



**Universidade de São Paulo  
Instituto de Ciências Matemáticas e de Computação (ICMC)  
SSC0903 - Computação de Alto Desempenho**

## **Projeto de Algoritmos Paralelos (PCAM) - Método Iterativo de Jacobi-Richardson**

Matheus Yasuo Ribeiro Utino - 11233689

Pedro Ribas Serras - 11234328

Vinícius Silva Montanari - 11233709

Docente: Dr. Paulo Sérgio Lopes de Souza

**São Carlos  
27 de maio de 2022**

# Sumário

<b>1</b>	<b>Projeto de Algoritmos Paralelos (PCAM)</b>	<b>2</b>
1.1	Particionamento . . . . .	2
1.1.1	Convergência do método . . . . .	2
1.1.2	Método Iterativo de Jacobi-Richardson . . . . .	4
1.2	Comunicação . . . . .	7
1.2.1	Convergência do método . . . . .	7
1.2.2	Método Iterativo de Jacobi-Richardson . . . . .	7
1.3	Aglomeracao . . . . .	8
1.3.1	Convergência do método . . . . .	8
1.3.2	Método Iterativo de Jacobi-Richardson . . . . .	10
1.4	Mapeamento . . . . .	11

## Lista de Figuras

1	Redução e particionamento para a convergência. . . . .	3
2	Redução e particionamento para a convergência de várias linhas. . . . .	3
3	Sistema linear isolando x. . . . .	4
4	Redução e particionamento - Método Iterativo de Jacobi-Richardson . . . . .	5
5	Redução e particionamento para a convergência de várias linhas - Método Iterativo de Jacobi-Richardson. . . . .	6
6	Critério de parada do algoritmo. . . . .	6

# 1 Projeto de Algoritmos Paralelos (PCAM)

Serão descritos os passos para o desenvolvimento do PCAM, que é composto por quatro etapas: particionamento, comunicação, aglomeração e mapeamento. Durante o processo será explicado detalhadamente como será desenvolvido o algoritmo paralelo, assim como as decisões de projeto que foram tomadas.

## 1.1 Particionamento

Essa etapa visa extrair as tarefas e suas dependências, no caso será realizado o particionamento de dados.

### 1.1.1 Convergência do método

Inicialmente precisamos verificar se o método converge para o sistema linear analisado. No caso, a matriz  $A$  será particionada em  $N^2 - N$  tarefas, em que cada tarefa vai possuir o elemento  $A[i, j]$ , desconsiderando o elemento da diagonal principal  $A[i, i]$ . Após isso, as tarefas que são responsáveis pela linha  $i$  de  $A$  devem fazer um somatório dos módulos dos elementos dessa linha, preferencialmente pelo processo de redução. Por fim, esse valor do somatório vai ser dividido pelo elemento  $A[i, i]$ , em que  $A[i, i] \neq 0$  para a convergência do método, caso o somatório seja superior a 1 o método não converge, por outro lado se for inferior a esse valor significa que o método converge para o sistema linear escolhido. A ideia da redução e do particionamento para uma linha  $i$  da matriz  $A$  pode ser visualizada na fig. 1.

