UNIOESTE – Universidade Estadual do Oeste do Paraná
Curso de Bacharelado em Ciência da Computação

Soluções para o Balanceamento de Carga em Arquiteturas Paralelas Heterogêneas

Luis Fernando Veronese Trivelatto

Orientador: Guilherme Galante

Cascavel

26 de junho de 2018

Roteiro

1. Introdução
   1.1 Objetivo
2. Computação Paralela
3. Escalonamento e Balanceamento de Carga
4. Biblioteca de Balanceamento Multiframework
   4.1 Modelo de Aplicação Paralela
   4.2 Multiframework Balance
   4.3 Algoritmo de Balanceamento
5. Experimentos: Estudos de Caso
6. Experimentos: Resultados
7. Conclusões

Introdução

- Busca por maior poder computacional
- Processadores modernos são poderosos, mas algumas aplicações necessitam mais
  - Ex.: modelagem climática, simulação de enovelamento de proteínas e desenvolvimento farmacêutico
- Historicamente, crescimento da capacidade computacional esteve relacionado ao desenvolvimento dos processadores
  - Limitações: consumo de energia, dissipação de calor, tamanho físico
- Como atender a demanda por maior capacidade computacional?
- Computação paralela

Introdução

- *Grids*, *clusters*, processadores *multicore*, aceleradores
- Diferentes níveis de paralelismo a serem explorados
- Popularização de arquiteturas heterogêneas

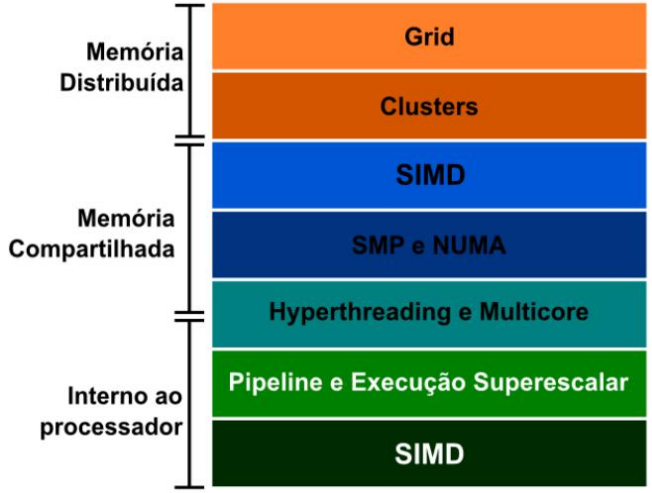


Figura 1: diagramação hierárquica dos diversos níveis de paralelismo. Fonte: (GALANTE, 2013).

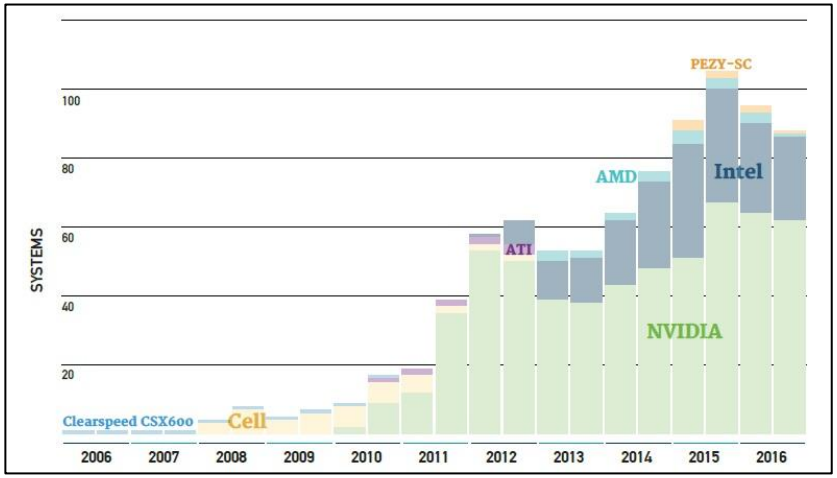


Figura 2: número de sistemas no Top500 que utilizam aceleradores/coprocessadores. Fonte: www.nextplatform.com

Introdução

- Maior dificuldade de programação
- Balanceamento de carga
- Frequentemente é definido em tempo de programação, sendo estático
  - Condições em tempo de execução não são consideradas:
    - Alterações na arquitetura paralela
    - Desconhecimento do programador do ambiente de execução
    - Distribuição manual sub-ótima

Introdução

- Modelo comum de aplicação: processamento iterativo, com sincronização ao fim de cada iteração
  - Ex.: produto de matrizes, método de Jacobi, programação dinâmica, simulações computacionais, e problemas de caminho mínimo
- Neste caso, desempenho é limitado pelo pior desempenho em cada iteração

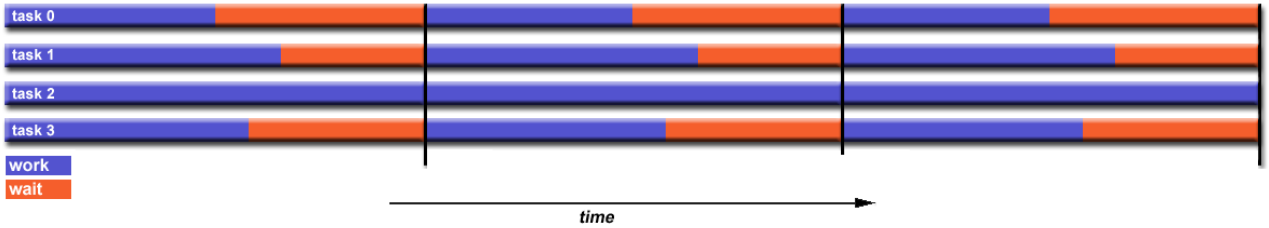


Figura 3: Desbalanceamento em processamento iterativo. Adaptado de: https://computing.llnl.gov/tutorials/parallel_comp/

- Uso dos recursos computacionais poderia ser otimizado se o balanceamento de carga ocorresse dinamicamente, em tempo de execução

Introdução | Objetivo

- Viabilizar uma solução para balanceamento de carga em arquiteturas paralelas heterogêneas em aplicações paralelas iterativas
  - Particionamento de dados e sincronização ao fim de cada iteração
- Explorar múltiplos níveis de paralelismo:
  - *Grids*/*clusters* (MPI)
  - *Multi-core* (OpenMP)
  - Aceleradores em geral
- Adaptação pouco custosa a códigos existentes

