Chapter 4
Issues in Shared Memory Programming

Module 4.1 - Correctness Issues in Shared Memory Parallel Programming

Issues of parallel programming

- Correctness Issues
  - Result preservation
  - Synchronization
  - Variable partitioning
- Parallelizing Compiler Limitations (refer to the textbook for detailed discussion)
- Performance Issues
  - Task granularity
  - Synchronization granularity
  - Lack of utilization of reduction variables

Correctness Issues

- Unlike sequential programs, bugs in parallel programs are more difficult to find
  - They may not result in observable anomalies such as crashes
  - They may manifest in errors intermittently
  - They are often timing-dependent
  - It may be difficult to recreate the conditions that produce the error

Correctness Issue: Result Preservation

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
sum = 0;
for (i=0; i<8; i++)
  if (a[i] > 0)
    sum = sum + a[i];
Print sum;
```

```
begin parallel // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter=4, sum=0;
shared double sum=0.0, a[], b[], c[];
shared lock_type mylock;

start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];
barrier;

for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0) {
    lock(mylock);
      sum = sum + a[i];
    unlock(mylock);
  }
barrier;    // necessary

end parallel // kill the child thread
Print sum;
```

Are both sum exactly the same?

Parallel Execution Changes Order of Operations

Decimal	Sign	Exponent	Mantissa
	Normalized Floating Point		
A= 1.3125	0	0111	(1)0101
B=0.03125	0	0010	(1)0000
C=0.03125	0	0010	(1)0000

Adding B and C first before adding A:

B+C:

(i) Equalize exponent	0	0010	(1)0001	→	0	0010	(1)0000
	0	0010	(1)0000	→	0	0010	(1)0000 +
(ii) Add mantissa					0	0010	(10)0000
(iii) Normalize result					0	0011	(1)0000

(B+C) + A:

(i) Equalize exponent	0	0011	(1)0000	→	0	0111	(0)0001
	0	0111	(1)0101	→	0	0111	(0)0101 +
(ii) Add mantissa					0	0111	(1)0110
(iii) Normalize result					0	0111	(1)0110

Adding A and B first before adding C:

A+B:

(i) Equalize exponent	0	0111	(1)0101	→	0	0111	(1)0101
	0	0010	(1)0000	→	0	0111	(0)0000(1) +
(ii) Add mantissa					0	0111	(1)0101
(iii) Normalize result					0	0111	(1)0101

(A+B) + C:

(i) Equalize exponent	0	0111	(1)0101	→	0	0111	(1)0101
	0	0010	(1)0000	→	0	0111	(0)0000(1) +
(ii) Add mantissa					0	0111	(1)0101
(iii) Normalize result					0	0111	(1)0101

Bug 1: Missing barrier

```
#pragma omp parallel for default(shared) nowait
for (i=0; i<n; i++)
  a[i] = b[i] + c[i];

#pragma omp parallel for default(shared) reduction(+:sum) nowait
for (i=0; i<n; i++)
  sum = sum + a[i];

print sum;
```

- Probable results
  - No crash
  - Wrong output

Bug 2: Missing synchronization

```
#pragma omp parallel for default(shared)
for (i=0; i<n; i++)
  sum = sum + a[i];

print sum;
```

- Probable results
  - No crash
  - Wrong output

Bug 3: Private/shared variables

```
#pragma omp parallel for shared(b,temp) private(a,c)
for (I=0; I<N; I++) {
  temp = b[I] + c[I];
  a[I] = a[I] * temp;
}
```

- c should be shared but declared private
  - Storage overhead
  - Non-deterministic outcome, depending on whether the private c is initialized to the global c
- a should be shared but declared private
  - Storage overhead and likely erroneous output
- temp should be private but declared shared
  - Possible wrong output due to overwrites, depending on timing

Module 4.2 – Limitations of Parallelizing Compilers

Limitations of Parallelizing Compilers

- Can compiler performs dependence analysis automatically and automatically extracts parallelism?
- Not easily
  - Compiler's main goal is to produce correct code within a reasonable compilation time
    - Costly analysis (e.g. inter-procedural) is often skipped
  - Compiler only has static information
    - Many things are not known at compile time: overheads, memory dependences, commutativity and associativity of user-defined operations

Case Study: Compiler Limitations

- MIPSpro compiler
- Automatic parallelization using pfa (Power Fortran Analyzer)

- Assume that we only want to maximize parallelism (reducing s, the serial fraction)
  - Don't care other factors
- Assume the loops are large enough that they are worth parallelizing

swim: Case 1

Parallelization Log for Subprogram MAIN__
  114: Not Parallel
    Scalar dependence on PCHECK.
    Scalar dependence on UCHECK.
    Scalar dependence on VCHECK.
  115: Not Parallel
    Scalar dependence on PCHECK.
    Scalar dependence on UCHECK.
    Scalar dependence on VCHECK.

