Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

# Modelagem de Gauss-Jordan

**Disciplina:** SCC0143 - Programação Concorrente
**Professor:** Dr. Paulo Sérgio Lopes de Souza


| **Alunos grupo 15** | **Número USP** |
|---|---|
| Camila Stenico dos Santos | 8530952 |
| Felipe Kazuyoshi Takara | 8921026 |
| Henrique F. M. Freitas | 8937225 |
| Lucas de Almeida Carotta | 8598732 |

# Sumário

# 1 - Objetivo

## 1.1 - Problema proposto

Utilizando a abordagem PCAM proposta por Ian Foster, modelar uma solução paralela para o problema de escalonamento de matrizes, usando o método de Gauss-Jordan.

## 1.2 - Estrutura de solução sequencial

Para iniciar uma discussão, faz-se necessária certa avaliação do método de resolução sequencial.

De maneira típica, ao utilizar o método, temos três possíveis operações para substituir uma linha sem prejudicar o resultado do sistema linear:

- Troca de Linhas;
- Multiplicação por escalar;
- Combinação linear.

Na proposta de solução sequencial, um elemento é selecionado como pivô (se tal elemento é diferente de zero). Os demais elementos da coluna deste pivô serão decrementados, a fim de aproximar o valor à matriz escalonada. Devido ao caráter do algoritmo, apenas soluções possíveis e determinadas serão evidenciadas.

Apesar de uma ideia intuitiva, a abordagem pragmática acima se faz necessária, deixando o algoritmo com um custo de operação maior que o necessário - $O(n^3)$. Com a paralelização, podemos reduzir o tempo de execução sem prejudicar os resultados.

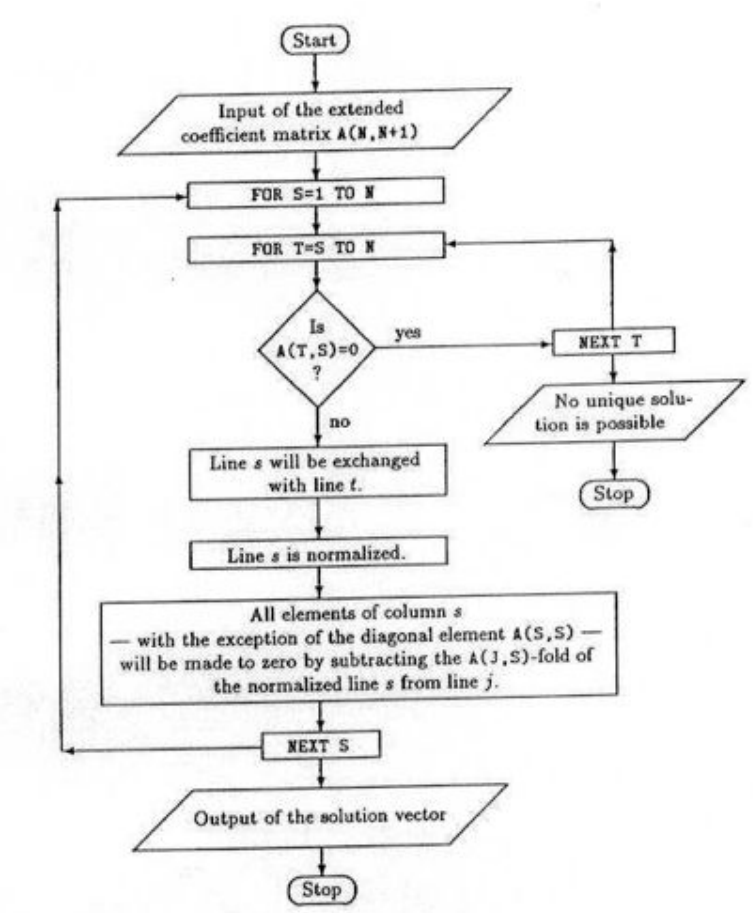


*Fig.1: Solução sequencial do escalonamento de matrizes usando o método Gauss-Jordan [1] [http://oregonstate.edu/instruct/ch590/lessons/lesson13.htm](http://oregonstate.edu/instruct/ch590/lessons/lesson13.htm)*

