Capítulo 2

Perfilamento e Visualização da Escalabilidade de Aplicações Paralelas com o Parallel Scalability Suite

Felipe Santos-da-Silva, Anderson B. N. da Silva, Vitor R. G. da Silva, Carlos A. Valderrama e Samuel Xavier-de-Souza

2.1. Desempenho vs. Escalabilidade

O principal objetivo ao criar um programa paralelo é, em geral, melhorar o **desempenho**, ou seja, reduzir o tempo necessário para realizar uma tarefa. No entanto, apenas o desempenho não é suficiente para avaliar a qualidade de um programa paralelo. É igualmente importante considerar a sua **escalabilidade**, que está diretamente relacionada a **eficiência** do programa à medida que variamos o número de recursos computacionais disponíveis e o tamanho do problema que ele resolve.

Para esclarecer melhor a diferença entre desempenho e escalabilidade, considere a seguinte analogia: imagine que em um restaurante há uma equipe responsável por lavar pratos. O desempenho é medido pela rapidez com que a equipe executa essa tarefa. Se apenas uma pessoa está lavando os pratos, o desempenho será avaliado pela quantidade de pratos que essa pessoa consegue lavar em um determinado período. Se ela trabalha rapidamente e sem cometer erros, dizemos que o desempenho é alto, ou seja, estamos analisando a eficiência do trabalho individual, sem considerar outros fatores.

Agora, imagine que o número de pratos para lavar aumenta significativamente, e para lidar com essa demanda extra, mais pessoas são adicionadas à equipe. A escalabilidade se refere à capacidade da equipe de continuar lavando os pratos de maneira eficiente à medida que mais pessoas são colocadas para ajudar. Se essas novas pessoas conseguirem dividir o trabalho de forma equilibrada, sem atrapalhar umas às outras, a equipe tem boa escalabilidade.

Por outro lado, se ao adicionar mais pessoas elas começarem a se esbarrar, disputar o espaço na pia ou esperar pelas mesmas ferramentas, a eficiência da tarefa será prejudicada. Isso acontece porque o aumento de pessoas não torna necessariamente o processo mais rápido, mas sim mais confuso. Da mesma forma, em um programa paralelo, a sobrecarga causada pela necessidade de coordenar o trabalho entre diferentes núcleos de processamento pode reduzir o benefício esperado de adicionar mais recursos.

2.1.1. Speedup

Um dos métodos mais comuns para medir o desempenho de um programa paralelo é através do conceito de ***speedup***, que compara o tempo de execução sequencial com o tempo de execução paralelo. Idealmente, se conseguirmos dividir o trabalho de maneira equilibrada entre p núcleos e minimizar a sobrecarga adicional, o programa paralelo deveria ser p vezes mais rápido que o programa sequencial. Se o tempo de execução sequencial for $T_{\text{sequencial}}$ e o tempo de execução paralelo for T_{paralelo}, o ***speedup*** pode ser expresso como:

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}}.$$

Quando o *speedup* é exatamente igual ao número de núcleos, ou seja, $S = p$, dizemos que o programa apresenta *speedup* linear. No entanto, na prática, obter essa aceleração ideal é raro, devido a fatores como sobrecarga de comunicação, sincronização entre *threads* e seções críticas que exigem mecanismos como *mutexes*. Esses elementos introduzem uma forma de execução sequencial dentro do programa paralelo, diminuindo o *speedup* real.

2.1.2. Eficiência e Escalabilidade

A **eficiência** de um programa paralelo está relacionada a como os recursos (como os núcleos) são utilizados em comparação com a aceleração obtida. Formalmente, podemos definir a eficiência como:

$$E = \frac{S}{P}.$$

Uma eficiência de 100% (ou 1) indicaria que cada núcleo está contribuindo de maneira ideal para o desempenho, algo que raramente ocorre devido à sobrecarga mencionada anteriormente.

Na analogia dos pratos 2.1, em um dos cenários, ao adicionar mais pessoas para lavar pratos, a eficiência da equipe diminuía devido à falta de espaço ou à sobrecarga de coordenação. Isso reflete o que ocorre em sistemas computacionais: muitas vezes, aumentar apenas os recursos (como mais núcleos ou máquinas) pode prejudicar a eficiência se não houver um aumento proporcional no tamanho do problema a ser resolvido. Máquinas maiores, como supercomputadores, precisam geralmente de problemas maiores para manter níveis elevados de eficiência. A eficiência, nesse caso, pode variar conforme os recursos e o tamanho do problema escalam, sendo necessário um equilíbrio cuidadoso para garantir que o ganho de desempenho compense a sobrecarga adicional de coordenação.

Além disso, a **escalabilidade** é um conceito essencial ao se avaliar o desempenho de programas paralelos. Ela refere-se à capacidade de um sistema ou algoritmo de sustentar ou melhorar sua eficiência à medida que o número de recursos disponíveis, como processadores ou *threads*, aumenta[1].

A análise de escalabilidade exige a avaliação de diversas configurações de uma aplicação, enquanto a análise de desempenho se concentra em uma avaliação detalhada de uma única configuração. Embora seja possível utilizar ferramentas de criação de perfis de desempenho para a análise de escalabilidade, essa abordagem apresenta certas limitações. Devido à natureza da análise de escalabilidade, o usuário precisa executar e coletar manualmente informações de múltiplas configurações. Além disso, essas ferramentas

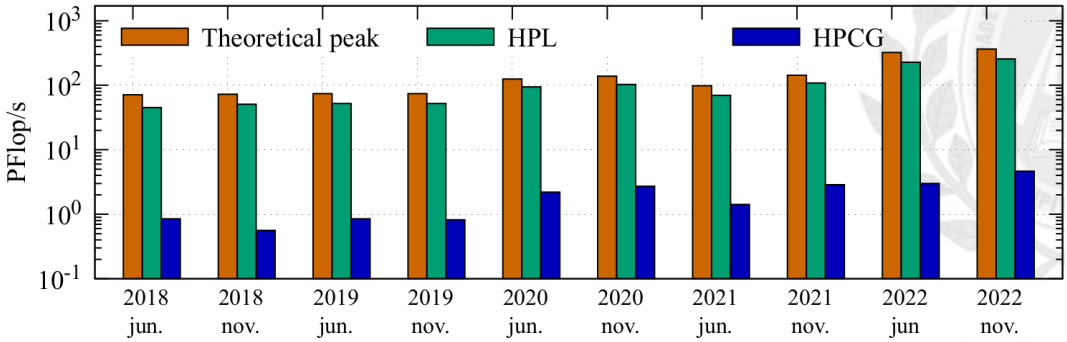

Figura 2.1. Comparação do desempenho médio dos 10 principais supercomputadores da lista Top500 (2018-2022) em termos de pico teórico, HPL e HPGC. O desempenho teórico de pico é consistentemente maior que os resultados medidos pelos benchmarks HPL e HPGC

fornecem dados excessivamente detalhados, muitas vezes além do necessário para avaliar escalabilidade. Outro ponto relevante é a ausência de artefatos visuais nativos que facilitem a interpretação dos resultados relacionados à escalabilidade.