```
      DO 3500 ICHECK = 1, MNMIN
       DO 4500 JCHECK = 1, MNMIN
       PCHECK = PCHECK + ABS(PNEW(ICHECK,JCHECK))
       UCHECK = UCHECK + ABS(UNEW(ICHECK,JCHECK))
       VCHECK = VCHECK + ABS(VNEW(ICHECK,JCHECK))
4500   CONTINUE
      UNEW(ICHECK,ICHECK) = UNEW(ICHECK,ICHECK)
   1  * ( MOD (ICHECK, 100) /100.)
3500   CONTINUE
```

Observations

- Compiler did not perform inter-procedural analysis or performed function inlining
- Solution:
  - Force parallelization by declaring reduction for PCHECK, UCHECK, and VCHECK

Case 2

```
204: Not Parallel
        Loop is contained within a parallel construct.
     DO 50 J=1,NP1
     DO 50 I=1,MP1
     PSI(I,J) = A*SIN((I-.5D0)*DI)*SIN((J-.5D0)*DJ)
     P(I,J) = PCF*(COS(2.D0*(I-1)*DI)
    1                +COS(2.D0*(J-1)*DJ))+50000.D0
  50 CONTINUE
```

Solution

- Compiler prefers to parallelize the outer loops
- This is a good heuristics in general
- However, if the parallelism at the outer loop is limited, force parallelization of the inner loops

Tomcatv: Case 3

```
TOMCATV
   83: Not Parallel
         Has IO statement on line 85.
      DO   10    J = 1,N
      DO   10    I = 1,N
           READ(10,600,END=990) X(I,J),Y(I,J)
   10 CONTINUE
```

Solution

- I/O read has to be serial
  - Part of the speedup limiting factor
  - However, if it is a bottleneck, may split the input file into several files, and each thread reads from its own file

```
133: Not Parallel
       Array dependence from RXM on line 135 to RXM on line 135.
       Array dependence from RYM on line 136 to RYM on line 136.
      DO     80    J = 2,N-1
        DO     80    I = 2,N-1
          RXM(ITER) = MAX(RXM(ITER), ABS(RX(I,J)))
          RYM(ITER) = MAX(RYM(ITER), ABS(RY(I,J)))
 80   CONTINUE
```

Obvservations

- Compiler detects loop-carried dependence on RXM[] and RYM[]
- Compiler did not inline or performed inter-procedural analysis
- or it could not infer if MAX is a reduction operator

Mgrid