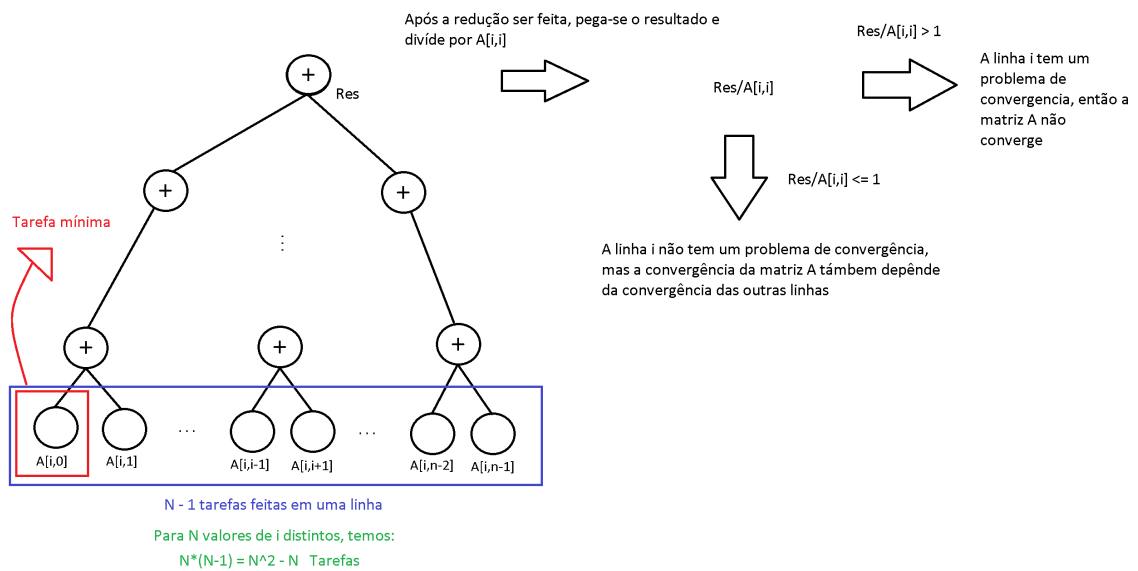


Figura 1: Redução e particionamento para a convergência.

Também podemos demonstrar o comportamento para mais de uma linha, conforme a fig. 2.

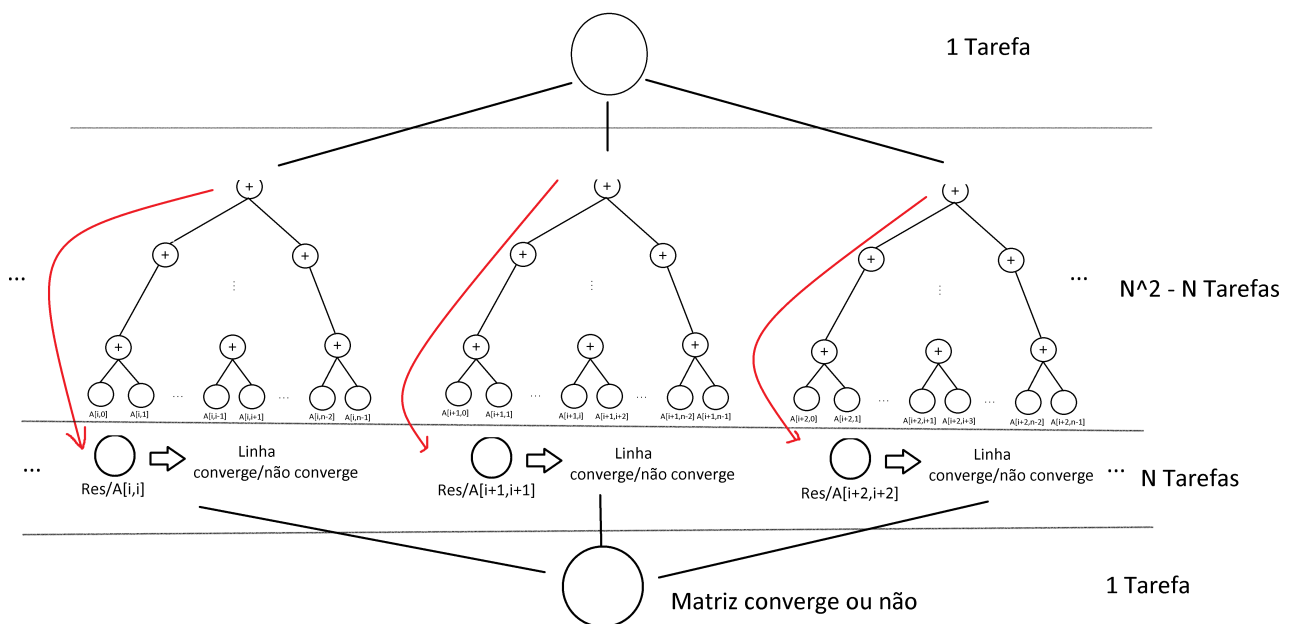


Figura 2: Redução e particionamento para a convergência de várias linhas.

