SSC0143 - Programação Concorrente
Professor: Paulo Sergio Lopes de Souza

Universidade de São Paulo

Design de Algoritmos Paralelos
-
Escalonamento de Gauss-Jordan

Matheus Gomes da Silva Horta	8532321
Hugo Moraes Dzin	8532186
Bruno Henrique Rasteiro	9292910
Luiz Eduardo Dorici	4165850

Instituto de Ciências Matemáticas e Computação - ICMC/USP

São Carlos

Setembro/2017

O Problema

O método de Gauss-Jordan é um algoritmo de escalonamento que consiste em aplicar operações elementares à matriz aumentada de um sistema de equações lineares até que ela esteja na forma escalonada reduzida.

As operações elementares são:

1. Multiplicar uma linha por uma constante
2. Permutar duas linhas
3. *Li = Li + c * Lj*, onde *Li, Lj* são linhas e *c* é uma constante numérica

Ilustramos o algoritmo no seguinte pseudocódigo:

```
Dado uma matriz aumentada M com n linhas e n+1 colunas

j = 0 (usado para pular colunas nulas)
Para cada linha i = 1 .. n:
      (1) Pivoteamento
      Enquanto M(i, i+j) (o pivô atual) == 0:
            Encontrar uma linha k tal que M(k, i+j) != 0
            Se houver k:
                  Permutar M(i, ) e M(k, )
            Senão:
                  Incrementar j para ignorar a coluna atual.

      (2) Dividir a linha atual pelo pivô
      Para cada coluna c = i+j+1 ... n+1:
            M(i,c) <- M(i,c) / M(i, i+j)

      (3) Zerar o resto da coluna do pivô (eliminação)
      Para cada linha k = 1 ... n, k != i:
            Para cada coluna c = i+j+1 ... n+1:
                  M(k,c) <-  M(k,c) - M(k, i+j) * M(i,c)
Solução <- coluna M( ,n+1)
```

Fontes de Paralelismo

Após análise do algoritmo, foi constatado que as iterações do loop não podem ser paralelizadas de maneira prática, e que cada iteração é composta três grandes partes interdependentes:

- Encontrar o próximo pivô (pivoteamento);
- Dividir a linha atual pelo pivô (normalização);
- Zerar a coluna do pivô (eliminação);

Cada uma dessas computações deve ser feita em ordem para garantir a corretude do algoritmo. Cada uma dessas partes é paralelizável internamente, com isso, as tarefas serão divididas com foco em explorar o paralelismo interno desses módulos. A representação gráfica dos componentes pode ser encontrada na figura 1.

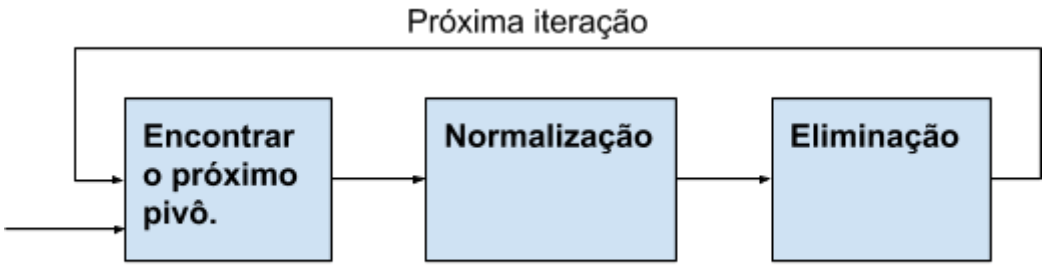


Fig 1 - Blocos dependentes do algoritmo

Particionamento

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 2 & 3 & 7 & 0 \\ 1 & 3 & -2 & 17 \end{array}\right]$$

Fig 2: No particionamento proposto, as tarefas operam sobre elementos individuais

