The Effect of Architecture on Parallel Programming

Arvind
Laboratory for Computer Science
MIT

A Tension

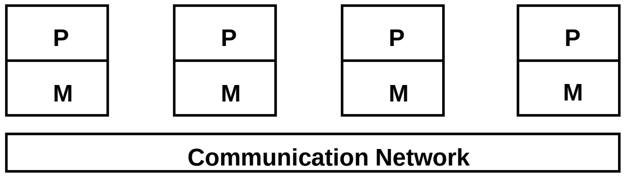
Parallel programming should be machine independent, so that

- **programs can run on a variety of machines**
- **software costs can be amortized**

However, *raison d'être* for parallel programming is high performance

$\Rightarrow$ **If a language cannot be compiled efficiently for widely available parallel machines then the language is unlikely to succeed.**

Differences in Parallel Architectures



Shared Memory
- *Cache-coherent*
  - Symmetric Multiprocessors (SMP)
    - memory bandwidth & latency, locks
- *Not Cache-coherent*
  - Cray T-3E

Distributed Memory, Message passing
- IBM SP-2
- Clusters of PC's and SMP's
  - message passing latency & bandwidth

Exotic features : presence bits, barriers, multithreading

System Software must be taken into account...

because often it can be used to overcome hardware deficiencies

- *global address spaces can be emulated*
  word vs. cache line vs. page granularity

- *split-phase transactions can be emulated*
  via multithreading in software

- *message passing can be moved from kernel to user space* to reduce overhead

- compiler can
  - *enhance locality* of memory references
  - *reduce the need for synchronization*
  - *deal with different cache-coherent models*

Architecture *Affects* the programming model

As an illustration, consider *Matrix Multiply* on

- **Shared memory machines (SMP's)**
  - ***Blocking transformations to enhance locality***
  - ***static vs. dynamic work distribution***
- **Distributed memory machines**
  - ***synchronous vs. asynchronous message passing***

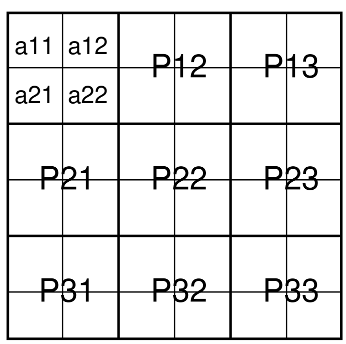
After studying the algorithmic issues we will consider the programming issues

- ***Message passing libraries - PVM, MPI, ...***
- ***Data parallel Fortran***

Matrix Multiply

All inner products can be computed in parallel

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$



b = 2
n = 3

Assume n^2 Processors and matrices of size bn x bn
Let processor $P_{i,j}$ compute the $(I,J)^{th}$ block of C.

Matrix Multiply on SMP's

A program for processor $P_{I,J}$

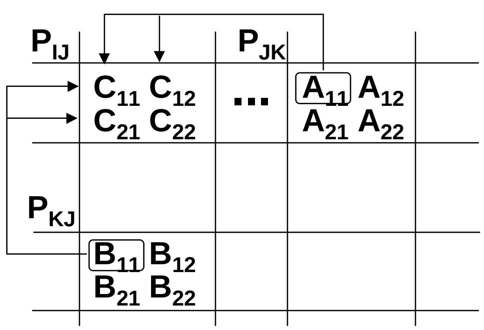
```
for i' = 1 to b do
  for j' = 1 to b do
    i = (I-1)*b+i';
    j = (J-1)*b+j';
     for k = 1 to n*b do
      C[i,j] = C[i,j] + A[i,k]*B[k,j]
     end;
  end
end
```

Inner Product

The *performance* depends on the latency of fetching matrix elements (e.g., `A[i,k]`).

How many times each element is used by a processor?

Communication Patterns in *Matrix Multiply*

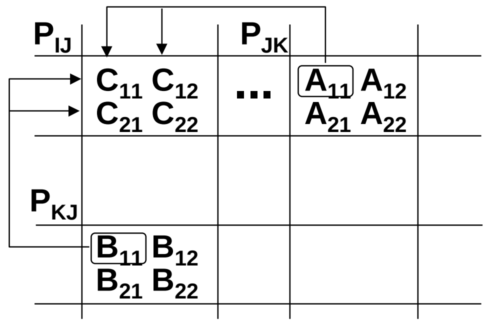


block size
b = 2

If the matrix is too large, `A[i,k]` may be thrown out of the cache before it can be reused

⇒ ***Transform the program to enhance temporal locality***
Blocked matrix multiply