### 1.1.2 Método Iterativo de Jacobi-Richardson

Nesse processo temos a matriz  $A$  e os vetores  $x$  e  $b$  como descrito no problema. Além deles, teremos o vetor  $xa$  para guardar os valores da última iteração de  $x$ . Nas iterações para encontrar um novo vetor  $x$ , inicia-se cada novo valor  $xa[i]$  como  $x[i]$ , sendo que o valor de  $x$  pode ser tomado inicialmente como zero para todos os seus elementos. Tendo o vetor  $xa$  pode-se encontrar os novos valores para o vetor  $x$  pelo processo da fig. 3 . Esse processo pode ser feito em paralelo.

$$\begin{cases} x_1^{(k+1)} = \left( b_1 - (a_{12}x_2^{(k)} + \dots + a_{1n}x_n^{(k)}) \right) / a_{11}, \\ x_2^{(k+1)} = \left( b_2 - (a_{21}x_1^{(k)} + \dots + a_{2n}x_n^{(k)}) \right) / a_{22}, \\ \vdots \\ x_n^{(k+1)} = \left( b_n - (a_{n1}x_1^{(k)} + \dots + a_{n,n-1}x_{n-1}^{(k)}) \right) / a_{nn}, \end{cases}$$

para  $k = 0, 1, \dots$

Figura 3: Sistema linear isolando x.

Agora a matriz  $A$  é particionada em  $N^2 - N$  tarefas, em que cada uma delas possui um elemento de  $A[i, j]$  e o elemento  $xa[j]$  correspondente a coluna  $j$  que a tarefa está computando. Após as multiplicações  $A[i, j] * xa[j]$ , no caso desconsiderando os elementos da diagonal principal, as tarefas que são responsáveis pela linha  $i$  de  $A$  devem fazer suas subtrações (preferencialmente por uma redução) das multiplicações realizadas, gerando assim o novo valor de  $x[i]$ , levando em conta que  $x[i]$  tinha recebido o valor de  $b[i]$  anteriormente. Esse processo pode ser verificado para um linha  $i$  conforme a fig. 4.

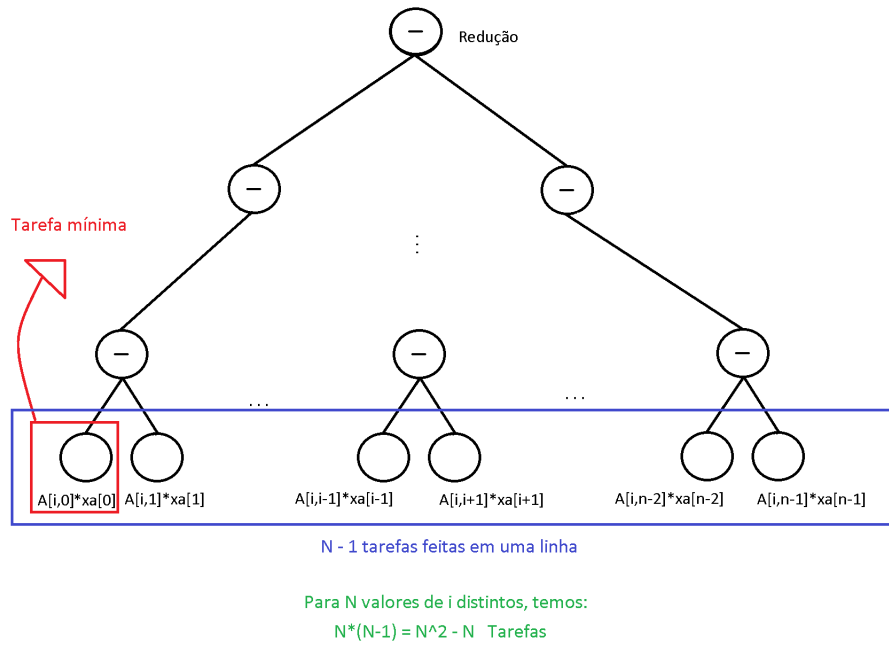


Figura 4: Redução e particionamento - Método Iterativo de Jacobi-Richardson

Também podemos demonstrar o comportamento para mais de uma linha, conforme a fig. 5.

