Parallel Computing - Lecture notes
HPC Lab - CSE - HCMUT
Thoại Nam

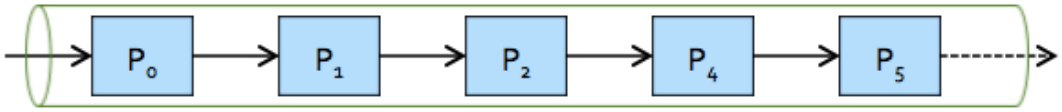
Kỹ thuật tính song song và phân tán

6.1. Tính toán song song biệt lập (Embarrassingly Parallel Computations - EPC)

6.2. Phân vùng - Chia để trị (Partitioning - Divide and conquer Strategies)

6.3. Tính toán dạng ống (Pipelined Computations)

6.3.1. Kỹ thuật tính toán dạng ống



Hình 6.3.1. Kỹ thuật tính toán dạng ống.

Kỹ thuật tính toán dạng ống (pipeline technique) gần giống ý tưởng tính toán tuần tự khi bài toán được phân chia thành một chuỗi các công việc trong đó việc trước truyền thông tin cần cho tính toán ở các việc sau. Trong tính toán song song, mỗi công việc được xem như một *đoạn ống* (*pipeline stage*) và được giải quyết trên một tiến trình hay một bộ xử lý như Hình 6.3.1. Như vậy mỗi đoạn ống tham gia giải quyết một phần việc trong bài toán và truyền thông tin cần thiết cho các đoạn ống sau. Ý tưởng giải quyết vấn đề là phân rã chức năng (functional decomposition). Bài toán được phân rã thành nhiều chức năng cần giải quyết và trong tính toán dạng ống thì các chức năng này được thực hiện theo thứ tự và có tính kế thừa.

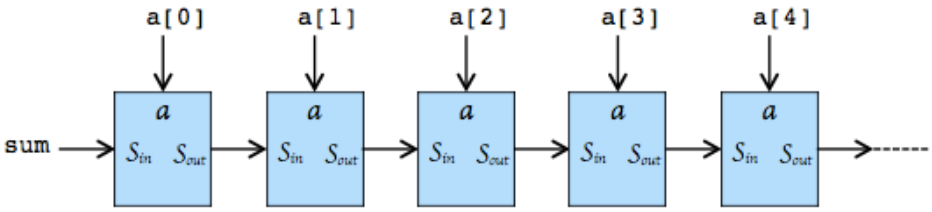
Một ví dụ tính một bài toán toán tuần tự có thể được xử lý theo tính toán dạng ống như đoạn chương trình sau:

```
for (i=0; i<n; i++)
    sum = sum + a[i];
```

Vòng lặp trên thực hiện việc cộng tích luỹ các giá trị `a[i]` vào tổng `sum`. Nhìn theo cách khác thì vòng lặp trên được phân rã thành các bước tính như sau:

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
...
sum = sum + a[n-1];
```

Các bước tính toán này có tính kế thừa vì kết quả của bước tính trước chứa trong biến `sum` được sử dụng lại trong bước tính sau. Cách tính này có thể xem như tính toán dạng ống như Hình 6.3.2 sau:

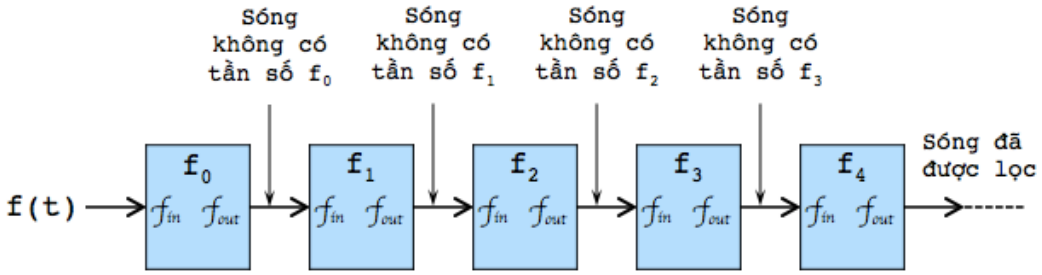


Hình 6.3.2. Bài toán tổng chuỗi bằng tính toán dạng ống.

Mỗi đoạn ống thứ k thực hiện việc cộng giá trị `a[k]` vào biến tích luỹ `sum` có được từ dữ liệu nhập s_{in}; và giá trị tổng `sum` mới này là dữ liệu xuất s_{out}. Vậy đoạn ống thứ k thực hiện:

