabalhos de *rollback-recovery*, estão o CoCheck, que foi a primeira implementação de *checkpoint-restart* e de migração de processos em MPI. Implementações como o Diskless *checkpoint* e o CoF buscam eliminar a necessidade de armazenamento estável através de codificações relacionadas ao *checkpoint*. O Multi-Level Checkpointing busca diminuir a sobrecarga do *checkpoint* coordenado realizando o *checkpointing* em diferentes níveis. Entre os trabalhos citados, ainda se destaca o Fenix. O Fenix emprega

checkpoints implícitos e a especificação ULFM para recuperar a aplicação em tempo de execução e de forma transparente. Importante notar que a maioria dos trabalhos que empregam a abordagem de registros de mensagens faz uso de um *event logger* centralizado e que não tolera falhas para armazenar os determinantes. Foi apresentado o primeiro *event logger* distribuído e tolerante a falhas que é baseado no algoritmo de consenso Paxos.

A técnica de replicação surge como uma possibilidade para fornecer alta disponibilidade aos sistemas HPC. O trabalho de Ferreira et al. emprega a replicação máquina de estado como principal mecanismo de tolerância a falhas para os sistemas HPC *exascale*. Naquele trabalho, o *checkpoint-restart* surge como segunda alternativa. Ferramentas como a rMPI e a redMPI usam a replicação para substituir um processo falho por uma réplica e para detectar e corrigir erros do tipo de computação incorreta, respectivamente. Uma grande desvantagem dessa técnica está na utilização de recursos extras pelas réplicas.

A técnica ABFT move a tolerância a falhas para o código da aplicação paralela. A técnica é altamente dependente da especificação MPI. Os trabalhos que defendem a inclusão de primitivas de tolerância a falhas na norma MPI usam como uma das justificativas a possibilidade de empregar a técnica ABFT. Apesar de eficiente, umas das desvantagens da técnica é a sua aplicação em um domínio específico. Grande parte dos trabalhos se apoia na verificação de *checksums* para detectar falhas.

Referências

[Aguilera et al. 2000] Aguilera, Chen, and Toueg (2000). Failure detection and consensus in the crash-recovery model. *Distributed Computing Journal*, 13.

[Aguilera et al. 1997] Aguilera, M. K., Chen, W., and Toueg, S. (1997). Heartbeat: A timeout-free failure detector for quiescent reliable communication. *Lecture Notes in Computer Science*, 1320:126–140.

[Avizienis et al. 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. E. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

[Batchu et al. 2004] Batchu, R., Dandass, Y. S., Skjellum, A., and Beddhu, M. (2004). MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 7(4):303–315.

[Bautista-Gomez et al. 2011] Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., and Matsuoka, S. (2011). Fti: High performance fault tolerance interface for hybrid systems. In *SC*.

[Bland et al. 2012a] Bland, W., Bosilca, G., Bouteiller, A., Herault, T., and Dongarra, J. (2012a). A proposal for user-level failure mitigation in the mpi-3 standard. Technical report, Department of Electrical Engineering and Computer Science, University of Tennessee.

[Bland et al. 2013] Bland, W., Bouteiller, A., Hérault, T., Bosilca, G., and Dongarra, J. (2013). Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of HPC Applications*, 27(3):244–254.

[Bland et al. 2012b] Bland, W., Bouteiller, A., Hérault, T., Hursey, J., Bosilca, G., and Dongarra, J. J. (2012b). An evaluation of user-level failure mitigation support in MPI. In *EuroMPI*, pages 193–203.

[Bland et al. 2012c] Bland, W., Du, P., Bouteiller, A., Hérault, T., Bosilca, G., and Dongarra, J. (2012c). A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI. In *Euro-Par*.

[Bland et al. 2015] Bland, W., Lu, H., Seo, S., and Balaji, P. (2015). Lessons learned implementing user-level failure mitigation in mpich. In *CCGrid*, pages 1123–1126.

[Bosilca et al. 2016] Bosilca, G., Bouteiller, A., Guermouche, A., Herault, T., Robert, Y., Sens, P., and Dongarra, J. (2016). Failure detection and propagation in hpc systems. In *SC*, pages 312–322.

[Bosilca et al. 2009] Bosilca, G., Delmas, R., Dongarra, J., and Langou, J. (2009). Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416.

[Bougeret et al. 2014] Bougeret, M., Casanova, H., Robert, Y., Vivien, F., and Zaidouni, D. (2014). Using group replication for resilience on exascale systems. *International Journal of High Performance Computing Applications*, 28(2):210–224.

[Bouteiller et al. 2010] Bouteiller, A., Bosilca, G., and Dongarra, J. (2010). Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211.

[Bouteiller et al. 2003] Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinier, P., and Magniette, F. (2003). MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *SC*.

[Bouteiller et al. 2005] Bouteiller, A., Collin, B., Herault, T., Lemarinier, P., and Cappello, F. (2005). Impact of event logger on causal message logging protocols for fault tolerant mpi. In *IPDPS*, pages 97–97.

[Bouteiller et al. 2013] Bouteiller, A., Hérault, T., Bosilca, G., and Dongarra, J. J. (2013). Correlated set coordination in fault tolerant message logging protocols for many-core clusters. *Concurrency and Computation: Practice and Experience*, 25(4):572–585.

[Bouteiller et al. 2006] Bouteiller, A., Hérault, T., Krawezik, G., Lemarinier, P., and Cappello, F. (2006). MPICH-V project: A multiprotocol automatic fault-tolerant MPI. *The International Journal of High Performance Computing Applications*, 20(3):319–333.

[Bouteiller et al. 2009] Bouteiller, A., Ropars, T., Bosilca, G., Morin, C., and Dongarra, J. (2009). Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure, recovery. In *Cluster*.

[Burns et al.] Burns, G., Daoud, R., and Vaigl, J. LAM: An open cluster environment for MPI.

[Camargo et al. 2017] Camargo, E. T., Duarte, E. P., and Pedone, F. (2017). *A Consensus-Based Fault-Tolerant Event Logger for High Performance Applications*, pages 415–427.

[Cappello et al. 2009] Cappello, F., Geist, A., Gropp, B., Kalé, L. V., Kramer, B., and Snir, M. (2009). Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23(4):374–388.

[Cappello et al. 2010] Cappello, F., Guermouche, A., and Snir, M. (2010). On communication determinism in parallel HPC applications. In *ICCCN*, pages 1–8.

[Chandra and Toueg 1996] Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

[Chandy and Lamport 1985] Chandy, M. and Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. *Transactions on Computer Systems*, 3(1):63–75.

[Charron-Bost et al. 2010] Charron-Bost, B., Pedone, F., and Schiper, A., editors (2010). *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer.

[Chen and Dongarra 2006] Chen, Z. and Dongarra, J. (2006). Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *IPDPS*, pages 10 pp.–.

[Chen and Dongarra 2008] Chen, Z. and Dongarra, J. (2008). Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions Parallel Distributed Systems*, 19(12):1628–1641.

[Chen et al. 2005] Chen, Z., Fagg, G. E., Gabriel, E., Langou, J., Angskun, T., Bosilca, G., and Dongarra, J. (2005). Building fault survivable mpi programs with ft-mpi using diskless checkpointing. In *PPoPP*, pages 213–223.

[Chen and Wu 2015] Chen, Z. and Wu, P. (2015). Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1323–1335.

[Davies et al. 2011] Davies, T., Karlsson, C., Liu, H., Ding, C., and Chen, Z. (2011). High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *ICS*, pages 162–171.

[Di et al. 2014] Di, S., Bautista-Gome, L., and Cappello, F. (2014). Optimization of a multilevel checkpoint model with uncertain execution scales. In *SC*.

[Di Martino et al. 2014] Di Martino, C., Kalbarczyk, Z., Iyer, R., Baccanico, F., Fullop, J., and Kramer, W. (2014). Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *DSN*, pages 610–621.

[Du et al. 2012] Du, P., Bouteiller, A., Bosilca, G., Herault, T., and Dongarra, J. (2012). Algorithm-based fault tolerance for dense matrix factorizations. In *PPoPP*.

[Duell 2003] Duell, J. (2003). The design and implementation of berkeley labâs linux checkpoint/restart. Technical report, Lawrence Berkeley National Laboratory.

[Egwutuoha et al. 2013] Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326.