```
 77: Not Parallel
         Call mg3p on line 78.
         Call resid on line 79.
      DO 20 IT=1,NIT
        CALL MG3P(U,V,R,N,A,C,NV,NR,IR,MM,LMI)
        CALL RESID(U,V,R,N,A)
CONTINUE
```

Solution

- Compiler refuses to perform inter-procedural analysis for parallelism
  - It cannot analyze side effects
  - or it' s computationally expensive (compiler needs to be reasonably fast)
- Inline the two functions and recompile
- or, examine dependences in the mg3p and resid, then hand-parallelize

Module 4.3 – Performance Issues

Amdahl's Law

- Parallel speedup $$speedup = \frac{T(1)}{T(n)}$$
- Suppose that *s%* of the execution is non-parallel, what is the maximum parallel speedup?

$$speedup = \frac{1}{s + \frac{1-s}{p}}$$

$$speedup_\infty = \lim_{p \to \infty} \frac{1}{s + \frac{1-s}{p}} = \frac{1}{s}$$

- The above is referred to as Amdahl's law. Example:

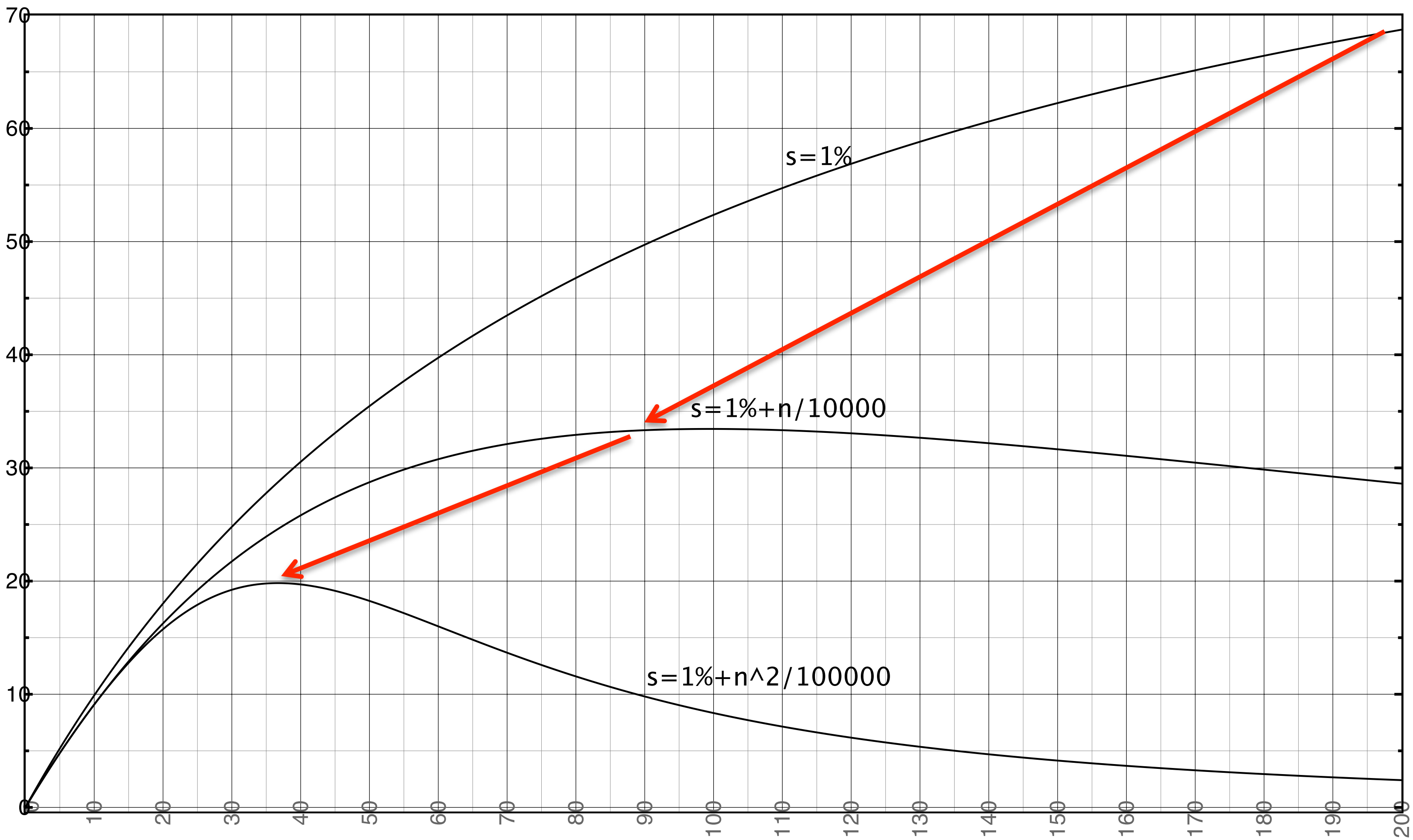
s	Max Speedup
10%	10
1%	100
0.1%	1000

Optimize program, or increase input size (referred to as "weak scaling")

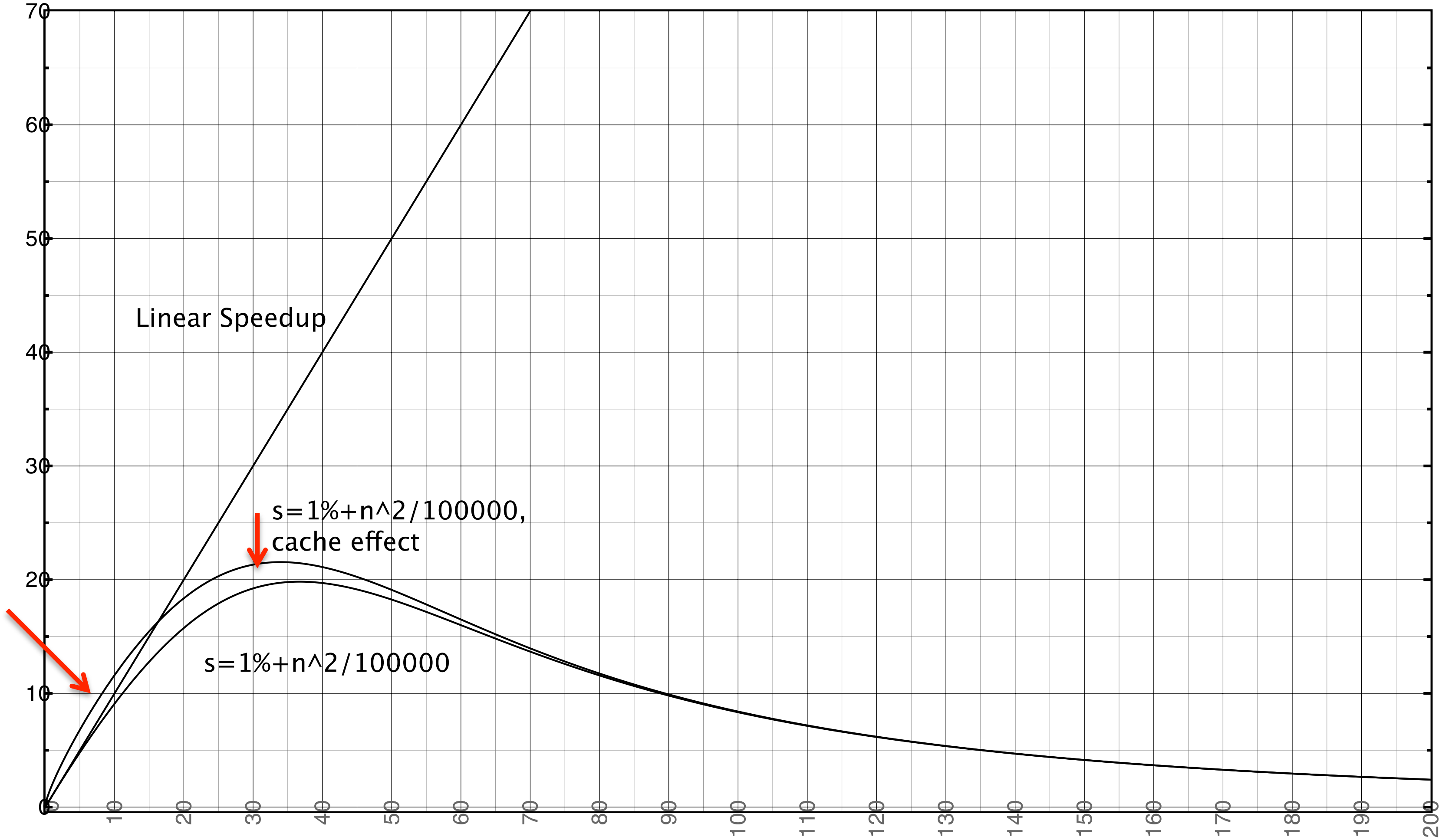
Reality

- The reality is often worse
  - Amdahl only considered sequential execution
  - Thread management overheads may scale at least linearly with num threads
  - Synchronization may scale quadratically to num threads
- However, it may be better, too
  - Parallel systems often have larger aggregate caches/ memories
  - Cache misses may decline significantly beyond a certain number of threads, producing a performance boost

Illustration



Illustration



Parallel Task Granularity

- outer loops vs. inner loops vs. both

