Capítulo 2

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

Felipe H. Santos-da-Silva, João B. Fernandes, Anderson B. N. da Silva, Ítalo A. S. Assis e Samuel Xavier-de-Souza

Abstract

Supercomputers play an important role in scientific and technological advances; however, many parallel applications fail to fully exploit their computational capacity. Traditional development tools for high-performance computing often focus on improving performance in specific configurations, without providing adequate support for analyzing how programs behave across different computing environments. In this context, the Parallel Architectures for Signal Processing Laboratory at UFRN developed the Parallel Scalability (PaScal) Suite, a toolset designed to analyze the scalability of parallel applications. The PaScal Suite enables the evaluation of scalability trends and the prediction of program behavior under different hardware configurations and workloads. The suite integrates the PaScal Analyzer and PaScal Viewer, which support execution, data collection, and visualization of performance metrics, helping developers identify bottlenecks and critical regions in parallel programs. This tutorial presents fundamental concepts of scalability and performance evaluation, along with practical activities demonstrating how the tool can be used for profiling and analyzing parallel applications in high-performance computing environments.

Resumo

Supercomputadores são fundamentais para o avanço científico e tecnológico, porém muitas aplicações paralelas não conseguem explorar plenamente sua capacidade de processamento. Ferramentas tradicionais de apoio ao desenvolvimento para sistemas de alto desempenho costumam focar na aceleração em configurações específicas, sem oferecer suporte adequado para analisar o comportamento do programa em diferentes ambientes computacionais. Nesse contexto, o Laboratório de Arquiteturas Paralelas para Processamento de Sinais da UFRN desenvolveu o Parallel Scalability (PaScal) Suite, um conjunto

de ferramentas voltado à análise da escalabilidade de aplicações paralelas. O PaScal Suite permite avaliar tendências de escalabilidade e prever o comportamento de programas em diferentes configurações de hardware e cargas de trabalho. O conjunto integra as ferramentas PaScal Analyzer e PaScal Viewer, que auxiliam na execução, coleta e visualização de métricas de desempenho, facilitando a identificação de gargalos e pontos críticos em aplicações paralelas. Neste minicurso, são apresentados os conceitos fundamentais de escalabilidade e métodos de avaliação de desempenho, além de atividades práticas que demonstram o uso da ferramenta no perfilamento e análise de aplicações paralelas em ambientes de computação de alto desempenho.

2.1. Análise de Desempenho e Escalabilidade

A avaliação de programas em Computação de Alto Desempenho (HPC) é composta por múltiplas métricas. Embora o objetivo primário seja frequentemente a melhoria do **desempenho absoluto**, ou seja, a redução do tempo de execução ou o aumento da vazão de tarefas, essa métrica isolada não é suficiente para caracterizar a qualidade de uma solução paralela. É fundamental analisar também a **escalabilidade** e a **eficiência**, que são medidas que descrevem como o desempenho se comporta mediante a variação dos recursos computacionais ou do tamanho do problema.

Portanto, enquanto o desempenho absoluto mede a velocidade em uma configuração específica, a escalabilidade mede a capacidade do sistema de sustentar o crescimento desse desempenho.

Para ilustrar a relação entre esses conceitos, considere a seguinte analogia: imagine uma cozinha de restaurante com uma equipe responsável por lavar pratos.

O **desempenho** pode ser observado na rapidez com que a tarefa é executada no estado atual. Se apenas uma pessoa está lavando os pratos, o desempenho é medido pela quantidade de pratos lavados por minuto. Se essa pessoa trabalha rapidamente, temos uma alta produtividade sequencial.

Já a **escalabilidade** refere-se ao comportamento dessa produtividade quando a equipe é expandida. Imagine que a demanda aumente e mais pessoas sejam contratadas. Se a equipe conseguir dividir o trabalho de forma equilibrada, onde cada novo membro adiciona uma capacidade de trabalho proporcional sem atrapalhar os demais, dizemos que o sistema possui boa escalabilidade.

Por outro lado, se a adição de novos membros fizer com que eles esbarrem uns nos outros, disputem espaço na pia ou aguardem por ferramentas compartilhadas, a eficiência do grupo cairá. No contexto computacional, isso representa o *overhead* de paralelização: o tempo gasto com comunicação e sincronização entre fluxos de execução que não contribui diretamente para a solução do problema, limitando o ganho de desempenho esperado.