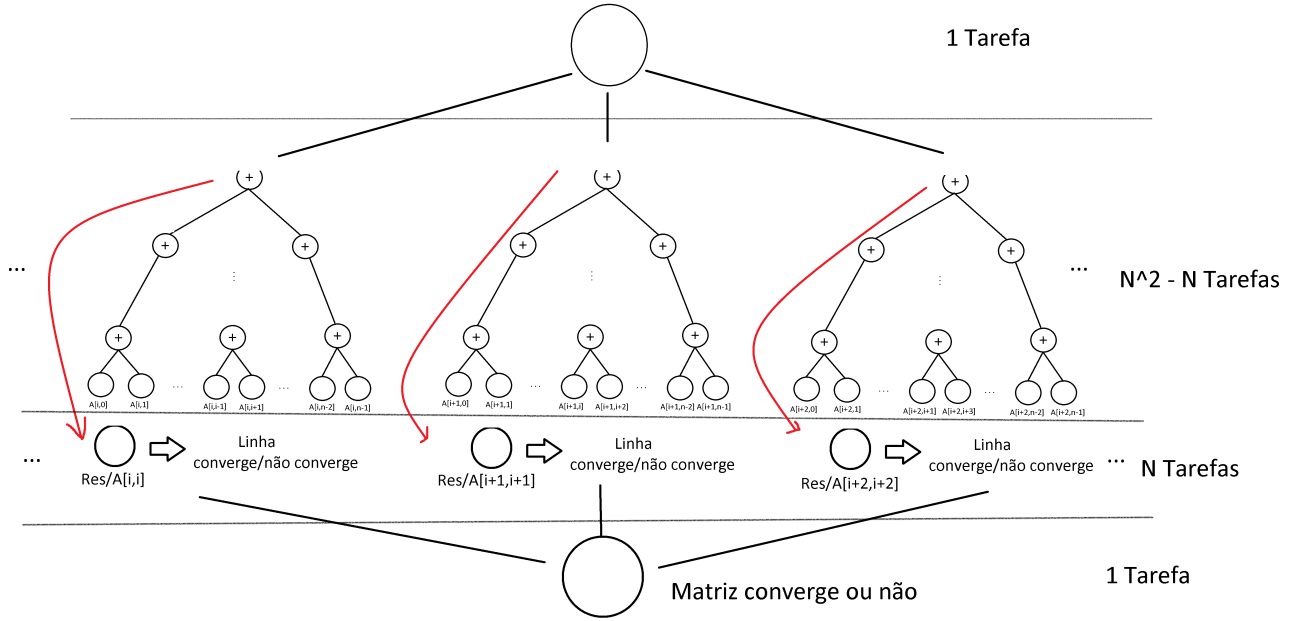


Figura 5: Redução e particionamento para a convergência de várias linhas - Método Iterativo de Jacobi-Richardson.

Tendo realizado a redução é preciso dividir cada  $x[i]$  pelo respectivo valor da diagonal principal de A ( $A[i, i]$ ). Esse processo pode ser dividido em N tarefas, cada uma com um  $x[i]$  diferente e seu respectivo  $A[i, i]$ .

Nesse momento é preciso verificar se a aproximação já é boa suficiente. Para isso, precisaremos dos valores dos vetores x e xa e o limiar escolhido. O processo que será feito segue a fig. 6.

$$D_r = \frac{\max\{|x_i^{(k+1)} - x_i^{(k)}|, i = 1, \dots, n\}}{\max\{|x_i^{(k+1)}|, i = 1, \dots, n\}} \leq \tau,$$

em que  $\tau > 0$  é uma certa tolerância.

Figura 6: Critério de parada do algoritmo.