[El-Sayed and Schroeder 2013] El-Sayed, N. and Schroeder, B. (2013). Reading between the lines of failure logs: Understanding how HPC systems fail. In *DSN*, pages 1–12.

[Elnozahy et al. 2002] Elnozahy, Alvisi, Wang, and Johnson (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surveys*, 34.

[Fagg and Dongarra 2000] Fagg, G. E. and Dongarra, J. (2000). FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent advances in PVM and MPI*, LNCS. Springer.

[Fagg and Dongarra 2004] Fagg, G. E. and Dongarra, J. (2004). Building and using a fault-tolerant MPI implementation. *The International Journal of High Performance Computing Applications*, 18(3):353–361.

[Felber et al. 1999] Felber, P., Défago, X., Guerraoui, R., and Oser, P. (1999). Failure detectors as first class objects. In *DOA*, pages 132–141.

[Ferreira et al. 2011] Ferreira, K. B., Stearley, J., Laros, III, J. H., Oldfield, R., Pedretti, K. T., Brightwell, R., Riesen, R., Bridges, P. G., and Arnold, D. (2011). Evaluating the viability of process replication reliability for exascale systems. In *SC*, page 44.

[Fiala et al. 2012] Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., and Brightwell, R. (2012). Detection and correction of silent data corruption for large-scale high-performance computing. In *SC*.

[Fischer et al. 1985] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382.

[Forum a] Forum, M. Mpi 4.0. `http://mpi-forum.org/mpi-40/`, year = 2017, note = Acessado em 01/02/2017,.

[Forum b] Forum, M. User-level failure mitigation. `http://fault-tolerance.org/ulfm/ulfm-specification/`, year = 2017, note = Acessado em 01/02/2017,.

[Forum 2017a] Forum, M. (2017a). Mpi forum fault tolerance working group. `https://github.com/mpiwg-ft`. Acessado em 01/02/2017.

[Forum 2017b] Forum, M. (2017b). Mpi forum website. `http://mpi-forum.org/`. Acessado em 26/01/2017.

[Forum 2017c] Forum, M. (2017c). Run-through stabilization process fault tolerance proposal. `https://github.com/mpi-forum/mpi-forum-historic/issues/276`. Acessado em 01/02/2017.

[Gabriel et al. 2004] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

[Gainaru et al. 2013] Gainaru, A., Cappello, F., Snir, M., and Kramer, W. (2013). Failure prediction for HPC systems and applications: Current situation and open issues. *The International Journal of High Performance Computing Applications*, 27(3):273–282.

[Gamell et al. 2014] Gamell, M., Katz, D. S., Kolla, H., Chen, J., Klasky, S., and Parashar, M. (2014). Exploring automatic, online failure recovery for scientific applications at extreme scales. In *SC*.

[Gamell et al. 2015] Gamell, M., Teranishi, K., Heroux, M. A., Mayo, J., Kolla, H., Chen, J., and Parashar, M. (2015). Local recovery and failure masking for stencil-based applications at extreme scales. In *SC*.

[Genaud et al. 2009] Genaud, S., Jeannot, E., and Rattanapoka, C. (2009). Fault-management in P2P-MPI. *International Journal of Parallel Programming*, 37(5):433–461.

[Gropp et al. 1996] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828.

[Gropp and Lusk 2004] Gropp, W. and Lusk, E. L. (2004). Fault tolerance in message passing interface programs. *The International Journal of High Performance Computing Applications*, 18(3):363–372.

[Guermouche et al. 2012] Guermouche, A., Ropars, T., Snir, M., and Cappello, F. (2012). HydEE: Failure containment without event logging for large scale send-deterministic MPI applicat. In *IPDPS*.

[Guerraoui et al. 2011] Guerraoui, R., Cachin, C., and Rodrigues, L. (2011). *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer.

[Huang and Abraham 1984] Huang, K.-H. and Abraham, J. A. (1984). Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers (TOC)*, C-33(7):518–528.

[Hursey and Graham 2011] Hursey, J. and Graham, R. L. (2011). Building a fault tolerant MPI application: A ring communication example. In *IPDPS Workshops*, pages 1549–1556.

[Hursey et al. 2011] Hursey, J., Graham, R. L., Bronevetsky, G., Buntinas, D., Pritchard, H., and Solt, D. G. (2011). Run-through stabilization: An MPI proposal for process fault tolerance. In *EuroMPI*, volume 6960, pages 329–332.

[Jalote 1994] Jalote, P. (1994). *Fault tolerance in distributed systems*. Prentice Hall.

[Jia et al. 2013] Jia, Y., Bosilca, G., Luszczek, P., and Dongarra, J. J. (2013). Parallel reduction to hessenberg form with algorithm-based fault tolerance. In *SC*, pages 1–11.

[Johnson and Zwaenepoel 1987] Johnson, D. B. and Zwaenepoel, W. (1987). Sender-based message logging. In *FTCS*.

[Kharbas et al. 2012] Kharbas, K., Kim, D., Hoefler, T., and Mueller, F. (2012). Assessing HPC failure detectors for MPI jobs. In *PDP*, pages 81–88.

[Kshemkalyani and Singhal 2011] Kshemkalyani, A. D. and Singhal, M. (2011). *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, Cambridge, UK.

[Lamport 2001] Lamport (2001). Paxos made simple. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 32.

[Laranjeira et al. 1991] Laranjeira, L., Malek, M., and Jenevein, R. (1991). On tolerating faults in naturally redundant algorithms. In *Reliable Distributed Systems, 1991. Proceedings., Tenth Symposium on*, pages 118–127.

[Lefray et al. 2013] Lefray, A., Ropars, T., and Schiper, A. (2013). Replication for send-deterministic MPI HPC applications. In *FTXS Workshop at HPDC*.

[Lemarinier et al. 2006] Lemarinier, P., Bouteiller, A., Krawezik, G., and Cappello, F. (2006). Coordinated checkpoint versus message log for fault tolerant MPI. *International Journal of High Performance Computing and Networking*, 2:146–155.

[Moody et al. 2010] Moody, A., Bronevetsky, G., Mohror, K., and d. Supinski, B. R. (2010). Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC*.

[MPI-Forum] MPI-Forum. User-level failure mitigation. `https://bitbucket.org/icldistcomp/ulfm/`, year = 2017, note = Acessado em 01/02/2017,.

[MPI Forum 2015] MPI Forum (2015). Document for a standard message-passing interface 3.1. Technical report, University of Tennessee, http://www.mpi-forum.org/docs/mpi-3.1.

[mpich.org 2017] mpich.org (2017). High-performance portable mpi. `http://www.mpich.org/`. Acessado em 26/01/2017.

[of Illinois] of Illinois, N. U. Blue waters. `https://bluewaters.ncsa.illinois.edu/`, year = 2017, note = Acessado em 01/02/2017,.

[open mpi.org 2017] open mpi.org (2017). Open mpi: Open source high performance computing. `https://www.open-mpi.org/`. Acessado em 26/01/2017.

[Pacheco 1996] Pacheco, P. S. (1996). *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Plank et al. 1998] Plank, J. S., Li, K., and Puening, M. A. (1998). Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, PDS-9(10):972–986.

[Rajachandrasekar et al. 2012] Rajachandrasekar, R., Besseron, X., and Panda, D. K. (2012). Monitoring and predicting hardware failures in HPC clusters with FTB-IPMI. In *IPDPS Workshops*, pages 1136–1143.

[Renesse et al. 1998] Renesse, R. V., Minsky, Y., and Hayden, M. (1998). A gossip-style failure detection service. Technical report, Cornell University.

[Riesen et al. 2012] Riesen, R., Ferreira, K., Silva, D. D., Lemarinier, P., Arnold, D., and Bridges, P. G. (2012). Alleviating scalability issues of checkpointing protocols. In *SC*.

[Ropars et al. 2013] Ropars, T., Martsinkevich, T. V., Guermouche, A., Schiper, A., and Cappello, F. (2013). SPBC: leveraging the characteristics of MPI HPC applications for scalable checkpointing. In *SC*, page 8.

[Ropars and Morin 2009] Ropars, T. and Morin, C. (2009). Active optimistic message logging for reliable execution of MPI applications. In *Euro-Par*.

[Ropars and Morin 2010] Ropars, T. and Morin, C. (2010). Improving message logging protocols scalability through distributed event logging. In *Euro-Par*.

