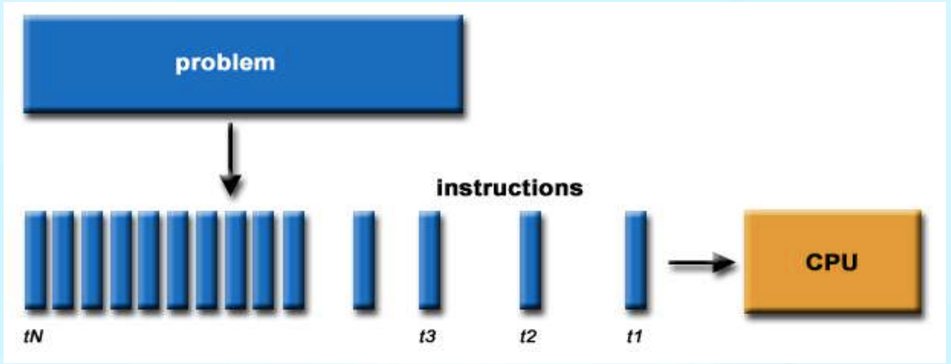
Tính toán song song

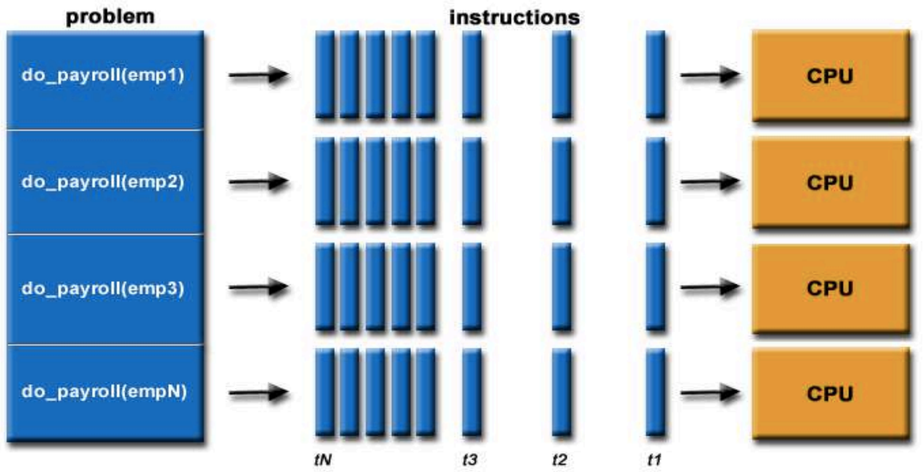
Parallel Computing

Dr. Tran Van Lang, A.Prof. in Computer Science

Tính toán song song là gì

- Theo truyền thống, phần mềm được viết theo cách tính toán tuần tự:
  - Được chạy trên một máy tính đơn có một CPU
  - Một bài toán được chia ra thành một chuỗi các câu lệnh rời rạc
  - Những câu lệnh này được thi hành sau câu lệnh khác
  - Vào một thời điểm chỉ có một câu lệnh thực thi.





- Trong một ngữ cảnh đơn giản, tính toán song song (Parallel Computing) là việc sử dụng đồng thời nhiều nguồn tài nguyên tính toán để giải quyết một vấn đề:
  - Được chạy bằng cách sử dụng nhiều CPU.
  - Một vấn đề có thể chia ra thành nhiều phần rời rạc được giải quyết một cách đồng thời.
  - Trong đó mỗi phần được tiếp tục phân chia thành một chuỗi các câu lệnh.
  - Những câu lệnh này được thi hành một cách đồng thời trên các CPU khác nhau.

- Nguồn tài nguyên tính toán có thể là:
  - Một máy tính duy nhất với nhiều bộ xử lý;
  - Một số tùy ý máy tính được kết nối qua mạng;
  - Tổ hợp cả hai loại trên.
- Bài toán tính toán có thể:
  - Được phân chia thành các phần công việc rời rạc mà có thể giải quyết một cách đồng thời
  - Thực hiện nhiều câu lệnh chương trình vào bất kỳ thời điểm nào
  - Với nhiều nguồn tài nguyên tính toán, bài toán được giải quyết ít thời gian so với chỉ một nguồn tài nguyên tính toán.

- Chẳng hạn:
  - Những bài toán thách thức lớn (Grand Challenge) được đặt ra đòi hỏi nguồn tài nguyên tính toán lên đến cả Peta phép tính trong một giây (số nguyên tố xuôi ngược: 1000000000000066600000000000001; số nguyên tố lớn nhất, hiện nay tìm ra $2^{13.466.017}$ -1)
  - https://www.extremetech.com/extreme/307998-ibm-supercomputer-identifies-77-compounds-that-could-fight-coronavirus
    - Máy này đã xác định được 77 hợp chất hóa học có thể giúp ngăn chặn Wuhan Coronavirus (SARS-CoV-2)
    - Mô phỏng trên hơn 8.000 hợp chất, tìm kiếm các phân tử làm cho virus này không tác động. Kết quả ban đầu đã xác định được 77 hợp chất có thể liên kết với protein Spike của Wuhan virus này, ngăn không cho nó liên kết với tế bào vật chủ.

Minh hoạ

- Cần dự báo thời tiết
  - Trên một vùng rộng chia thành các ô lưới, giả sử có:
    - 100 triệu ô lưới
    - để có kết quả, mỗi ô cần tính toán 100 phép toán
    - cần dự báo trước 2 ngày
    - bước tính toán là 1 phút (1 phút tính 1 lần)
  - Với máy tính có khả năng 100 triệu phép tính trong một giây. Hỏi bao nhiêu lâu thì có kết quả tính toán dự báo
  - Nên dùng máy tính nào là hợp lý

- Giải ra,
  - Số bước phải tính: 2ngày x 24giờ x 60phút /1 = 2.880 bước tính.
  - Số phép toán 10^2 x 10^8 x 2.880 = 2.880 x 10^{10} Flop
  - Với máy khả năng 10^8 FLops, thời gian tính toán là 2.880 x $10^{10}/10^8$ giây = 288.000 giây (gần bằng 3,33 ngày)
  - Để hợp lý, phải biết trước ít nhất là 1 ngày (=86.400 giây)
    - Thì cần máy có khả năng là 2.880 x 10^{10}/86.400 Flops = 2.880 x 10^{10}/864 x 10^2 Flops = 2.880/840 x 10^8 Flops = 3,43 x 10^8 Flops = 343 x 10^6 Flops
    - Như vậy cần máy tính có khả năng tính toán tương được 343 MFlops

- Tương tác giữa protein với phân tử nước
  - Trong cơ thể, một protein tương tác với một phân tử nước mất gần 1 giờ
  - Nếu mô phỏng,
  - một mô phỏng mất 10^{-12} giây
  - một mô phỏng cần 100 phép tính
- Nên nếu mô phỏng bằng máy tính nhanh nhất vào thánh 6/2018
  - Số phép tính cần: $100 \times 3600/10^{-12} = 36 \times 10^{16}$ Flop
  - Với máy tính năng lực 122.300 TFlop/s = $12{,}23 \times 10^{16}$ Flop/s, thì mất 36/12,23 giây (gần 2,94 giây)
  - Nhưng với máy tính thứ 500 – 715,6 TFlops (https://www.top500.org/list/2018/06/?page=5), mất 36/0,07156 giây (gần 503 giây tương đương 8,4 phút)

