@tópicos-em-arquiteturas-paralelas

Mestrado em Informática - UFPR

Diogo Cezar Teixera Batista
Daniel Weingaertner

Curitiba

12 de maio de 2010

1 Informações da Disciplina

- Nome: *Tópicos em Arquiteturas Paralelas*;
- Código: *CI 805*;
- Professor: *Daniel Weingaertner*;
- Horários: *3^a das 13:30 as 15:10 / 6^a das 13:30 as 15:10*;
- Método de Avaliação:
  - 1 prova + 1 trabalho: média = $(P + 2T) \div 3$
- Trabalho:
  - Trabalho individual;
  - Implementação computacional;
  - Artigo de 6 páginas, inglês (preferencialmente) ou português.
  - Apresentação de 20min em inglês ou português.
- Página da disciplina: http://www.inf.ufpr.br/danielw/grad/ci314/
- Datas:
  - Prova: 22 de junho;
  - Trabalho: 01 de junho;
  - Prova Final e Segunda Chamada: 06 de julho.

2 Multiprocessadores e Paralelismo a Nível de Thread

- Por que computadores paralelos vieram para ficar?
  - Limitação no tamanho dos componentes → latência, alimentação e dissipação;
  - Maior problema é dissipação.
- Limitação do paralelismo de instrução (ILP)
  - ILP: divisão em sub-unidades funcionais independentes;
  - Maior trabalho feito pelos compiladores: reordenamento de instruções;
  - Referência: descrição e problemas → Patterson.
- Desde 1980 computadores paralelos de memória compartilhada
  - Multithreading (Intel Hyperthread) permite execução intercalada de diversas threads, mantendo o pipeline de instruções cheio;
  - Multicore implementa processadores completos.
- Compiladores terão de aprender a lidar com essa realidade, mas o programador terá que descrever os pontos de concorrência da sua aplicação;
- Em 2005 Intel parou de desenvolver ILP para dedicar-se ao multicore, seguindo IBM e Sun;
- Fatores que impulsionam multiprocessamento:
  - Interesse crescente em performance de servidores (mainframe!?);
  - Programas que usam quantidades massivas de dados;
  - Compreensão de como usar multiprocessamento com efetividade, especialmente em servidores, que possuem paralelismo de threads intrínseco;
  - Redução de custo de design: replicação de um core mais simples.

3 Taxonomia de Arquiteturas Paralelas de Memória Compartilhada

- Single Instruction stream Single Data stream (SISD): uniprocessador;
- Single Instruction stream Multiple Data streams (SIMD):
  - Mesma instrução executada por multiplos processadores usando dados diferentes;
  - Exploram paralelismo de dados;
  - Cada processador tem seus próprios dados, mas há apenas uma memória de instruções;

  - Operações sobre multimídia (CUDA, SSE).

- Multiple Instruction streams Multiple Data streams (MIMD):
  - Cada processador tem seus próprios dados e instruções;
  - Explora o paralelismo em nível de Thread, uma vez que elas operam simultaneamente;
  - Mais flexível do que SIMD e por isso mais genéricamente aplicável;
  - Foco do início da disciplina (OpenMP).

3.1 Processadores MIMD

- São flexíveis: podem ser otimizados como multiprocessadores single-user para alta performance, multiprogramados, com diversas threads, ou uma combinação;
- Vantajosos do ponto de vista de custo pois podem ser construídos a partir da combinação de processadores existentes;
- Cada processador executa seu próprio conjunto de instruções. Eventualmente processos distintos ou threads distintas;
  - Diferenciar processos de threads;
  - Conceito de thread para o processador pode significar qualquer um dos dois;
- O programador precisa aprender a aproveitar o paralelismo a nível de threads;
  - Quantas, quando iniciar, quando sincronizar;
  - Também pode ser utilizado para explorar paralelismo a nível de dados, entretanto há um overhead que pode tornar proibitivo (comparado a SIMD);

3.2 Divisão dos MIMD quanto ao tipo de memória

- Arquiteturas de memória compartilhada centralizada;
  - Symmetric (shared-memory) multiprocessors (SMPs);
  - Uniform Memory Access (UMA): acesso à mesma latência;
  - Para atender à demanda de memória com poucos cores: cache grande, uma memória em múltiplos bancos;
  - Aumentando o número de cores (algumas dezenas): aumentar número de bancos de memória, usar conexões p2p ou switch;
  - É a arquitetura mais utilizada.
- Arquiteturas de memória distribuída;

  - Permite maior número de processadores. No modelo centralizado o barramento da memória não suportaria a demanda. Alta latência;
  - Quanto maior a velocidade dos processadores, menor o número que pode compartilhar barramento;
  - Necessita de uma interconexão de alta velocidade, tipicamente switches ou redes mesh multidimensionais;
  - Vantagens: baixo custo para alta banda de acesso, uma vez que a maioria dos acessos ocorre na mem. local. Redução da latência.
  - Desvantagem: comunicação entre processos mais complexa e aproveitamento da maior banda de memória exige esforço do software PROGRAMADOR;

3.3 Modelos de Comunicação e Arquitetura de Memória

Como compartilhar dados em sistemas de memória distribuída.