Inicialmente utilizamos o particionamento funcional, a fim de dividir as diferentes etapas do algoritmo. Ou seja, cada um dos 3 conjuntos descritos anteriormente têm seus próprios tipos de tarefas. Em seguida, fizemos uma partição de domínio dentro de cada um desses conjuntos, para aumentar a granularidade do nosso particionamento. Sendo assim, cada tarefa realiza uma computação bem pequena, em cima de apenas uma posição da matriz abaixo está uma lista com as tarefas propostas:

1. Pivoteamento
    a. Verificar se um dado elemento é não-nulo
    b. Dado dois elementos, decidir qual é melhor candidato para pivô **(pode não ocorrer)**
    c. Decidir quais linhas serão permutadas (redução) **(pode não ocorrer)**
    d. Permutar um par de elementos
    e. Pular coluna (no pseudocódigo é o incremento de j)
2. Normalização
    a. Dividir todos os elementos da linha do pivô pelo valor do mesmo
3. Eliminação
    a. A subtração com produto dentro do loop (3) do pseudocódigo

A cada iteração do algoritmo, uma coluna da matriz não precisa mais ser processada, sendo assim, cada iteração possui seu próprio conjunto de tarefas totalizando *(n - i) * (n + 1)* tarefas por iteração.

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & 1 & 5 & -6 \\ 0 & 2 & -3 & 14 \end{array}\right] \quad \left[\begin{array}{ccc|c} 1 & 0 & -4 & 9 \\ 0 & 1 & 5 & -6 \\ 0 & 0 & -13 & 26 \end{array}\right] \quad \left[\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & -2 \end{array}\right]$$

Fig 3: Representação gráfica a execução do algoritmo

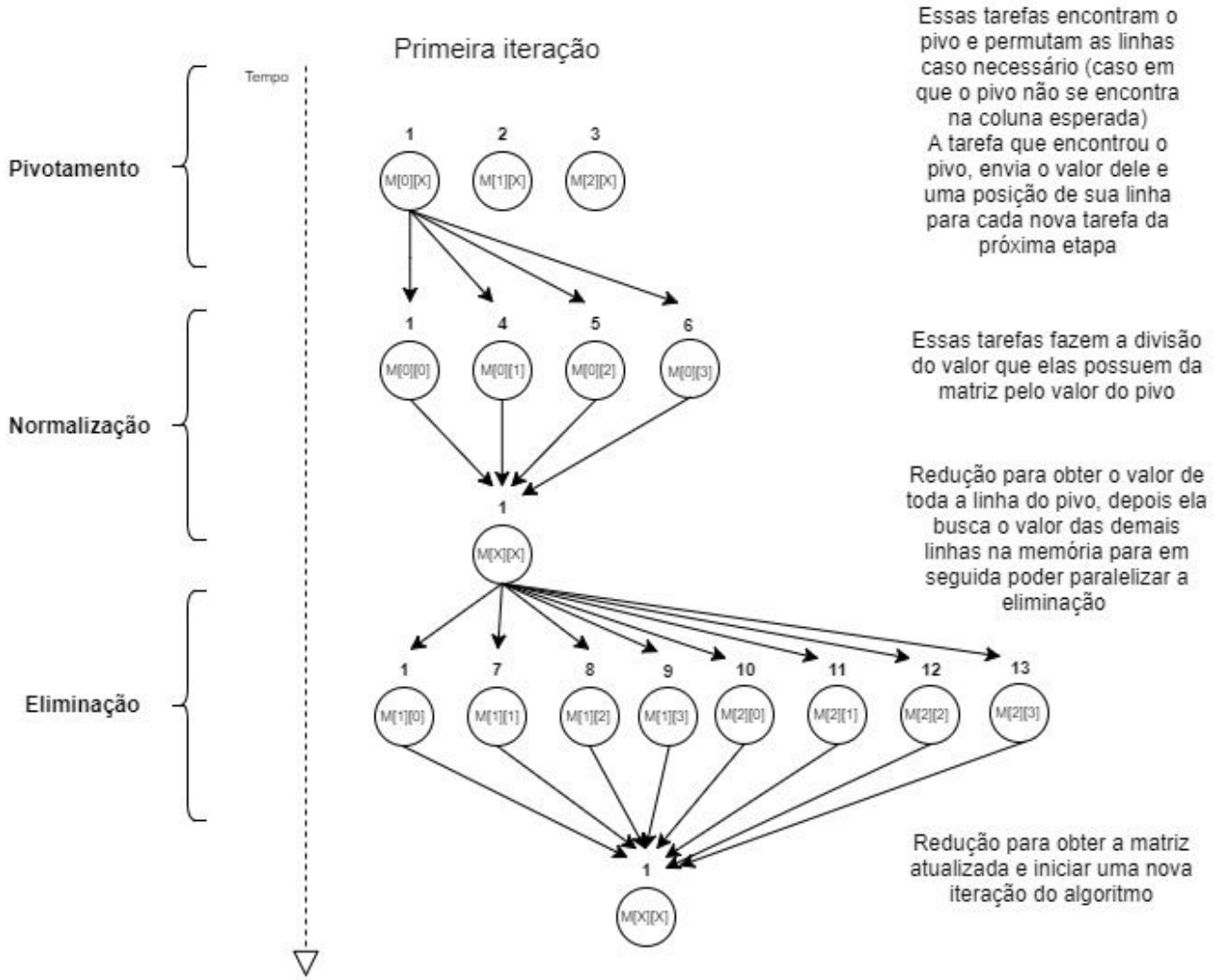
Cada tarefa realizará uma operação nos valores que receber, seja uma de eliminação ou normalização, com isso, pode-se

