Benchmark da linguagem Bend em comparativo com Python e OpenMP

Arthur M. Passos[1], **Natan M. Passos**[1], **Calebe P. Bianchini**[1]

[1]Programa de Pós-graduação em Computação Aplicada – PPGCA
Faculdade de Computação e Informática – FCI
Universidade Presbiteriana Mackenzie – SP – Brasil

{arthurmoreirap, natanmrrpassos}@gmail.com, calebe.bianchini@mackenzie.br

Resumo. *Este artigo apresenta uma análise comparativa do desempenho da linguagem de programação Bend em relação ao Python (utilizando* `multiprocessing`*) e C (utilizando OpenMP) no contexto de uma Simulação de N-Body, um problema clássico de física computacional. O estudo explora as capacidades da linguagem Bend, que se destaca por abstrair a complexidade do paralelismo, facilitando a escrita de código paralelo sem a necessidade de controle explícito pelo programador. A implementação foi comparada quanto ao tempo de execução, uso de CPU e memória. Os resultados evidenciam o potencial do Bend em proporcionar uma programação paralela eficiente, mesmo em cenários computacionalmente intensivos.*

1. Introdução

O avanço no desempenho dos sistemas de computação foi historicamente impulsionado pelo aumento das frequências dos processadores e pela miniaturização dos transistores. No entanto, esses métodos atingiram limitações físicas e térmicas, levando a indústria de tecnologia a explorar o paralelismo como uma solução para continuar melhorando o desempenho computacional [Sutter 2005].

Paralelismo refere-se à execução simultânea de várias tarefas ou processos. Isso permite o uso mais eficiente dos recursos de hardware ou reduzindo o tempo de execução, especialmente em arquiteturas de múltiplos núcleos e GPUs (*Graphics Processing Units*).

Essa necessidade de aproveitar ao máximo o hardware disponível emerge principalmente de aplicações como simulações científicas, aprendizado de máquina e processamento de imagens. Elas requerem capacidades de processamento que muitas vezes ultrapassam o que um único núcleo de CPU pode fornecer.

Apesar dos benefícios claros, o paralelismo é um problema intrinsecamente complexo de lidar. As dificuldades incluem a sincronização entre *threads*, a gestão de recursos compartilhados, a prevenção de condições de corrida e a minimização de sobrecarga de comunicação entre *threads*. Além disso, escrever código paralelo que seja eficiente e livre de erros é uma tarefa desafiadora que exige um profundo entendimento tanto do hardware subjacente quanto das técnicas de programação paralela.

Dessa necessidade, Bend surge como uma nova linguagem de programação projetada para simplificar o desenvolvimento de aplicações paralelas [HigherOrderCompany 2023]. Inspirada por paradigmas funcionais e sintaxe do Python,

a linguagem Bend visa reduzir a complexidade associada à programação paralela, oferecendo abstrações mais altas e seguras que ajudam a evitar erros comuns, como condições de corrida e *deadlocks*. Com um foco em facilitar a escrita e manutenção de código paralelo, Bend se apresenta como uma alternativa promissora para desenvolvedores que buscam aproveitar o paralelismo sem se perder em detalhes técnicos intrincados.

Para analisar comparativamente as capacidades do Bend, foram selecionados dois *frameworks* comuns de paralelismo: o OpenMP em C e a biblioteca `multiprocessing` de Python 3, que foram aplicados ao problema de Simulação de N-Body [Silva et al. 2022]. A Simulação de N-Body é um problema clássico na física computacional que modela a interação de um sistema de corpos sob a influência de forças, como a gravidade. Este problema é computacionalmente intensivo e exige uma gestão eficiente de recursos de CPU e memória para ser resolvido em um tempo razoável.

Comparar desempenho Bend (na última versão 0.2) e OpenMP (na última versão estável 5.2) por si só é uma comparação injusta em alguns aspectos [Pereira et al. 2017]: diferença de idade das linguagens (Bend de 2023 e OpenMP de 1997), maturidade de teste, maturidade de desenvolvimento, contraste de uma linguagem de alto nível (Bend) e baixo nível (C/C++) tratando os aspectos de paralelismo manualmente, que não é o caso no Bend. Nesse contexto o OpenMP representa um constaste de um *framework* de baixo nível maduro. No outro extremo, tem-se Python, uma linguagem de *script* focada em simplicidade e clareza, em detrimento de desempenho e controle de baixo nível, por mais que ainda seja necessário um controle e cuidado sobre a aplicação do paralelismo.

