Shared Memory Programming

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

Shared Memory Matrix Multiplication

Problem

$$C = A \times B$$

Sequential algorithm

```
for i = 1 to n do
  for j = 1 to do
    C[i,j] = 0
    for k = 1 to n do
      C[i,j] = C[i,j] + A[i,k]*B[k,j]
    end
  end
end
```

Which loop to parallelize?

- Outer loop, inner loop, both loops?
- Interchange i and j?
- How to map workload to processors?

Nested Loop Parallelization

- Parallelization of inner loop.
  - $O(n)$ processes.
  - n synchronizations (one per iteration).
  - grain size $O(n^2)$.
- Parallelization of outer loop.
  - $O(n)$ processes.
  - 1 synchronization.
  - grain size $O(n^3)$.
- Parallelization of both loops.
  - $O(n^2)$ processes.
  - 1 synchronization.
  - grain size $O(n)$.

Which strategy to choose?

Design Goals

1. Maximize parallelism.

   Large number of processes.

2. Optimize load balancing.

   Small grain size of processes.

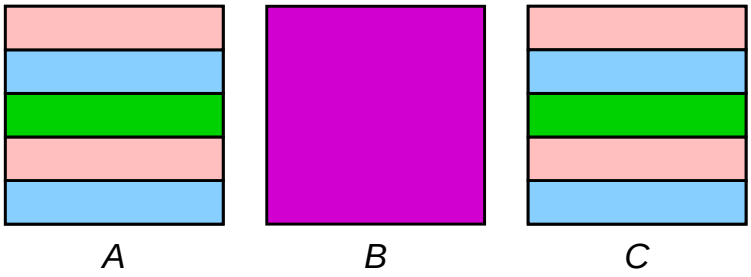
3. Minimize overhead.

   - Small number of processes.
   - Large grain size of processes.
   - Few synchronizations.

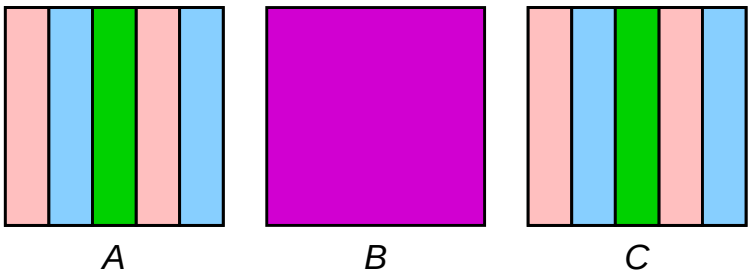
If load balancing is not a problem, maximize grain size.

Loop Interchange

- Parallelization of outer loop.
- Row-major layout of matrices (C).
- Outer loop i
  - Rows of A, C distributed among processors.
  - B accessed colum-wise by every processor.



- Outer loop j
  - Columns of A, C distributed among processors.
  - B accessed row-wise by every processor.

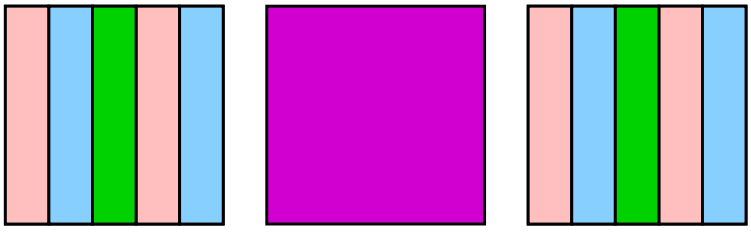


Is there any difference?

Mapping Strategies

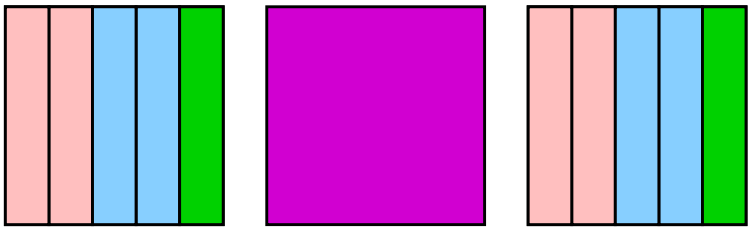
- Cyclic mapping

  Each procesor is in charge of every p-th rows/columns.