## 1.3 - Metodologia PCAM de Ian Foster

A metodologia PCAM de Ian Foster consiste dos seguintes passos:

1. Particionamento: dividir os problemas em tarefas menores e identificar as seções do programa
2. Comunicação: avaliação de como será a comunicação entre threads/processos.
3. Aglomeração: Combinar as tarefas que são dependentes entre si para que aumente a eficiência do código paralelo.
4. Mapeamento: atribuição das tarefas aglomeradas para os threads/processos.

# 2 - Abordagem

## 2.1 - Particionamento

No levantamento inicial das tarefas, podemos encarar tal algoritmo possuindo dois *overheads* bem claros:

- Percorrer a matriz a fim de trocar as linhas em cada elemento diferente de zero ($O(n^2)$, aproximadamente).
- A cada troca, efetuar a combinação linear em cada elemento pertencente a coluna do valor selecionado ($O(n^2)*O(n) = O(n^3)$). Para que ocorra o cálculo correto, é determinada uma constante *c* por iteração.

Como estamos trabalhando com uma matriz n x n, mais a coluna de resultado, podemos encarar a matriz como um conjunto de n+1 colunas e n linhas, onde haverão (n+1)*n elementos, onde em cada elemento será feito um conjunto de operações aritméticas por linha caminhada. Logo, a menor unidade de gauss-jordan é o elemento da matriz sendo alterado por linha, sendo assumido como objetivo paralelizar as mudanças por elemento.

Voltando a definição de Gauss Jordan, temos na linha *n,* com o pivô A[*n][n]*, todos os elementos serão modificados seguindo A[*n][j]* = A[*n][j]* / A[*n][n],* para que o pivô seja 1, considerando [*j]* o número da coluna. Para as linhas diferentes de *n*, temos que os elementos serão modificados seguindo A[*l][c]* = A[*n][c]* * -(A[*l][n]*) + A[*l][c]*, onde *l* é a linha com valor diferente de *n*, *c* é a coluna. Pensando em paralelizar todos as mudanças de elementos com todas as linhas sendo modificadas para 1 em pivô e zero em todos os elementos, é possível notar o problema de dependência presente.
Segue um exemplo mais ilustrativo da partição.