Criam-se 2N tarefas, duas para cada valor de x. Dividiremos em dois grupos de tarefas de tamanho N. Cada tarefa do primeiro grupo recebe um valor de  $x[i]$  e seu respectivo  $xa[i]$  e calcula o valor absoluto da diferença desses valores, depois é feita uma função de máximo dos resultados de cada tarefa (preferencialmente por uma redução). E assim, encontramos o numerador da verificação



de parada.

Cada tarefa do segundo grupo recebe um valor de  $x[i]$  e calcula o valor absoluto deste, depois é feita uma função de máximo dos resultados de cada tarefa (preferencialmente por uma redução). E assim, encontramos o denominador da verificação de parada. É importante salientar que todas essas tarefas podem ser feitas em paralelo, inclusive tarefas de grupos diferentes.

Caso o valor de verificação encontrado seja maior do que o limiar escolhido, vai para a próxima iteração do método. Caso contrário, chegamos nos valores de  $X$  requeridos dentro do limiar escolhido anteriormente.

## 1.2 Comunicação

Essa etapa visa identificar a comunicação e sincronização entre as tarefas. As comunicações entre as tarefas visam prover os dados de entrada para tarefas e posteriormente recuperar os resultados das computações realizadas.

### 1.2.1 Convergência do método

Inicialmente, cada tarefa criada receberá o seu respectivo valor da matriz  $A$ . Após será feita uma comunicação por meio de uma soma por redução, onde cada tarefa passa seu valor para uma tarefa superior, seus valores são somados e esse processo se repete até chegar em um valor final. Tendo a subtração de cada linha, sobram  $N$  tarefas finais (uma para cada linha), as quais receberão os respectivos valores de  $A[i,i]$  para fazerem suas divisões. Tendo as divisões feitas e as verificações de convergência por linha, acontece mais uma comunicação, onde todas as convergências por linha são passadas para uma tarefa que assimila se a matriz converge no todo (caso todas as linhas convergirem).

### 1.2.2 Método Iterativo de Jacobi-Richardson

Já para o caso do método iterativo cada tarefa recebe o seu elemento da matriz  $A$  e as tarefas responsáveis pelos dados do vetor  $x$  também receberão seus elementos. Cada tarefa, inicialmente faz a multiplicação  $A[i,j]*x[j]$  e nesse momento acontecerá a outra comunicação por meio de uma subtração por redução, onde cada tarefa passa seu valor para uma tarefa superior, seus valores são subtraídos e esse processo se repete até chegar em um valor final. Após isso, sobrarão  $N$  tarefas (uma para cada linha), que por outra comunicação recebem os respectivos valores da diagonal principal

de A para dividir o valor da respectiva redução e assim encontrar o valor final de  $x[z]$ . E essas são as comunicações necessárias para encontrar os novos valores de x.

Para fazer a cópia de x em xa, são passadas para cada uma das N tarefas um valor  $x[i]$  de x e seu valor é copiado para xa.

### 1.3 Aglomeração

Essa etapa visa aglomerar as tarefas em processos, com objetivo de reduzir o overhead de comunicação. Nesse caso, estamos usando uma máquina MIMD com memória compartilhada.

#### 1.3.1 Convergência do método

Considerando a plataforma alvo do algoritmo, vamos agrupar as  $N^2 - N$  tarefas pensando, principalmente, na quantidade de linhas  $N$ , já que o custo de comunicação é mais caro, pois estamos trabalhando com um *cluster* de computadores. Assim, vamos dividir as linhas em NC nós do *cluster*. Desta forma, são  $N/NC$  linhas para cada nó do *cluster*. Se N não for múltiplo de NC, as linhas restantes da divisão inteira devem ser distribuídas para os nós de maneira que cada nó receba no máximo uma linha a mais do que calculada anteriormente. Como cada nó é uma máquina MIMD, vamos dividir as  $N/NC$  linhas em NP processos/*threads* dentro de cada nó, assim chegando a, aproximadamente,  $\frac{N}{NC \cdot NP}$  linhas para cada processos/*threads* de um nó. Com todas as devidas linhas em seus devidos nós, a separação de linhas em *threads* é feita pelo próprio *OpenMP*, distribuindo da maneira mais homogênea possível (se o número de linhas não for múltiplo do número de *threads* criadas, ele distribui as linhas restantes entre as *threads*).

