Arquiteturas de Computadores

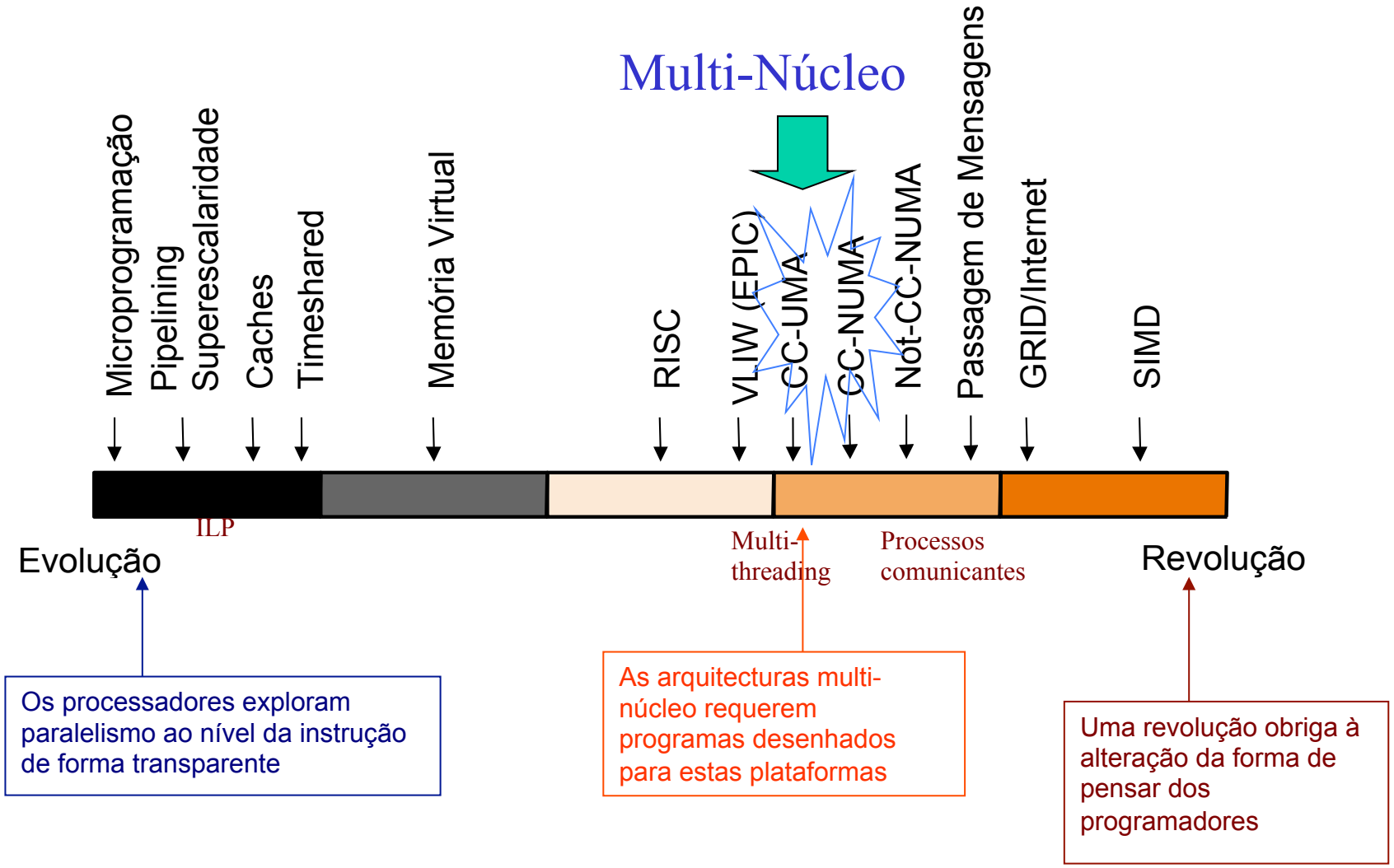
Programação de arquiteturas multi-núcleo

João Luís Ferreira Sobral
jls@...