Multi-core

- Múltiplos núcleos de processamento em um único *chip*

Multicomputadores

- Computadores independentes conectados por uma rede
  - Ex.: *clusters* e *grids*

Aceleradores

- Dispositivos auxiliares de *hardware*, tipicamente especializados em um tipo específico de computação
  - Ex: GPUs

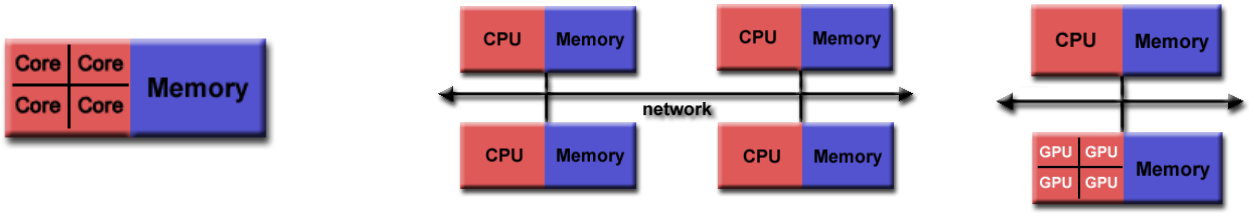


Figura 4: ilustrações de diferentes arquiteturas paralelas. Adaptado de: https://computing.llnl.gov/tutorials/parallel_comp/

Computação Paralela | Ferramentas

- Cada arquitetura apresenta um modelo de paralelismo
- Para cada modelo, há algumas ferramentas específicas para extração do paralelismo
- Ferramentas popularmente utilizadas:
- **OpenMP**
  - Sistemas de memória compartilhada
- **MPI**
  - Sistemas de memória distribuída
- **CUDA**
  - Aplicações em GPUs -> CPU faz *offloading* de trabalho para GPU

Computação Paralela | Arquiteturas Heterogêneas

- Diferentes níveis de paralelismo a serem explorados
- Criação de sistemas heterogêneos
  - Múltiplos níveis de paralelismo (ex. CPU + GPU)
  - Diferenças na capacidade computacional dos processadores
- Maior complexidade de programação
  - Diferentes modelos de programação
  - Ferramentas distintas
  - Otimização do uso dos recursos disponíveis
- Muitas aplicações são criadas combinando ferramentas específicas para cada nível
- Há ferramentas que buscam simplificar a programação em sistemas heterogêneos, mas pouco estabelecidas
- Código legado

Balanceamento e Escalonamento de Carga

- Escalonamento: refere-se a distribuição das tarefas para unidades de processamento
  - Escalonamento inadequado: tarefas podem ser distribuídas para processadores pouco eficientes para executá-las
- Balanceamento: divisão da carga de trabalho entre as unidades
  - Balanceamento inadequado: gera ociosidade dos processadores

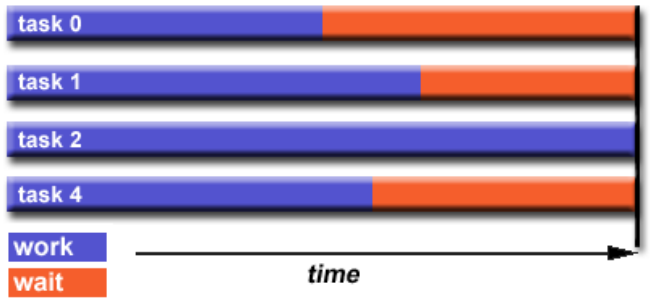


Figura 5: ilustração das consequências do balanceamento.
Fonte: https://computing.llnl.gov/tutorials/parallel_comp/

Trabalhos Correlatos

- (ACOSTA; BLANCO; ALMEIDA, 2013)
- Apresenta um esquema de paralelismo baseado em processamento iterativo
- Tenta igualar o tempo de execução em cada unidade de processamento
- Simples adaptação a códigos existentes

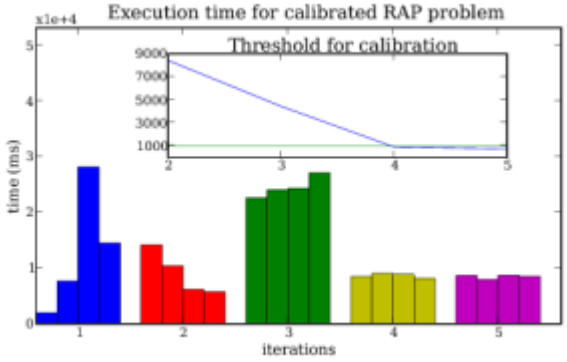


Figura 6: tempo de execução por processador nas primeiras iterações de um problema analisado.
Fonte: (ACOSTA *et al.*, 2013).

Trabalhos Correlatos

- Explorar todos os recursos computacionais disponíveis não é uma tarefa simples
- (ACOSTA; BLANCO; ALMEIDA, 2013) apresenta uma solução interessante para aplicações iterativas mas não explora arquiteturas mais escaláveis (ex.: *clusters* com máquinas *multicore* e aceleradores)

Biblioteca | Modelo de Aplicação

- Processamento iterativo sobre um domínio

```
for(int it = 0; it < num_iteracoes; it++)
{
    for(int i = 0; i < N; i++)
        processa(i);
}
```

Algoritmo 1: esqueleto de implementação sequencial de um problema iterativo.

- O domínio é particionado em subdomínios, que são processados paralelamente
- Após, os dados são sincronizados via comunicação coletiva/série de comunicações ponto a ponto
  - A partir da dependência de dados do problema
- Comunicação coletiva onde os dados dos processos são combinados e enviados a todos é comum: *MPI_Allgatherv*

Biblioteca | Modelo de Aplicação

```
int MPI_Allgatherv(const void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, const int recvcounts[], const int displs[],
MPI_Datatype recvtype, MPI_Comm comm);
```

Algoritmo 2: protótipo do método MPI_Allgatherv em C.