(É importante dizer que para o caso do problema proposto, temos que a matriz A e o vetor b devem ser gerados, isso pode vir a ser um problema, mas não deve ser levado em conta no cálculo do tempo de resposta do nosso algoritmo, já que a geração em si não é realmente importante para ele.

Existem duas principais abordagens possíveis para a geração. A matriz e o vetor serem gerados depois distribuídos pelos nós do *cluster*, mas nesse caso o tempo de transmissão das duas estruturas deveria ser levado em conta para o cálculo, já que a geração não é levada em conta, mas a transmissão de dados deveria ser levada em conta, o que geraria um gigantesco aumento no tempo de resposta. Outra possibilidade seria a geração dessas estruturas dentro de cada nó, sendo que um mesmo nó pode gerar tudo ou apenas o que ele utilizará, desta forma não sendo necessário comunicar esses nós

para transmitir A e b. É importante dizer que com altos valores de N o tanto de dados transferidos é gigantesco chegando a muito gigabytes e por isso o algoritmo acaba ficando lento.

No nosso programa, seguimos a abordagem onde cada nó gera as respectivas linhas que ele irá usar. Como tanto o a verificação de convergência do método quanto o próprio método iterativo, quando falamos dos dados de A e b, eles apenas necessitam dos dados das respectivas linhas que eles são responsáveis, esse dados não são comunicados para os outros nós, já que só serão utilizados dentro do mesmo nó, o que poupa o tempo necessário para a comunicação.

Por fim, é importante salientar como foi feita a geração da matriz A e do vetor b para que para um N específico ela sempre tenha os mesmo valores. Essa geração foi feita usando uma *seed* que varia com o número da linha, sendo que para uma linha i a *seed* sempre será a mesma (adotamos uma *seed* básica em comum para todas linhas e apenas somando o número da linha para poder calcular os valores para a respectiva linha), dessa forma, quando setamos um N, a geração não irá se repetir para os nós como repetiria se fosse colocada uma *seed* única para todos e também consegue se manter sempre igual para o mesmo N, mesmo em execuções diferentes.)

Com as distribuições de dados feita, deve-se iniciar o algoritmo de análise de convergência da matriz A. Nesse caso, cada processo fará a verificação para cada linha em separado e no final juntaremos todos esses dados para descobrir a convergência. Para iniciar é necessário fazer a soma de cada elemento da linha, sem somar o elemento relativo a diagonal principal da matriz. Após a soma da linha ter sido toda feita, deve-se dividir seu valor pelo respectivo valor da diagonal principal. Caso a divisão seja maior que 1 a linha não "converge" e caso seja menor ela "converge" ("converge" aqui foi usado entre aspas, pois estamos falando de uma linha e não da matriz, sendo que linhas em si não convergem, mas a ideia de "convergência" da linha está sendo usada como: se a linha "converge" é porque ela não é um empecilho para a convergência da matriz, mas caso a linha não seja "convergente" a matriz não irá convergir). Como, provavelmente, temos várias linhas por processo/*thread*, é preciso verificar a "convergência" de cada uma delas, para depois enviar apenas um *booleano* (na implementação foi usado um inteiro, mas poderia ter sido usado um bit único) verdadeiro, caso todas as linhas sejam "convergentes", ou falso, caso uma ou mais não seja, para o "processo mestre" dentro do nó do *cluster*. A análise de todas as linhas dentro de um processo/*thread* é feita de forma sequencial. Desta forma, quando as *threads* acabam, é feito um processo de redução entre todas as *threads* do nó, gerando um *booleano* de "convergência" das linhas do nó todo. De forma análoga a redução

das *threads*, cada nó deve enviar seus resultados da convergência para um nó mestre (o que tem o processo 0) em um processo de redução, chegando a um único *booleano*, sendo que se esse *booleano* for verdadeiro (1) a matriz A converge, caso contrário ela não converge.