A Figura 2.1 ilustra a comparação entre o desempenho teórico de pico e os resultados obtidos através dos benchmarks *HPL* (*High Performance Linpack*) e *HPGC* (*High Performance Conjugate Gradients*) para os 10 principais supercomputadores da lista Top500, no período de 2018 a 2022. Observa-se uma diferença significativa entre o pico teórico, representado em laranja, e os desempenhos efetivos, medidos pelo *HPL* (em verde) e *HPGC* (em azul), com o último apresentando os menores resultados. Isso evidencia que, embora grandes máquinas sejam capazes de resolver muitas tarefas simultaneamente, seu desempenho em problemas de grande escala, particularmente os mais complexos, como os medidos pelo *HPGC*, é consideravelmente inferior ao esperado. Além disso, o gráfico demonstra que, ao longo dos anos, essa tendência permanece constante, indicando limitações na eficiência dos sistemas em explorar todo o potencial teórico disponível.

A escalabilidade paralela tem se tornado uma preocupação crescente em razão do aumento da demanda por desempenho em sistemas computacionais. Com o avanço de processadores *multicore*, a eficiência no uso de múltiplos núcleos tornou-se crucial para garantir que aplicações possam lidar com volumes de dados cada vez maiores e tarefas mais complexas.

Adicionalmente, à medida que sistemas paralelos se tornam mais comuns em áreas como inteligência artificial, big data e simulações científicas, a capacidade de escalar eficientemente recursos paralelos afeta diretamente a produtividade e o custo-benefício das operações. No entanto, observa-se uma escassez de ferramentas dedicadas especificamente à análise e otimização da escalabilidade paralela, o que aumenta a complexidade desse processo.

As principais fontes de ineficiência em sistemas paralelos estão relacionadas à comunicação e sincronização entre os núcleos. A comunicação excessiva entre diferentes núcleos pode gerar um *overhead* significativo, diminuindo a eficiência global da aplicação. Além disso, a necessidade de sincronização frequente entre as *threads* pode causar

períodos de inatividade enquanto se espera a finalização de tarefas em diferentes unidades de processamento.

Outro fator importante é o desbalanceamento de carga, que ocorre quando diferentes partes da aplicação recebem quantidades desiguais de trabalho, levando a um uso desigual dos recursos. Isso resulta em computação extra para alguns processos, enquanto outros permanecem ociosos, aguardando a sincronização. Essas ineficiências aumentam o tempo total de execução e diminuem o *speedup*, dificultando a escalabilidade paralela. Esse ciclo de ineficiências, quando identificadas, podem ser mitigadas por técnicas de balanceamento de carga, que distribuem o trabalho de maneira uniforme, e técnicas de ocultação de latência, que permitem que a computação prossiga enquanto ocorrem transferências de dados.

2.1.3. Lei de Amdahl

Em 1967 Gene Amdahl fez uma observação que ficaria conhecida como a **Lei de Amdahl**[2]. A lei estabelece um limite teórico no *speedup* que pode ser alcançado com a paralelização, considerando que uma fração do código de um programa é "inerentemente sequencial"e, portanto, não pode ser paralelizada. De acordo com Amdahl, mesmo que uma grande parte do código seja paralelizada, o tempo de execução total será limitado pela parte que permanece sequencial. Isso significa que o *speedup* total não pode crescer indefinidamente, independentemente do número de processadores adicionados.

Suponhamos que 90% de um programa possa ser paralelizado e que o tempo de execução em um único processador seja $T_{\text{sequencial}} = 20$ segundos. Usando a Lei de Amdahl, podemos calcular o tempo de execução paralelizado da seguinte forma:

$$T_{\text{paralelo}} = 0,9 \times \frac{T_{\text{sequencial}}}{p} + 0,1 \times T_{\text{sequencial}} = \frac{20}{\frac{18}{p} + 2},$$

onde p representa o número de processadores. E o *speedup* será:

$$S = \frac{20}{0,9 \times \frac{T_{\text{sequencial}}}{p} + 0,1 \times T_{\text{sequencial}}} = \frac{20}{\frac{18}{p} + 2}.$$

Conforme p aumenta, o *speedup* se aproxima de um valor máximo. Neste exemplo, mesmo com 1000 processadores, o *speedup* não excede 10, evidenciando o limite imposto pela parte sequencial do código. Mais genericamente, se uma fração r do programa for intrinsecamente sequencial, o limite máximo de *speedup* será dado por $S \leq \frac{1}{r}$.

2.1.4. Lei de Gustafson

Embora a **Lei de Amdahl** seja amplamente utilizada para modelar o speedup em sistemas paralelos, ela pressupõe que o tamanho do problema permanece constante à medida que o número de processadores aumenta, o que pode limitar o potencial de paralelização. Para abordar essa limitação, John Gustafson propôs, em 1988, uma alternativa chamada **Lei de Gustafson**, que considera o aumento no tamanho do problema conforme mais processadores são adicionados.

A Lei de Gustafson sugere que, ao invés de focar na fração sequencial do código que não pode ser paralelizada, deveríamos considerar o fato de que o tamanho total do problema pode crescer com o aumento do número de processadores, fazendo com que a parte paralelizável do programa se torne mais significativa. Isso implica que, para problemas maiores, a fração do tempo gasto nas partes sequenciais do código diminui, possibilitando um *speedup* maior do que o previsto pela Lei de Amdahl.

A Lei de Gustafson é expressa pela seguinte fórmula para o speedup S:

$$S = p - (p-1) \times r,$$

onde p é o número de processadores e r, a fração sequencial do programa. Diferente da Lei de Amdahl, que impõe um limite rígido ao *speedup*, a Lei de Gustafson[3] sugere que o *speedup* cresce linearmente com o número de processadores, assumindo que o tamanho do problema cresce proporcionalmente à quantidade de recursos computacionais disponíveis.