```
s_out = s_in + a[k];
```

Trong một số bài toán, mỗi đoạn ống có thể thực hiện một hàm tính toán phức tạp hơn. Một ví dụ thực tế là bài toán xử lý tính hiệu số cần phải lọc một số tần số f_0, f_1, f_2, v.v. ra khỏi tín hiệu nhập $f(t)$. Một giải pháp xử lý tín hiệu số sử dụng tính toán dạng ống được trình bày ở Hình 6.3.3 với mỗi đoạn ống đảm trách việc khử một tần số.



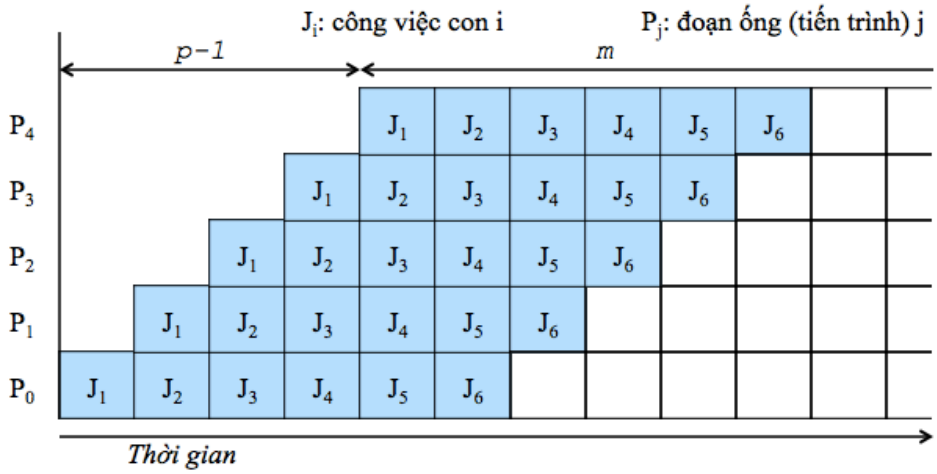
Hình 6.3.3. Tính toán dạng ống dùng lọc tần số.

Khi việc tính toán trong bài toán có thể phân rã thành một số bước tính toán nối tiếp nhau thì kỹ thuật tính toán dạng ống có thể được ứng dụng để tăng tốc tính toán với ba dạng bài toán sau:

- Dạng I: Có nhiều công việc con (instance) với các bước tính toán như nhau cần giải quyết trong bài toán;
- Dạng II: Có một chuỗi mẫu dữ liệu cần xử lý và mỗi mẫu cần xử lý qua nhiều bước;
- Dạng III: Thông tin khởi động một bước tính toán khác có thể được truyền đi trước khi bước tính toán hiện tại hoàn tất việc tính toán của nó.

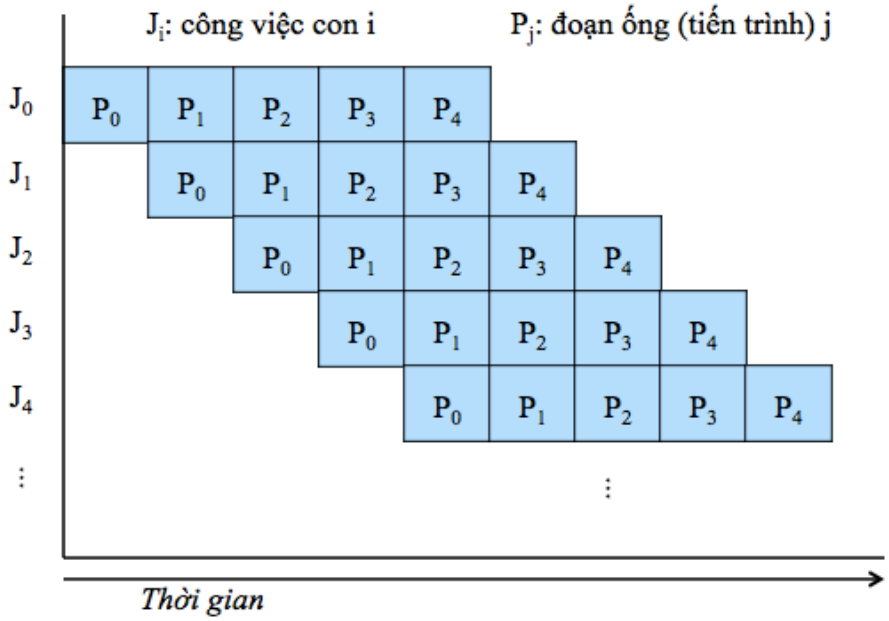
Thiết kế phần cứng tính toán như kiến trúc bộ xử lý hiện tại đều theo dạng tính toán dạng ống để giải quyết các bài toán ở Dạng I. Một chương trình là một bài toán với nhiều công việc con là các lệnh máy. Từng lệnh máy được thực thi trên bộ xử lý với kiến trúc

có nhiều đoạn ống (segment hay stage). Một ứng dụng khác thường thấy ở Dạng I là dạng bài toán quét tham số (parameter sweep application); khi đó một chương trình được chạy lại nhiều lần với các tham số đầu vào khác nhau trên một không gian đa chiều rộng lớn. Mô phỏng Monte Carlo hoặc tìm kiếm trên không gian tham số là một dạng của lớp bài toán quét tham số.



Hình 6.3.4. Biểu đồ 1 về không gian - thời gian trong tính toán dạng ống.

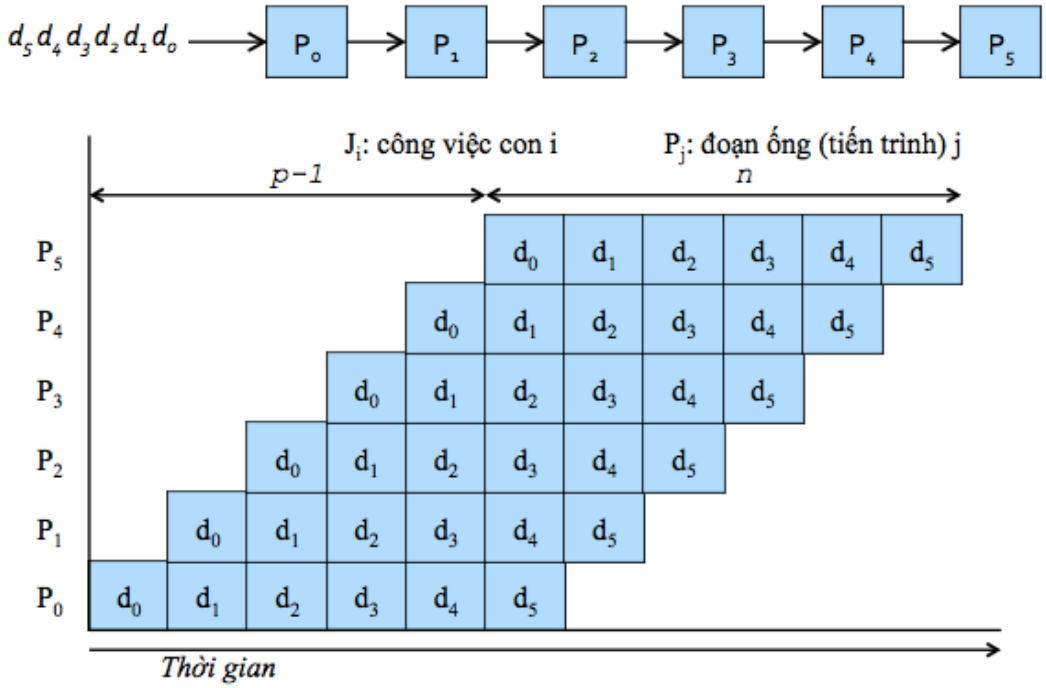
Kỹ thuật tính toán dạng ống theo Dạng I với nhiều tiến trình hay bộ xử lý được mô tả theo biểu đồ không gian-thời gian như Hình 6.3.4. Trong biểu đồ này, mỗi tiến trình đảm trách một bước tính toán tương ứng với một đoạn ống. Giả sử thời gian tính toán ở mỗi đoạn ống là như nhau và được gọi là *chu kỳ ống* (*pipeline cycle*). Mỗi công việc con (instance) trong bài toán giả sử cần năm tiến trình xử lý tuần tự theo thứ tự P_0, P_1, P_2, P_3, và P_4. Hiệu ứng bậc thang xuất hiện trong giai đoạn đầu cho đến khi công việc con đầu tiên hoàn thành sau năm chu kỳ ống. Sau đó mỗi công việc con được hoàn thành trong mỗi chu kỳ ống.



Hình 6.3.5. Biểu đồ 2 về không gian - thời gian trong tính toán dạng ống.

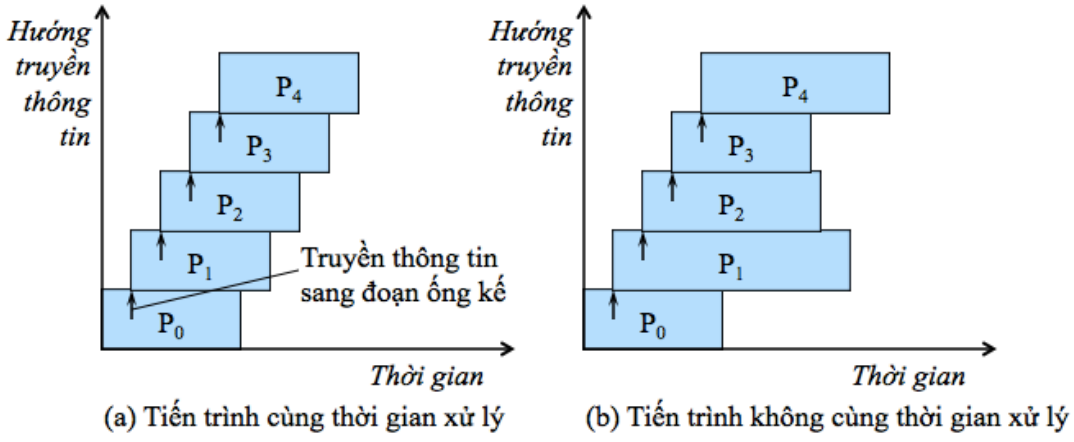
Hình 6.3.5 cho thấy một cách nhìn khác với biểu đồ Hình 6.3.4 khi xếp các công việc con (instance) trên trục tung. Biểu đồ này hữu ích khi chúng ta quan tâm thông tin chuyển giữa các bước tính toán trong mỗi đoạn ống trong một công việc con.