[Sankaran et al. 2005] Sankaran, S., Squyres, J. M., Barrett, B., Sahay, V., Lumsdaine, A., Duell, J., Hargrove, P., and Roman, E. (2005). The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *The International Journal of High Performance Computing Applications*, (4):479–493.

[Schneider 1990] Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(3):299.

[Schroeder and Gibson 2010] Schroeder, B. and Gibson, G. A. (2010). A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–351.

[Stellner 1996] Stellner, G. (1996). Cocheck: Checkpointing and process migration for MPI. In *IPPS*, pages 526–531.

[Suo et al. 2013] Suo, G., Lu, Y., Liao, X., Xie, M., and Cao, H. (2013). Nr-mpi: A non-stop and fault resilient mpi. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 190–199.

[Tiwari et al. 2014] Tiwari, D., Gupta, S., and Vazhkudai, S. (2014). Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *DSN*, pages 25–36.

[Wang et al. 2012] Wang, C., Mueller, F., Engelmann, C., and Scott, S. L. (2012). Proactive process-level live migration and back migration in HPC environments. *Journal of Parallel and Distributed Computing*, 72(2):254–267.

[Wang et al. 2011] Wang, R., Yao, E., Chen, M., Tan, G., Balaji, P., and Buntinas, D. (2011). Building algorithmically nonstop fault tolerant MPI programs. In *HiPC*, pages 1–9.

[Zheng et al. 2012] Zheng, G., Ni, X., and KalÃ©, L. V. (2012). A scalable double in-memory checkpoint and restart scheme towards exascale. In *DSN Workshop 2012*.

Capítulo

6

Introdução à Otimização de Desempenho para Arquitetura Intel Xeon Phi Knights Landing (KNL)

Silvio Stanzani, Jefferson Fialho, Raphael Cóbe, Rogério Iope e Igor Freitas

Abstract

The main features offered by the KNL architecture are: the heterogeneous memory system, implementation of the latest vector instruction set from Intel called AVX-512, and a large number of cores. Using these features is essential for performance gains, but adapting code to use those features together is a challenging task. In this sense, the purpose of this short course is to present the opportunities for performance optimization offered by the KNL architecture, demonstrating the use of these resources with code examples.

Resumo

Os principais recursos oferecidos pela arquitetura KNL são: o sistema de memória heterogêneo, implementação do conjunto de instruções vetoriais mais recente da Intel chamado AVX-512, e uma grande quantidade de núcleos. A utilização desses recursos é essencial para se obter ganhos de desempenho, porém adaptar um código para utilizar tais recursos em conjunto é uma tarefa desafiadora. Nesse sentido, o objetivo desse minicurso é apresentar as oportunidades para otimização de desempenho oferecidas pela arquitetura KNL, demonstrando o uso desses recursos com exemplos de código.

6. Introdução

Atualmente, arquiteturas computacionais paralelas têm sido montadas de modo heterogêneo, compostas por recursos computacionais que possuem diversos processadores e coprocessadores ou aceleradores que podem ser usados em conjunto por uma mesma aplicação em um mesmo nó. Mais recentemente tem surgido iniciativas para montar infraestruturas paralelas utilizando coprocessadores disponibilizados como nós interconectados por uma rede de alto desempenho, como por exemplo a arquitetura Intel Xeon Phi Knights Landing (KNL).

O KNL oferece um grande número de unidades de processamento vetorial e um sistema heterogêneo de memória. A utilização adequada de tais recursos para otimizar o desempenho de aplicações é um desafio. Nesse sentido, o objetivo deste minicurso é apresentar os conceitos básicos relacionados à arquitetura KNL e como utilizar as novas possibilidades oferecidas pela arquitetura KNL, com ênfase na vetorização e no uso do sistema heterogêneo de memória. A estrutura geral do minicurso é detalhada na Seção 6.1.1.

6.1.1. Estrutura do Minicurso

Esse minicurso está estruturado da seguinte forma: a Seção 6.2 apresenta o conceito de arquiteturas paralelas, um caso especial desse modelo chamado arquitetura paralela híbrida e alguns conceitos essenciais para explorar tais arquiteturas. A Seção 6.3 apresenta em linhas gerais as iniciativas da Intel para prover suporte à computação de alto desempenho, com foco na arquitetura KNL. A Seção 6.4 mostra as técnicas para otimizar desempenho de aplicações na Arquitetura KNL. Finalmente, a Seção 6.5 apresenta as conclusões.

O código fonte completo de todos os exemplos mostrados neste capítulo, bem como, algumas avaliações comparativas de desempenho estão disponíveis no repositório de arquivos github[1].

6.2. Arquiteturas Paralelas

Sistemas computacionais modernos são constituídos por uma combinação de recursos que incluem processadores multinúcleos, subsistemas de memória que podem apresentar múltiplos níveis de acesso, e subsistemas de entrada e saída[1]. Tais sistemas são ditos heterogêneos quando dispõem de processadores auxiliares, como

[1] https://github.com/intel-unesp-mcp/KNL-Short-Course

coprocessadores e/ou aceleradores (gráficos ou de uso geral), que podem acrescentar dezenas ou centenas de elementos de processamento extras, acessíveis ao programador. Diversos sistemas podem ainda ser agrupados formando agregados ou clusters. O paralelismo pode ser explorado em sistemas computacionais desse tipo em diversos níveis, conforme descrito a seguir:

- Núcleo de processamento (*processing core*), constituído de registradores, unidades lógicas e aritméticas, unidades vetoriais, caches de instruções e de dados;
- Processador (*chip multiprocessor*), que pode conter múltiplos núcleos de processamento e um ou mais níveis de cache;
- Nó computacional (*computing node*), constituído por múltiplos processadores e os mecanismos de comunicação entre eles, o subsistema de memória (em geral, compartilhado entre os processadores) e os subsistemas de entrada e saída;
- Conjunto de nós computacionais (*computing cluster*): caracterizado por dispor de múltiplos nós computacionais e de mecanismos de comunicação entre nós (em geral, de grande largura de banda e baixa latência).

Uma tendência no desenho de novas arquiteturas computacionais é o desenvolvimento de arquiteturas paralelas híbridas, que combinam recursos heterogêneos em um mesmo sistema computacional. Na seção 6.2.1 é descrito este caso especial.

6.2.1. Arquiteturas Paralelas Híbridas

As arquiteturas paralelas híbridas surgiram como uma forma de aumentar o poder computacional, por meio da agregação de recursos extras de processamento em um mesmo nó, que são os coprocessadores e/ou aceleradores (gráficos ou de uso geral) por meio de um recurso de comunicação interno ao nó, que pode ser NVLINK[2] ou PCI-Express por exemplo [2].

Mais recentemente, tem se observado uma variação no uso de coprocessadores e/ou aceleradores como nós independentes que podem ser agregados em um sistema computacional por meio de uma rede de alto desempenho seguindo por exemplo o padrão Infiniband [3]. Dois exemplos de uso de coprocessadores como nós independentes são o DGX-SATURNV[3] desenvolvido pela NVIDIA que utiliza nós do tipo NVIDIA DGX-1, e o projeto Aurora[4] que é um projeto de supercomputador que prevê o uso de nós do tipo Xeon Phi (KNL).

---

[2] http://www.nvidia.com/object/nvlink.html

[3] https://www.nvidia.com/en-us/data-center/dgx-saturnv/

[4] https://aurora.alcf.anl.gov/

A Figura 1 ilustra o conceito de arquitetura paralela híbrida montada em um nó e montada utilizando mais de um nó.

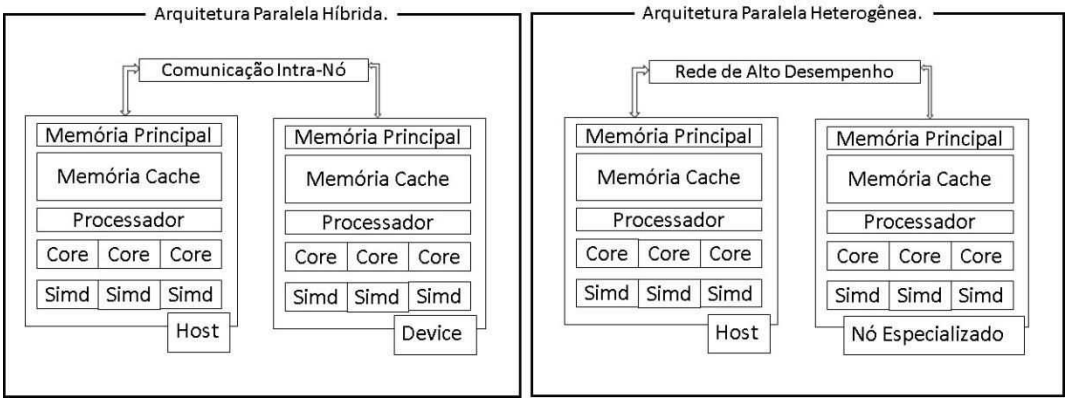


Figura 1. Arquiteturas Paralelas Híbridas.