Suponhamos que $r = 0,1$ ou seja, 10% do código é sequencial e o restante pode ser paralelizado. Para $p = 100$ processadores, o *speedup* S pode ser calculado como:

$$S = 100 - (100-1) \times 0,1 = 100 - 99 \times 0,1 = 100 - 9,9 = 90,1.$$

Esse resultado mostra que o *speedup* é de $90,1$, significativamente maior do que o limite previsto pela Lei de Amdahl para o mesmo número de processadores.

2.1.5. Tipos de Escalabilidade

Em 2.1.2 fomos apresentados rapidamente ao conceito de escalabilidade de maneira mais informal. Embora o termo "escalável"seja frequentemente usado dessa forma, aqui ele será definido de maneira mais precisa, conforme a literatura de computação paralela.

Suponha que um programa paralelo seja executado com um número fixo de *threads* e um problema de tamanho fixo, e que tenha uma eficiência inicial E. Agora, se aumentarmos o número de *threads* e também ajustarmos o tamanho do problema de maneira adequada, podemos verificar se a eficiência E é mantida. Caso seja possível encontrar uma razão correspondente de aumento do tamanho do problema para que a eficiência permaneça constante, o programa é considerado escalável.

A eficiência E de um programa paralelo pode ser descrita pela seguinte equação:

$$E = \frac{n}{\frac{n}{p} + 1} = \frac{n}{n+p}.$$

onde n é o tamanho do problema, p é o número de *threads*, $T_{\text{sequencial}} = n$ e $T_{\text{paralelo}} = \frac{n}{p} + 1$ são os tempos de execução.

Para determinar se o programa é escalável, aumentamos o número de processadores por um fator k e verificamos o fator x pelo qual o tamanho do problema deve aumentar para manter a eficiência E constante. Assim, o número de processadores se torna kp e o tamanho do problema se torna xn. A equação que devemos resolver para x é:

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}.$$

Quando $x = k$, a eficiência E permanece inalterada, ou seja, programa é escalável. Podemos simplificar a equação para:

$$\frac{xn}{xn+kp} = \frac{kn}{k(n+p)} = \frac{n}{n+p}.$$

Ou seja, se aumentarmos o tamanho do problema n na mesma proporção que aumentarmos o número de *threads* p, a eficiência se mantém constante, indicando que o programa é escalável.

2.1.6. Análise de Eficiência e Escalabilidade Paralela

Um programa é dito **fortemente escalável** quando consegue manter a eficiência E mesmo sem aumentar o tamanho do problema. Isso significa que, ao adicionar mais processadores ou I, o programa mantém uma alta eficiência sem precisar aumentar proporcionalmente o tamanho da tarefa a ser processada. Em um cenário de escalabilidade forte ideal, a eficiência E não se altera mesmo com o aumento número de processos p.

Por outro lado, um programa é considerado **fracamente escalável** quando consegue manter a eficiência E somente quando o tamanho do problema aumenta na mesma proporção que o número de processadores ou *threads*. Isso quer dizer que embora mais recursos sejam adicionados, o problema precisa crescer proporcionalmente para que a eficiência não decresça.

2.1.7. Avaliando a Escalabilidade com Tabelas

Uma forma de avaliar a escalabilidade de um programa é utilizar tabelas que correlacionam o número de núcleos de processamento com o tamanho do problema. O primeiro passo é coletar os tempos de execução do programa em diversas configurações de número de núcleos e tamanhos do problema como ilustrado na tabela abaixo.

# Núcleos	1x	2x	4x	8x	16x	32x
1	100.00	400.00	1600.00	6400.00	25600.00	102400.00
2	51.00	201.00	803.00	3201.00	12801.00	51201.00
4	27.00	102.00	402.00	1602.00	6402.00	25602.00
8	15.50	53.00	203.00	803.00	3203.00	12803.00
16	10.25	29.00	104.00	404.00	1604.00	6404.00
32	8.13	17.50	55.00	205.00	805.00	3205.00
64	7.56	12.25	31.00	106.00	406.00	1606.00
128	7.78	10.13	19.50	57.00	207.00	807.00

Tabela 2.1. Tempos de execução para diferentes números de núcleos e tamanhos de problema.

A equação que representa o tempo de execução sequencial é $T_{\text{sequencial}} = n$, on n é o tamanho do problema. Já o tempo de execução paralelo $T_{\text{paralelo}} = \frac{n}{p} + log_2 p$, onde p representa o número de núcleos utilizados.

Uma vez que os tempos de execução foram obtidos, como descrito na Tabela 2.1, podemos agora calcular o *speedup* pata cada combinação de número de núcleos e tamanho

do problema, O *speedup* é definido como a razão entre o tempo de execução sequencial $T_{\text{sequencial}}$ e o tempo de execução paralelo T_{paralelo}:

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}}.$$

Segue a tabela que representa os valores para *speedup*

# Núcleos	1x	2x	4x	8x	16x	32x
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.96	1.99	2.00	2.00	2.00	2.00
4	3.70	3.92	3.98	4.00	4.00	4.00
8	6.45	7.35	7.80	7.97	7.99	8.00
16	9.76	13.79	15.38	15.84	15.96	16.00
32	12.31	22.86	29.09	31.22	31.80	31.96
64	13.22	32.65	51.61	60.48	63.05	63.76
128	12.85	39.51	82.05	112.28	123.67	126.89

Tabela 2.2. Speedups obtidos para diferentes números de núcleos e tamanhos de problema.

Após o cálculo do *speedup* para cada valor p e n, podemos calcular a eficiência E, que definida como a razão entre o *speedup* S e o número de núcleos:

$$E = \frac{S}{p}.$$

A tabela abaixo representa o cálculo de eficiência para cada combinação possível de configuração.

# Núcleos	1x	2x	4x	8x	16x	32x
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.98	1.00	1.00	1.00	1.00	1.00
4	0.93	0.98	1.00	1.00	1.00	1.00
8	0.81	0.92	0.98	1.00	1.00	1.00
16	0.61	0.86	0.96	0.99	1.00	1.00
32	0.38	0.71	0.91	0.98	0.99	1.00
64	0.21	0.51	0.81	0.94	0.98	1.00
128	0.10	0.31	0.64	0.88	0.97	0.99

Tabela 2.3. Eficiências obtidas para diferentes números de núcleos e tamanhos de problema.