Dessa forma, o presente trabalho busca principalmente identificar a posição atual do desempenho e aproveitamento de recursos do Bend em relação a esses dois limiares do espectro dadas implementações simples, sem a necessidade de controle dedicado do paralelismo.

Para esse comparativo, será utilizada a Simulação de N-Body avaliando aspectos cruciais como tempo de execução, uso dos núcleos de CPU e uso de memória. Esta comparação não só demonstra a eficácia do Bend na simplificação da programação paralela, mas também oferece *insights* sobre o seu desempenho relativo em um cenário realista e desafiador.

Vale ressaltar que a documentação do Bend [HigherOrderCompany 2023] deixa claro que ainda há uma extensa trilha de trabalho para otimizações do compilador, mesmo no escopo de tarefas sequenciais. O projeto apenas enfatiza fortemente possuir a capacidade de gerenciar a criação de mais de 10.000 *threads* sem necessidade do controle do usuário, desde que a tarefa não seja intrinsecamente sequencial.

2. Linguagem Bend

Uma das características do Bend é seu alinhamento com o paradigma funcional, tendo forte inspiração Haskell. Esse paradigma é um modelo de programação que trata a computação como a avaliação de funções matemáticas. Em vez de usar estados mutáveis e variáveis, este enfatiza o uso de funções puras, que não têm efeitos colaterais. Isso torna o comportamento do código mais previsível e fácil de entender [Sebesta 2018]. Este paradigma facilita a decomposição de problemas em funções menores e independentes, que podem ser executadas em paralelo de maneira mais eficiente. Assim, oferece suporte completo para recursos como:

- Recursão Irrestrita: capacidade de uma função chamar a si mesma sem restrições.
- Funções de ordem superior: funções que aceitam outras funções como parâmetros e/ou retornam funções como resultado.
- *Closures*: funções que capturam variáveis do seu contexto de criação, permitindo acessar essas variáveis fora do contexto original.
- *Continuations*: representações do estado de um programa em um ponto específico, permitindo retornar a esse ponto posteriormente.

2.1. HVM2 – Higher-order Virtual Machine

O diferencial principal do Bend está por trás da linguagem, o HVM2 (*Higher-order Virtual Machine 2*) [Taelin 2024], que é uma máquina virtual projetada para execução massivamente paralela usando Redes de Interação, principalmente avaliadores de Combinadores de Interação.

Redes de Interação são um modelo baseado em grafos introduzido por Lafont [Lafont 1989], oferecendo uma alternativa mais eficiente às máquinas de Turing e ao cálculo *lambda*. Em redes de interação, a computação é representada por reescrita de grafos, onde os nós (denominados agentes) interagem entre si através de regras específicas.

Já os Combinadores de Interação são um tipo de Rede de Interação, introduzidos por Lafont [Lafont 1997] e que otimizam ainda mais este modelo. Eles permitem uma execução minimalista e altamente paralela de programas, reduzindo operações complexas em transformações locais mais simples. Cada interação entre os agentes é reduzida aplicando regras simples de aniquilação e comutação (exemplos de operações na Figura 1), permitindo uma computação paralela eficiente sem a sobrecarga de mecanismos tradicionais de sincronização, como *locks* e *mutexes*.

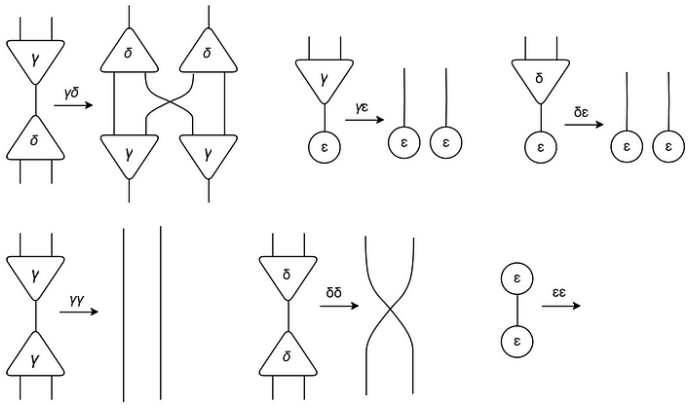

Figura 1. Ilustração de algumas operações dos Combinadores de Interação por grafos [de L. Nogueira 2024] .