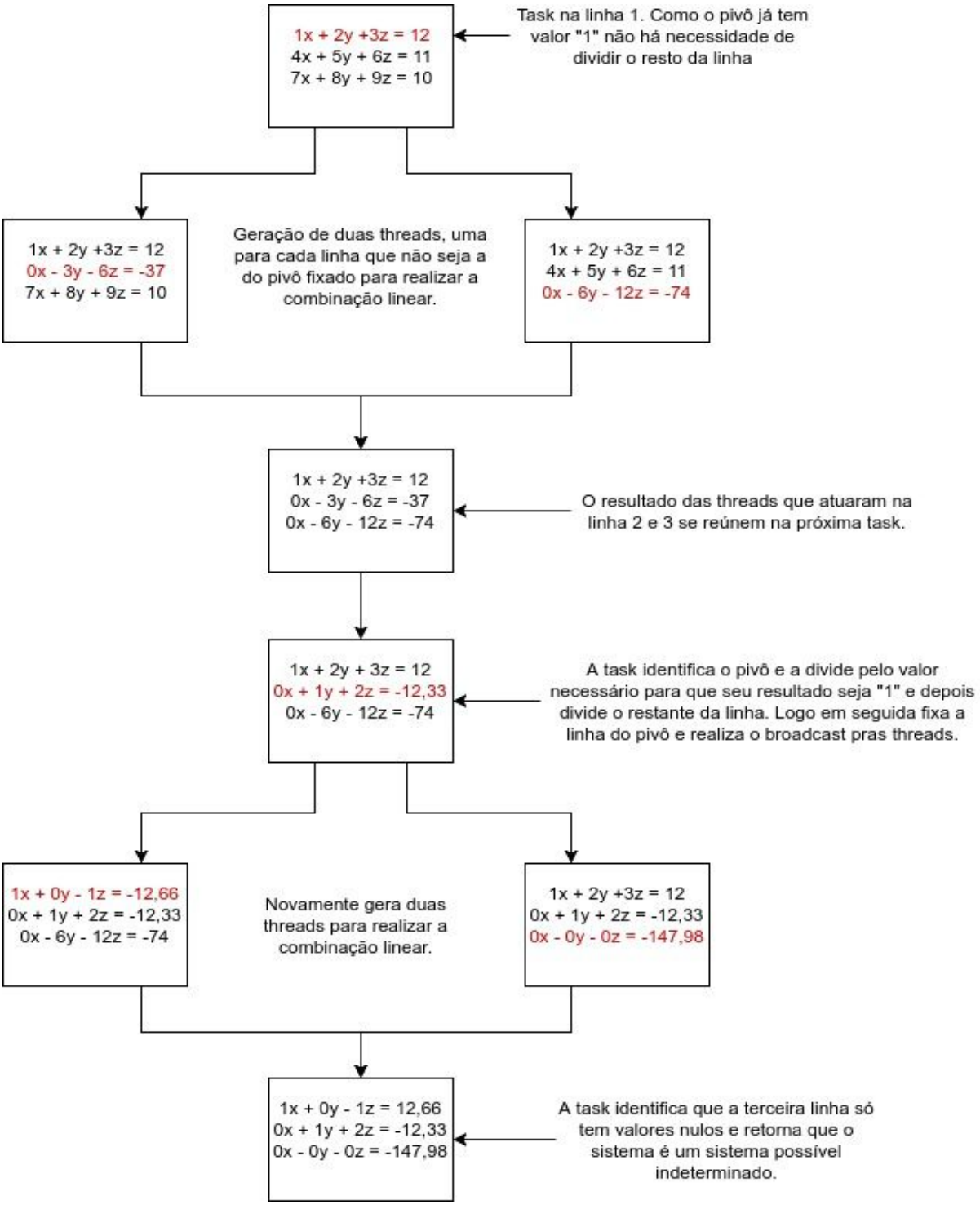


*Fig. 1: Exemplo de partição para uma matriz-exemplo.*
*Compilada pelos autores*

## 2.2 - Comunicação

O momento da troca de linhas é chave para a comunicação efetiva no algoritmo. Uma vez que a eliminação ocorre nas colunas e não há a modificação do elemento na posição da diagonal principal, as tarefas podem ser executadas de maneira concorrente sem perder consistência.

Cada tarefa de troca necessita, portanto, uma comunicação síncrona entre as tarefas que efetuam as operações nas colunas, enquanto as tarefas nas colunas recebem de maneira assíncrona a coluna e a constante sobre as quais devem operar. Abaixo, consideramos uma matriz de n=3 como exemplo de grafo de comunicação.