Với p tiến trình trong một ống tính toán (pipeline) và m công việc việc con (instance) trong bài toán cần giải quyết, số *chu kỳ ống* (hay gọi tắt là *chu kỳ*) cần để xử lý m công việc con là $m+p-1$. Trung bình thời gian xử một công việc con là $(m+p-1)/p$ chu kỳ và nó dần đến một chu kỳ khi m đủ lớn. Sau $p-1$ chu kỳ đầu tiên thì mỗi công việc con được hoàn thành trong mỗi chu kỳ. Vì vậy *độ trễ trong ống tính toán* (*pipeline latency*) là $p-1$ chu kỳ.

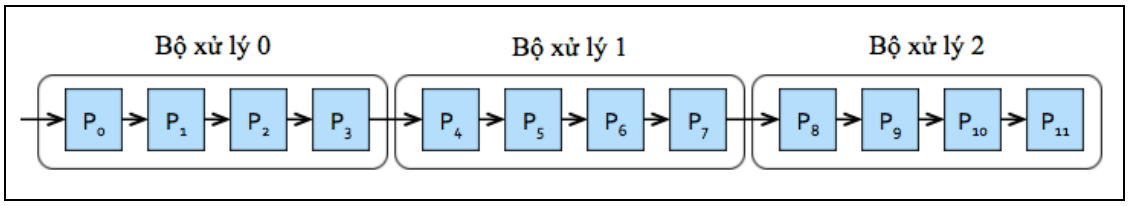


Hình 6.3.6. Tính toán dạng ống xử lý chuỗi dữ liệu nhập.

Trong Dạng II, từng đoạn ống là các phép tính số học xử lý chuỗi phần tử dữ liệu nhập chạy qua nó; ví dụ như tính tổng hay tích các phần tử dữ liệu đi qua nó. Hình 6.3.6 biểu diễn một kiến trúc ống tính toán có sáu tiến trình và sáu phần tử dữ liệu nhập d_0, d_1, d_2, d_3, d_4 và d_5. Với p tiến trình và n phần tử dữ liệu, thời gian thực thi là $(p-1)+n$ chu kỳ với giả thiết thời gian tính toán ở mỗi đoạn ống là như nhau và bằng một chu kỳ.



Hình 6.3.7. Tiến trình truyền thông tin trước khi hoàn thành.



Hình 6.3.8. Chia nhóm tiến trình vào nhiều bộ xử lý.

Một dạng khác cũng có thể sử dụng tính toán dạng ống nhưng bài toán không chia ra làm nhiều công việc con mà chỉ có một việc lớn. Việc này được chia ra nhiều bước tính toán tại các tiến trình theo kiến trúc ống tính toán. Mỗi tiến trình có thể truyền thông tin cho một tiến trình tiếp theo trước khi nó hoàn thành. Đây là Dạng III của tính toán dạng ống được biểu diễn ở Hình 6.3.7. Các tiến trình từ P_0 đến P_5 nối tiếp nhau và tiến trình sau cần thông tin từ tiến trình trước đó để bắt đầu thực hiện theo kiến trúc ống tính toán. Giải pháp truyền thông tin trước khi nó hoàn thành tại mỗi tiến trình cho phép các tiến trình này gối đầu nhau khi cùng giải quyết một công việc nên giảm thời gian hoàn thành công việc. Dạng III không yêu cầu khối lượng tính toán (hay thời gian thực thi) ở mỗi đoạn ống ở từng tiến trình phải bằng nhau.

Nếu số đoạn ống lớn hơn số lượng bộ xử lý thì một nhóm đoạn ống có thể được gán cho một bộ xử lý như ở Hình 6.3.8. Các tiến trình trong một bộ xử lý phải được thực thi tuần tự theo thứ tự.

6.3.2. Bài toán tổng tiền tố (Prefix sums hay Scan)

Một bài toán điển hình trong tính toán song song dữ liệu là bài toán tổng tiền tố. Cho một tập n giá trị $x_0, x_1, \ldots, x_{n-1}$ và và một phép toán kết hợp $\oplus$, bài toán tổng tiền tố trên n giá trị có kết quả như sau:

0: x_0

1: $x_0 \oplus x_1$

2: $x_0 \oplus x_1 \oplus x_2$

...

$n-1$: $x_0 \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1}$

Toán tử $\oplus$ là phép toán kết hợp (associative operation) có tính kết hợp và giao hoán nên có thể là phép cộng (+), phép nhân (.), phép tính lấy giá trị lớn nhất (max), phép tính lấy giá trị nhỏ nhất (min), phép giao luận lý ($\wedge$), phép hợp luận lý ($\vee$), v.v..

Ví dụ: Cho dãy x[] có giá trị là:

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	2	3	4	5	6	7	8

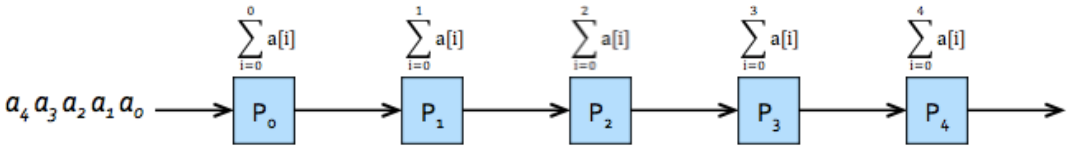
thì tổng tiền tố là:

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	3	6	10	15	21	28	36