Kiểu mẫu phân chia

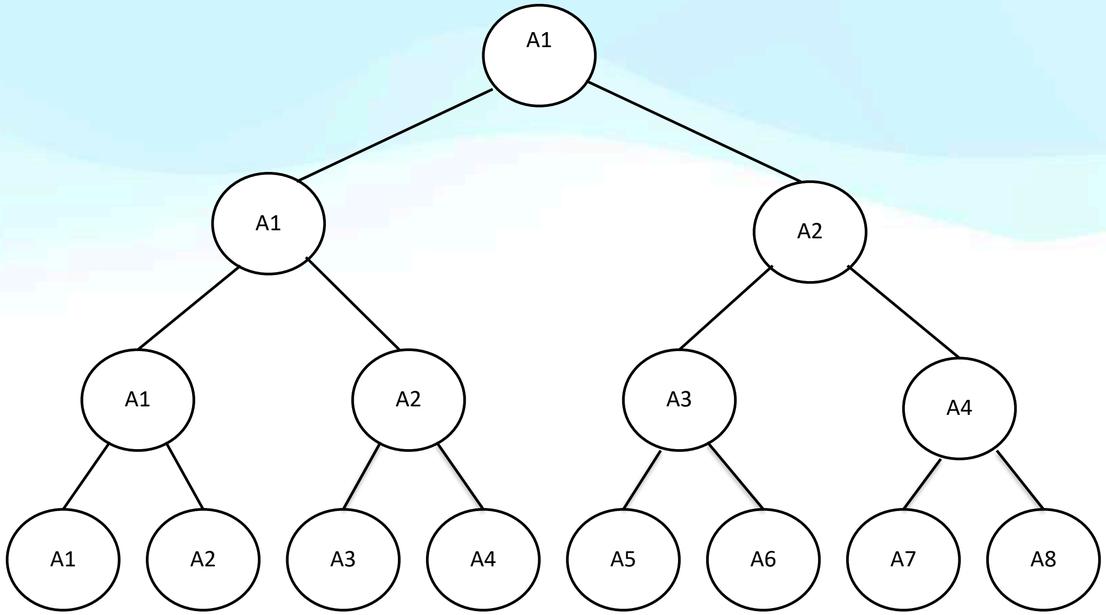
Paradigm

Dr. Tran Van Lang, A.Prof. in Computer Science

- Có 3 kiểu:
  - Mô hình cây nhị phân (Binary Tree Paradigm)
  - Chia để trị (Devide and Conquer Paradigm)
  - Phân hoạch (Partitioning Paradigm)
- Cây nhị phân:
  - Xét cây nhị phân đầy đủ với n lá có độ cao là $\log_2 n$ (hoặc ký hiệu $\log n$)
  - Dữ liệu đặt ở n nút lá.
  - Quá trình đi từ ngọn đến gốc mất $\log n$ thời gian.

Cây nhị phân

- Thuật giải tuần tự mất $O(n)$
  - Xét với $n = 2^k$ để có được cây nhị phân đầy đủ
  - Từ đây chia dữ liệu thành 2 nhóm
  - Số tiến trình cần thiết là $n/2$
- Minh hoạ với $n = 8 = 2^3$, mỗi nhóm có 4 phần tử
  - Nhóm 1: A_1, A_2, A_3, A_4
  - Nhóm 2: A_5, A_6, A_7, A_8
- Cần 4 task
- Với dãy gồm 8 phần tử, mô hình như hình vẽ bên dưới



- Bốn tiến trình đồng thời tính các giá trị tổng của nó theo yêu cầu
- Rồi lưu vào các biến tương ứng
  - $A_1 \leftarrow A_1 + A_2$
  - $A_2 \leftarrow A_3 + A_4$
  - $A_3 \leftarrow A_5 + A_6$
  - $A_4 \leftarrow A_7 + A_8$

- Giai đoạn tiếp theo chỉ còn lại 4 phần tử. Các phần tử lưu trong 2 nhóm
  - Nhóm 1: A_1, A_2
  - Nhóm 2: A_3, A_4
- Khi đó 2 tiến trình đồng thời cộng
  - $A_1 \leftarrow A_1 + A_2$
  - $A_2 \leftarrow A_3 + A_4$
- Các kết quả được lưu vào A_1, A_2

- Thuật giải

Algorithm: Tính tổng nhị phân

Input: Mảng A[1..n]

Output: A[1]

```
1. p = n/2
2. While p > 0 do
3.         For i = 1 to p doPar
4.                 A(i) = A(2i-1) + A(2i)
5.         endFor
6.         p = p/2
7. EndWhile
```

- Độ phức tạp:
  - Số process ban đầu là $p = n/2$
  - Trong mỗi lần thực hiện số tiến trình chỉ còn 1/2.
  - Nên số tiến trình cần thiết là
    $$P = n/2 = O(n)$$
  - Trong câu lệnh 4, chi phí thời gian là $O(1)$ cho 1 tiến trình, nên với $\log_2 n$ bước, chi phí thời gian $O(\log n)$.

Chia để trị

- Chia bài toán thành các bài toán con để dễ tìm ra lời giải cho tứng bài toán con riêng lẽ.
- Hợp nhất các lời giải này lại để được lời giải của bài toán.
- Theo cách thực hiện của mô hình cây nhị phân
  - Thuật giải song song chưa tối ưu, bởi:
    - Thuật giải tuần tự cần $O(n)$
    - Song song cần $O(\log n)$ thời gian với $O(n)$ task
- Nên hiệu suất $E_p(n)$ (Efficiency) quá nhỏ khi n khá lớn:

$$\frac{O(n)}{O(n)O(\log n)} = \frac{1}{\log n} < 1$$

- Chúng ta phải khảo sát bài toán sao cho $E_p(n)$ có giá trị là 1

- Với thuật giải tính tổng như trên, $1 = E_p(n) = \dfrac{O(n)}{p \times O(\log n)}$
- Suy ra $P = \dfrac{n}{\log n}$
- Minh hoạ:
  - Chia mảng $A(1{:}n)$ thành $r = n/\log n$ nhóm, để huy động r tiến trình
  - Mỗi nhóm có $\log n$ phần tử.
  - Ở đây vẫn giả thiết $n = 2^k$ và $n/\log n$ là số nguyên ($k = \log n$)
  - Các nhóm được phân bố như sau:

- $A_1, A_2, A_3, \ldots, A_k$
- $A_{k+1}, A_{k+2}, A_{k+3}, \ldots, , A_{2k}$
- .....
- $A_{(i-1)k+1}, A_{(i-1)k+2}, \ldots, A_{ik}$
- .....
- $A_{(r-1)k+1}, A_{(r-1)k+2}, \ldots, A_{rk}$

- Mỗi tiến trình cộng $\log n$ phần tử.
- Lưu kết quả lại trong mảng $B(1{:}r)$
- Sử dụng thuật giải song song (như phần 1) để tính tổng r phần tử của B.
- Thuật giải song song trên B này cần $O(\log n)$thời gian và $O(r)$ tiến trình.