Após a obtenção da Tabela 2.3, podemos avaliar a escalabilidade do programa. Observa-se que, para tamanhos de problema menores, como $1x$, a eficiência apresenta uma queda acentuada à medida que o número de núcleos aumenta. Esse comportamento vai contra o conceito de escalabilidade forte.

No entanto, ao analisar a escalabilidade de forma linear, isto é, aumentando o tamanho do problema proporcionalmente ao número de núcleos, percebe-se que a eficiência

se mantém mais estável, próxima de 1. Isso indica que o programa consegue distribuir melhor a carga de trabalho em cenários com maior demanda de processamento, mostrando uma capacidade de escalabilidade sob essas condições.

Portanto, podemos concluir que o programa exibe escalabilidade fraca para problemas de menor dimensão, mas aproxima-se de uma escalabilidade forte quando o tamanho do problema cresce na mesma proporção que o número de núcleos.

A análise por tabelas ainda é viável com uma pequena contagem de núcleos e taxas crescentes quadraticamente, agora para análises mais detalhadas, o uso de tabelas começa a ser um problema, a Tabela 2.4 ilustra uma parte da Tabela 2.3 agora com taxas lineares.

# Núcleos	1x	2x	3x	4x	5x	6x	7x	8x	9x	10x
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
3	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
4	0.93	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
5	0.87	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
6	0.81	0.95	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
7	0.84	0.94	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
8	0.81	0.92	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00
9	0.84	0.91	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
10	0.75	0.91	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
11	0.72	0.91	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
12	0.72	0.96	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
13	0.75	0.95	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
14	0.63	0.95	0.97	0.99	1.00	1.00	1.00	1.00	1.00	1.00
15	0.63	0.94	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
16	0.61	0.86	0.96	0.99	1.00	1.00	1.00	1.00	1.00	1.00
17	0.57	0.85	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
18	0.57	0.94	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00
19	0.83	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
20	0.85	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
21	0.83	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
22	0.50	0.81	0.96	0.99	1.00	1.00	1.00	1.00	1.00	1.00
23	0.43	0.79	0.94	0.99	1.00	1.00	1.00	1.00	1.00	1.00
24	0.48	0.78	0.93	0.99	1.00	1.00	1.00	1.00	1.00	1.00
25	0.46	0.78	0.94	0.99	1.00	1.00	1.00	1.00	1.00	1.00
26	0.71	0.88	0.93	0.99	1.00	1.00	1.00	1.00	1.00	1.00
27	0.44	0.76	0.93	0.97	1.00	1.00	1.00	1.00	1.00	1.00
28	0.43	0.75	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00
29	0.42	0.74	0.86	0.95	0.98	1.00	1.00	1.00	1.00	1.00
30	0.43	0.72	0.92	0.98	1.00	1.00	1.00	1.00	1.00	1.00
31	0.72	0.81	0.91	0.98	0.99	1.00	1.00	1.00	1.00	1.00
32	0.38	0.71	0.85	0.94	0.96	0.97	0.98	0.99	1.00	1.00

Tabela 2.4. Eficiências obtidas para diferentes números de núcleos e tamanhos de problema.

A medida que o detalhamento nas análises crescem, maiores ficam as tabelas, que vale ressaltar, não são geradas por nenhuma ferramenta de perfil de desempenho, essas ferramentas fornecem os dados utilizados para uma única célula dessa tabela. O desenvolvedor que se preocupa com a escalabilidade é deixado sozinho para rodar, coletar e de alguma forma visualizar os pontos críticos e gargalos da aplicação.

Embora tabelas sejam frequentemente utilizadas para essa tarefa, elas podem ocultar tendências importantes de escalabilidade. Além disso, gráficos de linha ou barras não são adequados para visualizar esse tipo de dado, já que muitas vezes disfarçam as tendên-

cias. No caso de perfis voltados para escalabilidade, seria necessário criar uma tabela para cada região de código de interesse, o que, manualmente, se torna ainda mais demorado e, com ferramentas de perfilamento, pode ser ainda mais trabalhoso.

2.2. Introdução ao PaScal Suite

O *Parallel Scalability Suite* (PaScal Suite) é um conjunto de ferramentas para avaliar as tendências de escalabilidade de programas paralelos. Ela simplifica a execução, medição e comparação de várias execuções de um programa paralelo, permitindo a análise de tendências de escalabilidade em ambientes de configuração com diferentes quantidades de elementos de processamento e diferentes cargas de trabalho, com elementos visuais que ajudam o usuário a entender o comportamento do programa e reconhecer gargalos de escalabilidade que podem exigir uma análise de otimização mais aprofundada. O conjunto de ferramentas pode ser utilizado para auxiliar o desenvolvimento de programas paralelos executados em um único nó computacional de memória compartilhada.

2.2.1. PaScal Analyzer

O *Parallel Scalability Analyzer* [4], ou **PaScal Analyzer**, é uma ferramenta projetada para perfilar aplicações, permitindo a medição e a comparação de execuções com diferentes configurações alternativas. Seu objetivo é facilitar o entendimento da capacidade de escalabilidade de uma aplicação paralela, observando como ela utiliza os recursos computacionais disponíveis. Além disso, a ferramenta se destaca por seu baixo nível de intrusão nos programas analisados, um aspecto fundamental para compreender com precisão o comportamento e a capacidade de escalabilidade da aplicação. Após sua execução, o *Analyzer* organiza os dados coletados para uma futura análise visual. Esses dados podem incluir tempo de execução, potência e outros medidores de desempenho; contudo, neste trabalho, vamos nos concentrar em tempo de execução.

No *Analyzer*, a instrumentação, seja manual ou automática, permite que os desenvolvedores observem, de forma incremental, como diferentes partes da aplicação paralela utilizam os recursos computacionais, medindo a eficiência e o desempenho conforme diferentes configurações são aplicadas. Essa flexibilidade é útil para localizar gargalos ou otimizar o uso dos recursos disponíveis, sem adicionar sobrecarga excessiva ao código em análise.

A instrumentação automática permite que o desenvolvedor execute a ferramenta sem a necessidade de modificar o código manualmente, facilitando o perfilamento. Já a instrumentação manual envolve a adição explícita de chamadas para marcar regiões de interesse no código-fonte, controlando exatamente quais partes da execução devem ser monitoradas.

O Algoritmo 2.1 ilustra a instrumentação manual com o *Analyzer*. Para isso, é necessário incluir a biblioteca `pascalops.h` no cabeçalho do seu código em C ou C++ e delimitar a zona perfilada com as funções `pascal_start(id)` e `pascal_stop(id)`, onde `id` é um número inteiro utilizado para identificar a região medida.