Grafo de Dependência

Iremos usar uma matriz de 3 linha e 4 colunas como exemplo ilustrativo. Os elementos serão nomeados da seguinte maneira:

$$\left[\begin{array}{ccc|c} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{array}\right]$$

Fig 4: Nome de cada elemento



Como mostrado no grafo de dependência, o máximo **grau de paralelismo** em um dado instante de tempo considerando todas as comunicações envolvidas, ocorre no momento da eliminação de uma coluna, onde todos os elementos da linha da matriz a partir do pivô estão executando suas respectivas operações de eliminação.

Comunicação

A comunicação ocorre de forma distinta em cada etapa do algoritmo. Nas figuras desta seção, os números nas tarefas representam a iteração e as setas a interação de cada uma em cada etapa do algoritmo.

1. Pivoteamento

Este é o passo mais complexo em termos de comunicação. Inicialmente, é necessário que a coluna do pivô faça uma redução para decidir quais linhas serão permutadas.

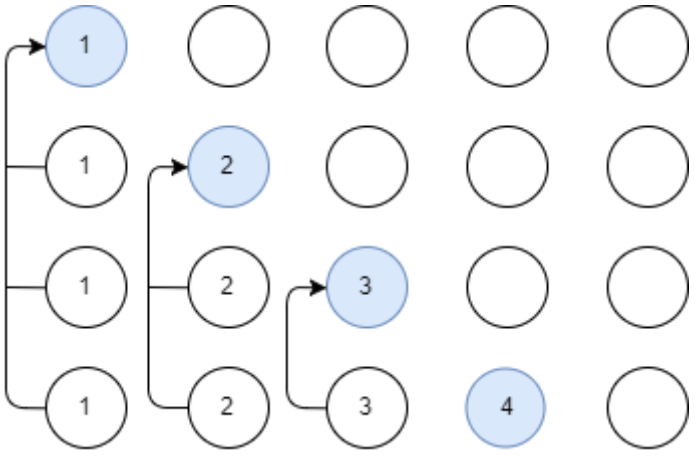


Fig 5: Reduções necessárias no pivoteamento.
Números indicam a iteração em que a comunicação ocorre.

Em seguida, a tarefa do pivô (nós em azul, na figura acima) decide como o pivoteamento irá proceder e precisa

2. Normalização

Cada iteração do algoritmo normaliza uma linha da matriz. Para isso, o pivô precisa enviar o seu valor para todos os elementos da mesma linha à sua esquerda, por meio de um broadcast. Os elementos à direita podem ser ignorados, pois sempre se tornam zero.