```
pragma omp parallel for {
for (i=0; i<n; i++)
  if (a[i][0] > 0)
    for (j=0; j<n; j++)
      a[i][j] = a[i][j] + 1;
}
```

```
for (i=0; i<n; i++)
  if (a[i][0] > 0)
    pragma omp parallel for {
      for (j=0; j<n; j++)
        a[i][j] = a[i][j] + 1;
    }
```

- outer loop iterations form larger tasks vs. the inner loop iterations, but
- Load balancing may be easier for the inner loop

- Examine the trade-offs (see Section 3.8)

Task Mapping

- How are tasks are grouped?
- How are tasks assigned to threads?
- How are threads assigned to cores?

Task and Thread Mapping

- How are tasks are grouped?
  - Too small tasks: task mgmt overheads too high
  - Too large tasks: load balancing become difficult
  - Solution: grouped tasks into larger ones
- How are tasks assigned to threads?
  - Static vs. dynamic (using a task queue)
- How are threads assigned to cores?
  - In NUMA systems, data and threads should be co-located

SCHEDULE directive in OpenMP

- *chunksize* specifies the size of task grouping
- Static: each chunksize is assigned to a processor statically
- Dynamic: each chunksize is a task in a task queue. Each worker thread fetches a task from the queue and executes it
- Guided: same as Dynamic, except that the task sizes are not uniform, early tasks are larger
- Runtime: check the environment variable OMP_SCHEDULE at run time to determine what scheduling to use

Task Scheduling in OpenMP

- SCHEDULE(Static|Dynamic|Guided|Runtime, chunksize)
- Static: Queue at DMV, Dynamic: Bank

```
sum = 0;
pragma omp parallel for reduction(+:sum) schedule (static, n/p) {
for (i=0; i<n; i++)
  for (j=0; j<i; j++)
    sum = sum + a[i][j];
Print sum;
```

- Suppose n = 1000 and p = 2; how many iterations each processor gets?

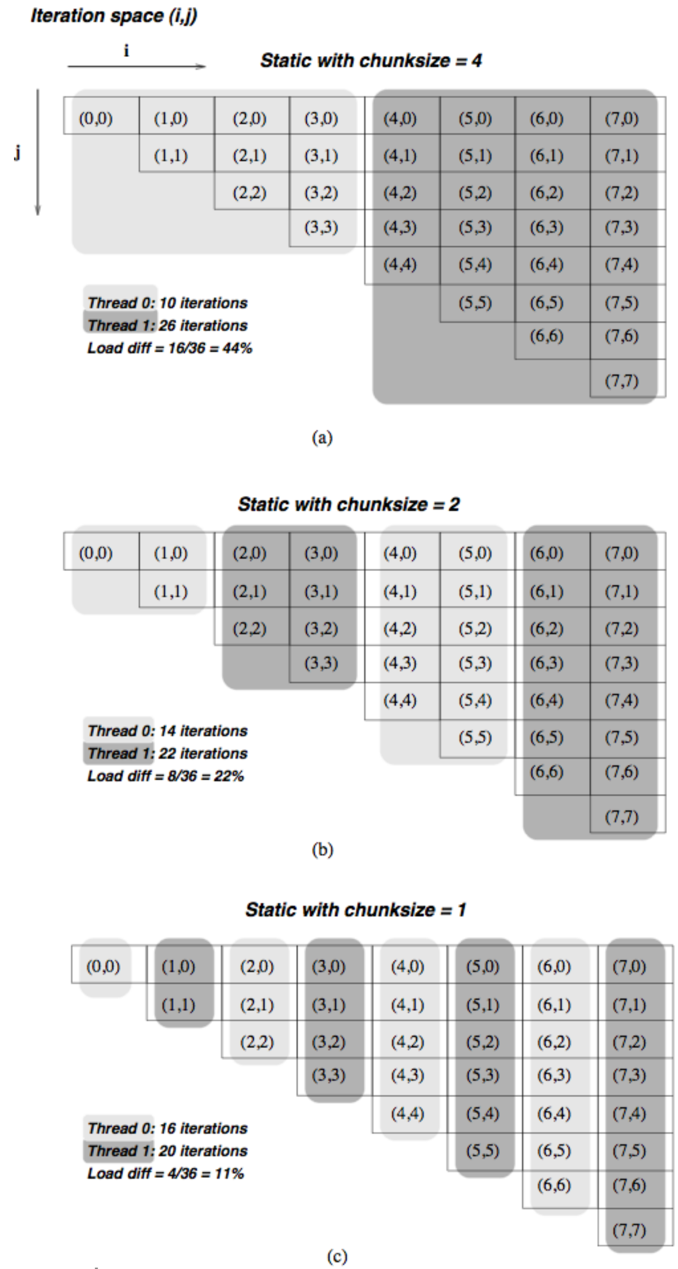
- How would this perform instead?