- Thuật giải:

Algorithm: Tính tổng chia và chế ngự

Input: A(1..n)
Output: B(1)
1. **For** i=1 **to** n/logn **doPar**
2. B(i) = 0
3. **For** j=1 **to** logn **do**
4. B(i) = B(i)+ A(ik+j-logn)
5. **EndFor**
6. **EndPar**

Algorithm: Nhị phân tính tổng B(1..r)

Input: B(1..r)
Output: B(1)
1. p = r/2
2. While p > 0 do
3. **For** i=1 **to** p **doPar**
4. B(i) = B(2i-1) + B(2i)
5. **EndPar**
6. p = p/2
7. **EndWhile**

- Độ phức tạp:
  - Các câu lệnh 2, 3, 4 cần $O(\log n)$ bước thời gian.
  - Khi đó, các bước tứ 1 đến 5 dùng song song mất $O(\log n)$ thời gian $O(n/\log n)$ tiến trình.
  - Các bước còn lại dùng thuật giải song song dạng cây nhị phân, cần thời gian với chi phí:

$$O\left(log(\frac{n}{\log n})\right) = O(\log n - \log\log n) = O(\log n)$$

  - Dùng $O\left(\frac{n}{\log n}\right)$ tiến trình.
  - Như vậy, thuật giải cần $O(\log n)$ thời gian với $O\left(\frac{n}{\log n}\right)$ tiến trình.

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

int main(){
  float *A, *B, *S;
  long int N, i, j;
  int R, K, p;

  printf("No. of elements:");
  scanf( "%ld", &N );
  K = log2(N);
  R = N/K;

  A = (float *)calloc( N, sizeof(float) );
  for( i = 0; i < N; i++ )
    A[i] = i+1;
  B = (float *)calloc( R, sizeof(float) );
  double wt = omp_get_wtime();
#pragma omp parallel for
  for( i = 0; i < R; i++ ){
    B[i] = 0.0;
    for(int j = 0; j < K;j++)
      B[i]+= A[(i+1)*K+j-K];
  }
  p = R/2;
  while( p > 0 ){
#pragma omp for
    for( i = 0; i < p; i++ )
      B[i]= B[2*i]+B[2*i+1];
    p /= 2;
  }
  printf( "Elapsed Parallel Time: %lf sec\n",
      omp_get_wtime()-wt );
  printf( "Sum of Sequence: %lf\n", B[0] );
  return 1;
}
```

- Mô hình phân hoạch:
  - Theo cách tiếp cận chia để trị, thuật giải phải bao gồm 2 bước, trong đó có bước quan trọng là việc hợp nhất các bài toán con đã chia ra để được lời giải của bài toán xuất phát.
  - Đôi khi việc thực hiện này làm ảnh hưởng đến kết quả của ban đầu.
  - Trong mô hình phân hoạch, người ta không quan tâm đến quá trình hợp nhất.
  - Tiến trình phân hoạch bảo đảm sao cho khi phân chia xong, việc giải các bài toán con cho kết quả là lời giải của bài toán đặt ra.

- Giả sử có 2 mảng A(1:n), B(1:n) đã được sắp xếp tăng, cần phải hợp nhất thành 1 mảng C(1:2n) cũng tăng.
- Ở đây giả thiết $n = 2^k$, và $r = n/\log n$ là số nguyên ($k = \log n$)
- Thuật giải như bên:

Algorithm: Merge //Thuật giải trộn tuần tự

```
Input: A(1..n), B(1..n)
Output: C(1..2n)
1. i = 1; j = 1; k = 1
2. While  k <= 2n do
3.        If A(i) < B(j) then
4.                 C(k) = A(i)
5.                 i = i + 1
6.        Else
7.                 C(k) = B(j)
8.                 j = j + 1
9.        EndIf
10.       k = k + 1
11. EndWhile
```

- Minh hoạ:
  - Phân hoạch A thành $r = n/\log n$ nhóm gồm k phần tử như sau:
    - $A_1, A_2, A_3, \ldots, A_k$
    - $A_{k+1}, A_{k+2}, A_{k+3}, \ldots, , A_{2k}$
    - .....
    - $A_{(i-1)k+1}, A_{(i-1)k+2}, \ldots, A_{ik}$
    - .....
    - $A_{(r-1)k+1}, A_{(r-1)k+2}, \ldots, A_{rk}$

- Tiếp theo cần tìm r số nguyên $j_1, j_2, \ldots, j_r$ sao cho
  - j_1 là chỉ số lớn nhất mà $A_k > B_{j_1}$
  - j_2 là chỉ số lớn nhất mà $A_{2k} > B_{j_2}$
  - .....
  - j_i là chỉ số lớn nhất mà $A_{ik} > B_{j_i}$
  - .....
  - j_r là chỉ số lớn nhất mà $A_{rk} > B_{j_r}$

- Từ đây, phân B thành r nhóm như sau:
  - $B_1, B_2, \ldots, B_{j_1}$
  - $B_{j_1+1}, B_{j_1+2}, \ldots, B_{j_2}$
  - .....
  - $B_{j_{i-1}+1}, B_{j_{i-1}+2}, \ldots, B_{j_i}$
  - .....
  - $B_{j_{r-1}+1}, B_{j_{r-1}+2}, \ldots, B_{j_r}$

- Các phần tử trong nhóm 1 của *A* đều nhỏ hơn hay bằng các phần tử trong nhóm 2, 3, ... của *B*.
- Có thể đồng thời hợp nhất các phần tử trong hai nhóm thứ i của *A* và *B*.

Algorithm: Thuật giải trộn

Input: A(1..n), B(1..n)
Output: C(1..2n)