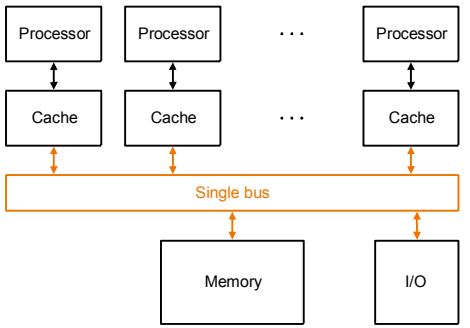
Programação de arquiteturas multi-núcleo

Conteúdos	**5 – Processamento paralelo**
	5.1 - Processadores Multi-Núcleo
Resultados de Aprendizagem	R5.2 – Identificar oportunidades de processamento paralelo R5.3 – Caracterizar as limitações inerentes ao processamento paralelo

Programação de arquiteturas multi-núcleo

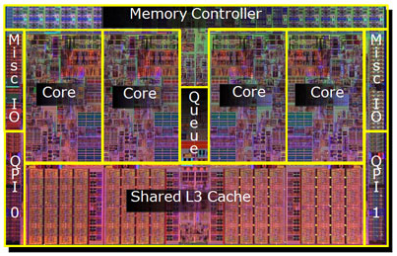


Programação de arquiteturas multi-núcleo



Conceitos

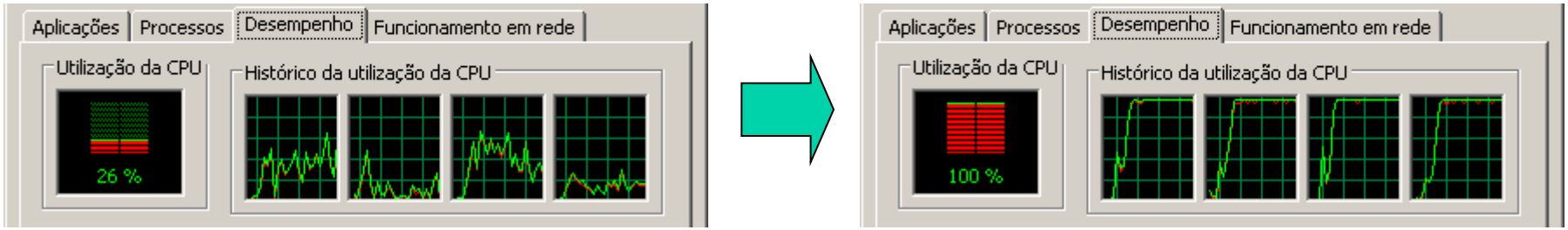
- Cada núcleo executa um fluxo de instruções
  - replica todos os recursos de um processador
    - Inclui um PC em cada núcleo
    - Não há coordenação implícita entre os núcleos
- Os vários núcleos podem executar:
  - Um mesmo fluxo de instruções (processando dados diferentes)
  - Fluxos diferentes de instruções
- Existe partilha de recursos externos:
  - cache, memória, barramento PCI, etc
- Qual o fluxo de execução/dados a executar por cada núcleo?



Os programas têm que ser escritos para tirarem partido destas arquiteturas

```
for(int i=0; i<N; i++)
   for(int j=0; j<N; j++)
      R[i][j]=G[i-1][j] + G[i+1][j] + G[i][j-1] + G[i][j+1] + G[i][j];
```

```
#pragma omp parallel for
for(int i=0; i<N; i++)
   for(int j=0; j<N; j++)
      G[i][j]=….
```



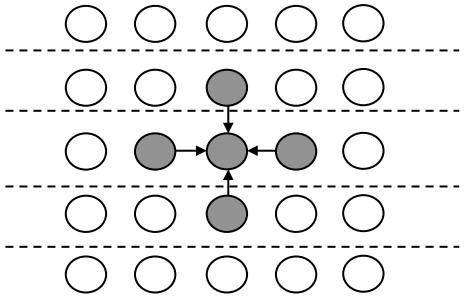
Programação de arquiteturas multi-núcleo

Desenvolver programas para processadores com vários núcleos

- **É necessário especificar tarefas que podem ser executas de forma independente**
  - **Exemplo (Método de Jacobi)**
    - **Os pontos da matriz podem calculados simultaneamente**

$$X_{i,j}^{(t+1)} = X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j+1}^{(t)}$$

```
for(int t=0; t<Niterações; t++) {
   for(int i=0; i<N-1; i++)
      for(int j=0; j<N-1; j++)
         r[i][j]=x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1] + x[i][j];
   // x = r na próxima iteração
```