2.1.1. Speedup

Um dos métodos fundamentais para quantificar o ganho de desempenho em processamento paralelo é o ***speedup***. Esta métrica estabelece a relação entre o tempo de execução da versão sequencial de um algoritmo e o tempo de sua execução paralela. Sendo $T_{\text{sequencial}}$ o tempo necessário para executar a tarefa em um único núcleo e T_{paralelo} o tempo

utilizando P núcleos, o *speedup* (S) é definido pela razão:

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}} \tag{1}$$

Teoricamente, o cenário ideal é denominado ***speedup* linear**, onde $S = P$. Nesse caso ideal, se o trabalho for dividido perfeitamente e sem custos adicionais, o programa paralelo seria exatamente P vezes mais rápido que o sequencial.

No entanto, na prática, obter essa aceleração linear é raro. Fatores intrínsecos à paralelização, como a sobrecarga (*overhead*) de comunicação, a sincronização entre *threads* e seções críticas que exigem mecanismos de exclusão mútua, tendem a degradar o desempenho. Esses elementos limitam o ganho, resultando frequentemente em um *speedup* sublinear ($S < P$).

2.1.2. Eficiência e Escalabilidade

A **eficiência** é a métrica que avalia o custo-benefício da paralelização, quantificando o quão efetivamente os recursos computacionais (como núcleos) são aproveitados para gerar o *speedup*. Matematicamente, ela é definida como a razão entre a aceleração obtida (S) e o número de processadores utilizados (P):

$$E = \frac{S}{P}. \tag{2}$$

Uma eficiência de 100% (ou 1) indicaria o cenário ideal onde o *speedup* é linear, significando que não há desperdício de recursos. Contudo, o valor de E tende a diminuir conforme P aumenta.

Retomando a analogia dos pratos (Seção 2.1), isso ocorre quando a adição de pessoas na cozinha gera conflitos por espaço. Em sistemas computacionais, aumentar apenas os recursos sem aumentar o tamanho do problema frequentemente prejudica a eficiência. Máquinas massivas, como supercomputadores, exigem problemas de grande escala para justificar o uso de seus milhares de núcleos e manter níveis elevados de eficiência.

A **escalabilidade**, portanto, é essencial para avaliar se um algoritmo consegue sustentar sua eficiência à medida que os recursos aumentam[1]. A análise de escalabilidade é mais complexa que a de desempenho pontual, pois exige a execução e coleta de dados de múltiplas configurações. Além disso, há uma escassez de ferramentas que forneçam artefatos visuais nativos e específicos para essa análise, muitas vezes entregando dados excessivamente detalhados que dificultam a interpretação rápida das tendências de escala.

A Figura 2.1 ilustra a discrepância entre o potencial teórico e a realidade prática. Ela compara o desempenho de pico teórico com os resultados dos benchmarks *HPL* e *HPGC* para os 10 principais supercomputadores (2018-2022). Observa-se que, enquanto o *HPL* (verde) se aproxima mais do pico teórico (laranja), o *HPGC* (azul), que representa problemas mais complexos e irregulares, apresenta resultados consideravelmente inferiores. Isso evidencia a dificuldade de manter a eficiência em problemas que exigem comunicação intensiva, mesmo em máquinas de ponta.