Para implementar esses combinadores, o HVM2 compila programas escritos em linguagens de alto nível para uma representação intermediária que pode ser executada em *hardware* paralelamente, como GPUs, com quase ideal aceleração linear baseada na contagem de núcleos. Essa abordagem permite que o Bend execute programas em *hardware* de maneira extremamente eficiente sem que o desenvolvedor precise se preocupar com a implementação de baixo nível do paralelismo, podendo gerar códigos sequenciais com Rust, paralelos para CPU com C e CUDA para GPU.

Com esse modelo de execução, o Bend e o HVM2 facilitam a criação de programas paralelos que são, ao mesmo tempo, eficientes e fáceis de entender, aproveitando o poder da execução massivamente paralela sem os desafios comuns associados à programação paralela tradicional.

3. N-Body

Na escolha do problema cogitou-se questões clássicas do paralelismo, como multiplicação de matrizes e algoritmos de ordenação. Contudo, foi priorizada a seleção de um problema com menor necessidade de aumentar a massa de dados para escalar o processamento, devido às limitações técnicas no Bend ao lidar com grandes conjuntos de dados quando utilizado o código gerado em C.

Dessa forma, foi selecionada a Simulação de N-Body por ser um algoritmo simples podendo escalar infinitamente a quantidade de iterações dado um conjunto inicial de corpos [Silva et al. 2022].

3.1. Descrição do Problema Simulação de N-Body

A Simulação de N-Body é um problema clássico na física computacional e astrofísica, onde se modela a evolução de um sistema de corpos sob a influência de forças mútuas, como a gravidade. Este problema é altamente computacional, pois requer o cálculo das interações entre cada par de corpos em um sistema de N corpos, resultando em uma complexidade de $O(N^2)$.

Nesse problema, cada corpo (ou partícula) no sistema exerce uma força sobre os outros corpos. No caso da gravidade, a fórmula básica para a força gravitacional entre dois corpos é dada pela Lei da Gravitação Universal de Newton:

$$F = G\frac{m_1 m_2}{r^2}$$

Onde F é a força gravitacional entre os corpos, G é a constante gravitacional, m_1 e m_2 são as massas dos corpos, r é a distância entre os centros dos corpos. O objetivo é calcular as posições e velocidades dos corpos ao longo do tempo, levando em consideração essas forças.

3.2. Algoritmo de Simulação

O algoritmo básico para a Simulação de N-Body consiste nos seguintes passos:

- Inicialização: definir as posições, velocidades e massas dos N corpos.
- Cálculo das Forças: para cada par de corpos, calcular a força gravitacional mútua.
- Atualização das Velocidades: usar as forças calculadas para atualizar as velocidades dos corpos.
- Atualização das Posições: usar as velocidades atualizadas para calcular as novas posições dos corpos.
- Iteração: repetir os passos 2-4 para cada intervalo de tempo até atingir a duração total da simulação.

Código 1. Trecho da compute_forces_parallel para separar processos em Python

```
1 def compute_forces_parallel(particles, G=6.67430e-11):
2     num_particles = len(particles)
3     num_chunks = cpu_count()
4     chunk_size = num_particles // num_chunks
5     args = [(particles, i * chunk_size, (i + 1) * chunk_size, G) for i
      in range(num_chunks)]
6     with Pool(processes=num_chunks) as pool:
7         forces_chunks = pool.map(compute_forces_chunk, args)
8     forces = [Vector3D(0, 0, 0) for _ in range(num_particles)]
9     for i in range(num_chunks):
10        start = i * chunk_size
11        end = (i + 1) * chunk_size
12        for j in range(start, end):
13            forces[j] = forces_chunks[i][j - start]
14    return forces
```

4. Implementação do Paralelismo

A Simulação de N-Body é um problema clássico em física computacional que se beneficia enormemente de abordagens paralelas, devido à necessidade de calcular interações entre múltiplos corpos. Implementações paralelas são essenciais para aproveitar ao máximo o *hardware* disponível e reduzir o tempo de execução. Para comparar a eficiência do Bend, Python com `multiprocessing` e C com OpenMP, foram desenvolvidas implementações paralelas da Simulação de N-Body para cada uma das linguagens.

4.1. Implementação em Python com `multiprocessing`

A implementação em Python faz uso da biblioteca `multiprocessing` para paralelizar o cálculo das forças entre as partículas. A abordagem consiste em dividir as partículas em *chunks* que são processados em paralelo por múltiplos processos. Cada processo é responsável por calcular as forças em um subconjunto das partículas. Esse cálculo é feito considerando a interação de cada partícula do subconjunto com todas as outras partículas. Após o cálculo, os resultados são combinados para obter a força total em cada partícula.