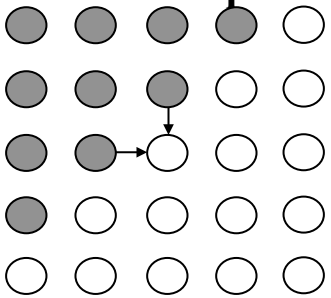


- **Devem ser preservadas as dependências entre os cálculos**
  - **Exemplo (Método de Jacobi):**
    - **não se pode calcular a iteração t+1 sem ter terminado a iteração t**

- **Existem casos em que as dependências limitam fortemente o paralelismo explorável**
  - **Exemplo (Successive Over-Relaxation)**

$$X_{i,j}^{(t+1)} = \Theta_1 X_{i,j}^{(t)} + \Theta_2 (X_{i-1,j}^{(t+1)} + X_{i,j-1}^{(t+1)} + X_{i+1,j}^{(t)} + X_{i,j+1}^{(t)})$$

valores calculados na mesma iteração



Programação de arquiteturas multi-núcleo

OpenMP (www.openMP.org)

- **"Standard" para programação de sistemas de memória partilhada**
  - **Promovido pela Intel para programação de processadores com hyper-threading**
    - **Suportado desde a versão 7.0 do compilador C++ da Intel**
  - **Suportado na versão de gcc 4.2 (e 4.1)**
- **Proporciona uma alternativa (mais simples) do que a utilização de *threads* para desenvolvimento de aplicações**
- **Usa diretivas que podem ser ignoradas pelos compiladores que não suportam o standard**
  - **Garante a compatibilidade com compiladores "legados" / arquiteturas sequenciais**
- **Exemplo:**

```
# pragma omp parallel for
for(int i=0; i<9; i++) {
    printf("%d",i);
}
```

Programação com OpenMP

- **Paradigma**

  – Baseado na especificação de ciclos ou secções de código cuja execução pode ser executada em paralelo (e.g., atividades paralelas)
    - Potencialmente cada atividade pode ser executada por um fio de execução diferente (e consequentemente executar num núcleo diferente)
    - A correção deve ser assegurada pelo programador

  – Os detalhes da criação e destruição atividades são geridos pelo compilador e sistema de execução

  – O número adequado de atividades é determinado pela biblioteca do OpenMP em função dos recursos de hardware disponíveis

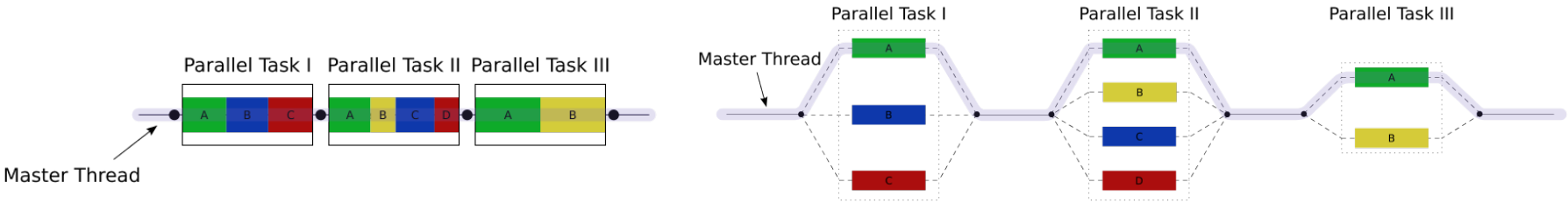
  – Exemplo – conversão de uma imagem a cores para tons de cinzento:

```
#pragma omp parallel
#pragma omp for
        for(int i=0; i<numPixels; i++) {
                grayScale[i] = 0.299*rgb[i].red + 0.587*rgb[i].green + 0.114*rgb[i].blue;
        }
```