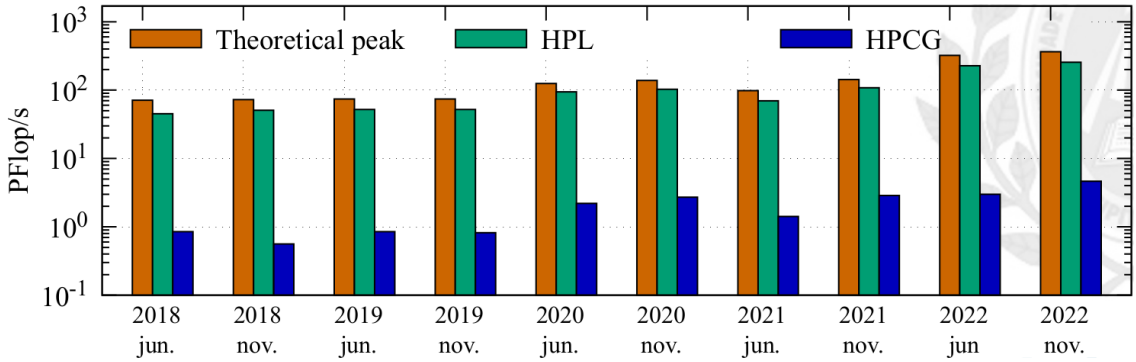

Figura 2.1. Comparação do desempenho médio dos 10 principais supercomputadores da lista Top500 (2018-2022) em termos de pico teórico, HPL e HPGC. O desempenho teórico de pico é consistentemente maior que os resultados medidos pelos benchmarks HPL e HPGC

2.1.3. Desafios na Escalabilidade de Sistemas Modernos

A escalabilidade paralela tem se tornado uma preocupação crítica devido à crescente demanda por desempenho. Com a onipresença de arquiteturas *multicore*, a eficiência no uso de múltiplos núcleos tornou-se determinante para lidar com volumes de dados massivos. Em áreas como inteligência artificial, *big data* e simulações científicas, a capacidade de escalar recursos afeta diretamente a produtividade e o custo-benefício. No entanto, observa-se ainda uma escassez de ferramentas dedicadas especificamente à análise e otimização da escalabilidade, o que eleva a complexidade desse processo.

As limitações para atingir essa escalabilidade ideal originam-se, principalmente, da comunicação, sincronização e desbalanceamento de carga. A comunicação excessiva entre núcleos gera um *overhead* significativo, degradando a eficiência global. Simultaneamente, a necessidade de sincronização frequente obriga *threads* a entrarem em períodos de inatividade (espera), subutilizando o hardware disponível.

Além disso, mesmo quando comunicação e sincronização são minimizadas, o desbalanceamento de carga pode tornar-se o principal responsável pela queda no *speedup*. Esse fenômeno ocorre quando diferentes unidades de processamento recebem quantidades desiguais de trabalho. O resultado é que algumas unidades de processamento finalizam suas tarefas prematuramente e permanecem ociosos aguardando os demais, reduzindo o *speedup* geral. Identificar e mitigar esse ciclo de ineficiências, através de técnicas como escalonamento de trabalho e ocultação de latência, é o desafio central na otimização de aplicações paralelas.

2.1.4. 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.

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:

$$T_{\text{paralelo}} = 0,9 \times \frac{T_{\text{sequencial}}}{P} + 0,1 \times T_{\text{sequencial}} = \frac{18}{P} + 2, \tag{3}$$

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

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}} = \frac{20}{\frac{18}{P} + 2}. \tag{4}$$

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

2.1.5. 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 (escalabilidade forte). Para abordar essa limitação, John Gustafson propôs, em 1988, uma alternativa chamada **Lei de Gustafson**, que considera o aumento do tamanho do problema conforme mais processadores são adicionados.

A Lei de Gustafson parte da premissa de que o tempo de execução se mantém constante enquanto aumentamos o tamanho do problema proporcionalmente ao número de processadores P. Normalizando o tempo de execução paralelo para 1, o trabalho total realizado seria a soma da parte serial s e da parte paralela, que agora é multiplicada por P (já que há mais processadores trabalhando no mesmo tempo). Assim, o *speedup* escalado (S_{scaled}) é calculado como:

$$S_{scaled} = s + P \times (1 - s). \tag{5}$$

Rearranjando os termos algebricamente, chegamos à forma mais comum da equação apresentada por Gustafson [3]:

$$S_{scaled} = P - (P - 1) \times s, \tag{6}$$

onde P é o número de processadores (frequentemente denotado como N na literatura original de Gustafson) e s é a fração sequencial do programa.

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

$$S = 100 - (100 - 1) \times 0,1 = 100 - 99 \times 0,1 = 100 - 9,9 = 90,1. \tag{7}$$

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

2.1.6. Tipos de Escalabilidade

Em 2.1.2 fomos apresentados rapidamente ao conceito de escalabilidade. Definindo de maneira mais precisa: 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.

Para ilustrar matematicamente a escalabilidade, considere um modelo hipotético simplificado onde o tempo de execução é linearmente proporcional ao tamanho do problema n (ou seja, $T_{\text{sequencial}} = n$) e o tempo paralelo possui um custo fixo de sobrecarga igual a 1 unidade de tempo (ou seja, $T_{\text{paralelo}} = \frac{n}{P} + 1$).