- Compartilhamento do espaço de endereçamento (Distributed Shared-Memory DSM);
  - Todos processadores tem acesso ao mesmo espaço de endereçamento, embora os espaços estejam fisicamente em locais diferentes;
  - Shared-Memory refere-se ao espaço de endereçamento, não a uma única memória;
  - Non-Uniform Memory Access (NUMA): tempo de acesso depende da distância entre banco de memória e processador;
  - Compartilhamento de dados é transparente/implícito através de operações de load/store;
- Espaços de endereçamento distintos;
  - Cada processador tem seu espaço em sua memória;
  - Comunicação deve ser feita explicitamente através de mecanismo;
  - Exemplo: Clusters;
- Memória Cache em Processadores SMP;
  - Não é compartilhada;
  - Problema da "inconsistência de memória";
  - Necessidade de mecanismo de "coerência de cache"
    * Fora do controle do programador, mas este deve saber fazer bom uso;
    * Exemplo 1: Snooping: cada processador tem o bloco e monitora o barramento;
    * Exemplo 2: Directory: status de cada bloco é mantido centralizado;
  - Infos em Patterson;

  - Sistemas que implementam coerência de forma transparente são chamados de "cache coherent";
  - OpenMP abstrai implementações físicas pois tem seu próprio modelo de memória, com dados privados e compartilhados (private, shared), e especifica quando há garantia de que um dado compartilhado está disponível.

- Programando SMPs;
  - Compiladores são projetados para fazer o melhor uso do ILP, mas não funciona para multicore, pois é difícil definir trechos de código/dados independentes e o compilador também não pode mudar o algoritmo para deixálo paralelizável;

4 OpenMP

- O que é OpenMP;
  - Interface de programação (API) para aplicações de memória compartilhada que facilita a programação paralela;
  - Não é uma linguagem de programação: extensão da linguagem C/C++, FORTRAN;
  - Possui diretivas que indicam como o trabalho será dividido entre threads e a ordem de acesso aos dados compartilhados;
  - Tornou-se um padrão de fato:
    * Ênfase em programação estruturada;
    * Simplicidade de uso;
    * Permite paralelização incremental de código existente;
    * É apoiado pelos principais fabricantes de SMPs.

- Como programar com OpenMP;
  - As "diretivas"OpenMP dizem ao compilador quais instruções devem ser executadas em paralelo e como distribuí-las ao longo das threads;
  - Como transformar um programa sequencial em um programa paralelo:
    * 1° passo: Identificar o paralelismo. Pode requerer reorganização do código;
    * 2° passo: Implementar no código o paralelismo identificado;
    * Entretanto: ganhos significativos de performance geralmente demandam que o programador "suje as mãos"e desenvolva algoritmos paralelos;