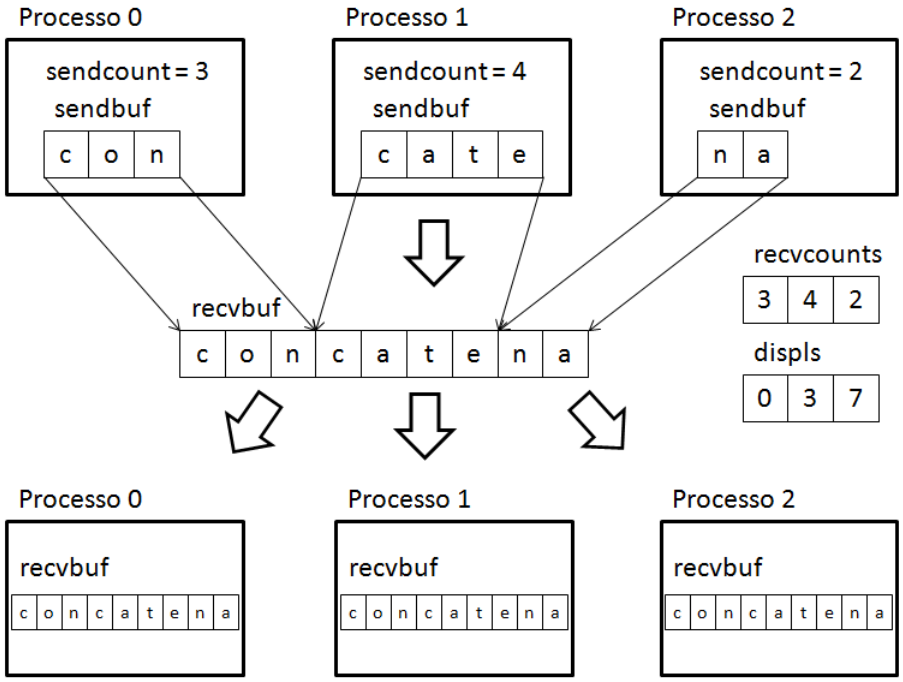


Figura 7: funcionamento do método MPI_Allgatherv. Fonte: Autoria própria

Biblioteca | Modelo de Aplicação

```
int proc;           // Id do processo MPI
int num_proc;       // Número de processos MPI
int num_threads;    // Número de threads neste nó
int displs[num_proc], counts[num_proc];
...
#pragma omp parallel num_threads(num_threads)
for(int it = 0; it < num_iteracoes; it++)
{
    int tid = omp_get_thread_num();  // Id da thread
    int ini = displs[proc];
    int fim = displs[proc] + counts[proc];

    #pragma omp for
    for(int i = ini; i < fim; i++)
    {
        if(thread_usa_GPU(tid)) CUDA_processa(i);
        else                    processa(i);
    }

    #pragma omp barrier
    #pragma omp single
    MPI_Allgatherv(&result[displs[proc]], counts[proc], MPI_DATATYPE,
                   result, counts, displs, MPI_DATATYPE,
                   MPI_COMM
    );
}
```

Algoritmo 3: esqueleto de implementação paralela de um problema iterativo.

Biblioteca | Multiframework Balance

- Biblioteca para balanceamento em aplicações MPI + OpenMP
- Extensão da biblioteca *ULL_MPI_calibrate* apresentada em (ACOSTA; BLANCO; ALMEIDA, 2013)
- Dois níveis de balanceamento
  - Processos MPI
  - *Threads* OpenMP
- Permite usar *threads* para incluir aceleradores no sistema
- Carga de trabalho é um intervalo discreto [0, N)
- Cada processo/*thread* recebe um subintervalo contíguo para processar

Biblioteca | Multiframework Balance

```
int proc;           // Id do processo MPI
int num_proc;       // Número de processos MPI
int num_threads;    // Número de threads neste nó
int displs[num_proc], counts[num_proc];
...
#pragma omp parallel num_threads(num_threads)
for(int it = 0; it < num_iteracoes; it++)
{
    int tid = omp_get_thread_num();  // Id da thread
    int ini = displs[proc];
    int fim = displs[proc] + counts[proc];

    Multiframework_init_section(it, counts, displs, THRESHOLD, tid, &ini, &fim);

    #pragma omp for
    for(int i = ini; i < fim; i++)
    {
        if(thread_usa_GPU(tid)) CUDA_processa(i);
        else                    processa(i);
    }

    Multiframework_end_section(it, tid);

    #pragma omp barrier
    #pragma omp single
    MPI_Allgatherv(&result[displs[proc]], counts[proc], MPI_DATATYPE,
                   result, counts, displs, MPI_DATATYPE,
                   MPI_COMM
    );
}
```

Algoritmo 4: esqueleto de implementação paralela de um problema iterativo usando Multiframework Balance.

Biblioteca | Multiframework Balance

```
void Multiframework_Init_lib(
    int problem_begin,
    int problem_end,
    int num_threads,
    int proc_id,
    int num_proc,
    MPI_Comm comm
);


void Multiframework_Finalize_lib()
```

```
void Multiframework_begin_section(
    int iteration,
    int *counts,
    int *displs,
    int threshold,
    int thread_id,
    int *my_begin,
    int *my_end
);

void Multiframework_end_section(
    int iteration,
    int thread_id
);
```

Algoritmos 5, 6, 7, 8: protótipo dos métodos da biblioteca.

Biblioteca | Algoritmo de Balanceamento

- Tentar igualar o tempo de execução entre os processos/*threads*
- T[] = vetor com tempo de trabalho de cada UP
- counts[], displs[] = vetores definindo carga de trabalho
- Calcula-se RP[] = poder relativo de cada UP

$$RP[i] = \frac{counts[i]}{T[i]}$$

$$SRP = \sum_{i} RP[i]$$

$$counts[i] = round\left(problemSize * \frac{RP[i]}{SRP}\right), \qquad se\ 0 \leq i \leq n-2$$

$$counts[i] = problemSize - \sum_{i=0}^{n-2} counts[i], \qquad se\ i = n-1$$

$$displs[0] = L$$

$$displs[i] = displs[i-1] + count[i-1], \qquad para\ i \geq 1$$

Experimentos: Estudos de Caso

- Método de Jacobi

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^k\right), \qquad i = 1,2, \ldots n$$

- RAP

$$G[i][x] = \max\{G[i-1][j-x] + f_i(x), 0 < x \leq j\}, \qquad para\ i \geq 2$$
$$G[1][x] = f_1(j), \qquad para\ 0 < j \leq M$$
$$G[i][0] = 0$$