Neste cenário hipotético, a eficiência E seria descrita por:

$$E = \frac{S}{P} = \frac{T_{\text{sequencial}}}{P \times T_{\text{paralelo}}} = \frac{n}{P(\frac{n}{P}+1)} = \frac{n}{n+P}. \tag{8}$$

Essa equação demonstra que, para manter E constante à medida que P aumenta, n também precisa aumentar. 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. O número de processadores se torna kP e o tamanho do problema se torna xn. A equação que devemos resolver para x é:

$$E = \frac{n}{n+P} = \frac{xn}{xn+kP}. \tag{9}$$

Quando $x = k$, a eficiência E permanece inalterada. Podemos simplificar a equação para:

$$\frac{kn}{kn+kP} = \frac{kn}{k(n+P)} = \frac{n}{n+P}. \tag{10}$$

Ou seja, se aumentarmos o tamanho do problema n na mesma proporção que aumentarmos o número de *threads* P, a eficiência se mantém constante (escalabilidade fraca).

2.1.7. Análise de Eficiência e Escalabilidade Paralela

Um programa é dito **fortemente escalável** quando consegue manter a eficiência fixa mesmo sem aumentar o tamanho do problema. Isso significa que, ao adicionar mais processadores, o programa mantém uma alta eficiência sem precisar aumentar proporcionalmente a carga de trabalho.

Por outro lado, um programa é considerado **fracamente escalável** quando consegue manter a eficiência fixa à medida que o tamanho do problema aumenta na mesma proporção que o número de processadores. Isso quer dizer que, embora mais recursos

sejam adicionados, o problema precisa crescer proporcionalmente para que a eficiência não decresça.

2.1.8. Avaliando a Escalabilidade com Tabelas

Uma forma de avaliar a escalabilidade é utilizar tabelas que correlacionam o número de núcleos com o tamanho do problema. Considere agora uma aplicação real cuja complexidade de tempo sequencial seja quadrática ($T_{\text{sequencial}} = n^2$) e a execução paralela sofra com uma sobrecarga logarítmica de comunicação ($T_{\text{paralelo}} = \frac{n^2}{P} + \log_2 P$).

Os tempos de execução para diversas configurações são apresentados na Tabela 2.1.

Núcleos (P)	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
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 (em segundos) variando núcleos (P) e tamanho (n).

A equação que representa o tempo de execução sequencial é $T_{\text{sequencial}} = n^2$, onde n é o tamanho do problema. Já o tempo de execução paralelo é $T_{\text{paralelo}} = \frac{n^2}{P} + \log_2 P$, 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* para cada combinação. O *speedup* é definido como:

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}}. \tag{11}$$

Segue a tabela que representa os valores para *speedup*

Após o cálculo do *speedup*, calculamos a eficiência E:

$$E = \frac{S}{P}. \tag{12}$$

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

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.

Núcleos (P)	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
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.

Núcleos (P)	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
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.

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.

À medida que o detalhamento nas análises cresce, maiores ficam as tabelas. Vale ressaltar que estas tabelas não são geradas automaticamente por ferramentas de perfilamento tradicionais; essas ferramentas geralmente fornecem apenas os dados para uma única célula da tabela por vez. 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 ocul-

Núcleos (P)	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$
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.

tar 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 tendências. 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 desenvolvido para avaliar tendências de escalabilidade em programas paralelos. O pacote simplifica a execução, medição e comparação de múltiplos cenários de teste, permitindo a análise de ambientes com diferentes quantidades de processadores e cargas de trabalho.

A ferramenta fornece elementos visuais que auxiliam o usuário a compreender o comportamento do programa e a reconhecer gargalos que exigem otimização. O PaScal Suite é composto por dois módulos principais que serão detalhados a seguir: o *PaScal Analyzer* (Seção 2.2.1), responsável pela coleta de dados, e o *PaScal Viewer*, responsável pela visualização gráfica das métricas.

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. 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 **no** tempo de execução.

No *Analyzer*, a instrumentação permite que os desenvolvedores observem como diferentes partes da aplicação utilizam os recursos. A instrumentação automática permite a execução sem modificar o código, enquanto a manual envolve a adição explícita de chamadas para marcar regiões de interesse.