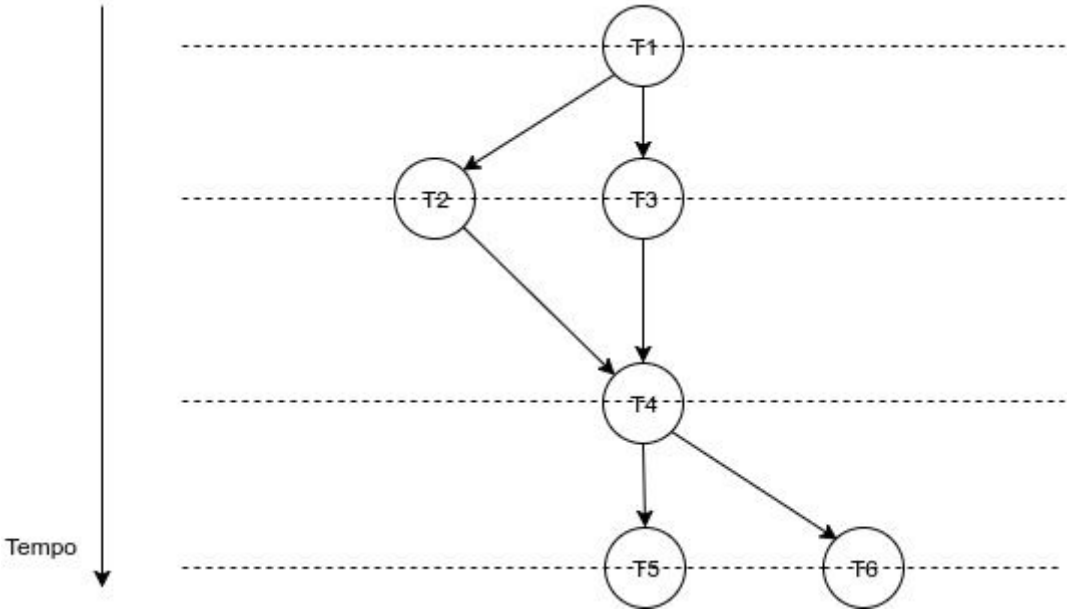


*Fig. 2: Simulação de dois passos do algoritmo numa matriz 3x3.*
*Compilada pelos autores.*

Note que em [2], as tarefas 1 e 4 estão atreladas a determinar o pivô. Para que 4 seja executada corretamente, as operações das tarefas 2 e 3 devem ser finalizadas. O número de tarefas geradas num caso genérico após determinar o pivô corresponde a *j-1*, onde *j* compreende o número máximo de linhas da matriz. Mais níveis podem ser introduzidos a fim de reduzir o overhead de comunicação - como [2] não contempla um caso grande o suficiente, tais níveis de comunicação foram omitidos. A proposta dos autores é de gerar tasks de acordo com a fórmula *ceilling(j/2)* para diminuir tal overhead.