- Transferência de calor

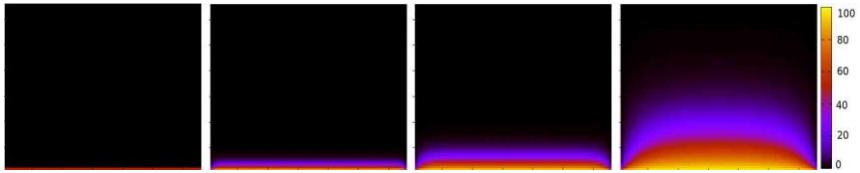


Figura 8: ilustração do problema de transferência de calor. Fonte: Autoria própria

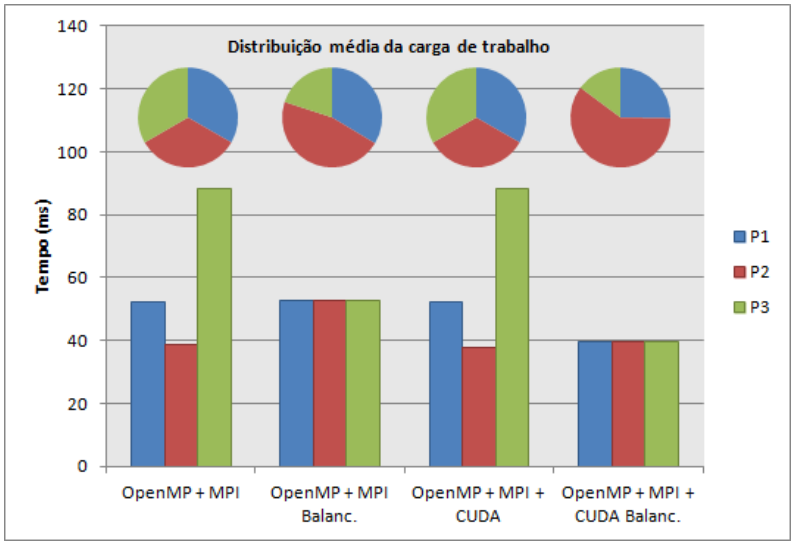
$$U_{i,j}^{n+1} = U_{i,j}^n + s_x\left(U_{i+1,j}^n - 2U_{i,j}^n + U_{i-1,j}^n\right) + s_z\left(U_{i,j+1}^n - 2U_{i,j}^n + U_{i,j-1}^n\right)$$

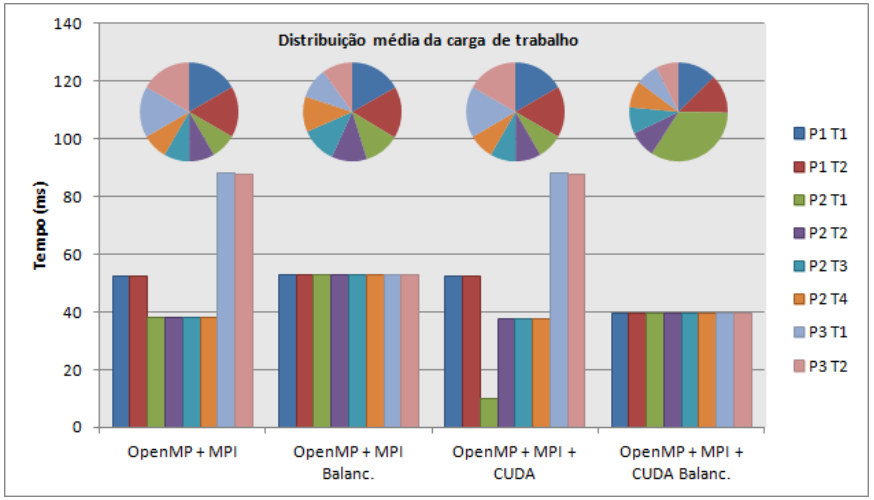
Experimentos: Resultados

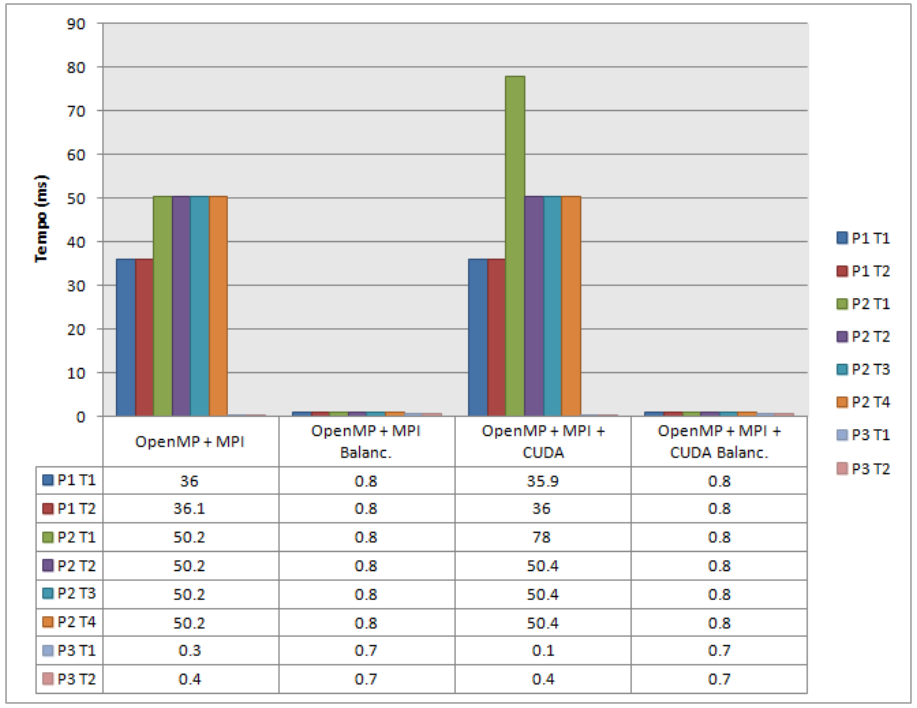
- 5 implementações dos problemas
  - Sequencial
  - OpenMP + MPI com e sem balanceamento
  - OpenMP + MPI + GPU (CUDA) com e sem balanceamento

- Execuções em um *cluster* com 3 máquinas:
  - Máquina 1:
    - 1 Intel Core i3-2100 (2 *cores*), frequência 3.10 GHz, 4 GB RAM
  - Máquina 2:
    - 2 Intel Xeon E5620 (4 *cores*), frequência 2.40 GHz, 16 GB RAM
    - GPU Tesla K20c (2560 núcleos CUDA), frequência 706 MHz, 5 GB RAM
  - Máquina 3:
    - 1 Pentium Dual-Core E5200 (2 *cores*), frequência 2.50 GHz, 1 GB RAM