Uma forma de otimizar a execução em uma arquitetura paralela híbrida consiste em descarregar partes do código para serem executadas em coprocessadores em um mesmo nó, esse processo é chamado de *offloading*. Uma variação desse processo é chamado de descarga pela rede (*offload over fabric)*, que significa fazer a descarga de código em um nó pela rede [4].

6.2.2. Explorando Paralelismo em Múltiplos Níveis

Dois mecanismos básicos para explorar o paralelismo presente nos diversos níveis do sistema computacional são os seguintes: vetorização que será descrito na seção 6.2.2.1 e programação MultiThreaded que será descrito na seção 6.2.2.2.

6.2.2.1. Vetorização

As instruções de uma arquitetura são classificadas em dois tipos: escalares ou vetoriais [5]:

- Instruções escalares executam uma operação em um conjunto de operandos;
- Instruções vetoriais executam uma operação para múltiplos conjuntos de operandos, que funcionam carregando todos os conjuntos de operandos de uma única vez e aplicando o operador simultaneamente em todos os operandos, conforme mostrado na Figura 2.

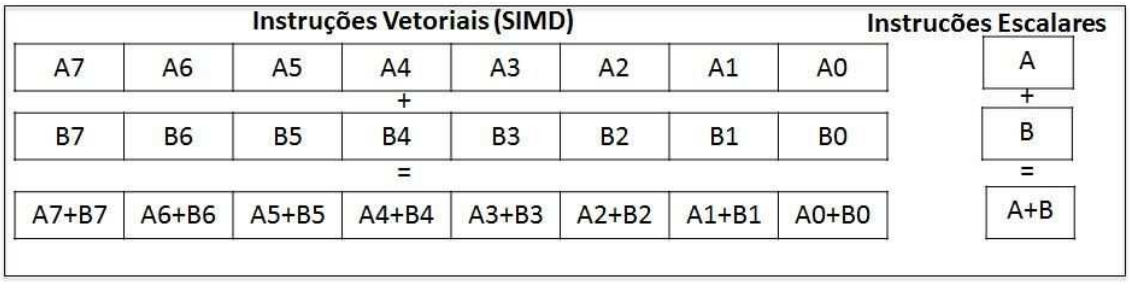


Figura 2. Execução de Instruções Vetoriais e Instruções Escalares.

A técnica de explorar paralelismo usando instruções vetoriais é chamada de vetorização e tem grande potencial para obter ganhos desempenho, pois aumenta consideravelmente a vazão de execução de instruções. Normalmente a vetorização é implementada em laços com alto custo computacional, por serem compostos por regiões fixas que são executadas repetidamente.

Um laço para ser vetorizado deve possuir duas características [6]: o corpo do laço deve ser composto por poucas linhas e por métodos elementares, e as iterações do laço devem ser independentes.

A seguir é descrito em linhas gerais três técnicas usadas para realizar a vetorização de um laço:

- Vetorização automática: está presente na maioria dos compiladores como um processo de otimização que ocorre em conjunto com processo de compilação, e consiste em utilizar instruções vetoriais no lugar das instruções escalares, mantendo a garantia de que o resultado da execução do programa não apresentará alteração;
- Semi-vetorização automática: em diversas situações o compilador é incapaz de garantir que o resultado da execução do programa se manterá o mesmo, ao utilizar instruções vetoriais, nesses casos, o compilador mantém as instruções escalares. Essa política é chamada de conservadora e serve para garantir que o processo de otimização do compilador não introduza erros no programa. No entanto, alguns laços que não são vetorizados por conta da política conservadora do compilador, podem ser vetorizados sem trazer quaisquer mudanças no resultado final. Nesses casos, o desenvolvedor pode utilizar diretivas de compilador, para indicar ao compilador regiões de código que podem ser vetorizadas. Ao utilizar tais diretivas, a responsabilidade pela geração de erros passa a ser do desenvolvedor.
- Vetorização explícita: é a utilização de bibliotecas que permitem a manipulação direta das unidades de processamento vetorial. Tais bibliotecas seguem um padrão chamado *intrinsics* [7], e tem como objetivo prover acesso direto aos recursos de vetorização presentes na arquitetura.

A Intel desenvolveu diversas bibliotecas *intrinsics*. A primeira é a *Multimedia Extensions* (MMX), que trabalha com tipos inteiros em conjuntos de 2, 4 e 8. A segunda versão é chamada *Streaming SIMD Extensions* (SSE), que permite trabalhar com precisão simples e dupla com conjuntos de até 128 bits, diversas versões da biblioteca SSE foram lançadas. A terceira versão é chamada *Advanced Vector Extensions* (AVX), que permite manipular registradores de 256 bits. A terceira versão é chamada *Initial Many Core Instructions* (IMCI) e foi a primeira implementação para Xeon Phi Knights Core (KNC), que permite manipular registradores de 512 bits. A versão mais recente é a AVX-512, que é implementada parcialmente no KNL.

6.2.2.2. Programação Multithreaded

A maioria dos sistemas operacionais modernos controla a execução de programas por meio de processos e threads. Processos representam programas em execução e sob controle do sistema operacional. Os processos são compostos por uma ou mais threads, que representam uma linha de execução formada por uma sequência de instruções que podem ser controladas de maneira independente pelo sistema operacional [8].

Todo processo tem pelo menos uma thread identificada como thread principal. A thread principal pode disparar novas threads, que podem ser executadas concorrentemente em diversos núcleos, e compartilhar dados do processo.

A programação multithreaded é uma técnica para explorar as oportunidades de paralelismo intrínseco de aplicações em arquiteturas multinúcleos, executando diferentes parte da carga de entrada, como por exemplo diferentes iterações de um laço, usando threads em diversos núcleos de processamento.

Uma das especificações mais tradicionais de programação multithreaded em arquiteturas multinúcleos é o OpenMP. Para atender os novos desafios presentes nas arquiteturas paralelas híbridas, a versão 4.0 do OpenMP foi lançada, disponibilizando mecanismos para semi-vetorização automática e *offloading* [9].

6.3. Arquiteturas para Processamento de Alto Desempenho da Intel

As plataformas computacionais montadas com processadores Intel têm seguido a tendência de serem montadas de forma heterogênea. Nesse contexto, as arquiteturas computacionais de alto desempenho da Intel são desenvolvidas em duas linhas: Multicore e Manycore.

A família de processadores da linha Multicore são chamados de Xeon e são desenvolvidos com objetivo de prover poder computacional a quaisquer tipos de carga de trabalho. As características comuns desses processadores é possuir múltiplos núcleos e sistema de memória em múltiplos níveis.

A família de processadores da linha Manycore são chamados de Xeon Phi e são especializados em processamento de cargas de trabalho massivamente paralelas, que pode ser vetorizada e que possuam alta demanda de transferência de dados entre memória e registradores vetoriais. A família Xeon Phi será descrita com mais detalhes na Sessão 6.3.1.

6.3.1. Família de processadores Intel Xeon Phi

A primeira geração da família Xeon Phi é chamada KNC [10] e é montada como uma placa que pode ser agregada em um nó. Nesse sentido, parte da carga de trabalho da aplicação, que possua características adequadas para processamento nessa arquitetura, pode ser descarregada para um ou mais coprocessadores KNC acoplados no nó computacional.

A segunda geração da família Xeon Phi é chamada KNL [11]. Essa arquitetura é montada como um nó de processamento independente, embora mantenha as mesmas características de ser projetada para ser usada como um recurso complementar a um processador de propósito geral, para cargas de trabalho massivamente paralelas e vetoriais.