Comparação de diferentes APIs

Código 1: Comparação de diferentes APIs

```
1  /* Comparação entre implementações Sequencial, MPI, PThreads e OpenMP
2     do produto escalar entre dois vetores
3
4     FONTE: Chapman, pg.17-21
5
6  */
7
8
9  /*  Sequential Dot-Product
10     The sequential program multiplies the individual elements of two arrays
11     and saves the result in the variable sum; sum is a so-called reduction
12     variable.
13 */
14
15 int main(argc,argv)
16 int argc;
17 char *argv[];
18 {
19   double sum;
20   double a [256], b [256];
21   int n, i;
22   n = 256;
23   for (i = 0; i < n; i++) {
24      a [i] = i * 0.5;
25      b [i] = i * 2.0;
26   }
27   sum = 0;
28   for (i = 0; i < n; i++ ) {
29      sum = sum + a[i]*b[i];
30   }
31   printf ("sum = %f", sum);
32 }
33
34
35 /* Dot-Product in MPI
36    Under MPI, all data is local. To implement the dot-product, each process
37    builds a partial sum, the sum of its local data. To do so, each executes
38    a portion of the original loop. Data and loop iterations are accordingly
39    manually shared among processors by the programmer. In a subsequent step,
40    the partial sums have to be communicated and combined to obtain the global
41    result. MPI provides the global communication routine MPI_Allreduce for
42    this purpose.
43 */
44 int main(argc,argv)
```

```
45 int argc;
46 char *argv[];
47 {
48   double sum, sum_local;
49   double a [256], b [256];
50   int i, n, numprocs, myid, my_first, my_last;
51   n = 256;
52   MPI_Init(&argc,&argv);
53   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
54   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
55   my_first = myid * n/numprocs;
56   my_last = (myid + 1) * n/numprocs;
57 /*  for (i = my_first; i < my_last; i++) { */
58   for (i = 0; i < n; i++) {
59      a [i] = i * 0.5;
60      b [i] = i * 2.0;
61   }
62   sum_local = 0;
63   for (i = my_first; i < my_last; i++) {
64         sum_local = sum_local + a[i]*b[i];
65   }
66   MPI_Allreduce(&sum_local, &sum, 1, MPI_DOUBLE, MPI_SUM,
67                                                  MPI_COMM_WORLD);
68   if (myid==0) printf ("sum = %f", sum);
69 }
70
71
72 /* Dot-Product in Pthreads
73    In the Pthreads programming API, all data is shared but logically distributed
74    among the threads. Access to globally shared data needs to be explicitly
75    synchronized by the user. In the dot-product implementation shown, each
76    thread builds a partial sum and then adds its contribution to the global sum.
77    Access to the global sum is protected by a lock so that only one thread at a
78    time updates this variable. We note that the implementation effort in
79    Pthreads is as high as, if not higher than, in MPI.
80 */
81
82 #define NUMTHRDS 4
83 double sum;
84 double a [256], b [256];
85 int status;
86 int n=256;
87 pthread_t thds[NUMTHRDS];
88 pthread_mutex_t mutexsum;
89
90 int main(argc,argv)
91 int argc;
```

```
92  char *argv[];
93  {
94    int i;
95    pthread_attr_t attr;
96    for (i = 0; i < n; i++) {
97        a [i] = i * 0.5;
98        b [i] = i * 2.0;
99    }
100   pthread_mutex_init(&mutexsum, NULL);
101   pthread_attr_init(&attr);
102   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
103   for(i=0;i<NUMTHRDS;i++)
104   {
105      pthread_create( &thds[i], &attr, dotprod, (void *)i);
106   }
107   pthread_attr_destroy(&attr);
108   for(i=0;i<NUMTHRDS;i++) {
109      pthread_join( thds[i], (void **)&status);
110   }
111   printf ("sum = %f \n", sum);
112   pthread_mutex_destroy(&mutexsum);
113   pthread_exit(NULL);
114 }
115
116 void *dotprod(void *arg)
117 {
118     int myid, i, my_first, my_last;
119     double sum_local;
120     myid = (int)arg;
121     my_first = myid * n/NUMTHRDS;
122     my_last = (myid + 1) * n/NUMTHRDS;
123     sum_local = 0;
124     for (i = my_first; i < my_last; i++) {
125       sum_local = sum_local + a [i]*b[i];
126     }
127     pthread_mutex_lock (&mutexsum);
128     sum = sum + sum_local;
129     pthread_mutex_unlock (&mutexsum);
130     pthread_exit((void*) 0);
131 }
132
133
134 /*  Dot-Product in OpenMP
135     Under OpenMP, all data is shared by default. In this case, we are able to
136     parallelize the loop simply by inserting a directive that tells the compiler
137     to parallelize it, and identifying sum as a reduction variable. The details
138     of assigning loop iterations to threads, having the different threads build
```

```
139     partial sums and their accumulation into a global sum are left to the
140     compiler. Since (apart from the usual variable declarations and
141     initializations) nothing else needs to be specified by the programmer,
142     this code fragment illustrates the simplicity that is possible with OpenMP.
143 */
144 int main(argc,argv)
145 int argc; char *argv[];
146 {
147   double sum;
148   double a [256], b [256];
149   int status;
150   int n=256;
151   for (i = 0; i < n; i++) {
152       a [i] = i * 0.5;
153       b [i] = i * 2.0;
154   }
155   sum = 0;
156   #pragma omp parallel for reduction(+:sum)
157   for (i = 1; i <= n; i++ ) {
158       sum = sum + a[i]*b[i];
159   }
160   printf ("sum = %f \n", sum);
161 }
```

- A idéia básica de OpenMP;
  - Modelo de execução;
  - Uma thread é uma entidade de tempo de execução capaz de executar uma sequencia de instruções de maneira independente:
    - compartilham espaço de endereçamento do processo;
    - possuem área privada de memória (registradores e pilha);
    - Program Counter individual;
    - Podem ser executadas concorrentemente num único processador (troca de contexto → multithreading)
- Modelo Fork and Join;
  - Ruud, slides 13-15;
  - Quando um bloco "parallel"é encontrado por uma thread, ela cria um novo time de threads (FORK) e torna-se master do time;
  - Ao final da execução do bloco, apenas a thread master continua;
- OpenMP possibilita:

  - Criação de time de threads para execução paralela;
  - Especificação de como dividir o trabalho entre as threads;
  - Declaração de variáveis privadas e compartilhadas;
  - Sincronização de threads e realização de operações exclusivas;

- Criação de threads
  - Ocorre ao encontrar um bloco "parallel";
  - Ruud, slides 14-15;
  - Fim do bloco implica em barreira para todas as threads;
  - Blocos encadeados de "parallel": cada thread vira master;