Experimentos: Resultados | Jacobi



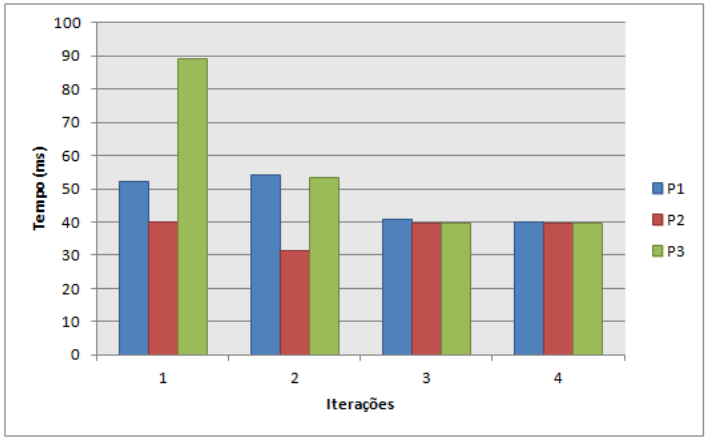


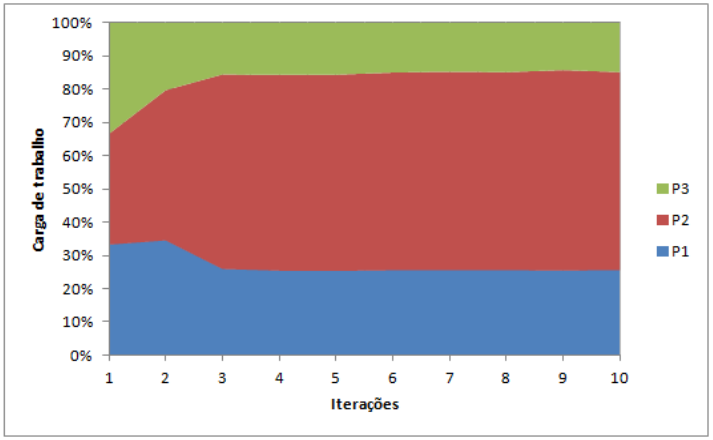


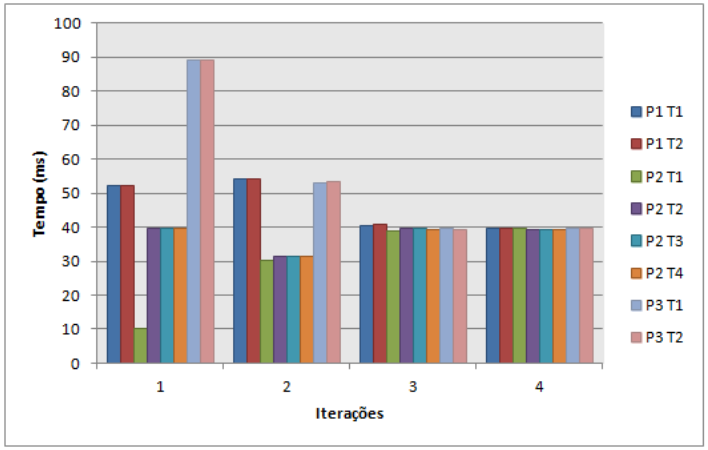
	OpenMP + MPI	OpenMP + MPI Balanc.	OpenMP + MPI + CUDA	OpenMP + MPI + CUDA Balanc.
P1 T1	36	0.8	35.9	0.8
P1 T2	36.1	0.8	36	0.8
P2 T1	50.2	0.8	78	0.8
P2 T2	50.2	0.8	50.4	0.8
P2 T3	50.2	0.8	50.4	0.8
P2 T4	50.2	0.8	50.4	0.8
P3 T1	0.3	0.7	0.1	0.7
P3 T2	0.4	0.7	0.4	0.7