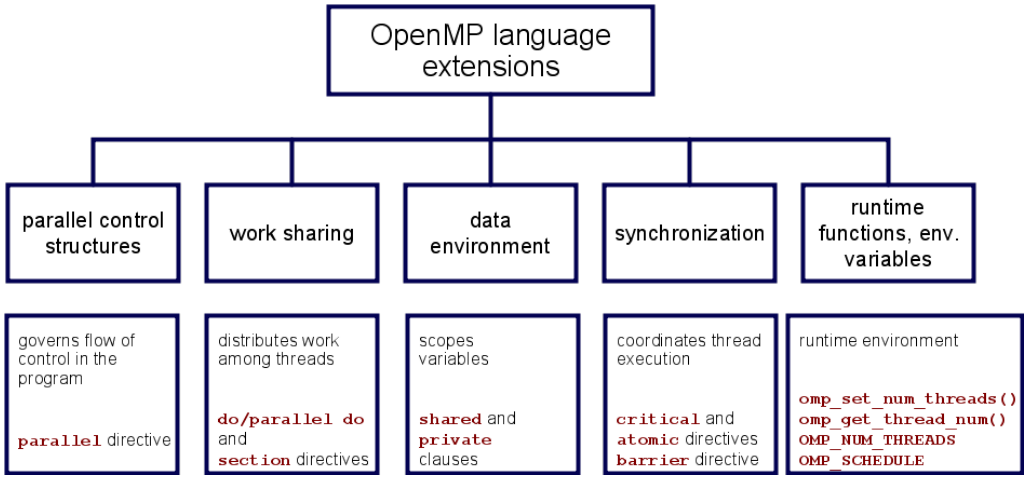
Programação com OpenMP

- **Modelo de programação/execução**

  – **A execução inicia com uma atividade principal (*master thread*)**
    - **As regiões *parallel* criam uma *equipa* de atividades paralelas**



- **Os construtores de partilha de trabalho geram tarefas para a equipa processar**
- **As cláusulas de partilha de dados indicam como partilhar variáveis nas regiões paralelas**



Programação com OpenMP

- **Directivas para criação/sincronização de tarefas**
  - **#pragma omp** *directive-name [clause[[,] clause]...] new-line*

  - **Criação de uma equipa de actividades**
    - **#pragma omp parallel**

  - **Partilha de trabalho**
    - **#pragma omp for => as iterações são distribuídas pelas actividades da equipa**
    - **#pragma omp sections => especifica tarefas heterogéneas**
    - **#pragma omp task => tarefas recursivas**

  - **Sincronização**
    - **#pragma omp master**
    - **#pragma omp single**
    - **#pragma omp ordered**
    - **#pragma omp critical**
    - **#pragma omp atomic**
    - **#pragma omp barrier**
    - **#pragam omp taskwait**
    - **#pragma omp flush**

    - **nowait (clausula que evita a sincronização no fim dos construtores)**

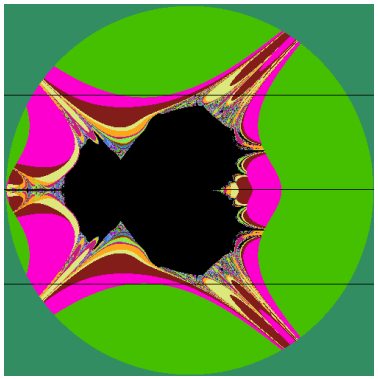
Programação com OpenMP

- **Secções de código com execução em paralelo**
  - **Permitem a especificação de tarefas heterogéneas (cada núcleo irá executar uma tarefa diferente)**

```
#pragma omp parallel
{
    #pragma omp sections
     {
            #pragma omp section {
                    taskA();
            }
            #pragma omp section {
                    taskB();
            }
            #pragma omp section {
                    taskC();
            }
    }
}
```

Programação com OpenMP

- **Desafios na paralelização (de ciclos)**



  - **Distribuição equitativa da carga computacional**

    - Problema:
      - as iterações do ciclo podem não demorar todas o mesmo tempo a executar

    - Tipos de escalonamento (dos ciclos)

      # pragma omp parallel for *schedule(kind [,chunk size])*

      - **Estático**: o número de iterações é dividido pelo número de actividades (*chunk size*)
      - **Dinâmico**: a várias iterações são colocadas numa fila de trabalho e cada fio de execução retira uma tarefa da fila de cada vez (ou o número especificado por *chunk size*)
      - **Guiado**: semelhante ao dinâmico mas o *chunk size* vai diminuindo durante a execução

    - Por defeito o escalonamento das iterações é estático
      - Adequado para situações em que todas as iterações demoram o mesmo tempo a executar
      - O escalonamento dinâmico introduz uma sobrecarga no tempo de execução