- Divisão de trabalho entre threads
  - Apenas a criação do bloco paralelo não divide o trabalho, apenas faz com que todas threads executem mesma tarefa;
  - Divisão de um laço FOR: mais comum;
    * Atribui uma ou mais iterações a cada thread;
    * Estratégia mais direta: atribui "chunks"consecutivos a cada thread *** diferenciar thread, chunk ***;
  - Para laço ser paralelizável:
    * Atribui uma ou mais iterações a cada thread;
    * Estratégia mais direta: atribui "chunks"consecutivos a cada thread *** diferenciar thread, chunk ***;
  - Blocos encadeados de "parallel": cada thread vira master;

- Para laço ser paralelizável:
  - Número de iterações deve ser conhecido antes e não pode mudar;
  - Iterações devem ser independentes;
  - Dependência de dados impede paralelismo quando o valor que é escrito em uma iteração é lido ou sobrescrito em outra;

- Divisão por pedaços de código;

- Pode-se especificar que numa região paralela apenas uma thread execute um trecho de código.

Modelo de memória

- Ruud, slide 11-12;
- Dados privados ou compartilhados para uma determinada região;
  * dados privados são mais rápidos, evitam lock e ajudam em cc-NUMA **
- Tamanho da pilha pode ser insuficiente;
- Dados compartilhados são coerentes em determinados pontos de sincronização, ou seja, temporariamente podem ser != (cache) Ruud, slide 18;

Sincronização de Threads

- Final de região paralela: sincronização implícita;
- Apenas uma thread executa determinado código;
  - Enquanto esperam, threads podem executar outros trechos;
- Atualização atômica de variáveis em mem. compartilhada;
- Sincronização de subconjunto de threads: não suportada, manual;
- operação flush sincroniza dado compartilhado;
- Evitar dead-lock e acesso simultâneo a dados compartilhados (data race) é função do programador;
  - Data race é difícil de detectar, pois pode não ser reprodutível: depende da ordem de execução das threads;
- Verificar se número de threads é o esperado, pois limitação nos recursos pode diminuir o número e quebrar o código.

Outras características

- Controle do número de threads: via var. de ambiente, durante execução ou antes de entrar na região paralela;
  - permite variação dinâmica do núm. de threads;
  - uma vez iniciada uma região paralela, num de threads não muda.
- Atualização atômica de variáveis em mem. compartilhada;
- Sincronização de subconjunto de threads: não suportada, manual;
- operação flush sincroniza dado compartilhado;
- Evitar dead-lock e acesso simultâneo a dados compartilhados (data race) é função do programador;

  - Data race é difícil de detectar, pois pode não ser reprodutível: depende da ordem de execução das threads;

- Verificar se número de threads é o esperado, pois limitação nos recursos pode diminuir o número e quebrar o código.

5 Escrevendo código em OpenMP

Sintaxe

- Diretivas iniciam com '#pragma omp';
- case sensitive;
- aceitam espaços entre as palavras;
- quebras de linha devem conter '\';
- Não há mensagem nem warning para erros de grafia (Utilizar '-Wall');

Código 2: Comparação de diferentes APIs

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5
6  void mxv(int m, int n, double * restrict a,
7           double * restrict b, double * restrict c);
8
9  int main(int argc, char *argv[])
10 {
11    double *a,*b,*c;
12    int i, j, m, n;
13    printf("Please give m and n: ");
14    scanf("%d %d",&m,&n);
15    if ( (a=(double *)malloc(m*sizeof(double))) == NULL )
16       perror("memory allocation for a");
17    if ( (b=(double *)malloc(m*n*sizeof(double))) == NULL )
18       perror("memory allocation for b");
19    if ( (c=(double *)malloc(n*sizeof(double))) == NULL )
20       perror("memory allocation for c");
21    printf("Initializing matrix B and vector c\n");
22    for (j=0; j<n; j++)
23       c[j] = 2.0;
24    for (i=0; i<m; i++)
```

```
25        for (j=0; j<n; j++)
26           b[i*n+j] = i;
27     printf("Executing mxv function for m = %d n = %d\n",m,n);
28 #ifdef _OPENMP
29     printf("Using OpenMP: %d\n",_OPENMP);
30 #endif
31     (void) mxv(m, n, a, b, c);
32     free(a);free(b);free(c);
33     return(0);
34 }
35 
36 #ifndef _OPENMP
37 void mxv(int m, int n, double * restrict a,
38           double * restrict b, double * restrict c)
39 {
40     int i, j;
41     for (i=0; i<m; i++)
42     {
43        a[i] = 0.0;
44        for (j=0; j<n; j++)
45           a[i] += b[i*n+j]*c[j];
46     }
47 }
48 
49 #else
50 
51 void mxv(int m, int n, double * restrict a,
52           double * restrict b, double * restrict c)
53 {
54     int i, j;
55 #pragma omp parallel for default(none) \
56          shared(m,n,a,b,c) private(i,j)
57     for (i=0; i<m; i++)
58     {
59        a[i] = 0.0;
60        for (j=0; j<n; j++)
61           a[i] += b[i*n+j]*c[j];
62     } /*-- End of omp parallel for --*/
63 }
64 #endif
```

Análise do código de aula04.c

- palavra-chave 'restrict': C99;
- garante acesso apenas através deste ponteiro para aquela região de memória;

- permite otimizações por parte do compilador;

Compilando código

- gcc -o ¡programa¿ -fopenmp -std=gnu99 ¡arq.c¿;

Principais Diretivas de OpenMP

- Construção 'parallel';
- Construções de divisão de trabalho;
  - loop;
  - section;
  - single;
- Cláusulas de compartilhamento de dados, 'no wait' e escalonamento;

5.1 Construção Parallel