```
pragma omp parallel for reduction(+:sum) schedule (dynamic, n/p)
pragma omp parallel for reduction(+:sum) schedule (guided, n/p)
pragma omp parallel for reduction(+:sum) schedule (runtime, n/p)
```

Effect of chunk size

```
sum = 0;

#pragma omp parallel for reduction
(+:sum) schedule (static, chunksz)
for (i=0; i<n; i++)
  for (j=0; j<i; j++)
    sum = sum + a[i][j];
Print sum;
```



Inherent vs. Artifactual Communication

- Communication is expensive!
- Important metric:
  - *communication to computation ratio*
  - Use this to infer the scalability
- Focus on *inherent communication*
  - Caused by task-to-process mapping
  - In actual communication, we also need to care about process-to-processor mapping

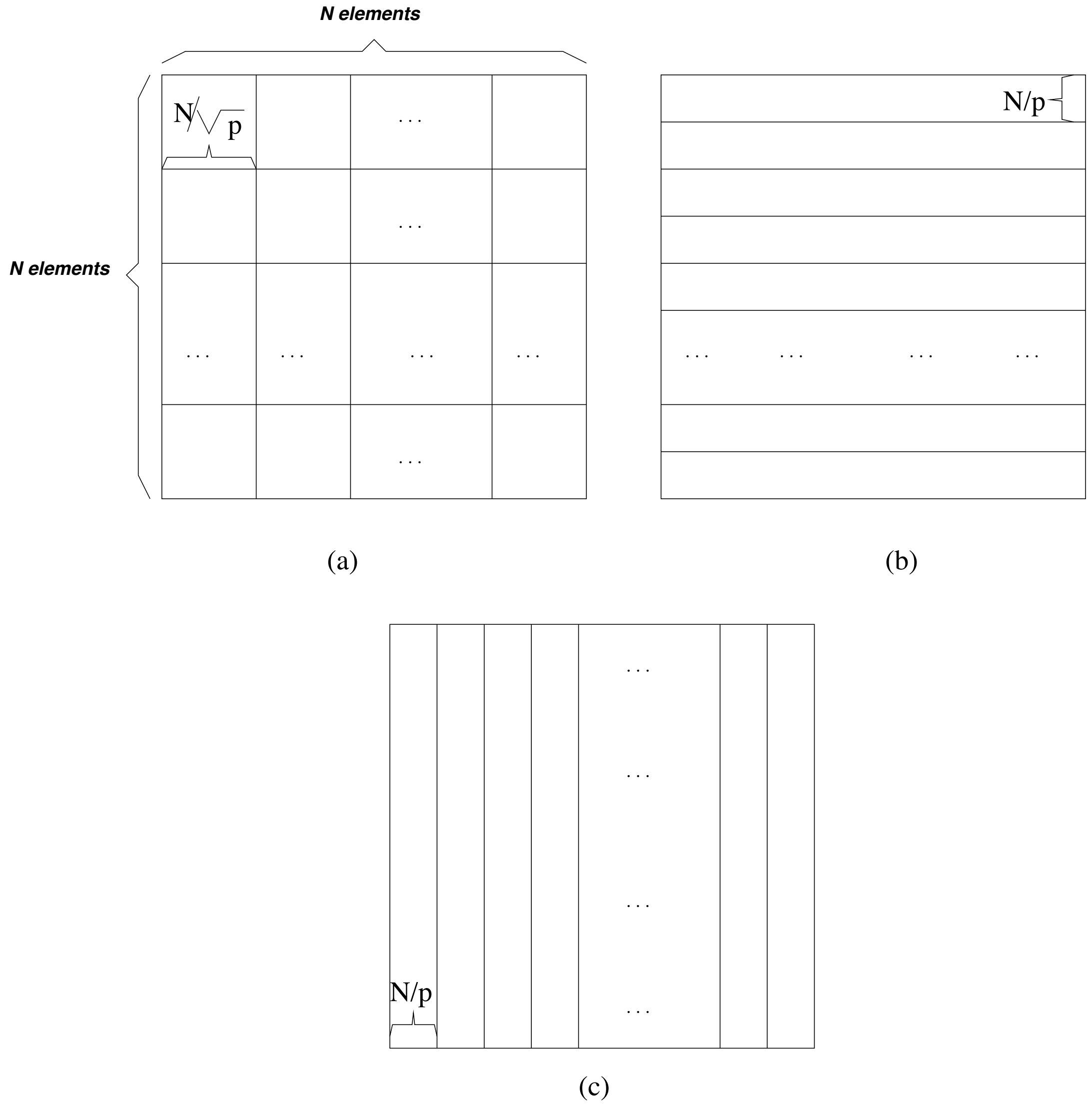


Figure 4.2: Assigning tasks to threads block-wise (a), row-wise (b), and column-wise(c).

Communication-to-computation Ratio

- Block-wise partitioning

$$CCR = \frac{Comm}{Comp} = \frac{4n/\sqrt{p}}{n^2/p} = \frac{4\sqrt{p}}{n}$$

- Row-wise partitining

$$CCR = \frac{Comm}{Comp} = \frac{2n}{n^2/p} = \frac{2p}{n}$$