Programação com OpenMP

- **Desafios na paralelização (de ciclos)**
  - **Dependências entre tarefas**

    Dependências entre iterações do ciclo (obrigam à reescrita do ciclo):

```
for(int i = 2;  i<10; i++) {
      fact[i] = i * fact[i-1];
}
```

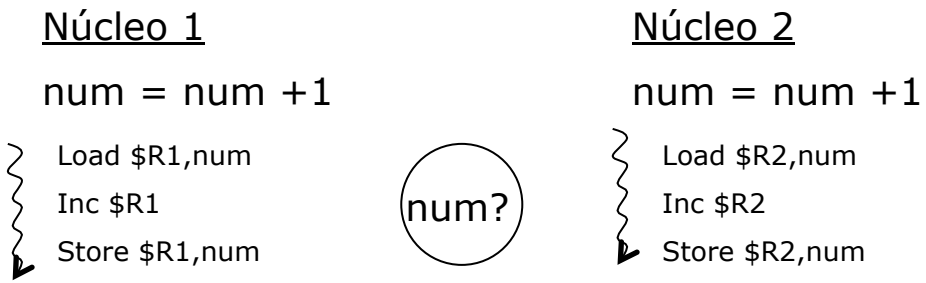
  - **As dependências limitam o paralelismo explorável**
    - **nos casos limite todo o código será executado de forma sequencial**
  - **Lei de Amdhal:**
    - Se a fração de um código executada de forma sequencial for s, o ganho máximo possível será *1/s*
      - **Exemplo: se 20% do código é executado de forma sequencial, ganho máximo possível será 5 vezes.**

Programação com OpenMP

- **Desafios na paralelização (de ciclos)**

Corridas entre fios de execução (obrigam à introdução de sincronização)

```
for(int i = 0;  i<6000; i++) {
     num++
}
```



Programação com OpenMP

- **Primitivas de sincronização**
  - **permitem restringir a ordem de execução das atividades paralelas**
  - **Acrescentar uma barreira**

```
# pragma omp parallel {
            …
   #pragma omp barrier
            …
}
```

  - **Seções críticas (executadas com exclusão mútua)**

```
# pragma omp critical
num++;
```

  - **Especificar uma operação a ser executada por um só fio de execução:**

```
# pragma omp single
```

Programação com OpenMP

- **Desafios na paralelização de ciclos**
  - **Redução de vectores de valores a um só valor:**
    - Cada atividade irá utilizar uma variável local "sum"; no final da região paralela os valores serão combinados num só (neste caso serão somados)

```
sum = 0;
# pragma omp parallel for reduction(+:sum)
      for(int i = 0;  i<100; i++) {
                  sum += array[i];
      }
```

  - **Execução de parte do ciclo pela ordem sequencial**

```
sum = 0;
# pragma omp parallel for ordered
      for(int i = 0;  i<100; i++) {
                  // ...
                  #pragma omp ordered
                  // ...
      }
```

Programação com OpenMP

- **Partilha de dados**
  - **O que acontece às variáveis nas regiões paralelas?**
    - **Variáveis declaradas na região são locais**
    - **Variáveis declaradas fora da região são partilhadas pelas atividades**
  - **Cláusulas para partilha de dados (exemplo: #pragma omp parallel private(w))**
    - **private(varlist) =>** varlist passam a privadas, o valor inicial não é especificado
    - **firstprivate(varlist) =>** idem mas inicia as variáveis com o valor fora da região
    - **lastprivate(varlist) =>** idem mas o valor final é o da ultima iteração dos ciclos (só aplicável a "for")
    - **reduction(op:var) =>** idem mas o valor final é o resultante da aplicação de *op* aos valores privados
  - **Diretivas para partilha de dados ao nível das *threads***
    - **#pragma omp threadlocal** => cada atividade (*thread*) possui uma cópia do valor.
    - **A clausula *copyin* pode ser utilizada para copiar o valor da atividade (*thread*) principal as outros**

Programação com OpenMP

- **Partilha de dados**
  - **Exemplo (implementação do *reduce(*:mul*))**

```
double gmul=1.0L;
double lmul=1.0L;

#pragma omp parallel shared(gmul), firstprivate(lmul)
{

    #pragma omp for
    for(int i=0; i<1000000;i++) {
        lmul *= sin(i);          // mul local
    }

    #pragma omp atomic
    gmul *= lmul;                // mul partilhada

}
```