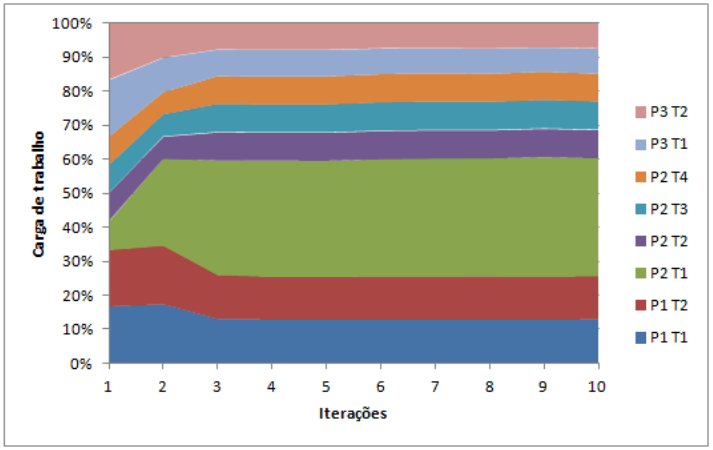
Experimentos: Resultados | Jacobi

- Primeiras iterações – OpenMP + MPI + CUDA

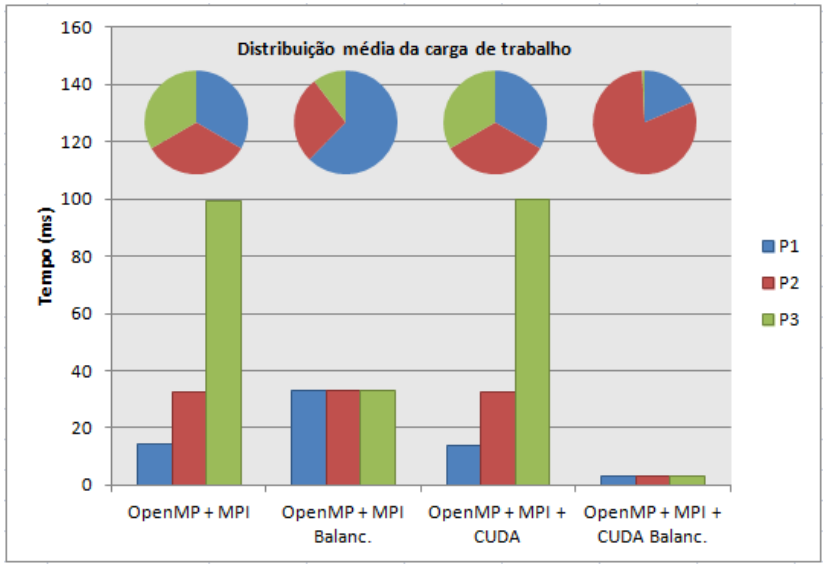


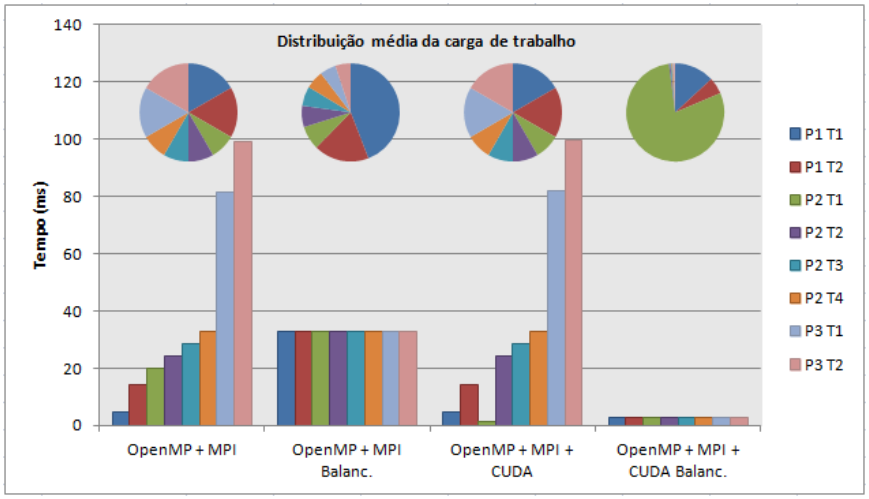


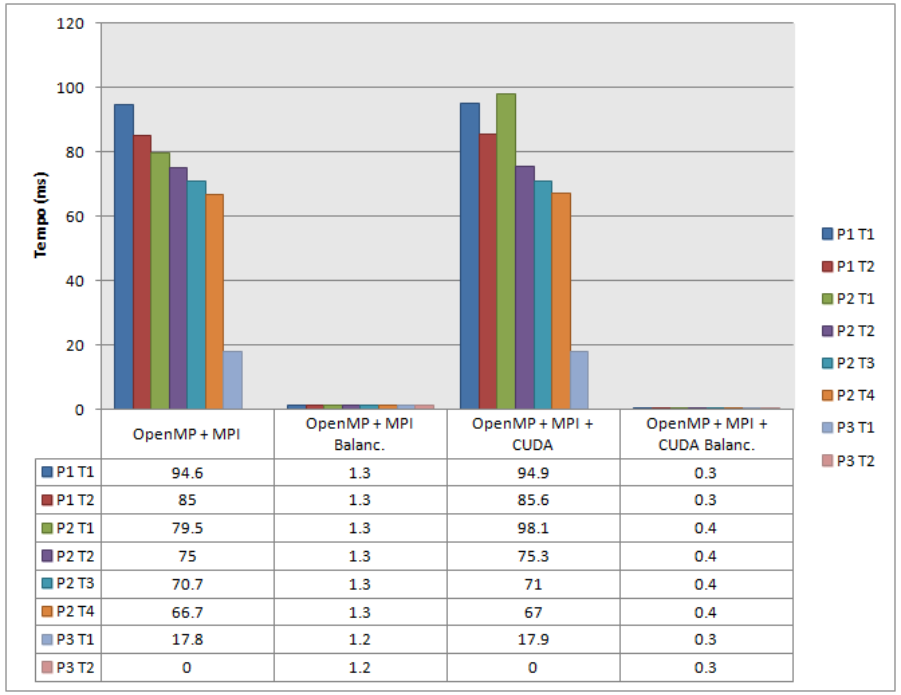




Experimentos: Resultados | RAP



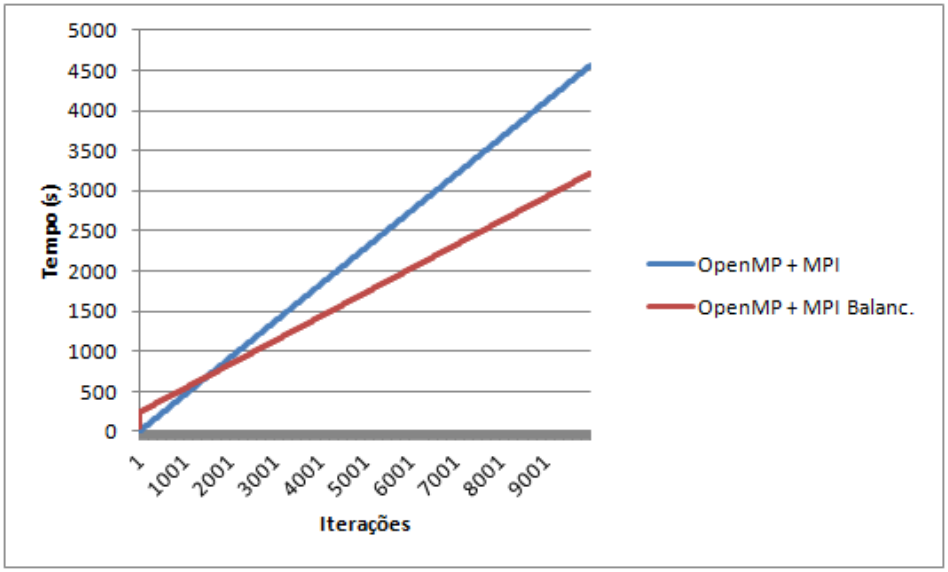




	OpenMP + MPI	OpenMP + MPI Balanc.	OpenMP + MPI + CUDA	OpenMP + MPI + CUDA Balanc.
P1 T1	94.6	1.3	94.9	0.3
P1 T2	85	1.3	85.6	0.3
P2 T1	79.5	1.3	98.1	0.4
P2 T2	75	1.3	75.3	0.4
P2 T3	70.7	1.3	71	0.4
P2 T4	66.7	1.3	67	0.4
P3 T1	17.8	1.2	17.9	0.3
P3 T2	0	1.2	0	0.3

Experimentos: Resultados | Transferência de Calor

Implementação	Tempo (s)
Sequencial	1,21
OpenMP + MPI	0,46
OpenMP + MPI Balanc., fase 1 (iterações 1-4)	58,87
OpenMP + MPI Balanc., fase 2 (iterações >4)	0,30