O KNL dispõe de até 72 núcleos (*cores*), e cada núcleo possui duas unidades de processamento vetorial de 512 bits. Os núcleos são organizados em pares chamados *tiles* conforme mostrado na Figura 3. Cada núcleo possui um cache de nível 1 (L1) e compartilha o cache de nível 2 (L2) com o outro núcleo que compõe o *tile*. Os *tiles* são interconectados, possuem coerência de cache, e são agrupados de forma padronizada. Os padrões de agrupamento da arquitetura KNL são chamados de modos de cluster [12], conforme descrito a seguir:

- All-to-All: os *tiles* não possuem subdivisão e os endereços de memória são distribuídos uniformemente entre eles. Normalmente esse modo é usado apenas para depuração.
- Quadrante / Hemisfério: no modo quadrante, os *tiles* são divididos em 4 partes, cada uma local ao seu respectivo controlador de memória. O modo hemisfério funciona da mesma forma, com a diferença que é subdividido em 2 partes.
- SNC-4 / SNC-2: semelhante ao Quadrante / Hemisfério, os modos SNC-4 / SNC-2 também subdividem o total de *tiles* em 4 ou 2 partes. Nestes modos, cada subdivisão é vista pelo sistema operacional como um nó do tipo Non-Uniform Memory Access (NUMA) [13].

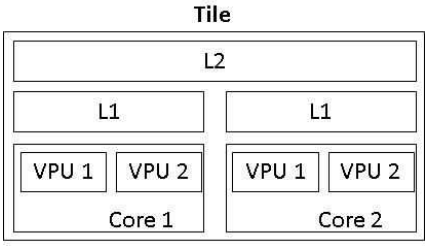


Figura 3. Organização dos Núcleos do KNL.

O subsistema de memória desenvolvido para o KNL consiste de uma memória principal *Dynamic Random Access Memory* (DRAM) e uma unidade de memória de banda larga chamada *Multi-Channel DRAM* (MCDRAM) [14]. A memória MCDRAM possui uma capacidade menor que a DRAM, porém com uma largura de banda maior. Nesse sentido, o uso da MCDRAM pode trazer ganhos de desempenho ao armazenar estruturas de dados com alta demanda de transferência entre memória e registradores do processador [15].

A MCDRAM pode ser configurada de 3 modos:

- Cache: a MCDRAM é usada como o último nível de cache, sendo gerenciada somente pelo sistema operacional;
- Flat: a MCDRAM pode ser usada como uma memória endereçável, que pode ser acessada por meio de APIs ou utilizando a ferramenta numactl[5];
- Hybrid: através deste modo é possível usufruir dos dois modos descritos. Este modo subdivide a MCDRAM em duas partes onde uma será configurada como memória endereçável (flat) e a outra parte como último nível de cache (cache).

A Figura 4 mostra os quatro modos de cluster combinado com MCDRAM configurada no modo cache. Quando é utilizado os modos Quadrante / Hemisfério apenas um nó NUMA é disponibilizado, e quando é utilizado o modo de cluster SNC-2 ou SNC-4 são disponibilizados dois ou quatro nós NUMA respectivamente.

A Figura 5 mostra os quatro modos de cluster combinado com MCDRAM configurada no modo flat. Nesse contexto, nós NUMA são criados para controlar o acesso à memória MCDRAM de modo exclusivo. Nos modos Quadrante / Hemisfério um novo nó é criado, e nos modos de cluster SNC-2 / SNC-4 são criados dois e quatro nós do tipo NUMA adicionais respectivamente.

[5] http://gnu.wiki/man8/numactl.8.php

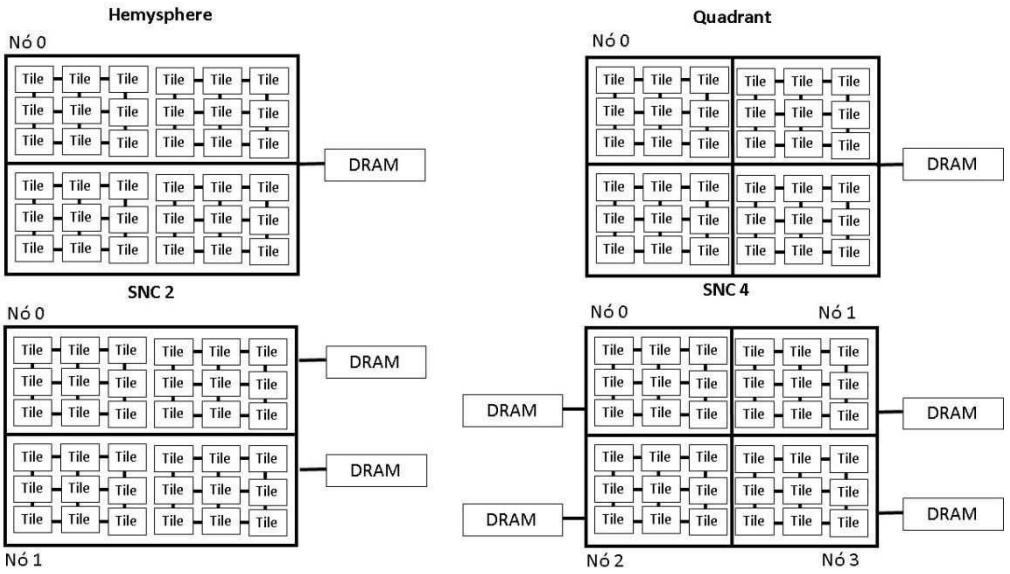


Figura 4. Diferentes Modos de Cluster Combinado com MCDRAM Configurada no Modo Cache.

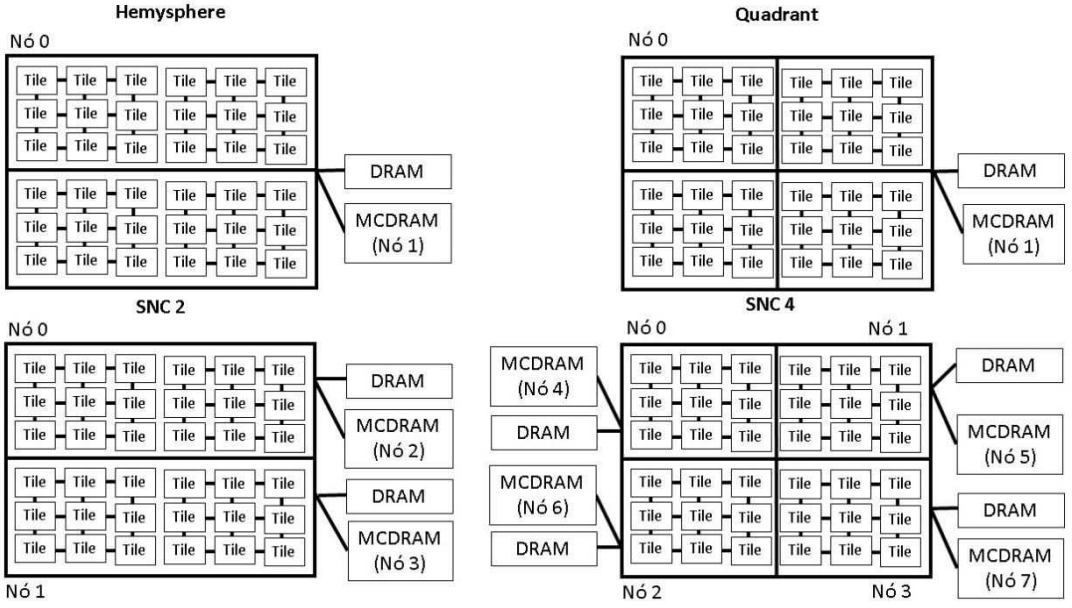


Figura 5. Diferentes Modos de Cluster e MCDRAM no Modo Flat.

O programa numactl disponibiliza um parâmetro para visualizar a configuração de nós NUMA no sistema (“numactl -H”). Um exemplo do retorno da execução desse comando em um servidor KNL configurado com modo cluster SNC-4 e MCDRAM configurada no modo flat, é mostrado na Figura 6.