A biblioteca `pascalops` fica visível para o *GCC* após a instalação do *Analayzer* e após delimitar as zonas a serem medidas, basta compilar o código com o argumento

```
#include "pascalops.h"
...
int main(){
    pascal_start(1);
    #pragma omp parallel
    { ... }
    pascal_stop(1);
    ...
}
```

Listing 2.1. Exemplo de uso do PaScal Analyzer com OpenMP

`-lmpascalops`:

```
g++ main.cpp -fopenmp -lmpascalops
```

Para realizar a instrumentação automática, basta executar o *pascalanalyzer* com o argumento opcional `-t aut`:

```
    pascalanalyzer -t aut -c 8,4,2,1 -i 100,200,400,800 -o out.
       json <binario_executavel>
```

No exemplo acima, além de utilizarmos o argumento `-t` para escolher o tipo de instrumentação, foram empregados outros argumentos como `-c`, para selecionar a quantidade de núcleos, e `-i`, para fornecer os inputs que serão usados no programa. O argumento `-o` é utilizado para especificar o nome do arquivo de saída, sendo que, caso não seja fornecido, o *Analyzer* salvará a saída com um nome padronizado.

2.2.2. PaScal Viewer

O *Parallel Scalability Viewer*[5], ou **PaScal Viewer**, é uma aplicação web projetada para visualizar métricas de desempenho em programas paralelos em sistemas de memória compartilhada. Ele aceita como entrada o arquivo de saída do *Analyzer*, fornecendo uma maneira simplificada de analisar e otimizar o código para escalabilidade paralela, introduzindo uma abordagem inovadora por meio de diagramas de cores que lembram mapas de calor. Esses diagramas são um suporte visual para identificar tendências de eficiência paralela do programa inteiro ou de partes específicas de um código paralelo. A interface do Viewer também possibilita a análise de zonas paralelas em hierarquia, permitindo o perfilamento de um código paralelo. O Viewer se encontra disponível no domínio: `https://pascalsuite.imd.ufrn.br/`.

A Figura 2.2 apresenta uma captura de tela da interface do *Viewer*, a qual exibe quatro diagramas que auxiliam na análise de desempenho de programas paralelos. Cada um desses diagramas tem uma função específica para avaliar diferentes aspectos da execução da aplicação.

O primeiro diagrama, localizado no canto superior esquerdo, mede a eficiência

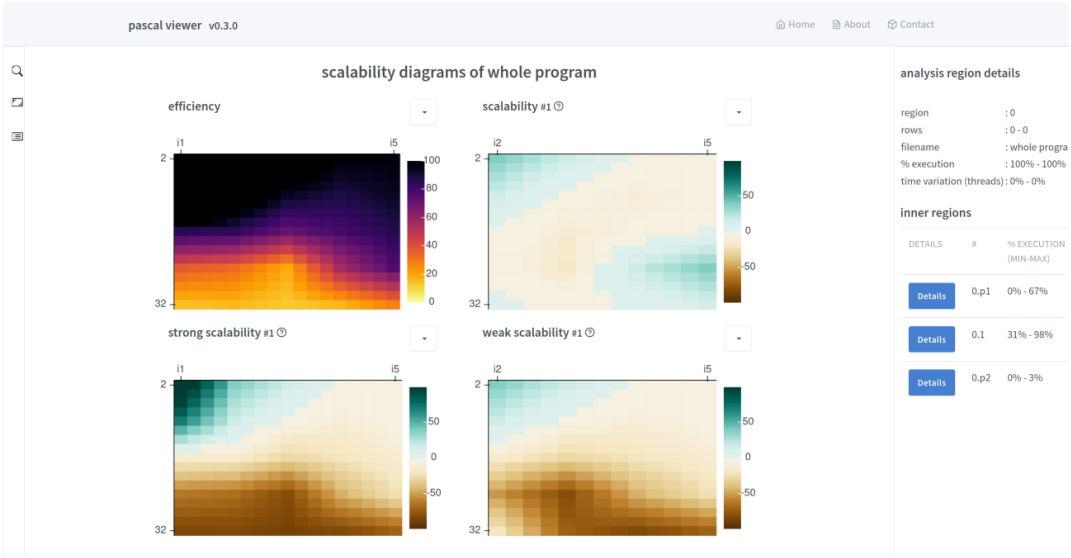

Figura 2.2. Captura de tela da interface da ferramenta web PaScal Viewer

global da aplicação paralela. Ele relaciona o tempo útil de execução ao tempo total gasto. Regiões com cores mais escuras indicam uma maior eficiência, enquanto áreas mais claras sugerem perda de desempenho.

O segundo diagrama, no canto superior direito, avalia a escalabilidade global da aplicação. Cores mais próximas do azul indicam trechos do código que são escaláveis, ou seja, que se beneficiam do aumento de recursos (como núcleos de processamento). Já os tons de marrom apontam ineficiência, sugerindo que a aplicação não está aproveitando de maneira adequada os recursos disponíveis.

Os dois diagramas no canto inferior da interface medem a escalabilidade forte (esquerda) e a escalabilidade fraca (direita). A escalabilidade forte avalia como o tempo de execução da aplicação é reduzido à medida que o número de núcleos aumenta, mantendo-se o mesmo tamanho de problema. Por outro lado, a escalabilidade fraca verifica se a aplicação consegue manter o mesmo tempo de execução ao se aumentar proporcionalmente tanto o número de núcleos quanto o tamanho do problema. Em ambos os diagramas, o esquema de cores segue o mesmo padrão do diagrama de escalabilidade global.

2.3. Estudo de Casos e Aplicabilidade

Agora, vamos avaliar a escalabilidade de um programa paralelo, o material utilizado nesta seção estará disponível no repositório público do PaScal Suite: `https://gitlab.com/lappsufrn/pascal-suite-tutorial/-/tree/minicurso2024`.