Experimentos: Resultados

Implementação		*Speedup* vs não balanceado
Método de Jacobi	OpenMP + MPI	1,55
	OpenMP + MPI + CUDA	1,92
RAP	OpenMP + MPI	2,68
	OpenMP + MPI + CUDA	11,18
Transferência de calor	OpenMP + MPI	1,41

Implementação		Taxa de utilização sem balanceamento	Taxa de utilização com balanceamento
Método de Jacobi	OpenMP + MPI	61,2%	98,5%
	OpenMP + MPI + CUDA	57,2%	98,0%
RAP	OpenMP + MPI	38,4%	96,3%
	OpenMP + MPI + CUDA	36,0%	89,7%

Implementação		*Threshold* = 5%		*Threshold* = 1%	
		Média	Maior	Média	Maior
Método de Jacobi	OpenMP + MPI	1,6	5	6	20
	OpenMP + MPI + CUDA	2,0	3	5,8	12
RAP	OpenMP + MPI	7,4	16	54,2	185
	OpenMP + MPI + CUDA	24,3	45	107,7	373

Implementação	*Speedup* biblioteca	*Overhead* por iteração
Método de Jacobi	0,96	2,0 ms
RAP	0,97	1,0 ms

Conclusão

- Realizar distribuição de carga adequada é fundamental para explorar ao máximo potencial da arquitetura
- Soluções para o balanceamento de carga com baixo custo de adaptação a códigos existentes
- Desenvolveu-se uma biblioteca para aplicações paralelas iterativas com OpenMP, MPI e aceleradores

Conclusão

- Obteve-se resultados satisfatórios
  - Speedup: 1,41 a 11,18
  - Taxa de utilização da arquitetura: 36% - 61% para 89% - 98%
  - Tempo para balanceamento: < 25 iterações em todas as implementações, < 10 em 3/4 implementações
  - *Overhead* de execução: 1-2 ms
  - Inclusão a códigos com alteração de por vezes 5 linhas de código

- Dificuldades e limitações
  - Ocupação das máquinas por outros processos, influenciando o balanceamento
  - Restrição da biblioteca a uma classe particular de aplicações paralelas

Trabalhos Futuros

- Desenvolvimento de técnicas para a ampliação da classe de aplicações paralelas alcançadas pela biblioteca
- Avaliação do algoritmo de balanceamento em problemas com carga de custo dinâmico
- Avaliação de outros métodos de minimização de oscilações na distribuição
- Avaliação de algoritmos baseados em outros critérios que não tempo de execução
- Extensão da biblioteca para outras tecnologias

¡Gracias!

- Perguntas?

1. Introdução
   1.1 Objetivo
2. Computação Paralela
3. Escalonamento e Balanceamento de Carga
4. Biblioteca de Balanceamento Multiframework
   4.1 Modelo de Aplicação Paralela
   4.2 Multiframework Balance
   4.3 Algoritmo de Balanceamento
5. Experimentos: Estudos de Caso
6. Experimentos: Resultados
7. Conclusão

Experimentos: Resultados

- Tempo de execução e *speedup*

- Jacobi

	Sequencial	OpenMP + MPI	OpenMP + MPI Balanc.	OpenMP + MPI + CUDA	OpenMP + MPI + CUDA Balanc.
Tempo (s)	1564	470	303	470	245
Speedup vs sequencial	-	3,32	5,17	3,32	6,39
Speedup vs não balanc.	-	-	1,55	-	1,92

- RAP

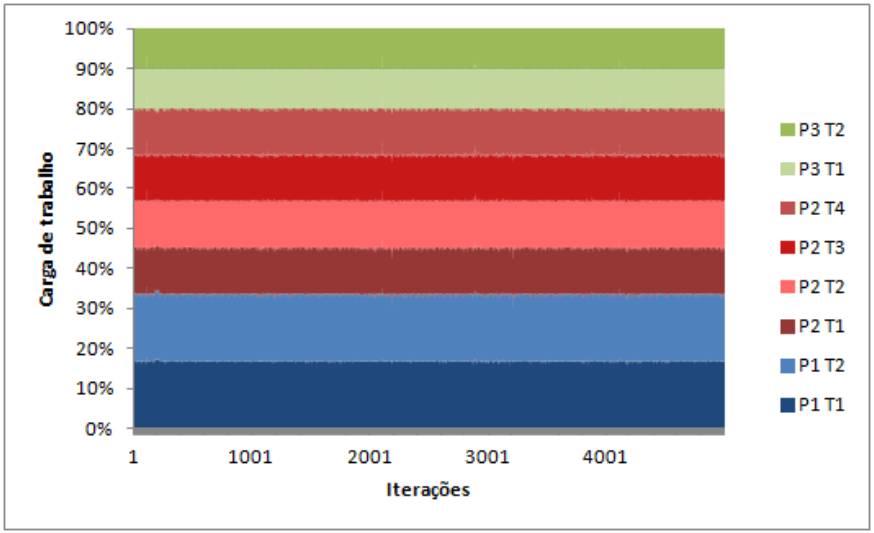
	Sequencial	OpenMP + MPI	OpenMP + MPI Balanc.	OpenMP + MPI + CUDA	OpenMP + MPI + CUDA Balanc.
Tempo (s)	874	513	191	515	46
Speedup vs sequencial	-	1,70	4,56	1,69	19,04
Speedup vs não balanc.	-	-	2,68	-	11,18

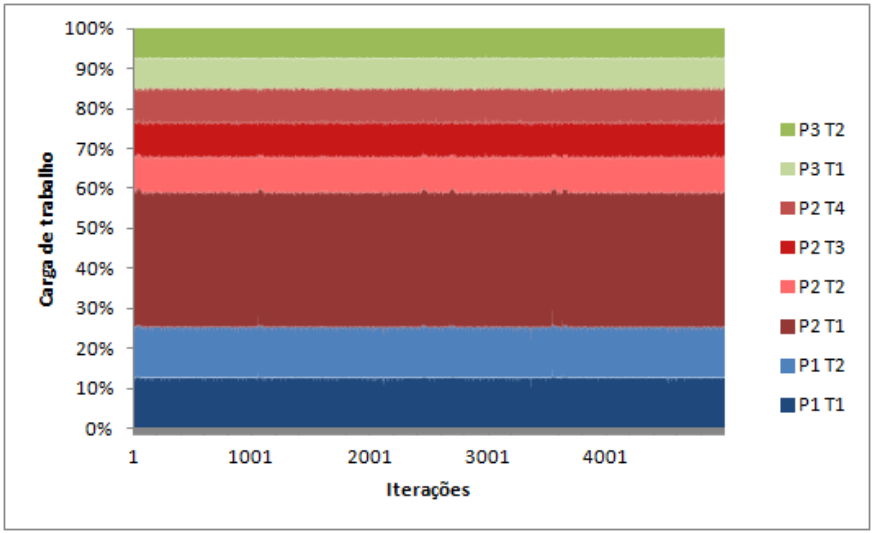
- Transf. de calor

	Sequencial	OpenMP + MPI	OpenMP + MPI Balanc.
Tempo (s)	12137	4612	3265
Speedup vs sequencial	-	2,63	3,72
Speedup vs não balanc.	-	-	1,41