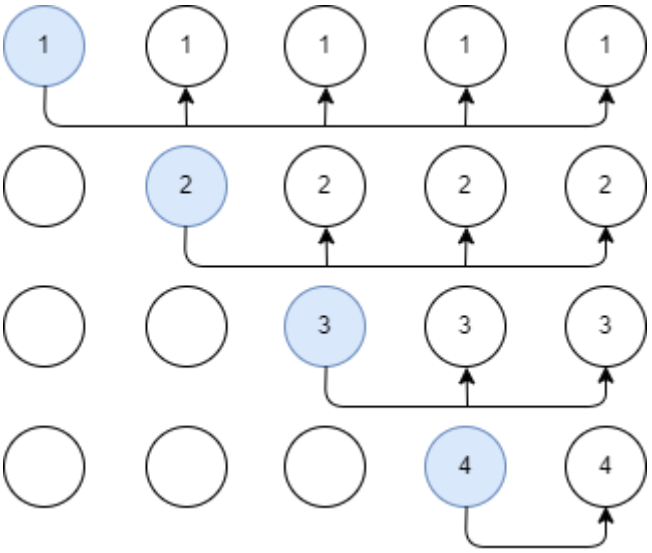


Fig 6: Comunicação necessária para normalizar, em um problema 4x5.
Números indicam em que iteração cada comunicação ocorre.

Como podemos ver na imagem acima, o broadcast mais