O Código 2.1 ilustra a instrumentação manual com o *Analyzer*. Para isso, é necessário incluir a biblioteca `pascalops.h` no cabeçalho do código e delimitar a zona perfilada com `pascal_start(id)` e `pascal_stop(id)`.

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

Código 2.1. Exemplo de uso do PaScal Analyzer com OpenMP

A biblioteca `pascalops` fica visível para o *GCC* após a instalação do *Analyzer*. Após delimitar as zonas, compila-se com o argumento `-lmpascalops`:

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

Para realizar a instrumentação automática, executa-se o *pascalanalyzer* com a opção `-t aut`:

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

No exemplo acima, utilizamos `-t` para o tipo de instrumentação, `-c` para a quantidade de núcleos e `-i` para fornecer **as entradas** que serão usadas no programa. O

argumento `-o` especifica o arquivo de saída; caso omitido, o *Analyzer* salvará a saída com um nome padronizado composto pelo **nome do binário seguido de data e hora da execução** (ex: `binario_2025-10-20_14-30.json`).

2.2.2. PaScal Viewer

O *Parallel Scalability Viewer*[5] é uma aplicação web para visualizar métricas de desempenho. Ele aceita como entrada o arquivo JSON do *Analyzer*, fornecendo diagramas de cores similares a mapas de calor. O Viewer encontra-se disponível no domínio: `https://pascalsuite.imd.ufrn.br/`.

A Figura 2.2 apresenta a interface do *Viewer*. Nos diagramas apresentados, o eixo vertical (Y) representa o número de núcleos utilizados, enquanto o eixo horizontal (X) representa os tamanhos do problema (ou entradas).

O primeiro diagrama (superior esquerdo) mede a eficiência global ($E = S/P$). O segundo (superior direito) avalia a escalabilidade global. Vale notar que, diferentemente da eficiência que possui fórmula fechada clássica, a métrica de escalabilidade no PaScal é derivada comparando a manutenção da eficiência da configuração atual em relação à configuração base. Cores próximas ao azul indicam alta escalabilidade (eficiência mantida), enquanto tons de marrom apontam degradação.

Os diagramas inferiores medem a escalabilidade forte (esquerda, onde o tamanho do problema é fixo e núcleos variam) e a escalabilidade fraca (direita, onde ambos variam proporcionalmente).

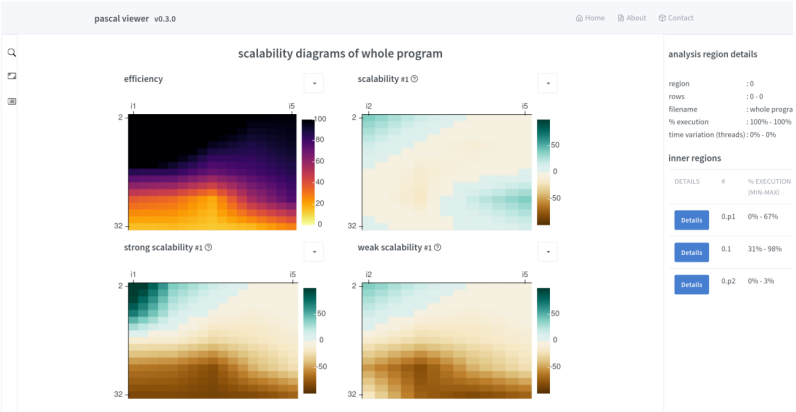

Figura 2.2. Captura de tela da interface da ferramenta web PaScal Viewer. O eixo Y indica contagem de núcleos e o eixo X as configurações de entrada.

2.3. Estudo de Caso e Aplicabilidade

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

Nosso objeto de estudo é o método de *Red-Black Gauss-Seidel*. O Código 2.2 apresenta a versão paralela utilizando *OpenMP*.

```
1 #include <omp.h>
```

```cpp
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <iostream>

#define MAX_ITER 1000

void initialize(double **A, int n) {
  // inicialize a matriz
}

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;
}
```

```
49
50 void solve_parallel(double **A, int n) {
51   int iters;
52   double diff = 0;
53   for (iters = 1; iters < MAX_ITER; ++iters) {
54     diff = matrix_calculation(A, n - 1);
55   }
56 }
57
58 int main(int argc, char *argv[]) {
59   int i;
60   double **A;
61   int n;
62
63   n = std::stoi(argv[1]);
64   A = new double *[n + 2];
65   for (i = 0; i < n + 2; i++) {
66     A[i] = new double[n + 2];
67   }
68
69   initialize(A, n);
70
71   solve_parallel(A, n - 1);
72 }
```