Experimentos: Resultados | Jacobi

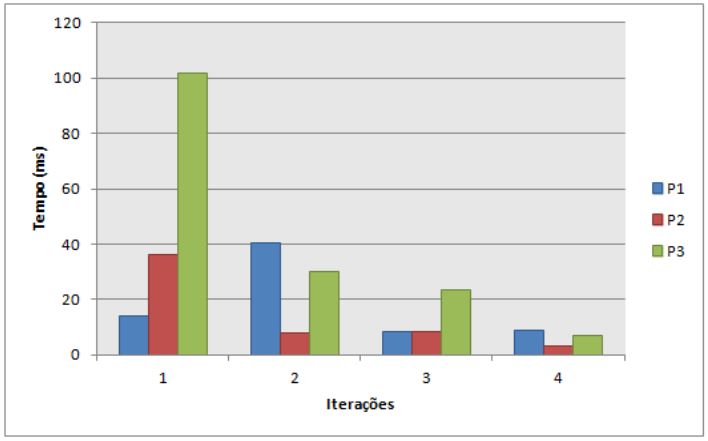
- Distribuição da carga de trabalho ao longo da execução

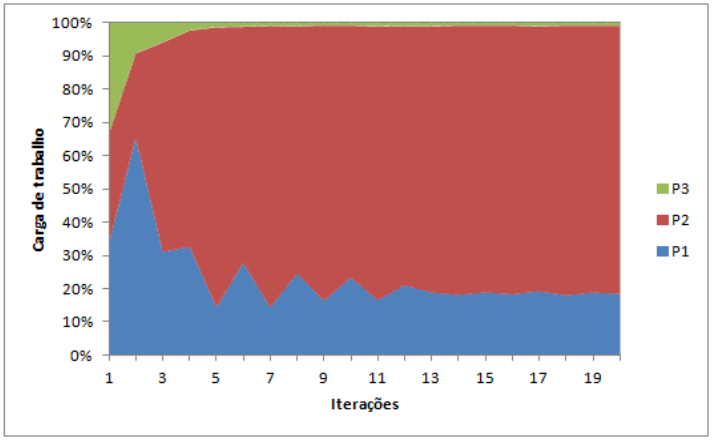


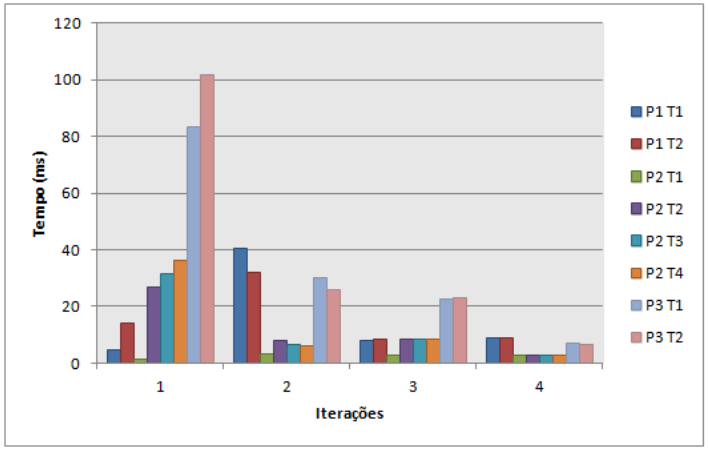


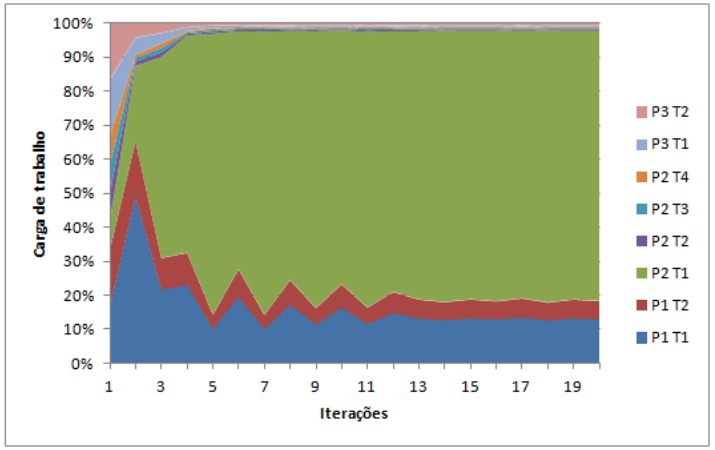
Experimentos: Resultados | RAP

- Primeiras iterações – OpenMP + MPI + CUDA









Biblioteca | Algoritmo de Balanceamento

```
int balance(int *count, int *displ, double *t, int n, int l, int r, int threshold) {
   double tmax = t[0], tmin = t[0];
   for(int i = 1; i < n; i++)
   {
      if(t[i] > tmax) tmax = t[i];
      if(t[i] < tmin) tmin = t[i];
   }
   if(100 - tmin * 100 / tmax <= threshold)
      return 0;
   double rp[n], srp = 0;
   for(int i = 0; i < n; i++)
   {
      rp[i] = count[i] / t[i];
      srp += rp[i];
   }
   int problem_size = r - l;
   int sc = 0;
   for(int i = 0; i+1 < n; i++)
   {
      count[i] = round(problem_size * (srp == 0 ? 0 : rp[i] / srp));
      sc += count[i];
   }
   count[n-1] = problem_size - sc;

   displ[0] = l;
   for(int i = 1; i < n; i++)
      displ[i] = displ[i-1] + count[i-1];
   return 1;
}
```