```
#pragma omp parallel [clause, clause, ...]
bloco estruturado}
```

- A partir desta construção é criado o time de threads;
- Inicia a execução paralela, mas não distribui o trabalho da região;
- Entre as threads. Cabe ao programador definir a divisão de trabalho;
- A thread que encontra uma região 'parallel' vira master do novo time;
- O ID dos threads varia de 0 (master) a n-1

Código 3: Construção Parallel

```
1 #pragma omp parallel
2 {
3         printf("The parallel region is executed by thread %d\n",
4         omp_get_thread_num());
5         if ( omp_get_thread_num() == 2 ) {
6                 printf(" Thread %d does things differently\n",
7         omp_get_thread_num());
8 }
9 }
```

Cláusulas suportadas

- if(scalar-expression)
- num threads(integer-expression)
- private(list)
- firstprivate(list)
- shared(list)
- default(none—shared)
- copyin(list)
- reduction(operator:list)

Restrições

- Não pode haver saltos para dentro ou para fora de uma região paralela
- O programa não pode depender da ordem das cláusulas
- Apenas uma cláusula 'if'
- Apenas uma cláusula 'num_threads'

Construções de Divisão de Trabalho

- dividem a computação entre as threads
- devem estar em uma região paralela ativa, senão são ignoradas (ex. Função)
- Regras:
  - Cada construção deve ser atingida por TODAS as threads ou por NENHUMA
  - A sequência de regiões de divisão de trabalho e barreiras deve ser a mesma para todas as threads de um time.
- Não constrói threads e não tem barreira de entrada. Por default barreira na saída.

Construção Loop

```
#pragma omp for [clause, clause, ...]
laço for
```

- Limitado a laços com número de execuções conhecido, construção com apenas uma variável de controle:

```
for (init-expr; var relop b; incr-expr)
```

- Cláusulas suportadas:
  - private(list)
  - firstprivate(list)
  - lastprivate(list)
  - ordered
  - schedule(kind[,chunk_size])
  - reduction(operator:list)
  - nowait
- Dois laços consecutivos: todos esperam na barreira (excessão:nowait)

Construção Sections

Código 4: Construção Section

```
#pragma omp sections [clause, clause, ...]
    {
    [#pragma omp section ]
        structured block
    [#pragma omp section
        structured block ]
        ...
}
```

- Cada thread executa um trecho diferente de código;
- Cada seção deve ter um bloco de código independete das outras;
- Cada bloco é executado apenas uma vez, e cada thread executa apenas um bloco por vez;
- Não há ordem de execução;
- Problemas:
  - Mais threads que blocos;
  - Balancemento de carga;
- Dois laços consecutivos: todos esperam na barreira (excessão:nowait)

Código 5: Exemplo de utilização das sections

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            (void) funcA();
        #pragma omp section
            (void) funcB();
    }
}
```

Construção Single

```
#pragma omp single [clause, ...]
structured block
```

Código 6: Construção Single

```
#pragma omp parallel shared(a,b) private(i)
{
    #pragma omp single
    {
        a = 10;
        printf("Single construct executed by thread %d\n",
        omp_get_thread_num());
    }
    /* A barrier is automatically inserted here */
    #pragma omp for
        for (i=0; i<n; i++)
        b[i] = a;
}
printf("After the parallel region:\n");
for (i=0; i<n; i++)
    printf("b[%d] = %d\n",i,b[i]);
```

- Possui barreira automática;
- Por que não deixar todas as threads escreverem o valor da variável?
  - Escrita não é atômica: resultado imprevisível;
  - Problema de performance;

Possível combinar 'parallel' com 'for' ou 'section'

- Cláusulas de controle das construções de Divisão de Trabalho:
  - São processadas ANTES de entrar na região paralela. São externas;
  - *shared*
    - Cuidar com uso simultâneo;
    - Cuidar com tamanho do cache;
  - *private*
    - Valor indefinido na entrada e depois da saída da região paralela;
  - *lastprivate*
    - Última thread na sequencia atualiza variável fora do bloco:
      - No caso de laço: último pedaço (thread depende do schedule);
      - No caso de section: última a aparecer no código;
      - Exemplo no código 12;
      - Poderia ser substituído por um if e uma variável shared;
      - Há aumento de custo em qualquer caso para determinar qual thread deve efetuar a cópia. (schedule)
    - *firstprivate*
      - inicializa o valor de todos elementos da thread;
      - geralmente, variáveis read-only podem ser shared;
      - no caso de cc-NUMA, melhor firstprivate!;
    - *default*
      - define o padrão a ser utilizado;
    - *nowait*
      - Retira a barreira ao final de uma região de divisão de trabalho
      - Barreiras ao fim de regiões paralelas não podem ser removidas
      - CUIDADO: race conditions!