Código 2.2. Implementação do método Red-Black Gauss-Seidel

Configuramos um ambiente variando núcleos entre 1, 2, 4 e 8. Primeiro compilamos o código (note que para a instrumentação automática, não é estritamente necessário vincular a biblioteca `pascalops`, mas o faremos por consistência):

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

Executamos o *Analyzer*:

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

Definimos tamanhos de problema (1001 a 2501) crescendo de forma não linear (aproximadamente quadrática) para tentar acompanhar o aumento da capacidade de processamento dos núcleos (escalabilidade fraca). O argumento `-r` define 5 repetições para cada teste.

A Figura 2.3 apresenta o resultado no *Viewer* para o **primeiro conjunto de testes**. O diagrama de eficiência global demonstra queda à medida que o problema cresce. O gráfico de escalabilidade global revela baixa escalabilidade em todos os pontos.

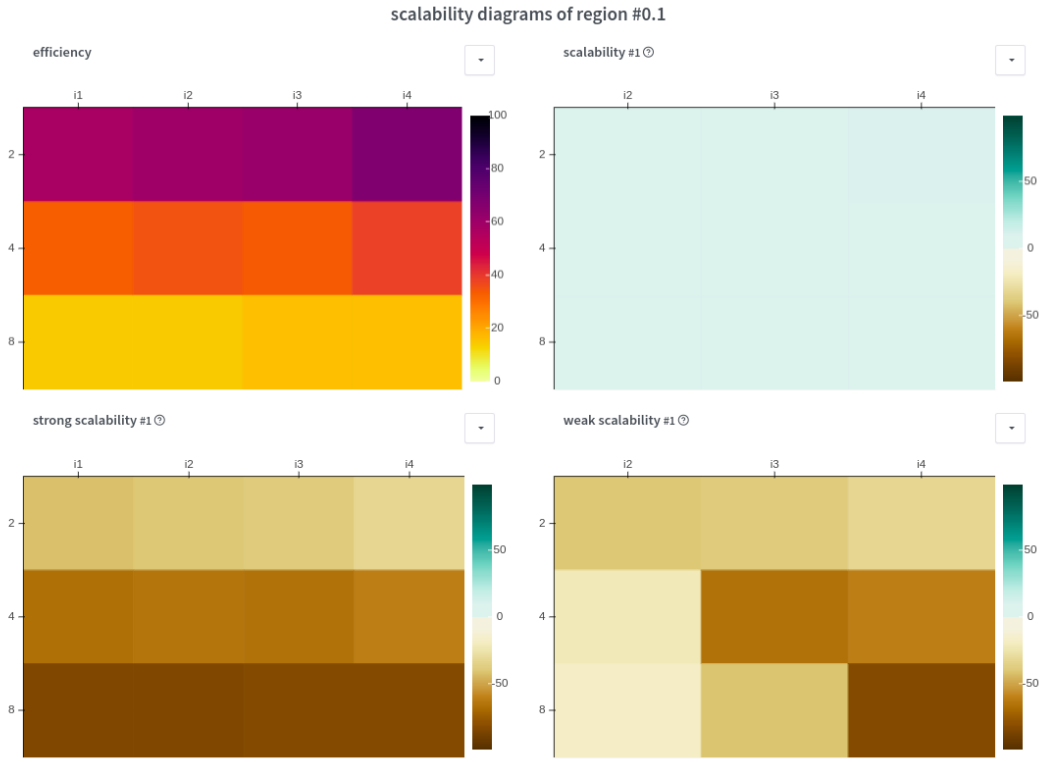

Figura 2.3. Diagramas do PaScal Viewer. A quantidade de núcleos varia em 1, 2, 4 e 8 (eixo Y) e o tamanho do problema varia nas configurações i1 (1001), i2 (1501), i3 (2001) e i4 (2501) no eixo X.