Primeiramente, definiu-se uma classe `Vector3D` para representar vetores tridimensionais, e uma classe `Particle` para representar as partículas, que têm posição, velocidade e massa. A função `compute_forces_chunk` calcula as forças em um subconjunto de partículas. A função `compute_forces_parallel` divide as partículas em *chunks* (linhas 1 a 5 do Código 1), cria processos paralelos para calcular as forças em cada *chunk* (linhas 6 e 7) e combina os resultados. No fluxo mais externo, a função `simulate` realiza a simulação completa, iterando sobre um número de passos de tempo.

Ainda há espaço para otimizações no processo do Python, por exemplo utilizando estruturas de dados baseadas em bibliotecas mais eficientes que as estruturas nativas do Python, como NumPy [Harris et al. 2020], contudo o objetivo deste experimento, inicialmente, é manter uma implementação simples do processo de simulação.

4.2. Implementação em C com OpenMP

A implementação em C utiliza OpenMP para paralelizar o cálculo das forças. OpenMP fornece uma interface de programação que facilita a criação de programas paralelos em

C, C++ e Fortran. A estrutura do programa é similar à do Python, mas em vez de usar `multiprocessing`, utilizou-se diretivas do OpenMP para paralelizar o laço de cálculo das forças.

Semelhante ao Python, foi definida uma estrutura `Vector3D` para representar vetores tridimensionais, e uma estrutura `Particle` para representar as partículas, que têm posição, velocidade e massa. A função `compute_chunk_forces` calcula as forças em todas as partículas, utilizando um laço paralelo com a diretiva `#pragma omp parallel for`. Na `update_chunk_particles` atualiza a posição e a velocidade das partículas com base nas forças calculadas, utilizando um laço paralelo. A função `simulate` realiza a simulação completa, iterando sobre um número de passos de tempo.

Ambas as implementações demonstram como o cálculo intensivo da Simulação de N-Body pode ser distribuído entre múltiplas unidades de processamento, melhorando significativamente o desempenho em comparação com uma abordagem sequencial. A utilização de `multiprocessing` no Python e de OpenMP no C permite que o programa utilize eficientemente os recursos de *hardware* disponíveis, resultando em uma execução mais rápida e eficiente. Contudo, é necessária a gestão e cuidado manual do que está acontecendo com os vetores na memória. Mesmo no caso do OpenMP, que não é necessário fazer a gestão da criação das instâncias paralelas, é necessário ficar atento aos dados passados para cada uma delas.

4.3. Implementação em Bend

A implementação da Simulação de N-Body em Bend é notável por sua simplicidade e eficiência na programação paralela, sem a necessidade de o desenvolvedor se preocupar com detalhes intrincados de sincronização e paralelismo. A linguagem Bend, juntamente com a *Higher-order Virtual Machine 2*, abstrai a complexidade do paralelismo, garantindo que o código seja livre de condições de corrida e *deadlocks*.

Na prática, a implementação do Bend, por ser uma linguagem principalmente funcional, se baseia em chamadas recursivas e replicação dos dados. Nesse caso, finaliza replicando a lista de partículas para chamada atualizando alguma partícula em específico. Pode parecer redundante e custoso, mas essa afinidade dos dados ao escopo facilita o processo de paralelismo de funções no Bend.

Além disso, isso fica totalmente oculto da preocupação do programador, como é possível ver no Código 2: na linha 2 é guardada a lista completa na `particles_cp` apenas para que seja possível consumir a lista *particles* pelo `fold` (que, de forma simples, representa um estrutura de "laço" para consumir estruturas de dados recursivas) e replicar a lista para cada contexto de chamada do `update_particle`, possibilitando o paralelismo.

Apesar de aparentar uma junção de conceitos complexos, tais abstrações limitam o programador de forma a fazer com que qualquer função que não tenha dependência imediata de outra seja capaz de ser processada de forma independente, habilitando o paralelismo sem nenhuma gestão direta.

4.4. Monitoramento de Métricas de Desempenho

Durante a execução dos programas, diversas métricas de desempenho foram monitoradas utilizando a ferramenta `time` do Linux. As métricas coletadas incluíram: tempo em

Código 2. Função para atualizar lista de partículas em Bend

```
def update_particles(particles, timestep):
  particles_cp = particles
  fold particles:
    case List/Nil:
        return List/Nil
    case List/Cons:
        return List/Cons { head: update_particle(particles.head,
   particles_cp, timestep), tail: particles.tail }
```

modo usuário, tempo em modo *Kernel*, tempo total, percentual de CPU durante a execução, memória RAM máxima consumida.