Blocked *Matrix Multiply*



block size
b = 2

The idea

Replace b remote fetches of an element by 1 remote fetch and (b-1) local fetches

$\Rightarrow$ ***Interleaving the computation of*** **C_{11}, C_{12}, C_{21} *and* C_{22}**

Transformation to Interleave *Inner products* step 1

Program for processor $P_{I,J}$

```
for i' = 1 to b do
  for j' = 1 to b do
     i = (I-1)*b+i';
     j = (J-1)*b+j';
     for K = 1 to n do
       for k' = 1 to b do
         k = (K-1)*b + k';
         C[i,j] = C[i,j] + A[i,k]*B[k,j]
       end;
     end;
  end
end
```

Inner Product

Loop splitting

For each K, the k'- loop fetches b elements of A and B each. These elements are also needed to compute the IP's for each (i',j') index.

Transform the program such that these other IP's are also partially computed before K is increased

Transformation to Interleave *Inner products* step 2

Program for processor $P_{I,J}$

```
for K = 1 to n do
  for i' = 1 to b do
    for j' = 1 to b do
      i = (I-1)*b+i';
      j = (J-1)*b+j';
      for k' = 1 to b do
        k = (K-1)*b + k';
        C[i,j] = C[i,j] + A[i,k]*B[k,j]
      end;
    end;
  end
end
```

Partial Inner Product

Loop Interchange

b references to `A[i,k]` are now *temporally closer*

By loop unrolling b-1 out of b references can be turned into local references

Blocked Computations

Transforms some global memory references into local memory references

Essential for good performance on all machines with caches

Work Distribution

```
forall I = 1 to n do
   forall J = 1 to n do
      Blocked Inner Product
   end
end
```

Static - **Iterations of `forall` loops are assigned to processors in a predetermined manner. Usually a *master processor* distributes the work to *worker processors*.**

Dynamic - **Each iteration of a `forall` loop is considered to be a *parallel thread*, which is assigned dynamically to a processor. Since the thread itself can be a `forall` loop, it may spawn more threads. ⇒ *multithreading***

Dynamic Work Distribution

Generally it is difficult to relate thread mapping and data mapping, e.g.,

The processor that computes the (i,j)th inner product may not own C[i,j]

**⇒ *synchronization* may be needed to manipulate a data structure, e.g.,
C[i,j] should be read and modified *atomically***

Matrix Multiply on Distributed Memory Multicomputers

What is *wrong* with the following program for processor P_{IJ}?

```
for i' = 1 to b do
  for j' = 1 to b do
    i = (I-1)*b+i';
    j = (J-1)*b+j';
    s = 0;
      for k = 1 to n*b do
        s = s + A[i,k]*B[k,j]
      end;
    C[i,j] = s;
  end
end
```

Most fetches are non local

A Multicomputer does not provide a direct way of addressing global memory, i.e., `A[i,k]`, etc.

⇒ message passing

Message Passing

Dominant models:

1. *Synchronous communication -*
CSP- style blocking sends and receives

send(dest-proc, local-address, size)
`receive`(source-proc, local-address, size)

2. *Asynchronous communication -*
non blocking sends,
FIFO order between a pair of ports or processes
⇒ a *demon to receive messages*
or *polling* by all application programs

Thorny issues - **matching sends and `receives`,**
- avoiding deadlocks
- buffer overflows,...

Matching Sends and Receives

MM revisited - a possible program on processor Pij

```
for k = 1 to n do
   receive(Pik, a)
   receive(Pkj, b)
   s = s + a * b
end                    ???
```

Makes sense only if

- **Processor Pik (the owner of `A[i,k]`) executes** `send(Pij, A[i,k])`
- **Processor Pij symmetrically sends `A[i,j]`to the processor(s) that need it.**

Can a compiler compute the *inverse* of the `receive` pattern?

In general, *NO !!!*

Matrix Multiply: On a Multicomputer

Program on Pij to collect row `i` and column j of matrix A
Assume processors are connected in 2D grid.

```
Arow[j] = A[i,j];
Bcol[i] = B[i,j];
   for ij = 1 to n-1 do
      send(left, Arow[j]);
      send(  up, Bcol[i]);
      i = (i+1) mod n;
      j = (j+1) mod n;
      receive(right, Arow[j]);
      receive( down, Bcol[i]);
   end
```

The *correctness* of this program depends on the global behavior of sends and `receives`.

Can sends cause buffer overflows?
Can this program deadlock?
Can `Arow` and `Bcol` be eliminated?