```
1. For i = 1 to r doPar
2.         j(i) = max{t/B(t)<A(ik)}
3.         Merge( A((i-1)k+1..ik), B(j(i-1)+1..j(i)) )
4. EndPar
```

- Ở bước thứ 2, dùng thuật giải tìm kiếm nhị phân, chi phí $O(\log n)$.
- Bước 3, tùy theo kích thước của $B_{j_{i-1}+1:j_i}$
- Nếu lớn hơn k, chúng ta dùng thuật giải song song này để hợp nhất hai phần tương ứng của A và B.
- Còn khi kích thước này nhỏ hơn hay bằng k, thuật giải cần chi phí $O(\log n)$.
- Như vậy, thuật giản cần $O(\log n)$ thời gian, $O(n/\log n)$ tiến trình.

- Minh hoạ:
  - A = {1,5,15,18,19,21,23,24,27,29,30, 31,32,37,42,49}
  - B = {2,3,4,13,15,19,20,22,28,29,38, 41,42,43,48,49}
  - n = 16 = 2^4, k =4, r = n/logn = 4
- A được phân thành 4 nhóm
- 1,5,15,18
- 19,21,23,24
- 27,29,30,31
- 32,37,42,49

- Khi đó, j = {5,8,10}
- B được phân thành 4 nhóm
  - 2,3,4,13,15
  - 19,20,22
  - 28,29
  - 38,41,42,43,48,49

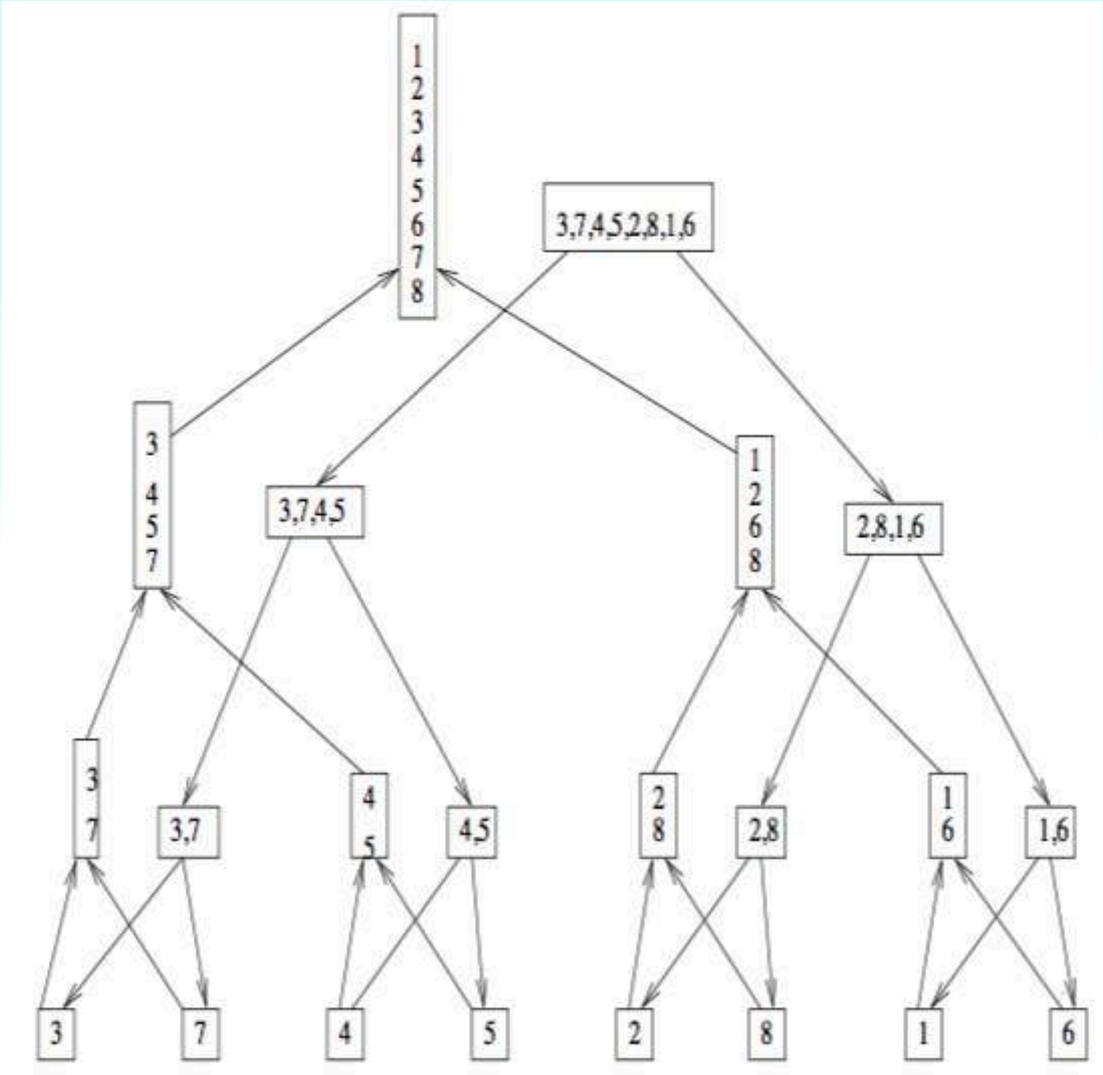
- Đồng thời hợp nhất các nhóm con của A và B.

Nhóm	1	2	3	4
A	1, 5, 15, 18	19, 21, 23, 24	27,29, 30,31	32,37, 42,49
B	2, 3, 4, 13, 15	19,20, 22	28,29	38,41,42, 43,48,49

- C(1:9) = {1,2,3,4,5,13,15,15,18}
- C(10:16) = {19,19,20,21,22,23,24}
- C(17:22) = {27,28,29,29,30,31}
- C(23:32) = {32,37,38,41,42,42,43,48,49,49}

Ứng dụng: thuật giải Merge-Sort

- Minh họa, với dãy 8 số nguyên, thuật giải trộn và sắp xếp như sau:
  - Các tiến trình đồng thời sắp xếp tuần tự phần dữ liệu liên quan.
  - Sau khi hoàn tất, trộn (merge) hai dãy đã sắp xếp thành một dãy sắp xếp
  - Các tiến trình tiếp tục cho đến hết dữ liệu



Ví dụ

- Phân tích thuật toán song song tính số π từ tích phân như sau:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$ bằng cách xấp xỉ

$$\pi \approx \sum_{i=0}^{N} \frac{4}{1+x_i^2} \Delta x,$$

- Trong đó $\Delta x = \frac{1}{N}, x_i = i\Delta x$

```
#include <stdio.h>
#define N  10000000
int main(int argc, const char *
argv[]){
   double delta, x, sum = 0.0, pi;
   delta = 1.0/N;
   for ( int i = 0; i < N; i++ ){
     x = i*delta;
     sum += 4.0/(1.0 + x*x);
   }
   pi = sum*delta;
   printf( "Pi = %20.18f\n", pi );
   return 0;
}
```

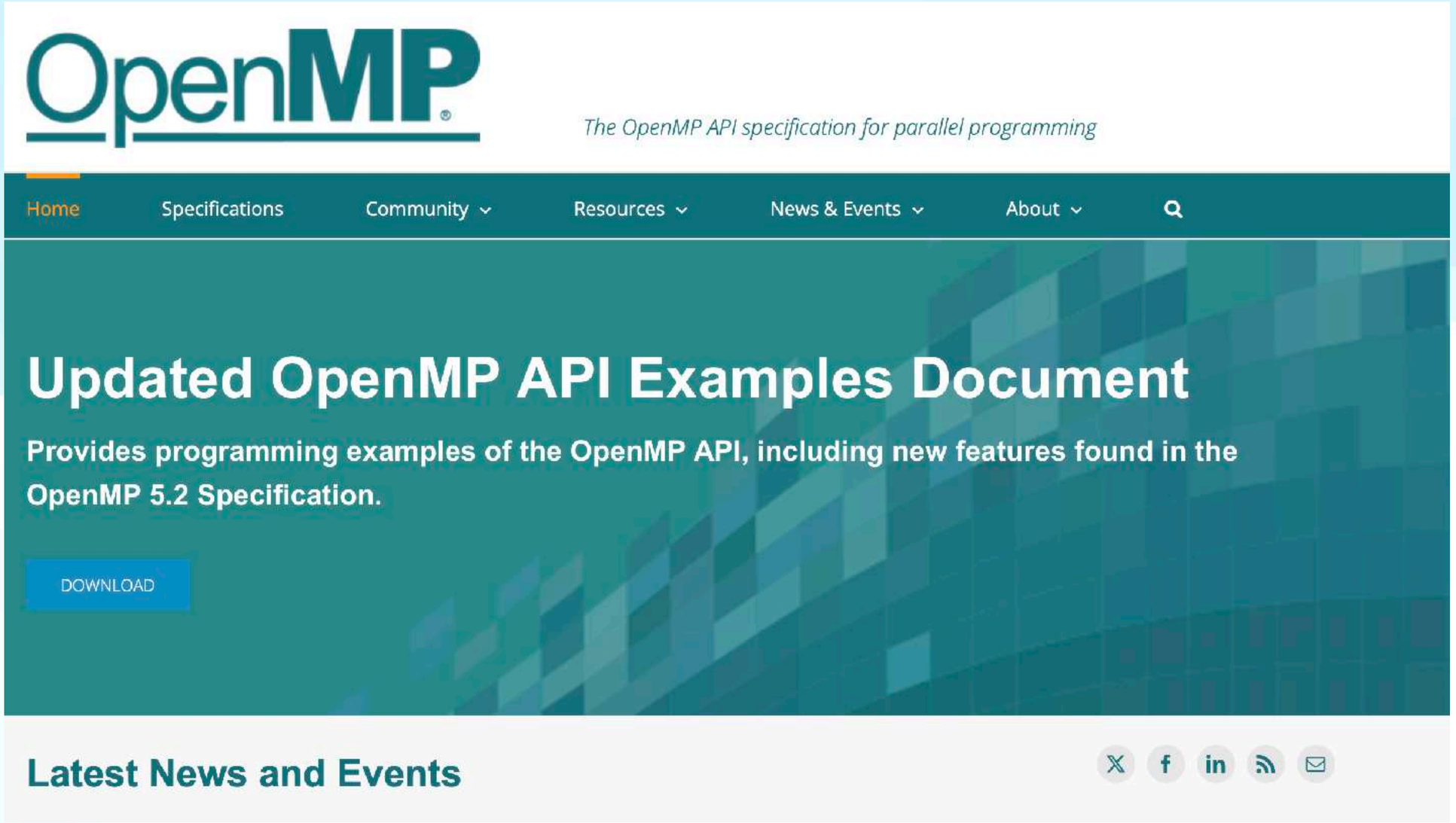
- Phân tích thuật toán tính số π dùng Phương pháp Monte Carlo:
  - Đường tròn nội tiếp trong một hình vuông
  - Nếu đặt số chấm ngẫu nhiên lên hình vuông
  - Thì tỷ lệ giữa số chấm nằm trong hình tròn và số chấm nằm trong hình vuông là: $\frac{r^2\pi}{2r \times 2r} = \frac{\pi}{4}$

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define npoints 10000000
int  main(int argc, const char * argv[]){
    int count = 0;
    double x, y, pi;
    srand( (unsigned int)time(NULL) );
    for ( int i = 1; i < npoints; i++ ){
      x = (double)rand()/RAND_MAX;
      y = (double)rand()/RAND_MAX;
      if ( x*x + y*y <= 1 )
      count += 1;
    }
    pi = 4.0*count/npoints;
    printf( "Pi = %20.18f\n", pi );
    return 0;
}
```