## 2.3 - Aglomeração

Para elaborar uma aglomeração coerente, utilizaremos um exemplo maior, com n=5

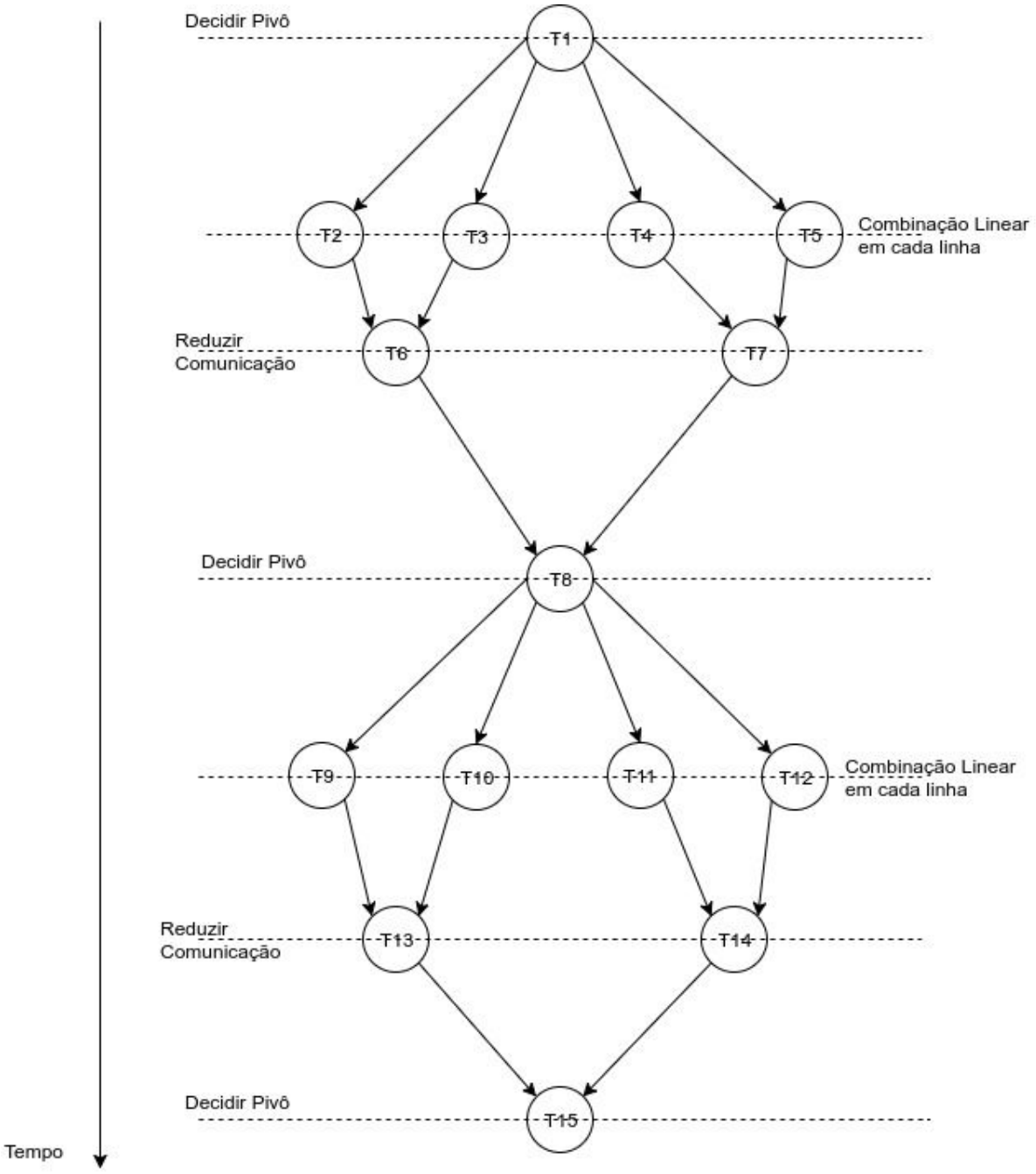


*Fig. 3: Simulação de dois passos do algoritmo sem aglomeração.*
*Note que há um passo para reduzir a sobrecarga de comunicação nos processos 8 e 15, de acordo com a fórmula proposta.*
*Compilada pelos autores.*

De fato, após um passo simples do algoritmo, temos o ciclo:

- Um pivô será definido;
- Todas as linhas sofrem combinação linear;
- Cada combinação será reunida até a última task.

Logo, podemos aglomerar um único processo para o controle dos pivôs, enquanto demais processos efetuam as combinações lineares e os passos intermediários para a consistência dos dados. Cada processo relacionado às combinações lineares será responsável por pelo menos duas atividades, tendo como possibilidade:

- Uma combinação linear e uma aglomeração;
- Duas combinações lineares.

Os processos que executam a segunda opção ainda farão duas comunicações, enviando seus resultados para processos pertencentes a primeira opção. No estágio de mapeamento, faz-se necessário um escalonamento destes processos, a fim de balancear a carga dos mesmos.

Abaixo, segue a aglomeração feita para o exemplo da figura 3.