demorado ocorre na 1a iteração, onde temos n remetentes. Devido à natureza 1-para-todos, podemos utilizar o seguinte esquema de broadcast:

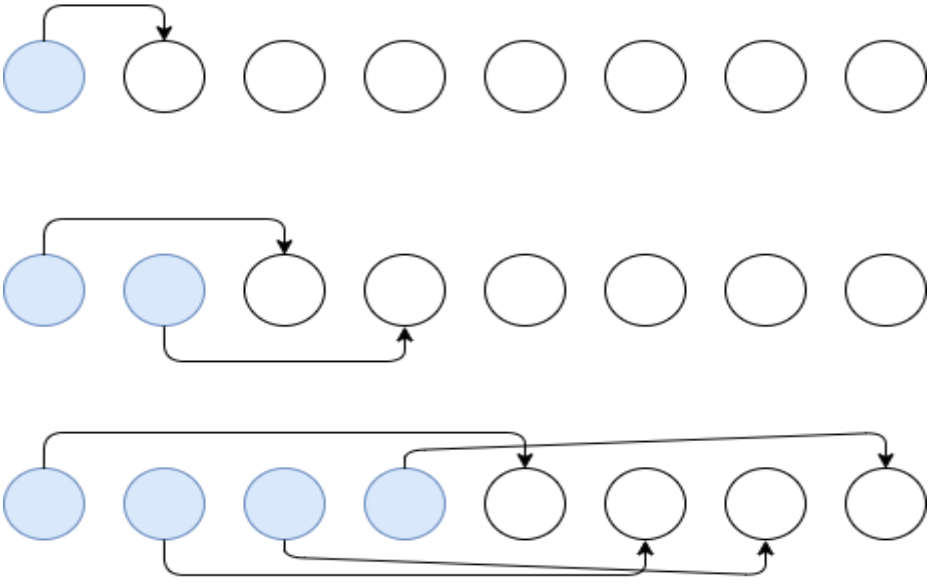
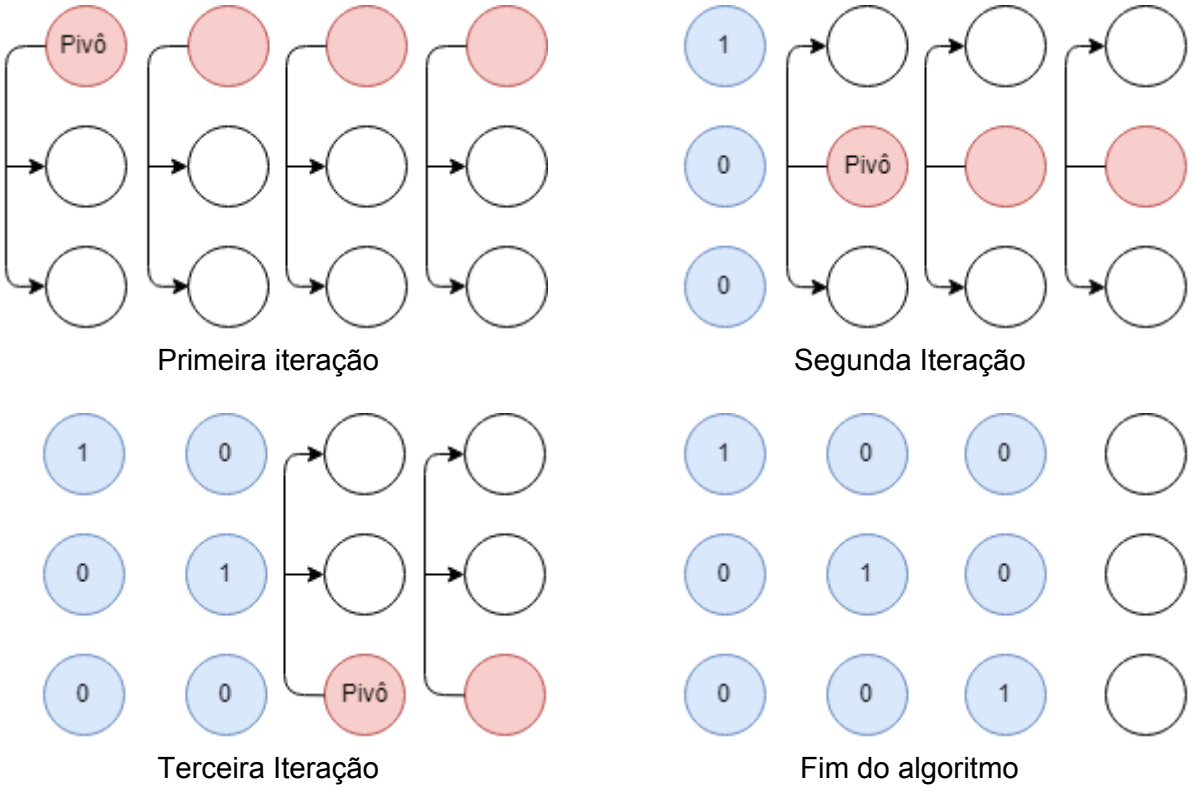