Lập trình OpenMP với shared-memory

- OpenMP (Open Multi-Processing) được cung cấp bởi The OpenMP Architecture Review Board (ARB) published its first API specifications,
- 10/1997: OpenMP for Fortran 1.0
- 10/1998. OpenMP for C/C++ standard
- 2000: version 2.0 of the Fortran specification
- 2002: version 2.0 of the C/C++ specifications
- 2005: Version 2.5 is a combined C/C++/ Fortran specification
- 5/2008: Version 3.0
- 7/2014: Version 4.0
- 11/2018: Version 5.9
- The current version is 5.2.2 (4/2024)
- **A preview of OpenMP 6.0, that will be released in 2024**

http://www.openmp.org



- Để install, có thể thông qua website https://releases.llvm.org để download phiên bản cần thiết.
- Hoặc install thư viện libomp dùng Homebrew ở terminal của hệ điều hành, bằng lệnh:
  - brew install libomp

- Với C/C++, có thể dùng:
  - GCC (GNU Compiler Collection)
  - Clang++
  - Intel C Compiler
  - Microsoft Visual C++
- Chẳng hạn, với macOS,
  - install GCC(vd gcc-11): brew install gcc
  - gcc-11 -o vidu vidu.c -fopenmp
  - **Lưu ý**: với gcc, đã có OpenMP để dùng

- OpenMP là dạng lập trình MultiThread
- Dùng để lập trình (Code Generation) song song trên bộ nhớ chia sẻ và multicore
- Chương trình viết trong OpenMP dùng mô hình SPMD (Single Program Multi Data)
- Các thành phần của OpenMP bao gồm:
  - Các chỉ thị biên dịch (Compiler Directives)
  - Các hàm thư viện (Runtime Library Routines)
  - Các biến môi trường (Environment Variables)

Chỉ thị parallel

- OpenMP được điều khiển thông qua các chỉ thị
  - Những chỉ thị của chương trình C/C++ được bắt đầu bởi từ khóa tiền xử lý #pragma.
  - Với OpenMP, chỉ thị được bắt đầu bởi #pragma omp
- Chỉ thị #parama omp parallel cho biết các câu lệnh trong khối được thực thi song song trên các thread của máy.
  - Ví dụ:

```
int main(){
#pragma omp parallel
    {
        printf( "OpenMP: Hello\n" );
    }
    return 1;
}
```

- Chương trình này xuất ra 8 dòng“OpenMP: Hello” nếu máy đó có 8 thread

```
lang@macBA OpenMP % ./first
OpenMP: Hello
OpenMP: Hello
OpenMP: Hello
OpenMP: Hello
OpenMP: Hello
OpenMP: Hello
OpenMP: Hello
OpenMP: Hello
lang@macBA OpenMP %
```

- Hoặc coi thông tin về máy

```
lang@macBA OpenMP % ./getinfo
So bo xu ly: 8
So thread: 8
Có 1 threads
Có 0 devices
Có 0 places
Có 1 teams
Total RAM: 24576 MB
Total RAM: 24576 MB
Thoi gian troi qua 0.001317 giay
lang@macBA OpenMP %
```

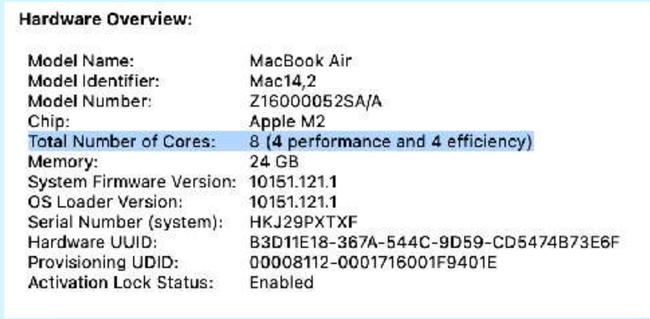
- Lưu ý:
  - Chương trình sử dụng các hàm thư viện, nên phải có `#include <omp.h>`
  - Khi biên dịch, phải chỉ định thư viện với –fopenmp. Chẳng hạn, để dịch tập tin first.c ra tập tin first khi dùng **GCC**:
    - gcc -o first first.c –fopenmp
    - g++ -o first first.cc –fopenmp
  - Hoặc dùng **Clang** phải chỉ định, chẳng hạn
    - clang –o first first.c –fopenmp

- Với C/C++, có thể dùng:
  - GCC (GNU Compiler Collection)
  - Clang++
  - Intel C Compiler
  - Microsoft Visual C++
- Chẳng hạn, với macOS,
  - install GCC(vd gcc-11): brew install gcc
  - gcc-11 -o vidu vidu.c -fopenmp
  - **Lưu ý**: với gcc, đã có OpenMP để dùng

Chỉ thị parallel for

- Chỉ thị này chia các công việc trong vòng lặp for cho các thread
- Ví dụ: Với máy có 8 core, chương trình với chỉ thị parallel for như sau:

```
#include <stdio.h>
#include <omp.h>
int main() {
   int n, tid;
#pragma omp parallel for
   for( n = 0; n < 16; ++n ){
      tid = omp_get_thread_num();
      printf( "(T#%d,%d) ", tid, n );
   }
   printf( "\n" );
   return 1;
}
```



- Trên máy có 8 core (thread), với 16 lần lặp
- Mỗi core đảm trách $16/8 = 2$ lần lặp để thực hiện lệnh printf()
  - Core 0: xuất giá trị 0, 1
  - Core 1: xuất giá trị 2, 3
  - Core 2: xuất giá trị 4, 5
  - Core 3: xuất giá trị 6, 7
  - Core 6: xuất giá trị 12,13
  - Core 7: xuất giá trị 14, 15