A Systolic *Matrix Multiply*

Initial Setup

$$\begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{array} \qquad \begin{array}{cccc} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \\ \hline \end{array}$$

1. Skew row i left by i-1

2. Skew column j up by j-1

$$\begin{array}{cccc} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{22} & A_{23} & A_{24} & A_{21} \\ A_{33} & A_{34} & A_{31} & A_{32} \\ A_{44} & A_{41} & A_{42} & A_{43} \end{array} \qquad \begin{array}{cccc} B_{11} & B_{22} & B_{33} & B_{44} \\ B_{21} & B_{32} & B_{43} & B_{14} \\ B_{31} & B_{42} & B_{13} & B_{24} \\ B_{41} & B_{12} & B_{23} & B_{34} \end{array}$$

3. Multiply element by element

$P_{2,2}$ $\boxed{A_{23} * B_{32}}$

A Systolic *Matrix Multiply*

Repeat steps 1 to 4 n-1 times

1. Shift A element left 1
2. Shift B element up 1
3. Multiply
4. Add

$P_{2,2}$ $\boxed{A_{23} * B_{32} \;,\; A_{24} \;,\; B_{42}}$

$P_{2,2}$ $\boxed{A_{23} * B_{32} + A_{24} * B_{42}}$

- Uses the associativity of +
- Avoids local storage for `Arow` and `Bcol`

Algorithm also works for blocked matrices

Systolic Matrix Multiply on Multicomputers

After the initial set up, each processor executes the following program:

```
s = a*b
for k = 1 to n-1 do
  send(left,a)
  send(up,b)
  receive(right,a)
  receive(down,b)
  s = s + a*b
end
```

where a, b, c and s represent local submatrices and + and * represent matrix operators.

Programming Tools

- ***Message passing libraries (PVM, MPI)***
  - Help the user to avoid learning machine -specific details of message passing
  - Provide portability because most vendors have committed to supporting them
  - Errors are common
  - The usage is generally restricted to SPMD model
- ***Data parallel languages (Fortran 90, HPF)***
  - Make message passing implicit, i.e., the user does not have to insert sends & receives
  - Much easier to use than PVM or MPI

performance?

PVM: Parallel Virtual Machine

Usually implemented on top of an operating system.

Supported on all types of parallel computers including networks of workstations.

The model

- *Tasks (coarse grain) and FIFO pipes*
- *Master and worker tasks*
- *Primitives for*
  - *- Spawning worker tasks*
  - *- Sending and Receiving messages*

PVM Primitives

General:

mytid(), parent(), exit()
spawn(code,args,where,no_pes,w_tids)
    returns worker task ids

Sending a message:

initsend(...)
    starts composing the “current outgoing msg.”
pkint(pointer to data, count, stride)
    packs data into the outgoing message
send(tid,tag), mcst(w_tids,tag)

Receiving a message:

recv(tid,tag)
    receives a message from task tid (with tag)
    to produce the “current input message”
upkint(pointer to data, count, stride)
    unpacks data from the incoming message

Systolic Matrix Multiply in PVM

n^2 worker tasks are set up by the master; each worker is given the tid's of the worker to its left, right, up and down

```
s = a*b
for k = 1 to n-1 do
  send(left,a)
  send(up,b)
  receive(right,a)
  receive(down,b)
  s = s + a*b
end
```

The sends and `receives` in the above code can be easily translated into PVM primitives.
+ and * become local loop operations.

Data Parallel Fortran

Data distribution specifications on a virtual grid

```
Real Array(N,N,N) :: A,B
Layout A(:NEWS, :NEWS, :Serial)
Layout B(:NEWS, :NEWS, :Serial)
```

The first two dimensions of arrays A and B are spread on a 2D grid of virtual processors while the elements in the third dimension are kept local to a processor.

Data movement is specified by shift operations

```
B(:,:,k) = CShift(A(:,:,1),Dim=2,Shift=k-1)
```

means `B(i, j, k)` will get `A(i, (j+k-1) mod N, 1)` for all `i` and `j`

Data Parallel *Systolic Matrix Multiply*

```
Real Array(N,N) :: A, B, C
Layout A(:NEWS, :NEWS), B(:NEWS, :NEWS)
       C(:NEWS, :NEWS)

A = CShift(A,Dim=2,Shift=[0:N-1:1])!skew
B = CShift(B,Dim=1,Shift=[0:N-1:1])!skew
C = A * B

Do i = 1,N-1
 A = CShift(A,Dim=2,Shift=1) !Shift rows left
 B = CShift(B,Dim=1,Shift=1) !Shift cols up
 C = C + A * B
End Do
```


The state of parallel programming is not very good !

- **Parallel programming is usually done only by experts**

- **Programming is easier on**
  shared memory machines
  ***than* distributed memory machines.**

- **Decent tools are available for data parallel or SPMD programming. However, the model is not suitable for many applications**

next multithreading ...