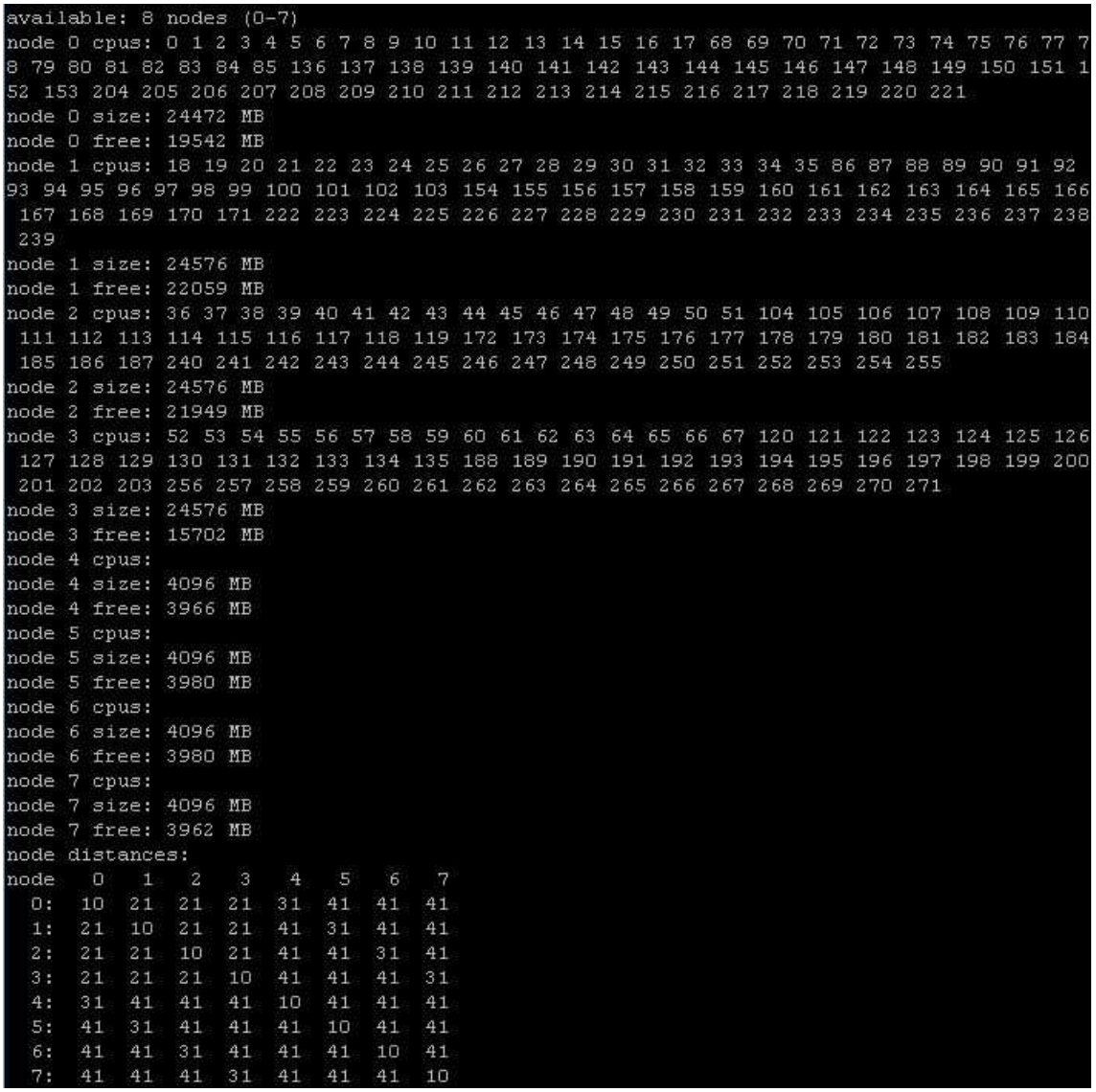


Figura 6. Configuração de nós NUMA de um servidor KNL.

A Seção 6.4 mostra algumas técnicas para explorar desempenho na arquitetura KNL.

6.4. Explorando os Recursos da Arquitetura KNL para Otimização de Desempenho

Para otimizar um programa para a arquitetura KNL é necessário explorar os seguintes recursos de modo combinado: paralelismo no nível de threads, otimização de transferência de dados da memória para registradores do processador e vetorização.

O paralelismo no nível de threads pode ser implementado com suporte de diversos *frameworks* e padrões de programação multithreaded, tais como, OpenMP **[16]**, TBB **[17]** e OpenCL **[18]**. A otimização de memória pode ser feita usando

ferramentas do Sistema Operacional ou bibliotecas de desenvolvimento, a Seção 6.4.1 descreve algumas dessas possibilidades. A Seção 6.4.2 descreve os recursos de vetorização que podem ser explorados na arquitetura KNL.

6.4.1. Explorando o Sistema de Memória Heterogêneo da Arquitetura KNL

A MCDRAM pode ser usada como suporte à otimização de desempenho de forma implícita ou de forma explícita.

Na forma implícita a MCDRAM é configurada no modo cache e o Sistema Operacional se encarrega de fazer as transferências de dados da memória DRAM para a memória MCDRAM, que atua nesse cenário como uma memória cache L3, usando critérios de predição de instruções para transferir os dados que serão utilizados nas próximas instruções que serão executadas, e dessa forma, diminuir a quantidade de transferências de dados da memória para os registradores do processador.

Na forma explícita a MCDRAM é configurada no modo flat ou híbrido e passa a oferecer a possibilidade de utilizar essa unidade de memória de modo mais flexível, permitindo alocar e desalocar estruturas de dados, da mesma forma como é feito na DRAM.

Nessa seção vamos abordar o uso da MCDRAM no modo flat. Nesse modo, a estratégia mais elementar para otimizar o desempenho é armazenar as estruturas de dados, com alta demanda por transferência da memória para os registradores do processador na MCDRAM.

A alocação de dados na MCDRAM no modo flat ou híbrido pode ser realizada utilizando dois mecanismos: o primeiro é mapear toda alocação e desalocação de uma aplicação para MCDRAM ou DRAM sem modificações no código fonte usando o programa numactl. A segunda é utilizando a biblioteca memkind[6] que permite definir para cada estrutura de dados o local de armazenamento MCDRAM ou DRAM.

O comando numactl permite mapear uma aplicação para usar a memória MCDRAM ou DRAM de modo exclusivo, para isso é necessário utilizar o parâmetro "-m" indicando em quais nós a aplicação deve ser executada. Considerando um servidor com modo de cluster SNC-4 e memória MCDRAM configurada em modo flat, a Listagem 1 mostra um exemplo de execução de uma aplicação teste na memória MCDRAM. Nesse exemplo toda alocação e desalocação do programa acontecerá na MCDRAM, caso o programa necessite alocar mais de 16 GB a aplicação termina com erro.

---

[6] http://memkind.github.io/memkind/

```
“numactl -m 4,5,6,7 ./teste”
```

Listagem 1. Exemplo de Execução de uma Aplicação na MCDRAM.

Outra forma de utilizar a memória MCDRAM é por meio da biblioteca memkind que disponibiliza instruções para alocar e desalocar dados na MCDRAM, por meio de duas funções chamadas *hbw_malloc()* e *hbw_free()*, que possuem os mesmos parâmetros de entrada e saída das funções *malloc()* e *free()* tradicionais, e portanto, podem ser usadas para substituir tais chamadas de modo transparente. Esse método exige alterações no código, porém permite um controle mais flexível, por exemplo, determinando quais estruturas de dados devem ser alocadas na MCDRAM e quais devem ser alocadas na DRAM.

Um exemplo de aplicação que pode ter impacto ao ser executada armazenando as estruturas de dados na MCDRAM é de transposição de matrizes **[19]**, pois envolve um grande número de movimentações de dados na memória. Uma avaliação de desempenho comparando a execução da aplicação em um servidor Xeon arquitetura Haswell, um servidor KNL usando DRAM e um serdiro KNL usando MCDRAM, mostrou que o KNL com MCDRAM apresenta o melhor desempenho[7], demonstrando que a MCDRAM trouxe um impacto positivo no desempenho dessa aplicação.

[7] https://github.com/intel-unesp-mcp/KNL-Short-Course

6.4.2. Explorando Vetorização na Arquitetura KNL

A arquitetura KNL apresenta novos recursos de vetorização que podem ser acessados usando a biblioteca de instruções da Intel chamada AVX-512 **[20], [21]**. A arquitetura KNL provê suporte aos seguintes módulos da biblioteca AVX-512: Foundation, Exponencial and Reciprocal (ER), Conflict Detection (CD) e Prefetch. Na Sessão 6.4.2.1 é descrito como utilizar a biblioteca AVX-512.