Para identificar gargalos, utilizamos a instrumentação manual. Incluímos `pascalops.h` e delimitamos as regiões. A Região 1 engloba a execução paralela total; a Região 2, o primeiro laço; e a Região 3, o segundo laço. O Código 2.3 ilustra essa instrumentação.

Recompilamos com `-lmpascalops` e executamos com `-t man`:

```
1    g++ -Wall -o RB -fopenmp RB.cpp -lmpascalops
2    pascalanalyzer ./RB -c 8,4,2,1 \
3                   -i 1001,1501,2001,2501 \
4                   -r 5 -t man -o RB.json
```

No *Viewer*, acessamos as sub-regiões (Figura 2.4). A interface indica que a Sub-Região 3 representa até 76% do tempo total.

A Figura 2.5 exibe a Sub-Região 2. Conclui-se que esse trecho é mais eficiente que a execução completa **(veja a Figura 2.3)**.

Já a Figura 2.6 apresenta a Sub-Região 3. Esse trecho apresenta baixa eficiência global, indicando ser o gargalo.

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 tre-

cho 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 um conjunto de testes com o *Analyzer* agora para identificar os possíveis ganhos de nossa solução.

A Figura 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 Figura 2.8 mostra os gráficos de desempenho referentes à Sub-Região 3 após a identificação de um possível gargalo e a implementação de nossa solução. É possível observar que houve um ganho de 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 capítulo, discutimos a relação e as distinções entre desempenho e escalabilidade no contexto da computação paralela, apresentando o PaScal Suite como ferramenta de apoio. Através de um estudo de caso, identificamos e mitigamos um gargalo de desempenho, ilustrando como a visualização de dados pode guiar a otimização de código paralelo.