- Kết quả những lần thực hiện khác nhau như sau

```
lang@macBA OpenMP % ./second
(T#2,4) (T#2,5) (T#3,6) (T#3,7) (T#4,8) (T#4,9) (T#6,12) (T#6,13) (T#7,14) (T#1,2) (T#7,15) (T#5,10) (T#0,0) (T#0,1) (T#5,11) (T#1,3)
lang@macBA OpenMP % ./second
(T#2,4) (T#2,5) (T#1,2) (T#3,6) (T#1,3) (T#3,7) (T#5,10) (T#5,11) (T#6,12) (T#7,14) (T#7,15) (T#4,8) (T#4,9) (T#6,13) (T#0,0) (T#0,1)
lang@macBA OpenMP % ./second
(T#1,2) (T#1,3) (T#0,0) (T#6,12) (T#0,1) (T#3,6) (T#3,7) (T#7,14) (T#7,15) (T#6,13) (T#4,8) (T#4,9) (T#2,4) (T#2,5) (T#5,10) (T#5,11)
lang@macBA OpenMP % ./second
(T#1,2) (T#1,3) (T#2,4) (T#2,5) (T#6,12) (T#6,13) (T#3,6) (T#3,7) (T#0,0) (T#0,1) (T#7,14) (T#7,15) (T#5,10) (T#5,11) (T#4,8) (T#4,9)
lang@macBA OpenMP % ./second
(T#1,2) (T#1,3) (T#3,6) (T#2,4) (T#3,7) (T#2,5) (T#6,12) (T#0,0) (T#6,13) (T#0,1) (T#5,10) (T#5,11) (T#4,8) (T#4,9) (T#7,14) (T#7,15)
lang@macBA OpenMP % ./second
(T#1,2) (T#1,3) (T#6,12) (T#6,13) (T#0,0) (T#0,1) (T#2,4) (T#2,5) (T#4,8) (T#4,9) (T#7,14) (T#7,15) (T#5,10) (T#5,11) (T#3,6) (T#3,7)
lang@macBA OpenMP % ./second
(T#2,4) (T#7,14) (T#2,5) (T#7,15) (T#1,2) (T#1,3) (T#3,6) (T#3,7) (T#5,10) (T#5,11) (T#4,8) (T#4,9) (T#6,12) (T#6,13) (T#0,0) (T#0,1)
lang@macBA OpenMP %
```

- Lưu ý:
  - Với chỉ thị này, ta có thể chỉ định số thread cần thực hiện. Chẳng hạn, chỉ muốn dùng 2 thread, bổ sung thêm num_threads() vào chỉ thị parallel for

```
#pragma omp parallel for num_threads(2)
```

  - Khi đó, với ví dụ trên, các câu lệnh in giá trị 0, 1, 2, 3, 4, 5, 6, 7 cho Core #0; và 8, 9, 10, 11, 12, 13, 14, 15 cho Core #1.
  - Việc khai báo sau đây là tương đương nhằm chỉ ra *n* là biến địa phương của từng thread

```
#pragma omp parallel for num_threads(2)
for( int n = 0; n < 16; ++n )
```

Ví dụ: khởi tạo giá trị

- Khởi tạo giá trị ban đầu cho một mảng dữ liệu có N phần tử, trong đó $N \gg 1$
- Cách giải quyết:
  - Chia thành P mảng con, mỗi mảng có $\frac{N}{P}$ phần tử
  - Từ đó P tiến trình đồng thời khởi tạo các mảng con này

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

void create( float *x,long int start,long int np ){
    long int i;
    for ( i = 0; i < np; i++ )
        x[start+i] = i*3.14159/np;
}
```

- Chương trình con khởi tạo của mỗi tiến trình và chương trình con phân chia cho mỗi tiến trình như sau (/Users/lang/Documents/Works/Training/Parallel Computing/Example/OpenMP/second3.c):

```c
void subroutine( float *x,long int n ){
    int p,P;
    long int np,start;
    #pragma omp parallel private(p,P,np,start)
    {
        p = omp_get_thread_num();
        P =  omp_get_num_procs();
        np = n/P;
        start = p*np;
        if (p == P-1)
            np = n - start;
        printf( "Call the create() function to create an array of %d elements starting from the %dth position.\n", np, start );
        create( x,start,np );
    }
}
```

Ví dụ: tính tổng (SimpleSum.c++)

- Gồm các khối:
  - Tạo dữ liệu ngẫu nhiên
  - Tính tổng theo thuật giải tuần tự
  - Tính tổng song song

```
void create( double* a, long n ){
    srand( time(0) );
    for ( int i = 0; i < n; i++ )
        a[i] = 1959 + rand() % 63;  // a[i] in [1959,2022]
}
```

```
void sequence( double* a, long n ){
    double s, ls, wt = omp_get_wtime();
    ls = 0.0;
    for ( int i = 0; i < n; i++ )
        ls += a[i];
    cout << "Elapsed time for Sequence computing "
    << omp_get_wtime() - wt << " seconds\n";
    cout << "Sum = " << ls << endl;
}
```

```
void parallel( double* a, long n ){
    double s, ls, wt = omp_get_wtime();
    ls = 0.0;
    s = 0.0;
#pragma omp parallel private( ls )
    {
#pragma omp for
        for ( int i = 0; i < n; i++ )
            ls += a[i];
        s += ls;
    }
    cout << "Elapsed time for Parallel computing "
    << omp_get_wtime() - wt << " seconds\n";
    cout << "Sum = " << s << endl;
}
```

```
void parallel2( double* a, long n ){
    double s, ls, wt = omp_get_wtime();
    s = 0.0;
#pragma omp parallel for reduction(+:s)
    for ( int i = 0; i < n; i++ )
        s += a[i];

    cout << "Elapsed time for Parallel computing "
    << omp_get_wtime() - wt << " seconds\n";
    cout << "Sum = " << s << endl;
}
```

Ví dụ: nhân 2 ma trận

- Chương trình tuần tự

```
int mult( double **a, double **b, double**c, int n, int m, int l ){
    double wt = omp_get_wtime();
    int i, j, k;
    for ( i = 0; i < n; i++ )
        for ( j = 0; j < m; j++ ){
            c[i][j] = 0.0;
            for ( k = 0; k < l; k++ )
                c[i][j] += a[i][k]*b[k][j];
        }
    cout << "Elapsed time for Sequential Implementation: " << setw(10) << omp_get_wtime() - wt << " seconds\n";
    return 1;
}
```

- Chương trình song song (multi_matrix.cc)

```
int par_mult( double **a, double **b, double**c, int n, int m, int l ){
    double wt = omp_get_wtime();
    int i, j, k;
#pragma omp parallel private( i,j,k )
    {
#pragma omp for
        for ( i = 0; i < n; i++ )
            for ( j = 0; j < m; j++ ){
                c[i][j] = 0.0;
                for ( k = 0; k < l; k++ )
                    c[i][j] += a[i][k]*b[k][j];
        }
    }
    cout << "Elapsed time for Parallel Implementation: " << setw(10) << omp_get_wtime() - wt << " seconds\n";
    return 1;
}
```

- Thời gian tính toán