Na Sessão 6.4.2.2 é descrito o módulo AVX-512 Foundation, na Sessão 6.4.2.3 é descrito o módulo AVX-512 CD, na Sessão 6.4.2.4 é descrito o módulo AVX-512 ER e na Sessão 6.4.2.5 é descrito o módulo AVX-512-Prefetch.

6.4.2.1. Utilizando a Biblioteca AVX-512

Para utilizar a biblioteca AVX-512 é necessário indicar explicitamente no momento da compilação, que se deseja utilizar o conjunto de instruções AVX-512, isso pode ser feito de duas formas:

- Utilizando o parâmetro de compilação -xMIC-AVX512
- Utilizando o parâmetro de compilação -xhost (apenas se estiver compilando a partir de um nó KNL)

Uma aplicação de exemplo que realiza soma de duas matrizes e armazena em uma terceira matriz foi utilizada para testar a compilação usando AVX-512[8]. A Listagem 2 mostra o comando de compilação para utilizar a biblioteca de vetorização AVX-512.

```
icc knl-ex1.c -o knl-ex1 -qopt-report=5 -xMIC-AVX512 -g
```

Listagem 2. Comando de Compilação Usando Biblioteca AVX-512.

A Listagem 3 mostra a mensagem após tentar executar o programa em um servidor que não possui suporte ao uso de AVX-512.

[8] https://github.com/intel-unesp-mcp/KNL-Short-Course/blob/master/knl-ex1.c

```
$ ./knl-ex1

Please verify that both the operating system and the processor
support Intel(R) AVX512F, ADX, RDSEED, AVX512ER, AVX512PF and
AVX512CD instructions.
```

Listagem 3. Resultado da Execução em um servidor sem suporte ao AVX-512.

Para identificar se a arquitetura computacional possui os recursos de hardware necessários para executar a biblioteca AVX-512 pode ser feita de duas formas:

- Consultando o parâmetro *flags* do arquivo /proc/cpuinfo;
- Utilizando o método "_may_i_use_cpu_feature" que identifica se um recurso está implementado no processador montado no computador. A Listagem 4 mostra um exemplo de como usar esse método.

```
#include <stdio.h>
#include "immintrin.h"

int main(int argc, char *argv[]) {

  printf("AVX512F  %d",_may_i_use_cpu_feature(_FEATURE_AVX512F));
  printf("AVX512ER %d",_may_i_use_cpu_feature(_FEATURE_AVX512ER));
  printf("AVX512PF %d",_may_i_use_cpu_feature(_FEATURE_AVX512PF));
  printf("AVX512CD %d",_may_i_use_cpu_feature(_FEATURE_AVX512CD));
  printf("AVX512VL %d",_may_i_use_cpu_feature(_FEATURE_AVX512VL));
  printf("AVX512BW %d",_may_i_use_cpu_feature(_FEATURE_AVX512BW));
  printf("AVX512DQ %d",_may_i_use_cpu_feature(_FEATURE_AVX512DQ));
  printf("AVX512VBMI %d",_may_i_use_cpu_feature(_FEATURE_AVX512VBMI));

  return 0;
}
```

Listagem 4. Exemplo de Uso da Função may_i_use_cpu_feature.

6.4.2.2. AVX-512 Foundation (F)

As funções da biblioteca AVX2 utiliza até 256 bits de registradores vetoriais, que são identificados como **YMM**. O AVX-512 Foundation (F) é um sub-conjunto da biblioteca AVX-512 que disponibiliza versões de diversas funções básicas da biblioteca AVX2 utilizando registradores de 512 bits, que são identificados como **ZMM**. A Listagem 5 mostra o fragmento de um programa que foi vetorizado usando AVX-512.

```
for (auxcont=0; auxcont<SIZE; auxcont++){
      o[auxcont]=a[auxcont] + b[auxcont];
}
```

Listagem 5. Fragmento de Código Compilado para AVX-512.

A Listagem 6 mostra dois fragmentos de código de máquina gerados a partir da compilação do fragmento de código da Listagem 5 para AVX-512 e para AVX2. No código da esquerda (compilado com AVX-512) é possível verificar que os métodos realizam chamadas para os registradores **ZMM**, e no código a direita (compilado com AVX2) é possível verificar que os mesmos métodos realizam chamadas para os registradores **YMM**.

```
vmovups (%rsi,%r14,4), %zmm2
vmovups 64(%rsi,%r14,4), %zmm3
vaddps (%rdi,%r14,4), %zmm2, %zmm4
vaddps   64(%rdi,%r14,4),   %zmm3,
%zmm5
vmovups %zmm4, (%r9,%r14,4)
vmovups %zmm5, 64(%r9,%r14,4)
```

```
vmovups (%rdi,%rcx,4), %ymm0
vaddps (%r10,%rcx,4), %ymm0, %ymm1
vmovups %ymm1, (%r9,%rcx,4)
vmovups 32(%rdi,%rcx,4), %ymm2
vaddps    32(%r10,%rcx,4),   %ymm2,
%ymm3
vmovups %ymm3, 32(%r9,%rcx,4)
```

Listagem 6. Fragmento do código de máquina gerado pela compilação com AVX-512 e AVX2 Foundation.

A aplicação N-body **[22]** é um exemplo de aplicação massivamente paralela que apresenta um impacto positivo ao ser executada utilizando funções vetorizadas. Uma avaliação comparativa executando o N-body no Xeon arquitetura Haswell e no KNL mostrou que um tempo de execução menor foi obtido no KNL[9], por conta da biblioteca AVX-512.

[9] https://github.com/intel-unesp-mcp/KNL-Short-Course

6.4.2.3. AVX-512 Conflict Detection (CD)

As instruções de detecção de conflito do módulo AVX-512 CD têm como objetivo resolver o problema de escrita concorrente em um endereço de memória, que impede com que o compilador realize vetorização automática em casos onde ocorrem acesso indireto à memória.

Um exemplo de laço com essas características é mostrado na Listagem 7. Nesse laço, a cada iteração é atualizada uma posição do vetor A, que é determinada pelo valor contido na posição i do vetor B. Esse tipo de acesso é chamado indireto, pois para se ter acesso a posição de A é necessário primeiro identificar o valor contido na posição i do vetor B.

```
for(i = 0; i < SIZE; i++) {
    A[B[i]] += 1.0f/C[i] + auxval;
}
```

Listagem 7. Exemplo de um Laço com Acesso Indireto à Memória.

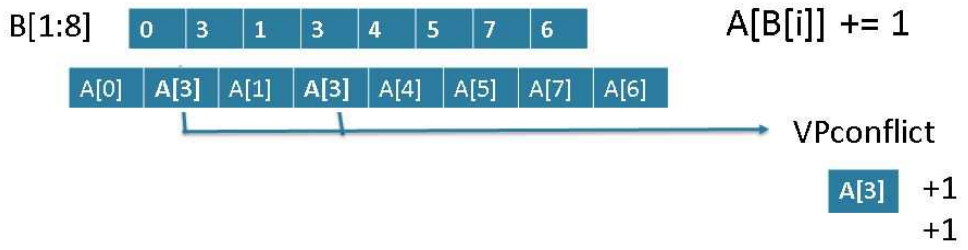


Figura 7. Exemplo do Uso das Instruções de Detecção de Conflito do AVX-512.

O compilador não consegue vetorizar esse código automaticamente, pois diferentes posições de B podem retornar valores iguais, e nesse caso, a instrução vetorial faria dois ou mais acessos simultâneos em uma mesma posição de A, o que pode gerar erros no resultado final. As instruções do módulo CD detectam os valores repetidos de B a cada execução vetorial e realizam a operação de modo sequencial para as posições repetidas de A, permitindo assim que o compilador vetorize o código. Uma condição para que essa vetorização seja feita corretamente é que o vetor de indireção, nesse exemplo o B, seja do tipo inteiro. A Figura 7 mostra um exemplo de execução do código da Listagem 7 considerando que a variável B possui o número 3 repetido em duas posições. Nesse contexto, a instrução VPconflict identifica essa situação e executa os passos em índices repetidos sequencialmente.