```
1.  Sequential_Prefix_sums (n, x[]) {
2.      for (int i = 0; i < n; i++)
3.          x[i] = x[i-1] ⊕ x[i];
4.      return x[];
5.  }
```

Hình 6.3.9. Giải thuật tổng tiền tố tuần tự.

Tổng tiền tố là bài toán cơ bản được sử dụng trong nhiều ứng dụng thực tế như nén dữ liệu, sắp xếp, đánh giá đa thức, v.v.. Giải thuật tuần tự đơn giản giải quyết bài toán tổng tiền tố được trình bày ở Hình 6.3.9. Thời gian chạy giải thuật là O(n).



Hình 6.3.10. Tổng tiền tố dùng tính toán ống.

Một giải pháp tính toán dạng ống (pipeline) khả thi cho bài toán trên là giao một tiến trình trong kiến trúc ống tính toán thực hiện tính toán tổng tiền tố tại một phần tử trong trong dãy số như Hình 6.3.10. Để tính tổng tiền tố chuỗi nhập có n giá trị $x_0, x_1, \ldots, x_{n-1}$ (x_0 ở đầu chuỗi nên đi vào ống tính toán trước), ống tính toán có n đoạn ống tương ứng với n tiến trình P_i ($0 \le i < n$) và mỗi tiến trình có một số định danh là i .

Đoạn chương trình tại P_i như sau:

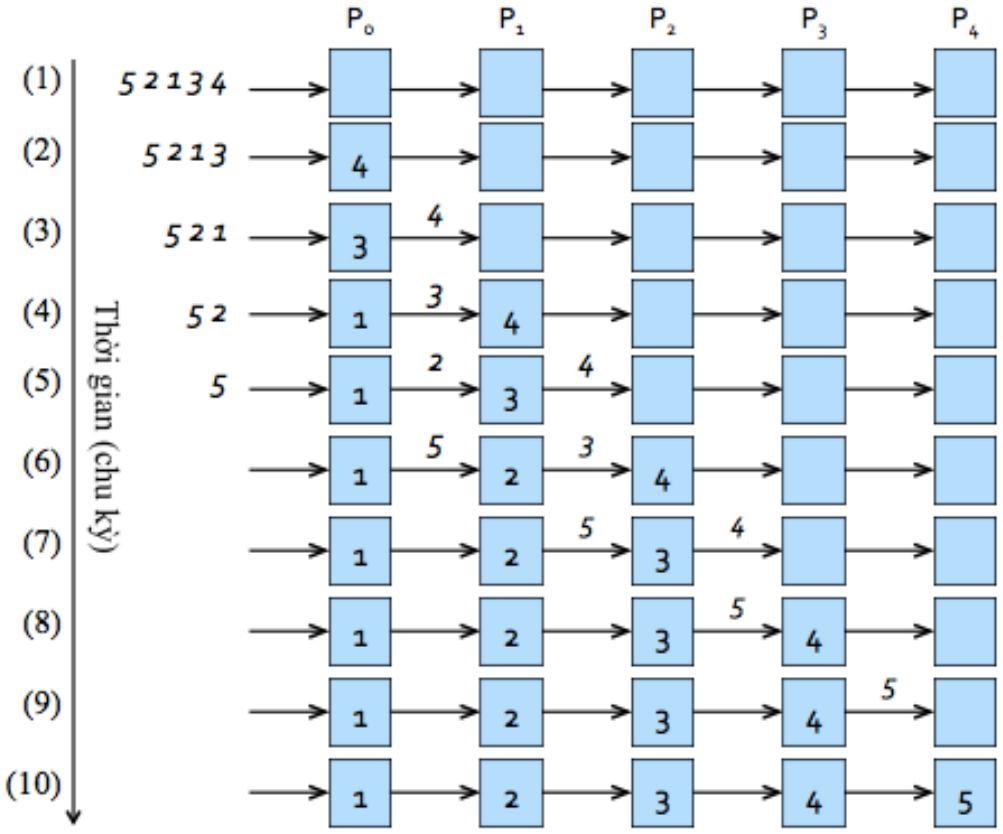
```
   Khởi động:
1.  k = 0;     // số giá trị nhận được
2.  psum = 0; // psum là giá trị tổng tiền tố tính tại Pi