```
lang@macBA OpenMP % ./mult_matrix
Number of   rows  in matrix A (n): 10000
Number of columns in matrix A (l): 1000
Number of columns in matrix B (m): 10000
Elapsed time for Parallel Implementation:    86.1547 seconds
Elapsed time for Sequential Implementation:    421.649 seconds
lang@macBA OpenMP % ./mult_matrix
Number of   rows  in matrix A (n): 1000
Number of columns in matrix A (l): 1000
Number of columns in matrix B (m): 1000
Elapsed time for Parallel Implementation:   0.723932 seconds
Elapsed time for Sequential Implementation:    3.54883 seconds
lang@macBA OpenMP % ./mult_matrix
Number of   rows  in matrix A (n): 10000
Number of columns in matrix A (l): 1000
Number of columns in matrix B (m): 1000
Elapsed time for Parallel Implementation:    7.29075 seconds
Elapsed time for Sequential Implementation:    35.5495 seconds
lang@macBA OpenMP %
```

Chỉ thị critical

- Ví dụ tính tổng của một dãy $\{a_n\}$ gồm n số thực (omp_sum.c++).
  - Khi đó, từng thread tính tổng riêng (khai báo là ls).
  - Thread chủ làm nhiệm vụ tính tổng (khai báo là s) từ các tổng riêng ls này rồi xuất ra màn hình
  - Khi đó, việc tính $s = s + ls$ được ngăn ngừa để các thread không tính được thực hiện thông qua chỉ thị *critical*.

```
int main(){
    long int n = 10000;
    double ls = 0.0, s = 0.0;
    long int i;
    double *a = (double *)calloc( n, sizeof(double) );
    for ( i = 0; i < n; i++ )
        a[i] = i+1;
#pragma omp parallel private(i,ls)
    {
#pragma omp for
        for ( i = 0; i < n; i++ )
            ls += a[i];
#pragma omp critical
        s += ls;
    }
#pragma omp end parallel
    printf( "Sum of sequence a: %lf\n", s );
    return 1;
}
```

Mệnh đề reduction

- Với cách tính tổng dãy số $\{a_n\}$ như trên, có thể dùng mệnh đề ***reduction***.

```
    s = 0.0;
#pragma omp parallel for reduction(+:s)
    for ( i = 0; i < n; i++ )
        s += a[i];
    printf( "Sum of sequence a: %lf\n", s );
```

Một số hàm thư viện

- double omp_get_wtime(): trả về thời gian hiện tại, tính ra giây
- int omp_get_thread_num(): để biết thread nào thực hiện
- int omp_num_threads(): co biết tổng số thread
- int omp_get_num_procs(): trả về số processor hiệu lực vào thời điểm hàm được gọi.
- int omp_get_max_threads(): trả về số thread tối đa được sử dụng mà không có mệnh đề num_threads trong chỉ thị #pragma.
- int omp_get_num_threads(): trả về số thread được sử dụng trong vùng tác động của lệnh thi hành song song mà nó được gọi.

- Ví dụ: chỉ chương trình chạy trên thread #0 mới xuất ra số lượng thread đang sử dụng dưới một chỉ thị song song (third3.c)

```
int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf( "Number of threads: %d\n", omp_get_num_threads() );
            printf( "Number of processors: %d\n", omp_get_num_procs() );
        }
    }
    if ( omp_get_thread_num() == 0 )
        printf( "Number of threads: %d\n", omp_get_num_threads() );
        printf( "Number of processors: %d\n", omp_get_num_procs() );
    return 1;
}
```

Lưu ý (cpucorethread.c)

- Processor (CPU):
  - **Processor** hoặc **CPU (Central Processing Unit)** là đơn vị xử lý trung tâm của máy tính, chịu trách nhiệm thực hiện các lệnh từ phần mềm.
  - Một CPU có thể chứa một hoặc nhiều core. Trong hệ thống đơn lẻ, CPU thường được gọi là một chip xử lý đơn hoặc đa lõi (multi-core).
- Core:
  - **Core** là một đơn vị xử lý độc lập bên trong CPU. Một CPU có thể có một hoặc nhiều core.
  - Mỗi core có khả năng thực hiện các tác vụ (task) riêng biệt cùng một lúc.

- Thread:
  - **Thread** là một đơn vị xử lý nhỏ hơn chạy trong một core. Một core có thể xử lý nhiều threads bằng cách sử dụng kỹ thuật đa luồng (multithreading).
  - Trong lập trình, thread là một dải lệnh có thể được thực thi độc lập. Thread cho phép một chương trình thực hiện nhiều tác vụ đồng thời, tận dụng tối đa tài nguyên của CPU.

- Sự khác biệt chính:
  - CPU vs Core:
    - Một CPU có thể chứa một hoặc nhiều core. Core là phần cứng vật lý bên trong CPU.
    - Một CPU với nhiều core có thể xử lý nhiều tác vụ đồng thời hơn so với CPU đơn lõi.
  - Core vs Thread:
    - Core là phần cứng vật lý thực hiện lệnh. Một core có thể chạy nhiều threads thông qua kỹ thuật đa luồng.
    - Thread là một dải lệnh phần mềm mà core thực hiện. Multithreading cho phép một core xử lý nhiều threads cùng một lúc, tăng hiệu suất.

Chỉ thị section

- Khi phân rã theo chức năng, có thể có những đoạn chương trình chạy ở những tiến trình khác nhau.
- Chẳng hạn,
  - Work0: thực thi ở tất cả các tiến trình và song song với các Work1, Work2, Work4, Work5
  - Work1: thực thi trong 1 tiến trình
  - Work2, Work3: thực thi tuần tự trong cùng 1 tiến trình, nhưng song song với Work4
  - Work4: thực thi song song với Work2
  - Work5: thực thi ở tiến trình chủ (tiến trình #0)

- Chương trình như sau (fifth.c)

```c
int main(){
    work0();
    #pragma omp parallel sections // starts a new team
    {
        {
            work1();
        }
    #pragma omp section
        {
            work2();
            work3();
        }
    #pragma omp section
        {
            work4();
        }
    }
    work5();
    printf( "Num of threads %d\n", omp_get_num_threads() );
    return 1;
}
```

- Hoặc ví dụ khác: giả sử có 2 section, thực thi 2 công việc tính toán song song và tính toán tuần tự đồng thời (multi_matrix_section.c++).

```
#pragma omp parallel sections
    {
    #pragma omp section
    par_mult( a, b, c, RA, CB, CA );
    #pragma omp section
    mult( a, b, c, RA, CB, CA );
    }
```

- Thay vì lần lượt tính toán

```
par_mult( a, b, c, RA, CB, CA );
mult( a, b, c, RA, CB, CA );
```

- Khi đó thời gian thực thi là thời gian lớn nhất (thời gian tuần tự)

Lập trình song song với Python

Parallel Programming with Python

- Python sử dụng cơ chế thông dịch (Interpreter) nên đôi khi chạy chậm không như mong muốn.
- Để tăng hiệu năng tính toán, Python có trang bị các gói multiprocessing và gói threading để lập trình với Process (tiến trình) và Thread (luồng).

- Ngoài `threading`, `multiprocessing` module; còn có
  - Concurrent Futures với `concurrent.futures` module
  - Asyncio trong việc lập trình không đồng bộ (Asynchronous Programming)
  - Parallel Computing với `joblib` và `dask.`
  - GPU Programming với `numba` và `cupy`
  - Distributed Computing với `Ray`

Hàm liên quan đến hệ thống (sysinfo.py)

- Trước hết, để biết về cấu hình của máy, ta viết một chương trình con `sysinfo`. Trong chương trình này sử dụng một số hàm có trong gói platform, socket và multiprocessing.