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;
}
```

Código 2.3. Instrumentação manual da região paralela 1 de nosso objeto de estudo.

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

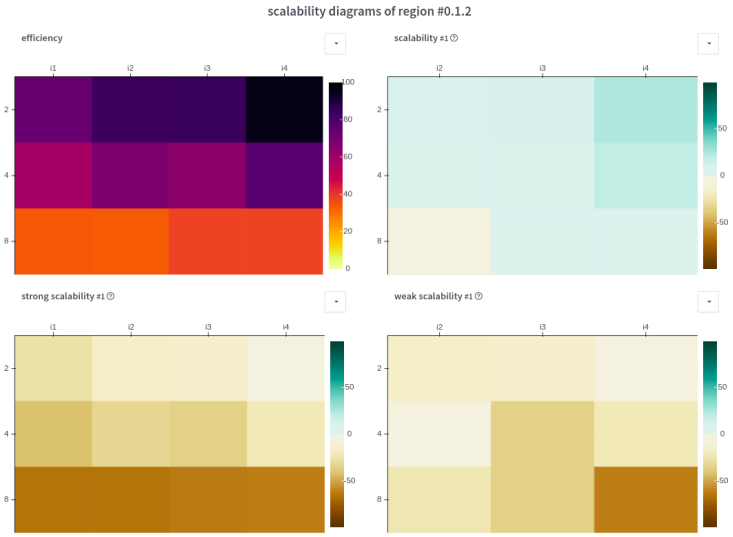

Figura 2.5. Diagramas de eficiência e escalabilidade da Sub-Região 2.

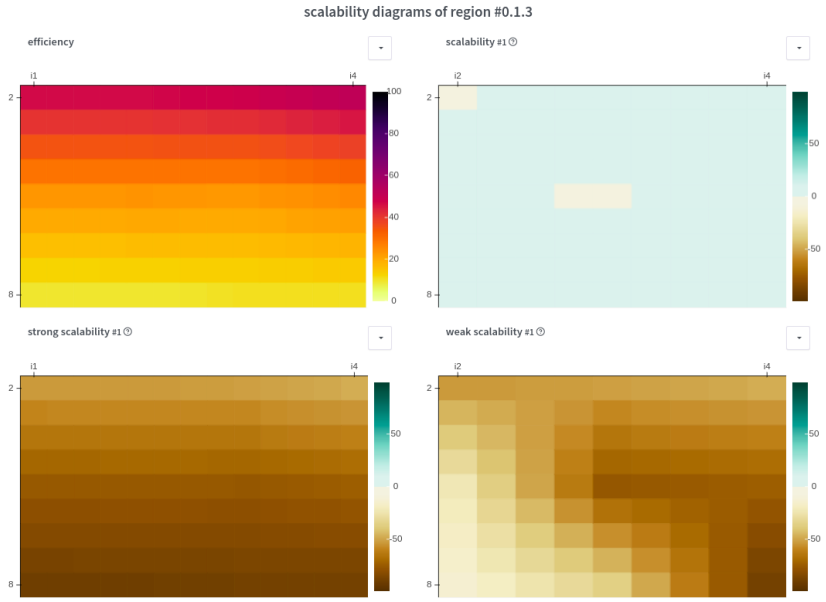

Figura 2.6. Diagramas da Sub-Região 3 no segundo conjunto de testes.

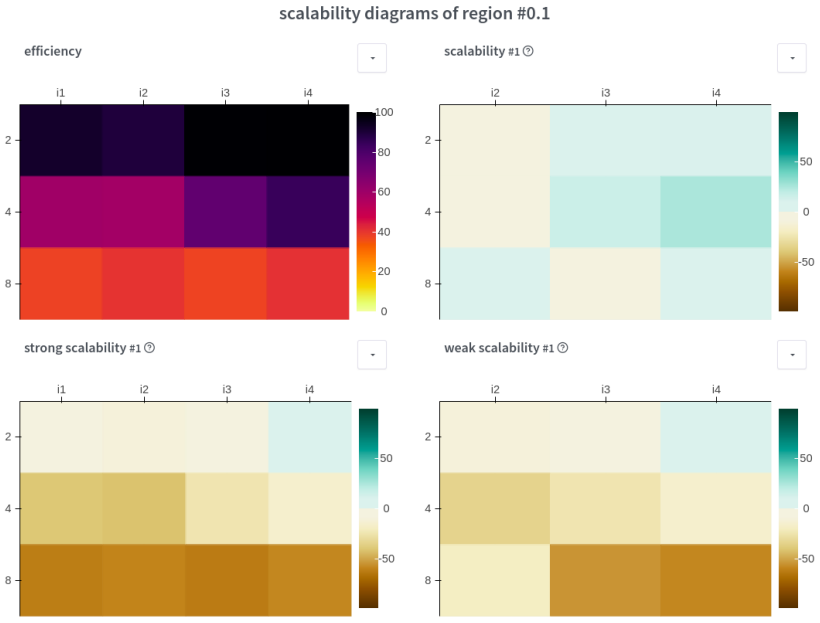

Figura 2.7. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Região 1 após nossa solução. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X

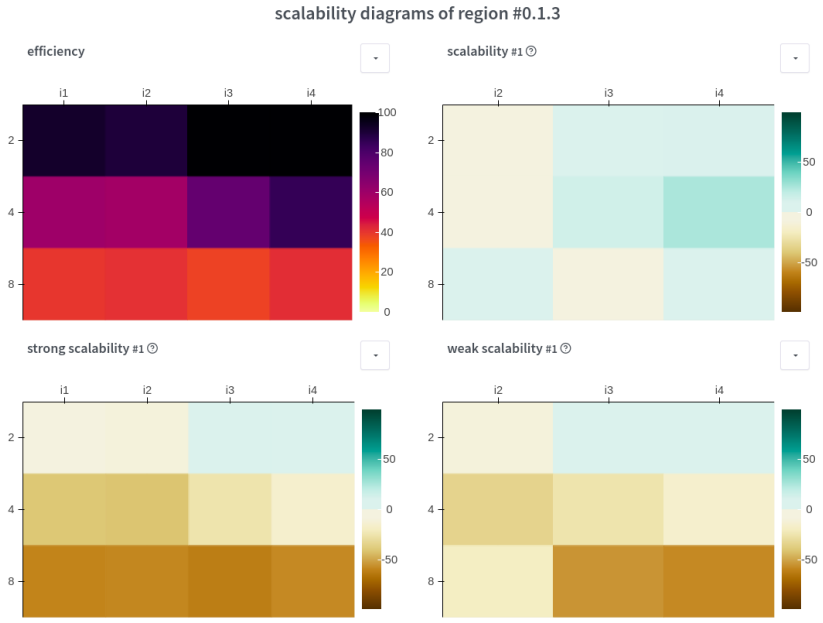

Figura 2.8. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Sub-Região 3 após nossa solução. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X