A Listagem 8 mostra o código fonte gerado a partir da compilação do código da Listagem 7. A instrução vpconflictd foi adicionada para vetorizar o código, resolvendo os possíveis acessos concorrentes à uma mesma posição na memória.

```
vmovups    (%rdi,%rsi,4)
vpxord     %zmm1, %zmm1, %zmm1
kmovw      %k1, %k2
vgatherdps (%r10,%zmm5,4), %zmm1{%k2}
vaddps     %zmm2, %zmm0, %zmm3
vpconflictd %zmm5, %zmm2
vptestmd   .L_2il0floatpacket.2(%rip), %zmm2, %k0

kmovw      %k0
```

Listagem 8. Fragmento de Código de Máquina Mostrando o Uso da Instrução de Detecção de Conflito.

6.4.2.4. AVX-512 Exponential and Reciprocal (ER)

Esse módulo provê instruções otimizadas para cálculo de funções exponenciais, funções inversas e raiz quadrada. Tais instruções estão disponíveis em precisão dupla e simples. Isso representa um grande avanço em relação as versões anteriores. Na versão IMCI tais funções são implementadas apenas para precisão simples, e na versão AVX2 apenas a inversa está disponível. Dessa forma, todas as outras implementações são possíveis apenas usando uma combinação de outras instruções.

Um exemplo de aplicação que tem grandes vantagens ao ser vetorizado com suporte desse módulo é a aplicação *Option Price* **[23]**. Que implementa um modelo de simulação de preços de ações. Um fragmento dessa aplicação é mostrado na Listagem 9.

```
for (  int no_iterations =0; no_iterations<100; no_iterations++)
{
      float c  = european_call_opt(Si,X,r,b,sigma,time);
      float d1 =
      (logf(Si/X)+(b+0.5f*sigma_sqr)*time)/(sigma*time_sqrt);
          float cndd1=cnd_opt(d1);
      g=(1.0f-rq2)*Si-X-c+rq2*Si*expbr*cndd1;
      gprime=(
      1.0f-rq2)*(1.0f-expbr*cndd1)+rq2*expbr*n_opt(d1)*(1.0f/(sig
      ma*time_sqrt));
      Si=Si-(g/gprime);
};
```

Listagem 9. Fragmento de Código da Aplicação Option Price.

No código de máquina mostrado na Listagem 10, a partir da compilação da aplicação Option Price, está destacado a instrução **vgetexpps** que tem como objetivo fazer o cálculo de funções exponenciais, e faz parte do módulo ER do AVX-512.

```
vgetmantps $0xb, %zmm5, %zmm18
vgetexpps %zmm5, %zmm5
vgetexpps %zmm18, %zmm16
vpsrld $0x13, %zmm18, %zmm0
vsubpsl  0x5e27(%rip){1to16}, %zmm18, %zmm17
vsubps %zmm16, %zmm5, %zmm18
vpermpsz  0x59a7(%rip), %zmm0, %zmm5
vmulpsl  0x5e11(%rip){1to16}, %zmm18, %zmm16
vpermpsz  0x5953(%rip), %zmm0, %zmm18
```

Listagem 10. Fragmento de Código de Máquina mostrando o Uso da Instrução vgetexpps.

6.4.2.5. Instruções para Prefetch (PF)

O *prefetch* é a ação de transferir de forma antecipada, da memória principal para a memória cache, conjuntos de dados que serão utilizados pelas próximas instruções que entrarão em execução. Em geral, os processadores modernos possuem mecanismos para realizar *prefetch* de modo automático, o critério para decidir quais dados devem ser transferidos é baseado em uma análise do laço.

Para laços que possuem padrões de acesso a memória complexo e, portanto, difíceis de predizer, o *prefetch* automático realiza a transferência antecipada com uma baixa taxa de acertos, e acaba trazendo pouco impacto de melhoria de desempenho na execução de instruções. Nesse sentido, desde a versão SSE são oferecidas instruções para realizar *prefetch* manualmente.

As instruções de *prefetch* do AVX-512 são capazes de transferir 8 ou 16 elementos simultaneamente para a cache L1 ou L2. Até as versões anteriores era possível transferir apenas uma linha de cache por vez (64 bytes).

6.5. Conclusões

Nesse minicurso foi apresentada as técnicas mais populares para explorar paralelismo em arquiteturas paralelas híbridas, foi apresentada as especificidades da arquitetura KNL, bem como as oportunidades para otimização de desempenho de aplicações na arquitetura KNL, mostrando exemplos práticos.

Referências

[1] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, 1 edition. Boca Raton, FL: CRC Press, 2010.

[2] A. Heinecke, M. Klemm, and H. J. Bungartz, "From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture," *Comput. Sci. Eng.*, vol. 14, no. 2, pp. 78–83, Mar. 2012.

[3] T. G. Robertazzi, "InfiniBand," in *Introduction to Computer Networking*, Springer, Cham, 2017, pp. 29–34.

[4] "Offload Computations from Servers with an Intel® Xeon Phi™ Processor | Intel® Software." [Online]. Available: https://software.intel.com/en-us/articles/how-to-use-offload-over-fabric-with-knights-landing-intel-xeon-phi-processor. [Accessed: 10-Oct-2017].

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[6] Silvio Stanzani, R. cóbe, R. Iope and Igor Freitas, " Introdução à Vetorização em Arquiteturas Paralelas Híbridas" in *XVII Simpósio em Sistemas Computacionais de Alto Desempenho*, 1st ed., vol. 1, D. Wanderson, Ed. Brasil: Sociedade Brasileira de Computação, 2016.

[7] "Overview: Intrinsics Reference | Intel® Software." [Online]. Available: https://software.intel.com/en-us/node/523353. [Accessed: 09-Oct-2017].

[8] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. Wiley Publishing, 2008.

[9] Silvio Stanzani, R. Cóbe, and R. Iope, "Introdução à Programação Multithreaded: explorando arquiteturas heterogêneas e vetorização com OpenMP 4," in *Escola Regional de Alto Desempenho do Rio Grande do Sul.*, 1st ed., vol. 1, R. Righi, Ed. Brasil: Sociedade Brasileira de Computação, 2016, pp. 89–112.

[10] S. Saini *et al.*, "Early Multi-node Performance Evaluation of a Knights Corner (KNC) Based NASA Supercomputer," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 57–67.

[11] A. Sodani, "Knights landing (KNL): 2nd Generation Intel; Xeon Phi processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, pp. 1–24.

[12] "Clustering Modes in Knights Landing Processors," *Colfax Research*, 11-May-2016. .

[13] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview," *Queue*, vol. 11, no. 7, p. 40:40–40:51, Jul. 2013.

[14] "MCDRAM as High-Bandwidth Memory (HBM) in Knights Landing Processors: Developer's Guide," *Colfax Research*, 11-May-2016. .

[15] S. Li, K. Raman, and R. Sasanka, “Enhancing Application Performance using Heterogeneous Memory Architectures on a Many-Core Platform,” *2016 Int. Conf. High Perform. Comput. Simul. Hpcs 2016*, pp. 1035–1042, 2016.

[16] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 4.0*. 2013.

[17] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, 1 edition. Beijing ; Sebastopol, CA: O’Reilly Media, 2007.

[18] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” *IEEE Test*, vol. 12, no. 3, pp. 66–73, May 2010.

[19] A. Vladimirov, “Chapter 24 - Profiling-Guided Optimization,” in *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, vol. 1, J. Reinders and J. Jeffers, Eds. Boston, MA, USA: Morgan Kaufmann, 2015, pp. 397–423.

[20] “Intel® AVX-512 Instructions | Intel® Software.” [Online]. Available: https://software.intel.com/en-us/blogs/2013/avx-512-instructions. [Accessed: 06-Oct-2017].

[21] “Capabilities of Intel® AVX-512 in Intel® Xeon® Scalable Processors (Skylake),” *Colfax Research*, 19-Sep-2017. .

[22] A. Duran and L. Meadows, “Chapter 9 - A Many-Core Implementation of the Direct N-Body Problem,” in *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, vol. 1, J. Reinders and J. Jeffers, Eds. Boston, MA, USA: Morgan Kaufmann, 2015, pp. 159–174.

[23] S. Li, “Chapter 8 - Parallel Numerical Methods in Finance,” in *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, vol. 2, J. Reinders and J. Jeffers, Eds. Boston, MA, USA: Morgan Kaufmann, 2015, pp. 113–137.