Fig 7: Broadcast 1-para-todos em 3 etapas com 8 nós.

Dessa maneira, em uma iteração a comunicação pode ser realizada com O(log(n)) etapas. Para a matriz completa teremos $O(\sum_{k=1}^{n} log(k))$ etapas no total, onde n é a ordem da matriz.

3. Eliminação



Assim como na seção anterior, a comunicação necessária reduz a cada iteração do algoritmo. A primeira iteração requer (n+1) broadcasts, todos de tamanho 1-para-n, já a última requer apenas 2 deles.

Aglomeração

Com o objetivo de reduzir a carga de comunicação nas etapas do algoritmo, foi decidido aglomerar as tarefas em colunas. onde cada aglomeração representa uma coluna da matriz. Desse modo, reduz-se a quantidade de comunicações de cada tarefa, uma vez que cada tarefa de cada coluna precisa se comunicar com todas as outras de sua coluna *(O(n-1))* e sua linha *(O(n))* em determinados intervalos de tempo. Com a aglomeração em colunas, a quantidade de comunicações se reduz de *n*(n-1)* no pior caso, para *n*. O broadcast pode ser feito da mesma forma como é feito na fig 7 da seção 2.

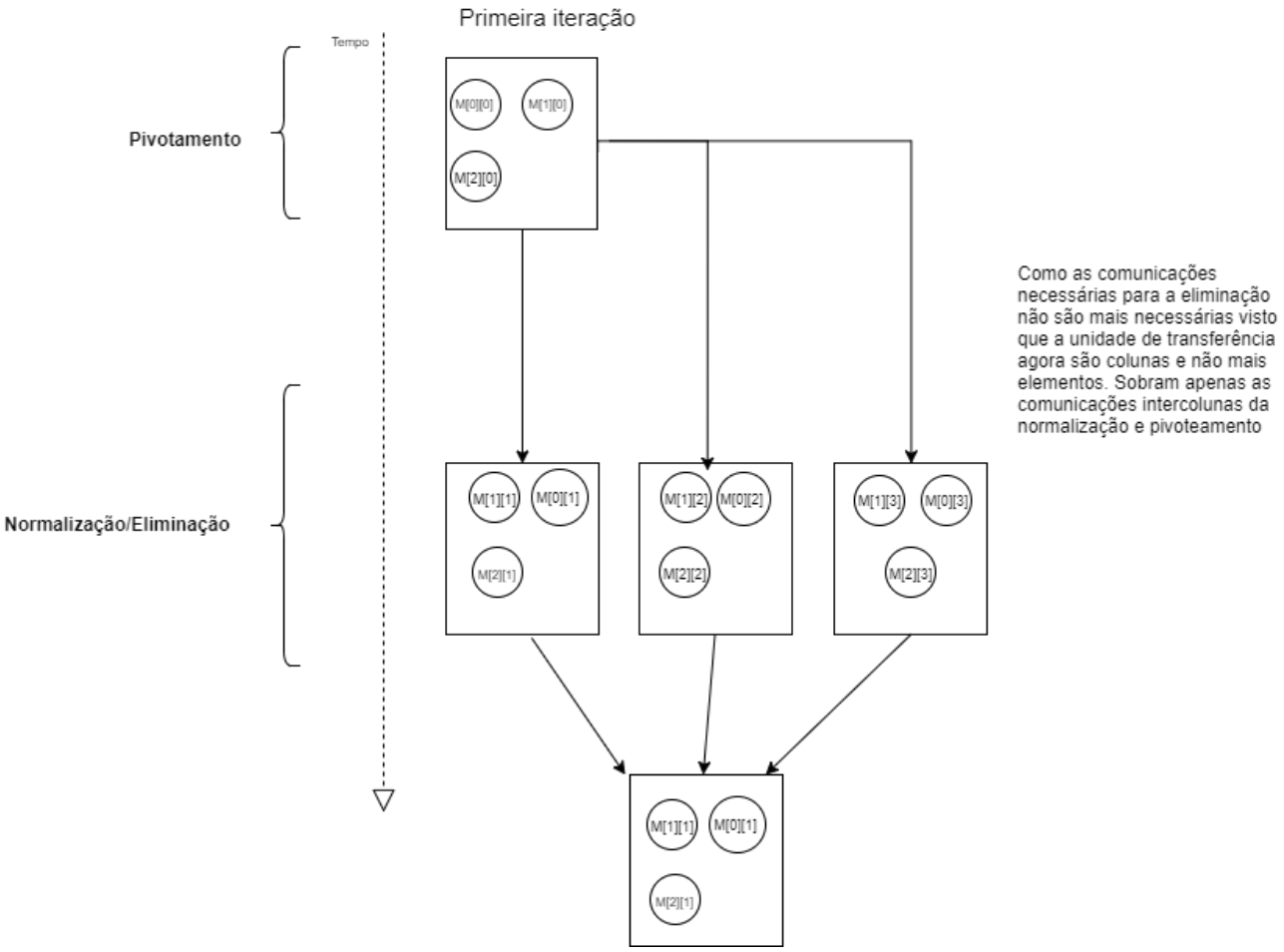


Fig 8: Grafo de dependências aglomerado

Mapeamento

A possibilidade de mapeamento mais intuitiva para esse modelo seria atribuir para cada processo uma tarefa e cada processo a um processador, dessa forma, cada processador ficaria responsável pela atualização de uma coluna na matriz. Entretanto, o custo de processamento para isso é baixo, embora a coluna cresce linearmente com o tamanho da matriz, as operações realizadas sobre os dados são básicas (soma, multiplicação etc.) e demoram somente o tempo de uma operação.