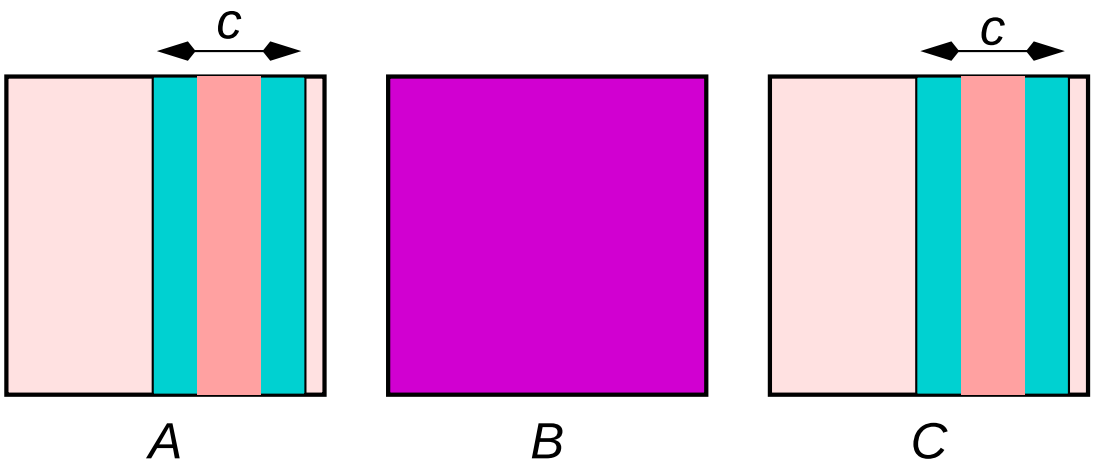
- Block-wise mapping

  Each processor is in charge of a consecutive "block" of $\lfloor n/p \rfloor$ rows/cols.



If workload depends on position of data, cyclic mapping achieves better load distribution.

Cyclic Distribution of Columns



- Memory accessed in form of cache lines of c words each.
- Each processor actually reads cn/p columns of A and C.
- But each procesor also modifies C.
- *False sharing* $\rightarrow$ many cache conflicts on C!

Inefficient solution!

Row Based Cyclic Distribution

```
for all P_m do in parallel
  for i = m to n by p do
    for j = i to n do
      C[i,j] = 0
      for k = 1 to n do
        C[i,j] = C[i,j] + A[i,k]*B[k,j]
      end
    end
  end
end
```

- Synchronization overhead $O(p)$
- Total complexity $O(n^3/p + p)$

Memory contention is ignored in this analysis!

Memory Access Analysis

Each process

- Read n/p rows of A.

  n^2/p words.
- Read every word of B

  n^2 words.
- Read/write n/p rows of C.

  n^2/p words.
- $M = n^2 + 2n^2/p$ memory area used.
- $C = 2n^3/p$ computations performed.
- $C/M = 2n/(p+2)$

Ratio computation/memory area $= O(n/p)$.

Block Matrix Multiplication

Decomposition of matrices in submatrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} =$$

$$\begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

- Submatrices of size $k \times k$.
- $p = (n/k)^2$ processes.
- Per process $M = 5k^2$ memory area used.
- Per process $C = 2k^3$ computations.
- Ratio $C/M = \frac{2}{5}k = \frac{2}{5}n/\sqrt{p}$.

Ratio computation/memory $= O(n/\sqrt{p})$.