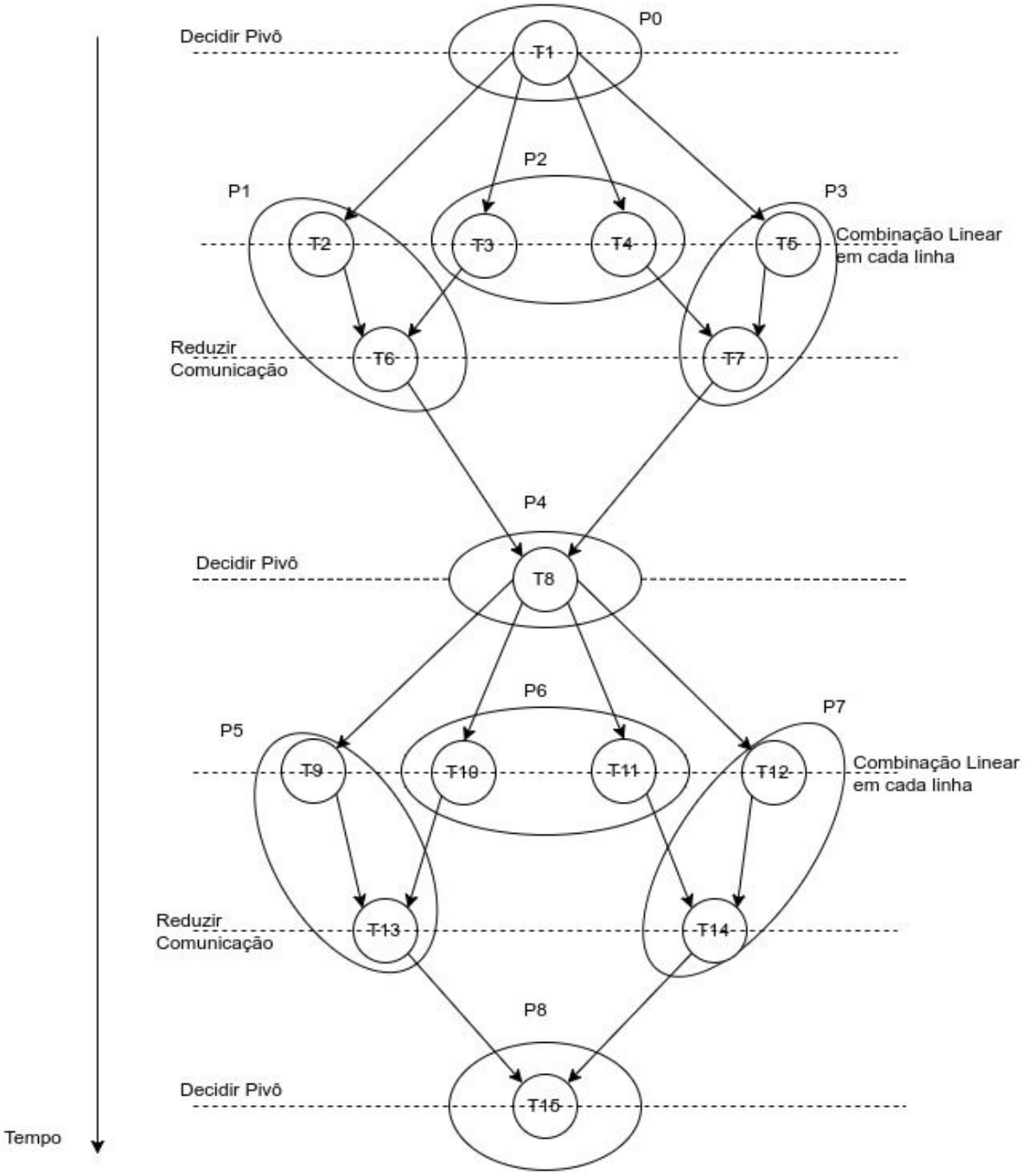


*Fig. 4: Aglomeração das tarefas em processos. Apesar de um aparente desbalanceamento, será utilizado um método de escalonamento destes processos, a fim de minimizar os possíveis efeitos de uma estrutura como a observada em P2 e P4.*
*Compilada pelos Autores.*

## 2.4 - Mapeamento

O método de mapeamento é o estático, nele se particiona o vetor de acordo com o número de processos disponíveis, sejam eles threads ou cores, isso será de acordo com a máquina disponível para rodar o algoritmo.

Essa abordagem é a melhor visto que antes da execução do algoritmo sabe-se ao certo o número de equações e variáveis apresentadas, um número que não muda *on the fly* de acordo com a recebimento de novos dados de outros processos, ou seja, a dependência dele é bem fechada. Com isso, o algoritmo não tem que se adaptar durante a execução para dar conta de tamanhos variáveis de dados, facilitando o gerenciamento do uso de memória.