O nosso objeto de estudo principal é a análise de escalabilidade em um problema clássico de suavização de matriz utilizando o método de *Red-Black Gauss-Seidel* em um ambiente paralelo. Esse tipo de abordagem é comum em problemas de simulação numérica, como difusão de calor, simulação de fluidos e outros sistemas que utilizam equações diferenciais parciais (EDPs). O código apresentado a seguir é uma versão paralela do

algoritmo utilizando *OpenMP* para paralelização das operações sobre a matriz.

```
#include <omp.h>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <iostream>

#define MAX_ITER 1000

void initialize(double **A, int n) {
  // inicialize uma matriz de tamanho n x n
}

double matrix_calculation(double **A, int n) {
  double tmp;
  double diff = 0;
  int i, j;

#ifdef _OPENMP
#pragma omp parallel private(tmp, i, j)
  {
#pragma omp for reduction(+ : diff)
    for (i = 1; i <= n; ++i) {
      for (j = 1; j <= n; ++j) {
        if ((i + j) % 2 == 1) {
          tmp = A[i][j];
          A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j] +
              A[i][j + 1] +
                          A[i + 1][j]);
          diff += fabs(A[i][j] - tmp);
        }
      }
    }
#pragma omp single
    {
      for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j) {
          if ((i + j) % 2 == 0) {
            tmp = A[i][j];
            A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j]
                + A[i][j + 1] +
                            A[i + 1][j]);
            diff += fabs(A[i][j] - tmp);
          }
        }
      }
    }
  }
```

```
#endif // _OPENMP
  return diff;
}

void solve_parallel(double **A, int n) {
  int iters;
  double diff = 0;
  for (iters = 1; iters < MAX_ITER; ++iters) {
    diff = matrix_calculation(A, n - 1);
  }
}

int main(int argc, char *argv[]) {
  int i;
  double **A;
  int n;

  try {
    n = std::stoi(argv[1]);
} catch (const std::invalid_argument& e) {
    std::cerr << "Invalid argument: " << argv[1] << " is not a
       valid integer." << std::endl;
    return 1;
}
  A = new double *[n + 2];
  for (i = 0; i < n + 2; i++) {
    A[i] = new double[n + 2];
  }

  initialize(A, n);

  solve_parallel(A, n - 1);
}
```

Para conduzir o estudo de caso, configuramos um ambiente com diferentes quantidades de núcleos de processamento, variando entre 1, 2, 4 e 8 núcleos, utilizando matrizes de diferentes dimensões (tamanho do problema) para observar o comportamento da escalabilidade forte e fraca da aplicação.

Primeiro compilamos o código:

```
    g++ -Wall -o RB -fopenmp RB.cpp
```

E com ele compilado executamos o *Analyzer*:

```
    pascalanalyzer ./RB -c 8,4,2,1 -i 1001,1501,2001,2501 -r 5 -
       t aut -o RB.json
```

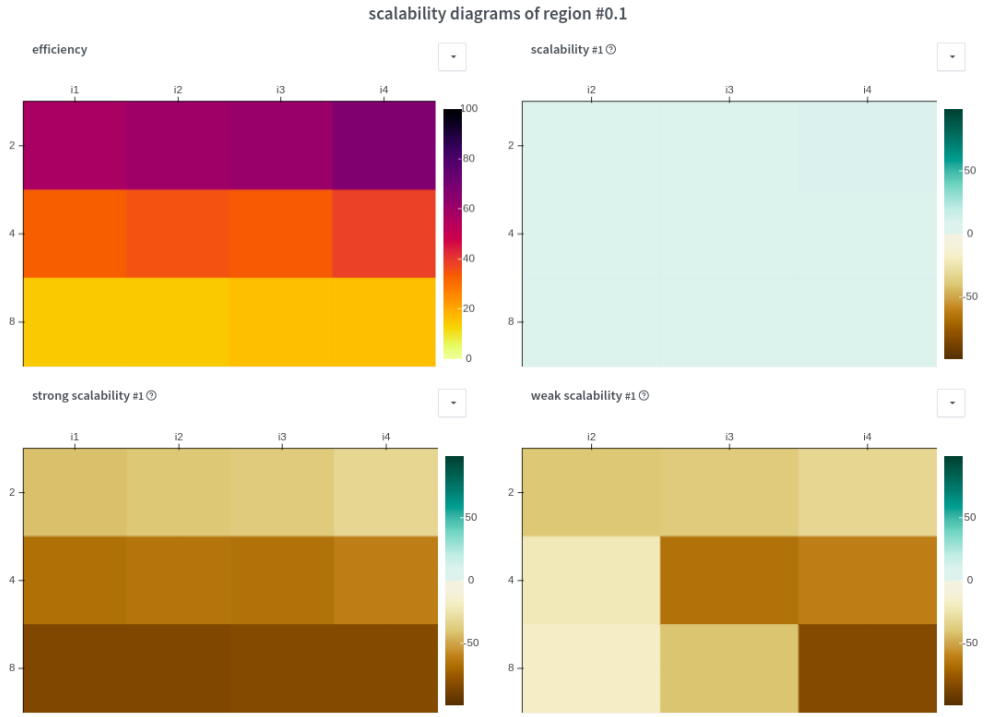

Figura 2.3. Captura de tela do PaScal Viewer com diagramas para nosso objeto de estudo na primeira leva de testes. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X

Definimos o tamanho do problema crescendo quadraticamente similar a quantidade de núcleos, o argumento `-r` define o número de repetições para cada configuração, caso não seja definido, o valor padrão é 1.

A Figura 2.3 apresenta uma captura de tela do *Viewer* contendo o arquivo de saída gerado pelo *Analyzer*. O diagrama de eficiência global demonstra uma queda de eficiência à medida que o tamanho do problema cresce proporcionalmente ao número de recursos alocados. Além disso, observa-se também uma perda de eficiência ao se manter o tamanho do problema fixo e apenas aumentar a quantidade de recursos.

Adicionalmente, o gráfico de escalabilidade global revela uma baixa escalabilidade em todos os pontos analisados. Tanto o gráfico de escalabilidade forte quanto o de escalabilidade fraca indicam que o programa não atingiu os critérios de nenhuma das duas formas de escalabilidade.

Para obter uma visão mais precisa dos possíveis gargalos em nosso código, podemos utilizar a instrumentação manual para perfilar as regiões paralelas identificadas na primeira rodada de testes com a instrumentação automática. Convenientemente, o nosso objeto de estudo possui apenas uma região paralela. Como demonstrado na Seção 2.2.1,

para realizar a instrumentação manual, precisaremos incluir a biblioteca `pascalops` em nosso código e delimitar as regiões de interesse utilizando `pascal_start(id)` e `pascal_stop(id)`.

O Algoritmo 2.2 ilustra a instrumentação manual. A Região 1 delimita toda a zona paralela, a Região 2 o primeiro laço de repetição, e, por fim, a Região 3 delimita o último laço de repetição dentro da Região 1.