Memory Access

- Row algorithm has computation/memory ratio $O(n/p)$.
- Block algorithm uses computation/memory ratio $O(n/\sqrt{p})$.
- Block algorithm has higher *data locality*.
- Cache performance of algorithm improves.
  - Large input matices.
  - Row algorithm: subsequent accesses to B cannot be cached $\rightarrow O(n^3/p)$ memory operations.
  - Block algorithm: subsequent accesses to B can be cached $\rightarrow O(n^3/pc)$ memory operations.
- Important especially for *distributed shared memory* architectures.

Reduce average memory latency time by increasing locality.

Shared Memory Synchronization

Problem: need *mutual exclusion* on shared data structures

```
int stack[];
int sptr = 0;
P: loop
     v = produce()
     stack[sptr] = v  | C1
     sptr++           |
C: loop
     sptr--           | C2
     v = stack[sptr]  |
     consume(v)
```

- C_1 and C_2 are *critical regions*.
- Execution must not overlap.
- Must be executed *atomically*.

Synchronization Mechanisms

Lock variable ("mutex")

```
mutex lock = INIT();
P: loop                    C: loop
     v = produce()              LOCK(lock)
     LOCK(lock)                 C2
     C1                         UNLOCK(lock)
     UNLOCK(lock)               consume(v)
```

- Shared variable
- States: "locked" and "unlocked"
- If locked, `LOCK()` blocks until unlocked.

Only one process can have lock $\Rightarrow$ only one process can enter critical region!

Lock Variables

Implementation by "Test&Set" operation

```
typedef shared int mutex;
LOCK(mutex *lock)
  do
    old = TestAndSet(lock);
  while (old == 1)
UNLOCK(mutex *lock)
  *lock = 0
```

- *Atomic* operation `TestandSet(lock)`
  1. sets lock to 1 and
  2. returns previous value of lock.
- Implemented by hardware mechanism.

Constant polling of lock variable!

Semaphores

Edgar Dijkstra.

```
Semaphore s = INIT(1)
P1: loop     P2: loop
      P(s)         P(s)
      C1           C2
      V(s)         V(s)
```

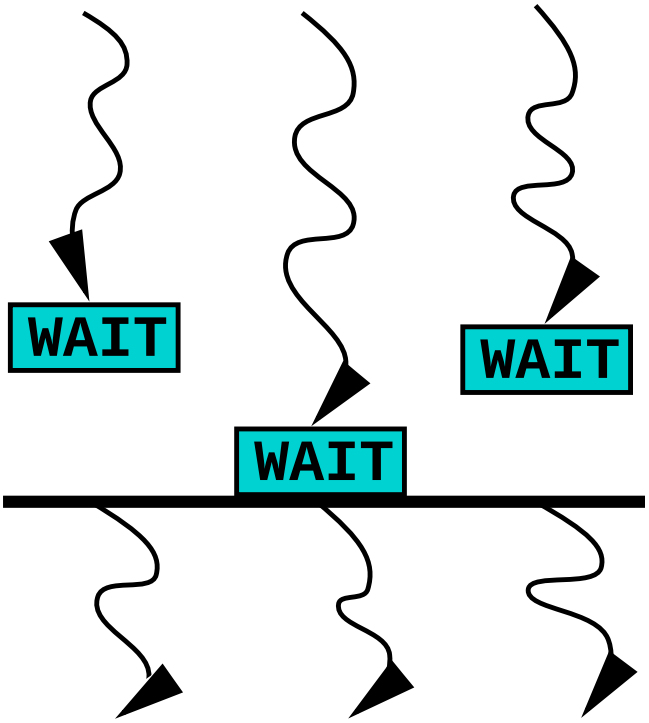
- Counter variable holding natural number.
- Atomic operations P and V.
- $P(s)$: if $s > 0$ then $s \leftarrow s - 1$ else block.
- $V(s)$: if processes blocked on s then release some process else $s \leftarrow s + 1$.

Typically implemented by locks!

Barriers

```
barrier b = INIT(n)
WAIT(barrier)
```

- Synchronization of n processes.
- `WAIT()` blocks until all n processes have reached barrier.



See Convex SPP documentation.