Código 7: Construção Single

```
1 #pragma omp parallel for private(i) lastprivate(a)
2 for (i=0; i<n; i++)
3 {
4     a = i+1;
5     printf("Thread %d has a value of a = %d for i = %d\n",
6     omp_get_thread_num(),a,i);
7 } /*-- End of parallel for --*/
8 printf("Value of a after parallel for: a = %d\n",a);
```

EXERCÍCIO: Cada thread em uma região paralela precisa acessar uma seção específica de um vetor, mas o acesso inicia a partir de um offset (!= 0). Seja 'a' o vetor, 'indx' o offset, 'n' o tamanho do chunk

Código 8: Exercício

```
#pragma omp parallel    default(none) \
                        private(i,TID,indx) \
                        shared(n,offset,a)
{
    TID = omp_get_thread_num();
    indx = offset + n*TID;
    for(i=indx; i<indx+n; i++)
    a[i] = TID + 1;
} /*-- End of parallel region --*/
```

5.2 Cláusulas de sincronismo

- `void omp_FUNC_lock(omp_lock_t *lck);`
- Similar a semáforos;
- FUNC pode ser: init, destroy, set, unset, test;
- Modo de uso:
  1. Declarar variáveis de lock;
  2. Inicializar o lock com omp_init_lock;
  3. Atribuir o lock com omp_set_lock ou omp_test_lock;
  4. Liberar lock após trabalho completo com omp_unset_lock.
- Cuidado com dead-lock;

Código 9: Utilização do Lock

```
#include <stdio.h>
#include <omp.h>

void work(int);
void skip(int);

int main() {
   omp_lock_t lck;
   int id;

```

```
11    omp_init_lock(&lck);
12    #pragma omp parallel shared(lck) private(id)
13    {
14       id = omp_get_thread_num();
15
16       omp_set_lock(&lck);
17       printf_s("My thread id is %d.\n", id);
18
19       // only one thread at a time can execute this printf
20       omp_unset_lock(&lck);
21
22       while (! omp_test_lock(&lck)) {
23          skip(id);   // we do not yet have the lock,
24                      // so we must do something else
25       }
26       work(id);      // we now have the lock
27                      // and can do the work
28       omp_unset_lock(&lck);
29    }
30    omp_destroy_lock(&lck);
31 }
```

5.3 Master

```
#pragma omp master
{ bloco estruturado }
```

- Garante execução pela thread master, em contraposição a single, em que qualquer uma pode executar;
- Não há barreira implícita de sincronismo.

5.4 Controle do ambiente de execução

Segue-se a seguinte prioridade para definição do número de threads:

```
OMP_NUM_THREADS < omp_set_num_threads() < threads() in Parallel
```

Código 10: Definindo número de threads

```
1 (void) omp_set_num_threads(4);
2 #pragma omp parallel if (n > 5) num_threads(n) default(none) private(TID) shared(n)
3 {
```

```
4       TID = omp_get_thread_num();
5       #pragma omp single
6       {
7       printf("Value of n = %d\n",n);
8       printf("Number of threads in parallel region: %d\n",
9           omp_get_num_threads());
10      }
11      printf("Print statement executed by thread %d\n",TID);
12  }l region --*/
```

Comandos:

- omp_get_num_threads;
- omp_get_thread_num;
- omp_get_num_procs;
- omp_in_parallel.

5.5 Cláusula Reduce

- reduction(operator:list);