4.5. Procedimento de Teste

O procedimento de *benchmark* foi estruturado para automatizar a execução dos programas e a coleta das métricas mencionadas. A execução de cada programa foi realizada com o monitoramento contínuo até a conclusão da execução. As entradas das execuções foram geradas com 12 corpos gerados aleatoriamente que foram reutilizados para todos os teste em todas as linguagens. Assim, foram definidas 50.000 iterações por execução.

Para garantir a consistência dos resultados, cada programa foi executado doze vezes com as mesmas entradas aleatórias. Após cada série de execuções, os dois valores mais extremos foram removidos, com maior e com o menor tempo de execução, eliminando potenciais *outliers* que poderiam distorcer a análise.

4.6. Ambiente dos testes

O testes foram executados em uma máquina Windows 11 e WSL2 com Ubuntu 22.04.4, processador Intel i5-1235U (10c/12t) e 26 GB de RAM alocados no WSL e 500 GB SSD. Foram usadas as seguintes versões dos programas: HVM 2.0.21, Bend 0.2.36, Python 3.10.12, gcc 11.4.0. Vale ressaltar que todos os testes foram feitos no contexto do WSL2, já que Bend ainda não possui suporte nativo para Windows.

5. Resultados e discussão

Tabela 1. Média das métricas das execuções da Simulação N Body

	user_time	system_time	elapsed_time	cpu_percent	max_resident_size
Bend	718,80s	1.322,09s	255,28s	799,0%	352.223,77 KB
Python	1.637,01s	886,81s	971,87s	259,3%	44.364,40 KB
OpenMP	9,66s	0,02s	0,82s	1.182,8%	2.804,4 KB

As métricas representam respectivamente a média de: tempo em modo usuário, tempo em modo Kernel, tempo total, percentual de CPU durante a execução, memória RAM máxima consumida.

Como esperado, os resultados mostram que a implementação em C com OpenMP apresentou o melhor desempenho, principalmente em termos de tempo total e consumo de memória, sendo que o tempo foi três ordens de magnitude mais rápido que as outras. Dessa forma, é definitivamente a mais eficiente em aproveitar do hardware.

Já nas linguagens de mais alto nível, a implementação em Python foi a menos eficiente, apesar de ainda ser necessário um cuidado maior comparado a solução com paralelismo no Bend. Apesar de também ser uma linguagem de alto nível com abstrações para programação paralela, Bend teve um desempenho intermediário, destacando-se pela simplicidade de implementação, mas com um *overhead* consideravelmente maior em termos de uso de memória que o Python.

6. Conclusão

É evidente a supremacia de eficiência do C com OpenMP. Apesar disso, ao tratar de linguagens com uma abstração maior, e consequentemente mais simples para programar no geral, Bend se mostrou uma alternativa viável quanto a Python. Por mais que ainda haja espaço para otimizações no código do Python, a linguagem Bend ainda não atingiu a maturidade e tem muito espaço para otimizações internas.

Referências

de L. Nogueira, L. (2024). How bend works: A parallel programming language that "feels like python but scales like cuda". https://towardsdatascience.com/how-bend-works-a-parallel-programming-language-that-feels-like-python-but-scales-like-cuda-48be5bf0fc2c. Acesso em: Julho 2024.

Harris, C. R., Millman, K. J., van der Walt, S. J., et al. (2020). Array programming with numpy. *Nature*, 585:357–362. Acesso em: Agosto 2024.

HigherOrderCompany (2023). Bend: A high-level, massively parallel programming language. https://github.com/HigherOrderCO/Bend. Acesso em: Julho 2024.

Lafont, Y. (1989). Interaction nets. In *POPL '90*. Acesso em: Julho 2024.

Lafont, Y. (1997). Interaction combinators. *Information and Computation*, 137(1):69–101.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. a. P., and Saraiva, J. a. (2017). Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA. Association for Computing Machinery.

Sebesta, R. W. (2018). *Conceitos de Linguagens de Programação-11*. Bookman Editora.

Silva, G., Bianchini, C., and Costa, E. B. (2022). *Programação Paralela e Distribuída*. Ed. Casa do Código. https://www.casadocodigo.com.br/pages/sumario-programacao-paralela.

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm. Acesso em: Julho 2024.

Taelin, V. (2024). Hvm2: A parallel evaluator for interaction combinator. https://docs.google.com/viewer?url=https://raw.githubusercontent.com/HigherOrderCO/HVM/main/paper/PAPER.pdf. Acesso em: Julho 2024.