```
import platform
import socket
import multiprocessing as mp

# Sub Program
def sysinfo():
    print()
    print('Python version:', platform.python_version())
    print('Compiler   :', platform.python_compiler())
    print('System     :', platform.system())
    print('Release    :', platform.release())
    print('Machine    :', platform.machine())
    print('Processor  :', platform.processor())
    print('CPU count  :', mp.cpu_count())
    print('Interpreter:', platform.architecture()[0])
    print('Host name  :', socket.gethostname())
    print('IP Address :', socket.gethostbyname(socket.gethostname()))
    print()

if __name__ == '__main__':
    sysinfo()
```

elapsedtime.py

- Đo thời gian thực thi một đoạn lệnh có thể dùng hàm `time()` có trong gói time.
- Hàm này trả về một số thực cho biết số giây kể từ lúc 0 giờ ngày 01/01/1970 đến thời điểm gọi hàm.
- Chẳng hạn, đo thời gian thực thi một số thao tác trên ma trận

```
import time
import numpy as np

N = 100000
M = 1000
t0 = time.time()
# Tạo ma trận NxM với các phần từ có giá trị là 1
A = np.ones((N,M))
t1 = time.time()
# Tạo ma trận đơn vị MxM
I = np.identity(M)
t2 = time.time()
# Tạo ma trận MxM có giá trị là 3
B = np.full((M,M),3)
t3 = time.time()
# Tạo ma trận NxM có giá tri ngẫu nhiên trong [0,1)
C = np.random.random((N,M))
t4 = time.time()
print()
print( "Elapsed time to create matrix A is %7.5f seconds" % (t1-t0) )
print( "Elapsed time is create matrix I is %7.5f seconds" % (t2-t1) )
print( "Elapsed time is create matrix B is %7.5f seconds" % (t3-t2) )
print( "Elapsed time is create matrix C is %7.5f seconds" % (t4-t3) )
print()
```

Process và Thread

- Process (hay là tiến trình) và Thread (luồng) là hai khái niệm thường gặp nhất khi lập trình nâng cao hiệu quả tính toán.
- Cũng cần phân biệt rõ hai khái niệm này:
  - Process: có thể coi là chương trình được nạp vào bộ nhớ máy tính để thực thi
  - Thread: là một khối các câu lệnh trong một process.
    - Thread còn được gọi là sub-process (tiểu tiến trình) để thực hiện một nhiệm vụ riêng lẽ nào đó của Process, mỗi Thread trong một Process thực hiện nhiệm vụ độc lập với Thread khác trong Process này.
  - Như vậy, một Process sinh ra các Thread, và các Thread này chia sẻ không gian bộ nhớ với nhau và là của Process. Còn các Process không chia sẽ không gian bộ nhớ với nhau.

- Chẳng hạn, khi mở một ứng dụng soạn thảo văn bản; có nghĩa là chúng ta tạo ra một Process.
- Khi bắt đầu nhập, Process sinh ra các Threads chẳng hạn như:
  - đọc các phím bấm
  - hiển thị văn bản
  - tự động lưu tập tin lại
  - làm nổi bật (highlight) các lỗi chính tả khi nhập
- Bằng cách sinh ra nhiều Thread, Chương trình soạn thảo văn bản tận dụng thời gian nhàn rỗi của CPU (chờ đợi các phím bấm hoặc tải tập tin về) qua đó làm cho công việc có năng suất hơn.

- Việc sử dung bộ nhớ: Threads và Processes là khác nhau:
  - Threads dùng shared-memory,
  - trong khi đó Processes thì có thể không.
  - Sự đồng bộ hóa về dữ liệu là cần thiết đối với các Threads,
  - nhưng với Processes là có thể không cần

- Một Process có thể có nhiều Thread
- Process được tạo ra bởi hệ điều hành để thực thi chương trình
- Hai Process có thể thi hành đồng thời một đoạn code trong chương trình Python
- Khi mở và đóng Process tốn nhiều thời gian hơn so với mở đóng Thread
- Trong trường hợp không chia sẻ không gian bộ nhớ, thì việc chia sẻ thông tin giữa các Process chậm hơn giữa các Thread

- Thread như là những tiến trình nhỏ (sub/mini-processes) tồn tại bên trong Process
- Thread chia sẻ không gian bộ nhớ nên các biến được dung chung trong việc đọc ghi
- Hai Threads không thể thực thi đồng thời cùng một đoạn code trong chương trình Python

Ví dụ: tính tổng

- Chẳng hạn cần tính tổng của *N* số thực có giá trị được tạo ngẫu nhiên trong khoảng [0,1).
- Dùng sum() của Python, hoặc có thể viết trực tiếp
- ***summing.py***

```
# Hàm tính tổng có thể viết
def psum( a, ib, ie, result, index ):
    s = sum(a[ib:ie])
    result[index] = s

def nsum( a, ib, ie ):
  s = 0.0
  for i in range(ib,ie):
    s += a[i]
  return s
```

```
# Để tính tổng tuần tự của N số này:
t0 = time.time()
s = nsum(a,0,N)
print( "\nTuần tự: Thời gian trôi qua %8.6f giây" % (time.time()-t0) )
print( "Tổng là %10.2f" %s )

# Dùng sum()
t0 = time.time()
result = [0]
psum(a,0,N,result,0)
print( "Dùng sum(): Thời gian trôi qua %8.6f giây" % (time.time()-t0) )
print( "Tổng là %10.2f" %result[0] )
```

- Khi cần song song, có thể dùng Thread với gói threading

```
# Dùng Thread
t0 = time.time()
threads = []
result = [0]*P  # Lưu các tổng riêng

# Tạo và khởi động threads
for i in range(P):
    ib = i * K
    ie = (i + 1) * K if i != P - 1 else N
    thread = th.Thread(target=psum, args=(a,ib,ie,result,i))
    threads.append(thread)
    thread.start()

# Đợi cho tất cả các thread hoàn thành
for thread in threads:
    thread.join()

print( "Dùng Thread: Thời gian trôi qua %8.6f giây" % (time.time()-t0) )
print( "Tổng là %10.2f" %sum(result) )
```

- Dùng Process với gói multiprocessing

```
# Dùng Process
t0 = time.time()
processes = []

# Dùng Manager để quản lý việc chia sẻ mảng
result = mp.Manager().list([0] * P)

# Tạo và khởi động các Process
for i in range(P):
    ib = i * K
    ie = (i + 1) * K if i != P - 1 else N
    process = mp.Process(target=psum, args=(a,ib,ie,result,i))
    processes.append(process)
    process.start()

# Đợi cho các processes hoàn thành
for process in processes:
    process.join()

print( "Dùng Process: Thời gian trôi qua %8.6f giây" %(time.time()-t0) )
print( "Tổng là %10.2f" %sum(result) )
```

- Hoặc dùng kỹ thuật Pool, với hàm tính tổng được viết riêng

```
def poolsum(a_segment):
    return np.sum(a_segment)
```

```
# Dùng Pool, tạo các Pool
with mp.Pool(processes=P) as pool:
   # Chia mảng thành từng đoạn
   segments = [a[i*K:(i+1)*K] for i in range(P)]

   # Map hàm đến Pool
   partial_sums = pool.map( poolsum,segments )

print( "Dùng Pool: Thời gian trôi qua %8.6f giây" %(time.time()-t0) )
print( "Tổng là %10.2f" %sum(partial_sums) )
```