Código 11: Exemplo Redução

```
1  // omp_reduction.cpp
2  // compile with: /openmp
3  #include <stdio.h>
4  #include <omp.h>
5
6  #define NUM_THREADS 4
7  #define SUM_START   1
8  #define SUM_END     10
9  #define FUNC_RETS   {1, 1, 1, 1, 0}
10
11 int bRets[5] = FUNC_RETS;
12 int nSumCalc = ((SUM_START + SUM_END) * (SUM_END - SUM_START + 1)) / 2;
13
14 int func1( ) {return bRets[0];}
15 int func2( ) {return bRets[1];}
16 int func3( ) {return bRets[2];}
17 int func4( ) {return bRets[3];}
18 int func5( ) {return bRets[4];}
19
20 int main( )
21 {
```

```
22      int nRet = 0,
23          nCount = 0,
24          nSum = 0,
25          i,
26          bSucceed = 1;
27
28      omp_set_num_threads(NUM_THREADS);
29
30      #pragma omp parallel reduction(+ : nCount)
31      {
32          nCount = nCount + 1;
33  //        printf("nCount = %d para TID = %d\n", nCount, omp_get_thread_num());
34
35          #pragma omp for schedule(static, 2) reduction(+ : nSum)
36          for (i = SUM_START ; i <= SUM_END ; ++i)
37          {
38              nSum += i;
39              printf("nSum = %d para TID = %d e i=%d\n", nSum, omp_get_thread_num(), i);
40          }
41
42          #pragma omp sections reduction(&& : bSucceed)
43          {
44              #pragma omp section
45              {
46                  bSucceed = bSucceed && func1( );
47              }
48
49              #pragma omp section
50              {
51                  bSucceed = bSucceed && func2( );
52              }
53
54              #pragma omp section
55              {
56                  bSucceed = bSucceed && func3( );
57              }
58
59              #pragma omp section
60              {
61                  bSucceed = bSucceed && func4( );
62              }
63
64              #pragma omp section
65              {
66                  bSucceed = bSucceed && func5( );
67              }
68          }
```

```
69      }
70
71      printf("The parallel section was executed %d times "
72              "in parallel.\n", nCount);
73      printf("The sum of the consecutive integers from "
74              "%d to %d, is %d\n", 1, 10, nSum);
75
76      if (bSucceed)
77          printf("All of the the functions, func1 through "
78                  "func5 succeeded!\n");
79      else
80          printf("One or more of the the functions, func1 "
81                  "through func5 failed!\n");
82
83      if (nCount != NUM_THREADS)
84      {
85          printf("ERROR: For %d threads, %d were counted!\n",
86                  NUM_THREADS, nCount);
87          nRet |= 0x1;
88     }
89
90      if (nSum != nSumCalc)
91      {
92          printf("ERROR: The sum of %d through %d should be %d, "
93                  "but %d was reported!\n",
94                  SUM_START, SUM_END, nSumCalc, nSum);
95          nRet |= 0x10;
96      }
97
98      if (bSucceed != (bRets[0] && bRets[1] &&
99                       bRets[2] && bRets[3] && bRets[4]))
100     {
101         printf("ERROR: The sum of %d through %d should be %d, "
102                 "but %d was reported!\n",
103                 SUM_START, SUM_END, nSumCalc, nSum);
104         nRet |= 0x100;
105     }
106 }
```

5.6 Paralelismo Encadeado

No paralelismo encadeado cada thread que irá formar um novo time com n novas threads assume a nova sequencia como master, ou seja, seu novo $ID = -$.

Código 12: Exemplo Paralelismo Encadeado

```
1  printf("Nested parallelism is %s\n", omp_get_nested() ? "supported" : "not supported");
2  #pragma omp parallel
3  {
4      printf("Thread %d executes the outer parallel region\n",
5      omp_get_thread_num());
6      #pragma omp parallel num_threads(2)
7      {
8          printf(" Thread %d executes inner parallel region\n",
9          omp_get_thread_num());
10     }
11 }
```

- Cuidado com omp_get_thread_num(): retorna 0–N-1, onde N é o time;

5.7 Flush

```
#pragma omp flush [lista]
```

- Modelo de consistência relaxada: permite valores temporários para cada thread;
- Garantia de valor único somente em pontos de sincronismo;
- Diretiva flush atualiza memória global (lista) com valores da thread;
- Se T modificou var, valor vai p/ memória global. Senão, var é atualizado com valor da mem. global;
- FLUSH implicitos em: barreiras, I/O de regiões críticas e lock.

6 Introdução ao modelo de Programação CUDA

CUDA é um modelo de programação paralela escalável e um ambiente de software para a computação paralela.

- Acesso ao Hardware das GPUs através de extensões mínimas da linguagem C/C++;
- Programação Heterogênea (CPU + GPU);

6.1 Modelo de programação CUDA (Execução + Memória)

CUDA foi desenvolvido para centenas de núcleos de processamento e milhares de threads paralelas.

Definições:

- Kernel = função que executa na GPU;
- Device = GPU;
- Host = CPU;

Porções paralelas de uma aplicação executam na GPU como Kernels

- Imagem (Guia de Programação, pág 21);
- Um Kernel por vez;
- Muitas threads por Kernel;

Diferenças entre threads CUDA e CPU

- Threads CUDA são extremamente leves;
- CUDA usa de milhares de threads para executar de forma eficiente, enquanto as CPUs multicore podem utilizar apenas algumas threads;
- Um Kernel CUDA é executado por um array multidimensional de threads:
  - Exemplo (Getting Started witk CUDA, pág 6);
  - Cada thread possui uma identificação que é usada para computar endereços de memória e tomar decisões de controle;
- Cooperação de threads:
  - Compartilhar resultados evita computação desnecessária;
  - Compartilhar acessos a memória reduz drasticamente a nececessidade de acessos a memória global;