Com o código devidamente modificado, vamos compilá-lo inserindo o argumento `-lmpascalops` e repetir os testes com o Analyzer, dessa vez indicando a instrumentação manual na linha de comando:

```
1    g++ -Wall -o RB -fopenmp RB.cpp -lmpascalops
```

Listing 2.3. Compilar o código com o argumento necessario para instrumentação manual

```
1    pascalanalyzer ./RB -c 8,4,2,1 -i 1001,1501,2001,2501 -r 5 -
        t man -o RB.json
```

Listing 2.4. Executar o PaScal Analyzer com -t man"para indiicar instrumentação manual

No *Viewer* é possível acessar os diagramas de cor da Região 2 e 3 como Sub-Regiões da Região 1 (Figura 2.4. Além da identificação das regiões, a iterface do *Viewer* nos fornece quanto esse trecho representa em porcentagem do tempo de execução. Essas informações indicam que Sub-Região 3 representa até 76% do tempo de execução total de nosso programa.

A Figura 2.5 apresenta uma captura de tela do *Viewer* da Sub-Região 2. Embora o diagrama de eficiência global demonstre determinada queda de eficiência a medida que os núcleos e tamanho do problema escalam, é possível concluir que esse trecho é mais eficiente que a execução por inteira (veja a Figura 2.3. Além disso, observa-se também uma perda de eficiência ao se manter o tamanho do problema fixo e apenas aumentar a quantidade de recursos.

O gráfico de escalabilidade global também revela uma baixa escalabilidade em todos os pontos analisados dessa zona.

A Figura 2.6 apresenta uma captura de tela do *Viewer* da Sub-Região 3. Diferentemente da Sub-Região 2, esse trecho apresenta uma baixa eficiência em todos os pontos do diagrama de eficiência global, o que irá refletir nos outros diagramas. Apesar da ineficiência, esse trecho também representa boa parte da porcentagem da execução de nossa aplicação. Por isso, acreditamos que o gargalo se encontre nessa zona.

Avaliando o trecho de código delimitado por `pascal_start(3)` e `pascal_stop(3)`, identificamos que o laço aninhado responsável por calcular o valor de `diff` nos índices pares de nossa matriz, está em uma diretiva `omp single` que indica que, aquele trecho deverá ser executado por somente uma thread. É possível que, se subtrairmos essa diretiva desnecessária desse trecho e adicionarmos uma diretiva `omp for` com a cláusula `reduction(+ : diff)`, vamos conseguir otimizar essa operação de redução e garantir que o laço alvo seja paralelizado.

Aplicando as modificações pensadas, a região 3 ficaria da seguinte forma:

```
pascal_start(3);
#pragma omp for reduction (+ : diff)
      for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j) {
          if ((i + j) % 2 == 0) {
            tmp = A[i][j];
            A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j]
                + A[i][j + 1] +
                              A[i + 1][j]);
            diff += fabs(A[i][j] - tmp);
          }
        }
      }
    pascal_stop(3);
  }
```

Após isso, recompilamos o código e refizemos mais uma leva de testes com o *Analyzer* agora para identificar os possíveis ganhos de nossa solução.

A 2.7 mostra que após nossa implementação, a Região 1 apresentou ganhos de eficiência em todos os pontos, embora ainda apresente baixa escalabilidade, abrindo portas para uma futura investigação mais profunda.

A 2.8 mostra Sub-Região 3 após a identificação de um possível gargalo e a implementação de nossa solução. É possível observar que, tivemos um ganho eficiência em todos os pontos de nosso diagrama de eficiência global. Os valores também melhoram para os gráficos de escalabilidade forte e fraca, embora nosso código ainda não tenha atingido um nível de escalabilidade desejável.

2.4. Conclusão

Neste capitulo, discutimos a diferença entre desempenho e escalabilidade no contexto da computação paralela e foi apresentada o PaScal Suite, um conjunto de ferramentas que auxiliam na automatização de testes e na visualização de tendências de escalabilidade paralela.

Foi feito um estudo de caso com o PaScal Suite, onde identificamos um gargalo de desempenho em um código de teste. Com as otimizações, conseguimos melhorar a eficiência, mas o programa ainda não apresenta escalabilidade ideal. O próximo passo seria uma análise mais profunda para entender melhor os motivos da baixa escalabilidade em outros pontos do código.

Referências

[1] Pacheco, P. S. (2011) *An introduction to parallel programming*. Morgan Kaufmann.

[2] Amdahl, G. M. (1967) *Validity of the single processor approach to achieving large scale computing capabilities*. In *Proceedings of the April 18-20, 1967, Spring Joint*

Computer Conference (pp. 483–485). Association for Computing Machinery.

[3] John L. Gustafson. 1988. Reevaluating Amdahl's law. Commun. ACM 31, 5 (May 1988), 532–533. https://doi.org/10.1145/42411.42415

[4] Silva, Vitor, Nóbrega-da-Silva, Anderson, Valderrama Sakuyama, C., Manneback, Pierre, and Xavier-de-Souza, Samuel (2022). "A Minimally Intrusive Approach for Automatic Assessment of Parallel Performance Scalability of Shared-Memory HPC Applications". *Electronics*, vol. 11, no. 5. DOI: 10.3390/electronics11050689.

[5] Nóbrega-da-Silva, Anderson, Cunha, Daniel, Silva, Vitor, Araújo Furtunato, Alex Fabiano, and Xavier-de-Souza, Samuel (2019). "PaScal Viewer: A Tool for the Visualization of Parallel Scalability Trends". In: *Handbook of Research on Emerging Developments and Applications of High Performance Computing*. pp. 250-264. ISBN: 978-981-13-6209-5. DOI: 10.1007/978-3-030-17872-7_15.

```
\\ outras bibliotecas
#include "pascalops.h"

double matrix_calculation(double **A, int n) {
double tmp;
double diff = 0;
int i, j;

  pascal_start(1);
#ifdef _OPENMP
#pragma omp parallel private(tmp, i, j)
  {
    pascal_start(2);

#pragma omp for reduction(+ : diff)
    for (i = 1; i <= n; ++i) {
      for (j = 1; j <= n; ++j) {
        if ((i + j) % 2 == 1) {
          tmp = A[i][j];
          A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j] +
              A[i][j + 1] +
                          A[i + 1][j]);
          diff += fabs(A[i][j] - tmp);
        }
      }
    }

    pascal_stop(2);
pascal_start(3);
#pragma omp single
    {
      for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j) {
          if ((i + j) % 2 == 0) {
            tmp = A[i][j];
            A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j]
                + A[i][j + 1] +
                            A[i + 1][j]);
            diff += fabs(A[i][j] - tmp);
          }
        }
      }
    }

    pascal_stop(3);
  }
  pascal_stop(1);
#endif // _OPENMP

  return diff;
}
```

Listing 2.2. Instrumentação manual da região paralela 1 de nosso objeto de estudo.