O maior problema desse modelo é a comunicação e o desbalanceamento de carga, durante as etapas de normalização e eliminação faz-se necessária a comunicação da tarefa que encontrou seu pivô com todas as que ainda não encontraram. Em relação ao desbalanceamento é notável que após a execução pela primeira vez das três primeiras etapas do algoritmo, ou seja, após a finalização da primeira tarefa, o processador destinado a executá-la ficará ocioso durante o resto do algoritmo.

Para exemplificar esse cenário vamos supor uma matriz M de dimensão N x N, para a normalização e eliminação do pior caso (a primeira coluna), o broadcast da coluna será para N-1 processadores, esse número irá decrementar de uma unidade a cada iteração do algoritmo, ou seja, sempre na realização de uma nova normalização, ficando assim com um número total de

mensagens enviadas igual a $\sum_{i=1}^{N}(N - i)$ para a execução do algoritmo. A figura X mostra como seria tal distribuição.

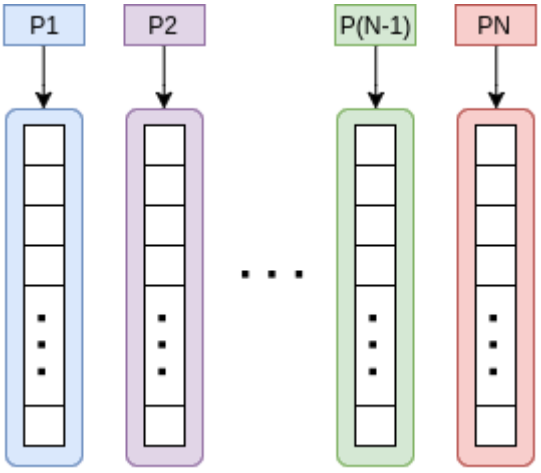


Fig 9 : Mapeamento trivial de processos em processadores

A abordagem que compensa essas falhas e que será utilizada para esse trabalho será a de atribuir mais de uma tarefa aos processadores, a distribuição deve ser feita escolhendo de forma que cada processador fique com colunas localizadas em extremos da matriz, por exemplo, suponha uma matriz 4 x 4 e um cluster com 2 processadores (A e B), o processo P1 é responsável pelas colunas 1 e 4, enquanto que o processo P2 pelas colunas 2 e 3, dessa forma os processadores A e B executam respectivamente os processos P1 e P2. Considerando que os processadores são multicore, as tarefas atribuídas aquele processo serão executadas paralelamente em threads nos seus respectivos processadores.

A fim de compararmos o ganho na comunicação, vamos pensar em uma matriz M de dimensão N x N, para a normalização e eliminação do pior caso o broadcast da coluna será para (N-2)/2 processadores, a cada iteração esse número será reduzido, o que pode ser expresso pela fórmula $\frac{1}{2}\sum_{i=0}^{N-1}(N-i-2)$. Se comparamos com a proposta de mapeamento anterior vimos que há uma diminuição de aproximadamente 50% do número de mensagens a serem enviadas, a figura X demonstra tal mapeamento.

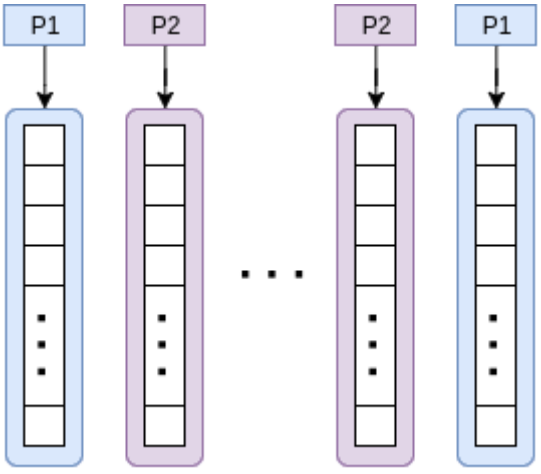


Figura 10 - Mapeamento de processos em processadores