- Hierarquia/Organização de Threads:
  - Imagem (Getting Started with CUDA, pág 8);
  - Um Kernel lança um grid de blocos de threads;
  - Threads em um bloco cooperam memória compartilhada;
  - Threads em um bloco podem sincronizar;
- Escalabilidade em CUDA:
  - Imagem (Guia de Programação, pág 78);
  - CUDA permite que programas escalem transparentemente em diferentes GPUs;
  - O HW é livre para escalonar os blocos de threads em qualquer;
  - Dispositivos podem executar menos ou mais blocos por vez dependendo da arquitetura;
- Arquitetura das GPUs:
  - Série 8 (G80):
    - Imagem (Getting Started with CUDA, pág 10);
    - 128 processadores (executam threads);
    - 16 multiprocessadores com 8 processadores e 1 memória compartilhada cada;
  - Série 10:
    - Imagem (Getting Started with CUDA, pág 11);
    - 240 processadores (executam threads);
    - 30 multiprocessadores com 8 processadores, 1 memória compartilhada e 1 unidade de dupla precisão cada;
- Modelo de Memória:
  - Imagem (Getting Started with CUDA, pág 12);
- Disposição Física da Memória:
  - Imagem (Guia de Programação, pág 80);
  - A memória Local o é no sentido de acesso de cada thread, na verdade é uma abstração da memória global;
  - A única memória do Device que a CPU pode acessar é a Global;
- Modelo de Execução:
  - Imagem (Getting Started with CUDA, pág 14);
  - Threads são executadas nos processadores;

- Blocos são executados nos multiprocessadores;
- Blocos não migram (executam até o fim no mesmo multiprocessador);
- Podem residir em um multiprocessador quantas threads forem permitidas limitado pelos recursos de hardware do multiprocessador (memória compartilhada,registradores);
- Kernel é lançado como um grid de blocos de threads;
- Apenas um Kernel pode executar por vez na GPU;

6.2 Gerenciamento de Memória

`cudaMalloc(void** pointer, size_t nbytes)`

- Aloca um ponteiro em GPU;
- pointer deve ser alocado previamente em CPU.

`cudaMemset(void* pointer, int value, size_t nbytes)`

- Preenche os primeitos nbytes do endereço de memória com o value.

`cudaFree(void* pointer)`

- Libera o ponteiro em GPU.

`cudaMemcpy(void* dst, void* src, size_t nbytes, enum cudaMemcpyKind direction)`

- copia os nbytes de src para dst;
- direction indica a direção da transferência:
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`
- Exemplo de gerenciamento de memória (Getting Started, pág 25).

6.3 Executando Código em GPU

Kernels são funções em C com restrições

- Não podem acessar memória do Host;
- Deve retornar void;
- O número de argumentos não pode ser variável (é fixo);
- Não possui recursão;

- Não possui variáveis estáticas.

Os argumentos são automaticamente transferidos para a memória da GPU.

As funções podem ser qualificadas em:

- `__global__` : chamada pelo Host e executada pelo Device;
- `__device__` : chamada pelo Device e executada pelo Device;
- `__host__` : chamada pelo Host e executada pelo Host.

Pode ser combinada com o qualificador `__device__` para gerar uma função que pode executar tanto em CPU e quanto em GPU.

Lançando Kernels: `kernel<<<dG, dB>>>(...)`

- dim3 dG: dimensões do Grid;
- dim3 dB: dimensões do Bloco.

Variáveis Pré-definidas:

- dim3 gridDim
- dim3 blockDim
- dim3 blockIdx
- dim3 threadIdx

Identificadores de Threads: Imagem (Getting Started, pág 40)

Exemplo de Código: Imagem (Getting Started, pág 42)

Tipos vetoriais pré-definidos:

u char[1..4]

u short[1..4]

u int[1..4]

u long[1..4]

- float[1..4]
- double[1..2]
- dim3

Qualificadores de Variáveis:

`__device__`: indica que a variável em questão será alocada na memória global.

Variáveis alocadas com cudaMalloc() possuem este qualificador implicitamente.

`__shared__`: indica que a variável em questão será alocada na memória compartilhada.

É acessível por todas as threads no mesmo bloco onde ela foi criada.

Variáveis não qualificadas:

- Escalares e tipos vetoriais pré-definidos são alocados nos registradores.
- O que não couber nos registradores é alocado na memória local.

6.4 Memória Compartilhada

- Todos os Kernels são assíncronos:
  - O controle volta imediatamente para a CPU assim que um kernel é lançado;
  - cudaMemcpy é síncrono;
  - O controle volta para a CPU após a cópia terminar;
  - cudaThreadSynchonize() bloqueia (espera) até que todas as chamadas CUDA terminem.
- Exemplo (Getting Started, pág 44).