Figura 2.4. Acesso a Sub-Regiões 2 e 3 após instrumentação manual

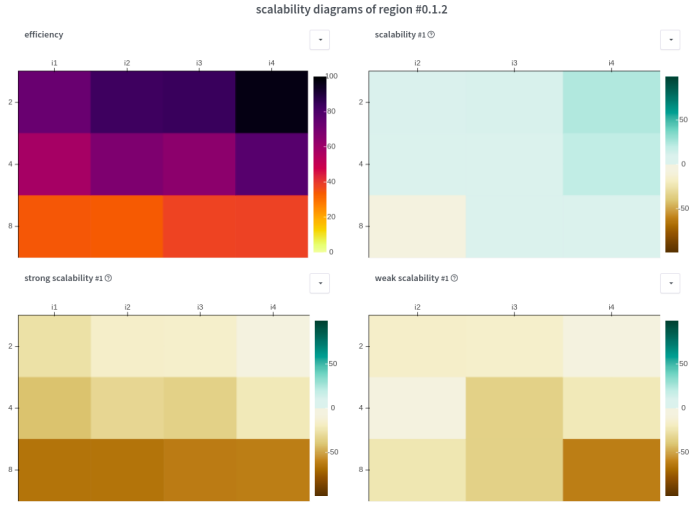

Figura 2.5. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Sub-Região 2 na segunda leva de testes. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X

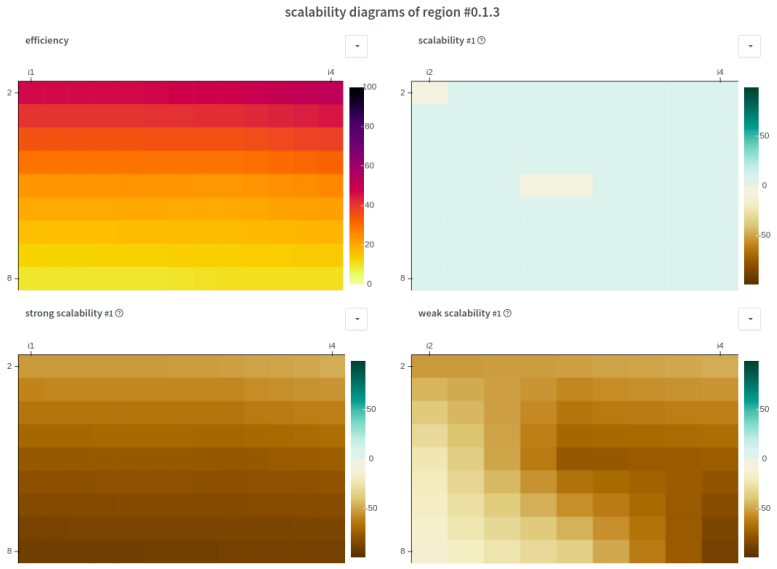

Figura 2.6. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Sub-Região 3 na segunda leva de testes. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X

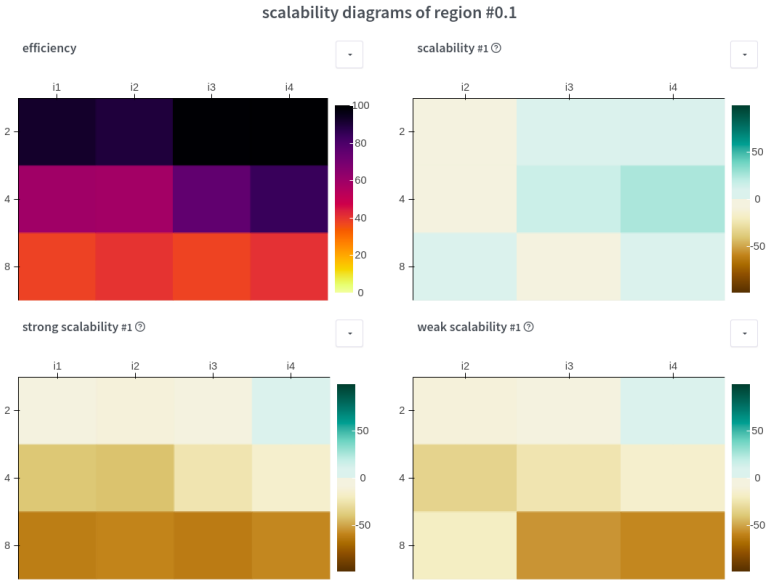

Figura 2.7. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Região 1 após nossa solução. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X

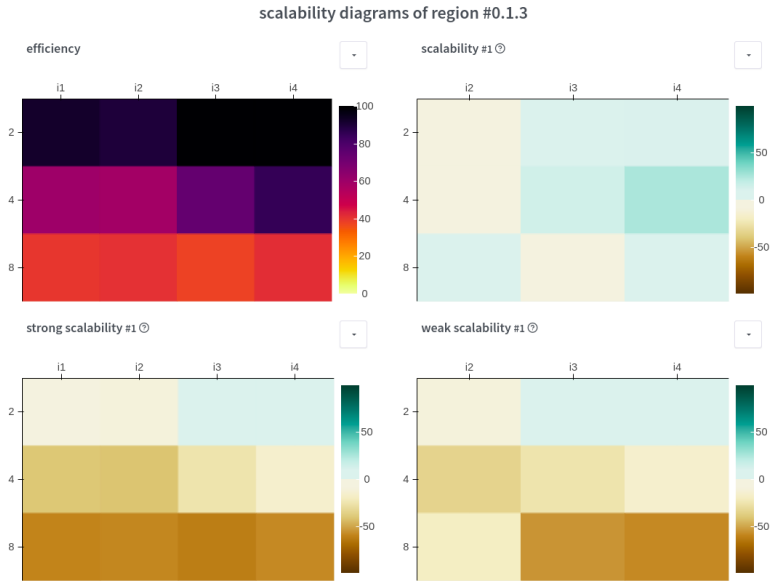

Figura 2.8. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Sub-Região 3 após nossa solução. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X