### 1.3.2 Método Iterativo de Jacobi-Richardson

Nessa parte, tanto a matriz A quanto o vetor b já estão distribuídos nos nós como feito anteriormente, assim tendo a mesma distribuição de linhas e dados para os nós e processos/*threads* que a feita anteriormente. É importante dizer que diferente da análise de convergência, nessa parte precisaremos do vetor x e xanterior. Sendo que os processos gerarão apenas os valores de x[i] das suas respectivas linhas i. Já o vetor xanterior é necessário por inteiro para todos os processos/*threads* e ele deve ser inicialmente preenchido de zeros, então cada nó gerará o seu xanterior inteiro e irá preenche-lo.

O método Jacobi deve acontecer como descrito no particionamento, mas feito em  $NC \cdot NP$  processos/*threads*, sendo que cada processo cuida de suas respectivas linhas. Após o processo calcular seus respectivos x[i], eles devem ser mandados para o nó mestre (processo 0 do *MPI*) por meio de um *gatherv* para que uma matriz x total, que chamaremos de xf (de tamanho N), seja formada. Após o *gatherv* o processo 0 é o único que tem o vetor xf. Com xf deve-se fazer a verificação de parada do método. Isso deve-ser feito apenas no nó mestre (processo 0) para não diminuir a comunicação entre nós.

Assim, para as tarefas de verificação do critério de parada atribuiremos as  $2N$  tarefas á NP processos/*threads*, sendo que os máximos locais serão calculados pelos processos e, no final, acontecerá a redução para encontrar o máximo global. É importante salientar que para diminuir a comunicação, é indicado que um valor do iterador i seja passado apenas para um processo, sendo que esse processo deve fazer todos os cálculos que usam essa posição nos vetores.

Tendo a verificação feita, caso o método Jacobi deva continuar, no nó mestre deve ser feita a cópia de xf para xanterior, em apenas um processo/*thread* para não ter problema de falso compartilhamento. Com o novo valor de xanterior o nó com o processo mestre deve fazer um broadcast para os outro nós enviando esse vetor. Após isso o uma nova iteração deve ser iniciada.

Caso a verificação mostre que deva-se parar as iterações do método Jacobi, então o no mestre faz um *broadcast* avisando todos os nós que o processo chegou ao fim.

(É importante salientar que para esse trabalho deve-se pedir que o usuário coloque uma linha

para verificar se o resultado nela foi condizente. Isso não entra no cálculo da velocidade do algoritmo em si porque não faz parte dele propriamente dito, mas é importante descrever como vai ser feito já que ele utilizará os dados distribuídos entre os nós, então é necessário comunicar os nós de acordo com o número da linha pedida.

Para fazer isso, ao final do método, é perguntado para o usuário que linha ele quer verificar. Tendo a resposta, o nó mestre (processo 0) calcula qual nó tem a linha desejada e envia o número desse nó para todos os nós por um *broadcast*. Todos os nós tirando o nó mestre (processo 0) e o nó com a linha desejada não serão mais usados, então eles são finalizados. O nó mestre manda o vetor  $xf$  para o nó que tem os valores de  $A[i]$  e  $b[i]$  da linha  $i$  escolhida e também manda o número da linha (os dois envios por comunicação ponto-a-ponto bloqueante). O esse nó imprime os resultados e todos são finalizado.)

## 1.4 Mapeamento

Cada nó do *cluster* em uso deve receber inicialmente apenas 1 processo, um total de  $NN$  processos, já que temos  $NN$  nós em uso. Esse processo será responsável por criar os outros processos (threads) dentro de seu específico nó. É importante dizer que um dos nós estará rodando o processo 0 e esse processo é equivalente a um processo mestre, coordenando os outros nós e seus respectivos processos, sendo que qualquer nó pode rodar o processo 0.

Só é possível controlar o mapeamento dos  $NN$  processos dentro dos  $NN$  nós, já que quando olhamos para um nó separadamente, que é uma MIMD de memória compartilhada, e suas *threads*, é responsabilidade do sistema operacional distribuir as *threads* pelos processadores. Idealmente, essas *threads* deveriam ser distribuídas homogeneamente pelos processadores do nó, mas essa decisão só pode ser tomada pelo SO.