Neste caso, o mapeamento estático será agregado da execução baseada em particionamento de dados com *task-scheduling*. Cada processo divide a linha pelo o valor do pivô. Cada tarefa possuirá requisitos locais.

| P0 |
|---|
| P1 |
| P2 |

Quanto ao caso de se “zerar” as colunas das outras linhas dado o pivô, a distribuição de processos se dará assim:

| P3 | P4 | P5 | P6 |
|---|---|---|---|

Utilizando esse método de blocos para o mapeamento a distribuição fica feita de maneira mais balanceada, se evitando distribuições falhas e consequentemente com que cores fiquem ociosos devido a finalização de tarefas antes dos outros cores finalizarem.

## 2.5 - Grafo Final

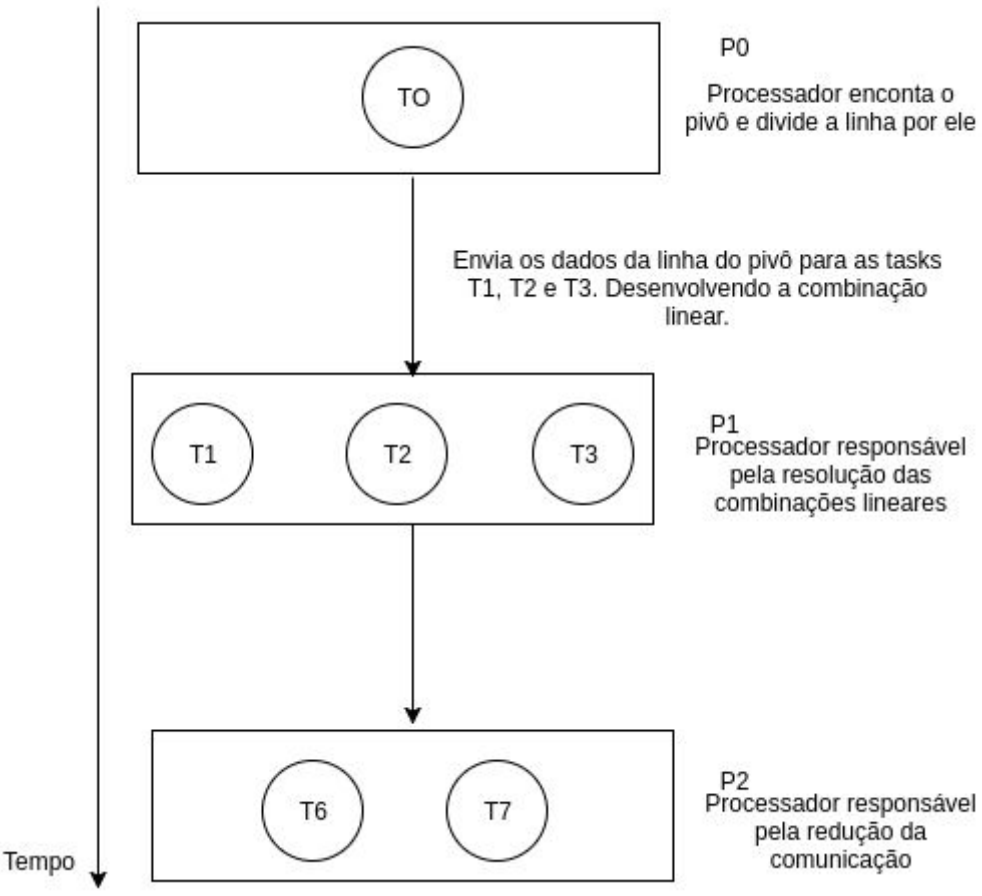


*Fig. 5: Representação da atribuição das tarefas em processadores. Apesar de um aparente desbalanceamento, será utilizado um método de distribuição* round-robin *para se equilibrar a execução e não se ter processadores em* idle.
*Compilada pelos Autores.*

# 3 - Bibliografia

FOSTER, Ian. Designing and building parallel programs . Vol. 78. Boston: Addison Wesley Publishing Company, 1995

[1] http://www.codewithc.com/gauss-jordan-method-algorithm-flowchart/