   Tính toán tại đoạn ống khi nhận dữ liệu:
3.  x = sin;  // nhận dữ liệu nhập
4.  if (++k <= i)
5.    psum += x;
6.  sout = x;

   Kết quả:
```

```
7.  return(psum);    // psum = ∑_{k=0}^{i} x_k
```

Chuỗi nhập $x_0, x_1, \dots, x_{n-1}$ lần lượt đi qua mỗi tiến trình P_i trên ống tính toán. Mỗi P_i lần lượt nhận giá trị nhập là x_k như đầu vào s_{in} tại dòng 3 và đẩy ra như đầu xuất s_{out} tại dòng 6 để làm đầu vào s_{in} cho P_{i+1}. Một biến đếm k để đếm số phần tử x_k đã nhận. Nếu k còn bé hơn hay bằng chỉ số định danh i của P_i thì cộng dồn x_k vào biến $psum$ tại dòng 4-5. Sau khi chuỗi nhập $x_0, x_1, \dots, x_{n-1}$ lần lượt đi qua P_i thì $psum=\sum_{k=0}^{i} x_k$ và như vậy chúng ta có kết quả tính tổng tiền tố $\sum_{k=0}^{0} x_k$, $\sum_{k=0}^{1} x_k$,..., $\sum_{k=0}^{n-1} x_k$ tương ứng tại P_0, P_1,..., P_{n-1}.

6.3.3. Sắp thứ tự (Sorting)

Đây là bài toán sắp một dãy số theo thứ tự tăng dần (hay giảm dần) dựa trên giá trị các phần tử là các con số cho phép một số phần tử có cùng giá trị. Không mất tính tổng quát, chúng ta xem xét giải thuật tính toán dạng ống sắp xếp tăng dần.



Hình 6.3.11. Sắp thứ tự dùng tính toán ống.

Ý tưởng giải quyết rất đơn giản: mỗi tiến trình P_i nhận một chuỗi (dãy) số như dữ liệu nhập và nó chỉ giữ lại giá trị nhỏ nhất đã từng nhận và chuyển các số có giá trị lớn hơn sang cho P_{i+1}. Như vậy một dãy số $x_0, x_1, \dots, x_{n-1}$ đi vào ống tính toán này lần lượt từng số sẽ được giữ lại tại P_0 đến P_{n-1} với giá trị từ nhỏ đến lớn. Hình 6.3.11 minh họa

giải thuật sắp thứ tự dãy số. Đây chính là một phiên bản giải thuật song song sắp xếp chèn (insertion sort).

Đoạn chương trình tại P_i như sau:

```
   Khởi động:
1.  flag = 0;  // Chưa có giữ số nào
   Tính toán tại đoạn ống khi nhận dữ liệu:
2.  x = s_in;   // nhận dữ liệu nhập
3.  if (flag == 0) {
4.    p = x;  // p giữ một số trong dãy số
5.    flag = 1;
6.  } else  if (x < p) {
7.            p = x;
8.            s_out = p;
9.          } else s_out = x;
   Kết quả:
10. return(p);
```

6.3.4. Tạo số nguyên tố

Phương pháp cổ điển để tìm các số nguyên tố chính là sàng Eratosthenes có từ hàng ngàn năm trước của nhà toán học Eratosthenes ở Cynere - Hy lạp cổ (Bokhari, 1987). Giải thuật theo phương pháp này như sau:

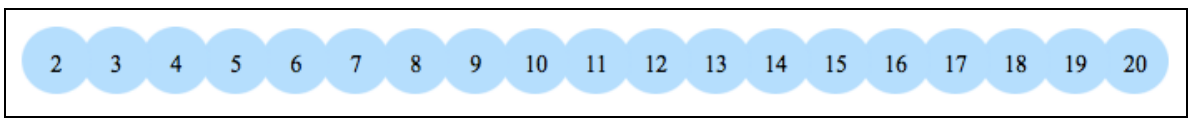
(1) Một chuỗi số bắt đầu 2 được sinh ra: 2, 3, 4, 5, 6, ...

(2) Giá trị đầu tiên 2 là số nguyên tố nên được giữ lại. Tất cả các số là bộ số của 2 được bỏ đi vì không phải số nguyên tố;

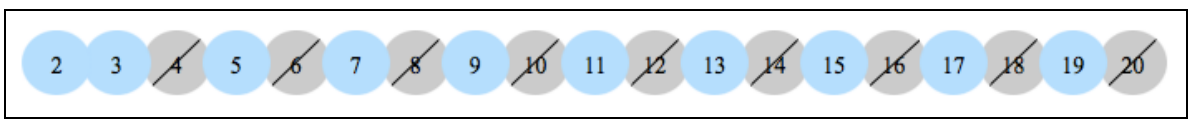
(3) Số kế tiếp là 3 là số nguyên tố nên được giữ lại. Tất cả các số là bội của 3 được bỏ đi vì không phải số nguyên tố;

(4) Quá trình cứ thế tiếp diễn với các số còn lại.

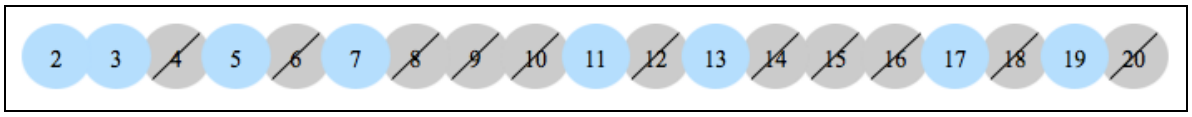
Giải thuật này loại đi các số không phải số nguyên tố và để lại có số nguyên tố. Ví dụ chúng ta cần tìm các số nguyên tố từ 2 đến 20. Chúng ta bắt đầu với dãy số từ 2 đến 20 như sau:



Bắt đầu với 2, chúng ta loại bỏ các bội số của 2:



Số kế tiếp là 3, chúng ta loại bỏ các bội số của 3:



Sau đó chúng ta lần lượt thực hiện tương tự với các số 5, 7, 9, 11, 13, 17, 19 nhưng không có bội số nào trong nhỏ hơn 20 chưa bị loại bỏ. Như vậy các số nguyên tố từ 2 đến 20 cuối cùng tìm được là 2, 3, 5, 7, 9, 11, 13, 17 và 19. Tuy nhiên để tìm các số nguyên tố đến n thì chúng ta chỉ cần bắt đầu với số có đến $\sqrt{n}$ để tìm và loại bỏ bội số của nó. Nhiều số lớn $\sqrt{n}$ đã được loại bỏ trước đó bởi vì chúng là bội số của một số nguyên tố nhỏ hơn $\sqrt{n}$. Ví dụ nếu $n=1024$ thì $\sqrt{n}=32$ nên không cần loại bỏ các bội số của 33 vì (33 x 2) đã được bỏ trước đó bởi (2 x 33), (33 x 3) đã được bỏ trước đó bởi (3 x 33), tương tự đối với các bội số khác lớn hơn 32. Vì vậy trong ví dụ tìm các số nguyên tố không lớn hơn 20 chỉ cần loại bỏ các bội số của 2 và 3 là đủ. Bước loại bỏ các bội số 5, 7, 9, 11, 13, 17, 19 là không cần thiết.

Phương pháp trên không thích hợp để tìm các số nguyên tố lớn bởi vì thời gian tính toán tuần tự lớn do nhiều lần quét bội số của những số nguyên tố nhỏ. Một cách cải tiến là chỉ xét những số lẽ vì số chẵn ngoại trừ 2 đều không phải là số nguyên tố.

Giải thuật tuần tự hiện thực sàng Eratosthenes như sau:

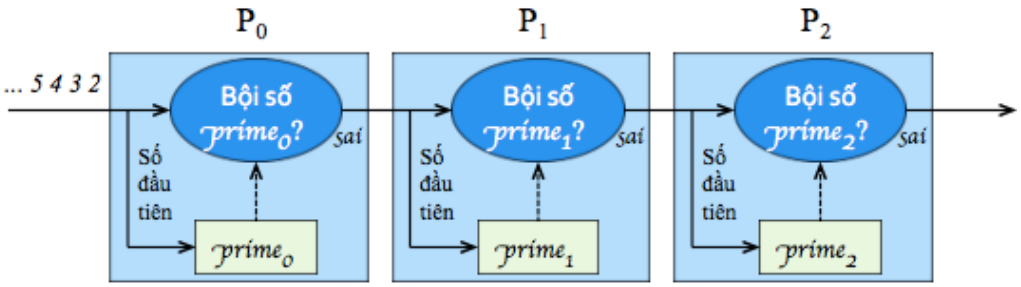
```
  // Khởi tạo xem tất cả là số nguyên tố
1.  for (i=2; i<=n; i++)
2.    prime[i] = 1;
  // Loại bỏ các bội số của các số nguyên tố từ 2 đến √n
3.  for (i=2; i<=sqrt(n); i++)
4.    if (prime[i] == 1)
5.      for (j=i+i; j<=n; j=j+i)
6.        prime[j] = 0;
```

Khởi đầu xem tất cả các số từ 2 đến n đều có thể là số nguyên tố tại dòng 1-2. Cho i bắt đầu từ 2 đến $\sqrt{n}$, nếu i là số nguyên tố thì loại bỏ tất cả các bội số của i không lớn hơn n.

Số lần lặp để loại bỏ các bội số của một số nguyên tố ở dòng 3-4 phụ thuộc vào số lượng số nguyên tố; chi phí tính toán trong mỗi vòng lặp ở dòng 5-6 phụ thuộc giá trị của số nguyên tố. Có $\lfloor n/2 - 1 \rfloor$ bội số của 2, $\lfloor n/3 - 1 \rfloor$ bội số của 3, v.v.. Vì vậy tổng thời gian tính toán tuần tự là:

$$T_s = \lfloor n/2 - 1 \rfloor + \lfloor n/3 - 1 \rfloor + \lfloor n/5 - 1 \rfloor + \ldots + \lfloor n/\sqrt{n} - 1 \rfloor$$

với giả sử tính toán tại mỗi vòng lặp là một đơn vị thời gian. Độ phức tạp thời gian của giải thuật tuần tự là $O(n^2)$. Một bội số của 2 là 30 cũng có thể là bội số của 3 và 5 nên Quinn (1994) đã phân tích hiệu quả của cách giải quyết của sàng Eratosthenes.



Hình 6.3.12. Tìm số nguyên tố dùng tính toán ống.

Phương pháp tìm các số nguyên tố sử dụng sàng Eratosthenes rất phù hợp với kỹ thuật tính toán dạng ống so với các những kỹ thuật song song khác. Hình 6.3.12 biểu diễn ý tưởng của kỹ thuật tính toán dạng ống giải quyết bài toán này.. Một dãy số liên tục bắt đầu từ 2 (2, 3, 4, 5,...) đi vào ống tính toán. Từng đoạn ống tương ứng với tiến trình P_i sẽ sẽ phụ trách giữ lại số nguyên tố thứ i và loại bỏ các bội số của nó; các số lớn không phải bội số của số nguyên tốt thứ i được truyền sang P_{i+1}. Giải thuật tính toán tại P_i như sau:

```
   Khởi động:
1.  prime = 0;  // Chưa có giữ số nào nguyên tố nào

   Tính toán tại đoạn ống khi nhận dữ liệu:
2.  x = s_in;   // nhận dữ liệu nhập
3.  if (prime == 0)
4.    prime = x;  // Giữ lại số nguyên tố thứ i+1
   // Loại bỏ bội số của số nguyên tố thứ i+1 (prime)
5.  else  if ((x % prime) != 0)
6.            s_out = x;

   Kết quả:
7.  return(prime);
```

Như ở Hình 6.3.12, khi số 2 đến P_0 thì được giữ lại như số nguyên tố thứ nhất tại dòng 3-4. Các số đến sau 3, 4, 5, 6,... sẽ bị loại bỏ nếu là bội số của 2 ở dòng 5-6 và ngược lại thì được đẩy ra như đầu xuất s_{out} tại dòng 6 để làm đầu vào s_{in} cho P_1. Như vậy vai trò của P_0 là giữ lại số nguyên tố thứ nhất là 2 và loại bỏ các bội số của 2. Chuỗi nhập cho P_1 là 3, 5, 7, 9,... (khác chuỗi nhập tại P_0) không có bội số của 2 và số đầu tiên 3 là số nguyên tố thứ hai. P_1 giữ lại số nguyên tố thứ hai là 3 và loại bỏ các bội số của 3; các số không bị loại bỏ là chuỗi nhập cho P_2. Quá trình cứ thế tiếp diễn nên lần lượt các số nguyên tố được 2, 3, 5, 7,... được giữ tại P_0, P_1, P_2, P_4,...

Việc kiểm tra bội số tại dòng 5 có chi phí cao (điều này được loại trừ trong giải thuật tuần tự ở dòng 5-6). Các bạn có thể có nghĩ ra biện pháp khác loại bỏ bội số với chi phí thấp hơn. Một vấn đề khác là việc gán các tiến trình P_0, P_1, P_2, P_4,... vào các bộ xử lý.

Chúng ta có thể gán một tiến trình vào một bộ xử lý hoặc gán một nhóm tiến trình vào một bộ xử lý đều không ảnh hưởng đến giải thuật tính toán dạng ống nêu trên.

6.3.5.Giải phương trình tuyến tính

Đây là một bài toán có thể giải quyết bằng dạng thứ III trong kỹ thuật tính toán dạng ống; nghĩa là một tiến trình vẫn tiếp tục tính toán sau khi truyền thông tin cho tiến trình kế cận. Hệ phương trình tuyến tính cần giải quyết có dạng *ma trận tam giác trên* (*upper-triangular*) như sau:

$$
\begin{aligned}
&a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \quad \ldots \quad + a_{n-1,n-1}x_{n-1} = b_{n-1} \quad (n\text{-}1)\\
&\ldots\\
&a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = b_2 \quad (2)\\
&a_{1,0}x_0 + a_{1,1}x_1 = b_1 \quad (1)\\
&a_{0,0}x_0 = b_0 \quad (0)
\end{aligned}
$$

trong đó a, b là các hằng số và x là ẩn số. Phương pháp giải để tìm nghiệm x_0, x_1, x_2, … , x_{n-1} là lần lượt tìm và thay thế các nghiệm đã có vào các phương trình từ (*0*) đến (*n-1*). Đầu tiên, nghiệm x_0 có thể tìm từ phương trình (*0*):

$$x_0 = \frac{b_0}{a_{0,0}},$$

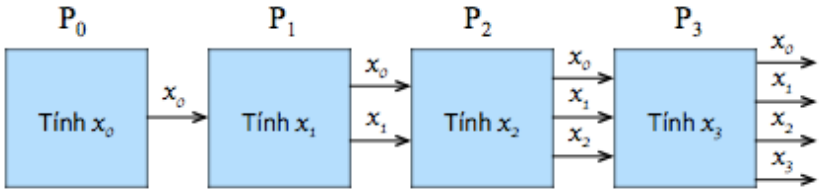
nghiệm x_0 được thế vào phương trình (*1*) để tìm x_1:

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}},$$

và nghiệm x_0 và x_1 được thế vào phương trình (*2*) để tìm x_2:

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}},$$

và cứ thế tiếp tục tìm tất cả các nghiệm.



Hình 6.3.13. Giải hệ phương trình dùng tính toán ống.

Giải thuật trên thể hiện rõ ý tưởng của một ống tính toán (pipeline) và được biểu diễn như Hình 6.3.13. Đoạn ống đầu tiên tương ứng với tiến trình P_0 tính toán tìm nghiệm x_0

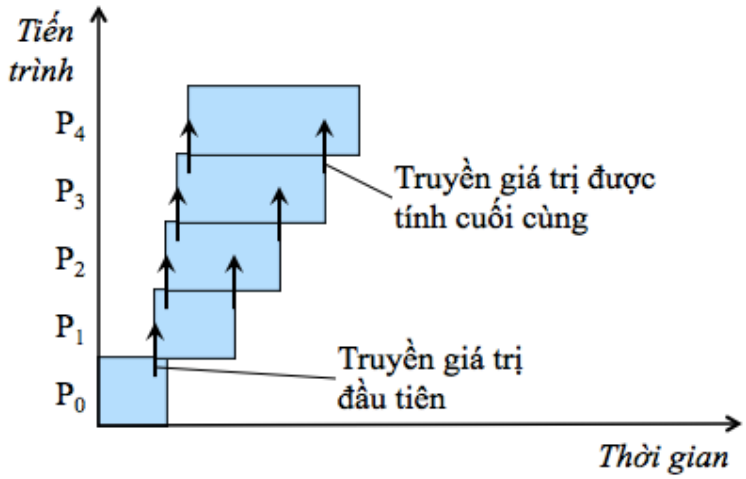
và truyền x_o sang P_1 để tính x_1 từ x_o, sau đó x_o và x_1 được truyền sang P_2 để tính x_2 từ x_o và x_1 và quá trình cứ thế tiếp tục để tìm tất cả các nghiệm của hệ phương trình. Để tính tất cả n nghiệm thì cần n tiến trình. Tiến trình P_0 tìm nghiệm x_o trực tiếp từ phương trình (o). Các tiến trình P_i ($0 < i < n$) nhận các nghiệm x_o, x_1, x_2, ... , x_{i-1} từ P_{i-1} và tính nghiệm x_i từ phương trình thứ (i):

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j}x_j}{a_{i,i}}.$$

Giải thuật tính toán tại P_i như sau:

```
   Khởi động:
1.  sum = 0;

   Tính toán tại đoạn ống khi nhận dữ liệu:

   // Nhận dữ liệu nhập x0, x1, x2, ... , xi-1
2.  for (j=0; j<i; j++) {
3.    x[j] = sin;   // Nhận dữ liệu
4.    sout = x[j];  // Truyền x[j] sang cho Pi+1
5.  }

   // Tính nghiệm xi (x[i]) từ phương trình thứ (i)
6.  for (j=0; j<i; j++)
7.    sum = sum + a[i][j]*x[j];
8.  x[i] = (b[i] - sum)/a[i][i];
9.  sout = x[i];  // Truyền x[i] (nghiệm xi) sang cho Pi+1

   Kết quả:
10. return(x[i]); // Nghiệm xi
```

Tiến trình P_i lần lượt nhận được `x[j]` ($0 \le j < i$) từ P_{i-1} thì chuyển ngay cho P_{i+1} ở dòng 2-5 trước khi thực hiện các tính toán bên trong P_i. Đây chính là Dạng III của tính toán dạng ống. Giải pháp này có nhược điểm là P_i phải đợi nhận tất các `x[j]` với $0 \le j < i$ từ P_{i-1} thì mới thực hiện việc tính toán `x[i]`. Vì vậy P_i luôn trì hoãn việc truyền `x[i]` sau khi đã truyền `x[j]` ($0 \le j < i$) do phải thực hiện việc tính toán ở dòng 6-8. Việc này làm các tính toán tại P_i và P_{i+1} không thể thực hiện đồng thời. Tính toán Dạng III của tính toán dạng ống chưa phát huy hiệu quả.



Hình 6.3.14. Giải hệ phương trình dùng tính toán ống Dạng III.

Một giải thuật tính toán tại P_i cải tiến như sau:

```
   Khởi động:
1.  sum = 0;

   Tính toán tại đoạn ống khi nhận dữ liệu:

   // Nhận dữ liệu nhập x0, x1, x2, ... , xi-1
2.  for (j=0; j<i; j++) {
3.    x[j] = sin;    // Nhận dữ liệu
4.    sout = x[j];  // Truyền x[j] sang cho Pi+1
5.    sum = sum + a[i][j]*x[j];
6.  }

   // Tính nghiệm xi (x[i]) từ phương trình thứ (i)
7.  x[i] = (b[i] - sum)/a[i][i];
8.  sout = x[i];  // Truyền x[i] (nghiệm xi) sang cho Pi+1

   Kết quả:
9.  return(x[i]); // Nghiệm xi
```

Việc tính toán nghiệm `x[i]` được thực hiện từng bước mỗi khi nhận được `x[j]` ($0 \le j < i$) ở dòng 5 nên việc tính toán tại P_i và P_{i+1} có thể thực hiện đồng thời. Ý tưởng này được thể hiện ở Hình 6.3.14.

Phân tích hiệu suất: Với giải thuật tính toán dạng ống cải tiến trên, chi phí tính toán tại mỗi P_i là khác nhau như Hình 6.3.14. Tiến trình P_0 chỉ thực hiện phép tính chia và

một thao tác gởi (gọi là hàm `send()`). Tiến trình P_i ($0 \leq i < n-1$) thực hiện i thao tác nhận (gọi là hàm `recv()`), i hàm `send()`, i phép tính nhân/chia, một phép tính chia/trừ, và một hàm `send()` ở cuối. Giả sử các phép tính cộng, trừ, nhân và chia có cùng thời gian tính toán là một đơn vị tính toán (u_{comp}); và thao tác gởi, nhận tương ứng các hàm `send()/recv()` cùng thời gian một đơn vị truyền nhận (u_{comm}). Như vậy P_i thực hiện tất cả $(2i+1)u_{comm}$ và $(2i+2)u_{comp}$. Tiến trình P_{n-1} thực hiện tất cả $(n-1)u_{comm}$ và $(2n-1)u_{comp}$. Trường hợp này chúng ta xem như đạt một chơ chế đồng bộ lý tưởng giữa `send()` và `recv()` và tất cả tiến trình thực thi song song và thời gian tính toán song bằng thời gian tính toán ở tiến cùng cuối cộng thêm $(p-1)u_{comm}$ (`send()`) và một u_{comp} (phép chia để tính x_o).

Độ phức tạp của giải thuật song song trên là $O(n)$ khi có số lượng bộ xử lý bằng n. trong khi giải thuật tuần tự có độ phức tạp là $O(n^2)$. Độ tăng tốc thực tế không thể là n vì phụ thuộc nhiều về hệ thống máy tính vật lý. Thường chi phí u_{comm} cao hơn chi phí tính toán u_{comm}. Để giảm chi phí truyền nhận, cơ chế truyền nhận *không bị chặn* (*nonblocking*) để tiến trình gởi tiếp tục công việc tính toán nhanh nhất có thể.