Artifactual Communication

- Actual communication depends on number of cache blocks communicated between processors

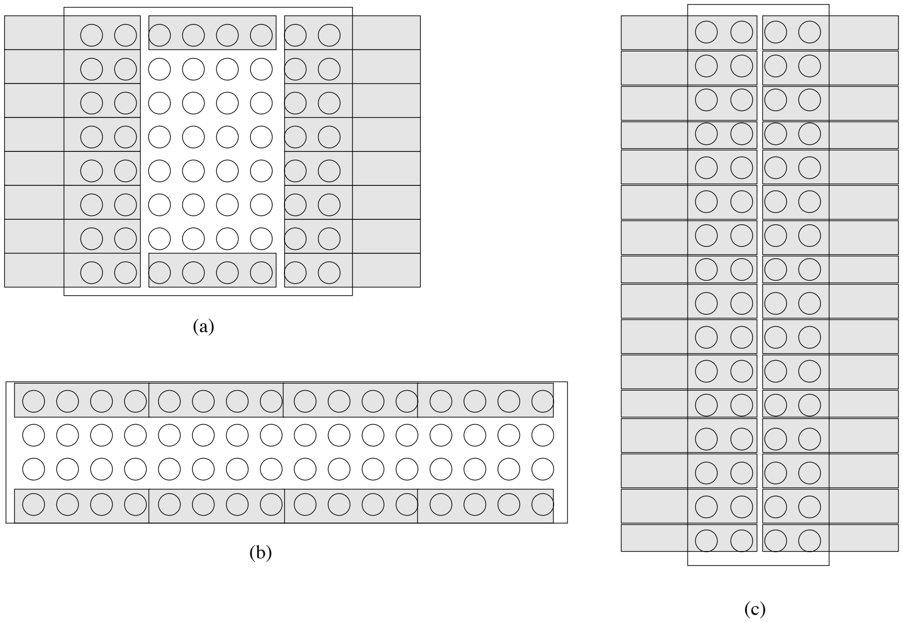


Figure 4.3: Number of cache blocks shared by more than one processors in the block-wise partitioning (a), row-wise partitioning (b), and column-wise partitioning (c), assuming the memory layout is as shown.

- Column-wise the worst, not clear if block-wise is always better than row-wise

Memory Hierarchy Issue

- Multiprocessor as Extended Memory Hierarchy
    - as seen by a given processor
- Levels in extended hierarchy:
    – Registers, caches, local memory, remote memory (topology)
    – Glued together by communication architecture
    – Lower level: higher size, higher communication cost
- Thus, key to performance: spatial and temporal locality at each hierarchy level

Reuse/Locality Patterns

- Data reuse/locality patterns:
  - Temporal reuse: a data recently accessed tends to be accessed again in the near future
  - Spatial reuse: the neighboring location of recently accessed data tends to be accessed again in the near future

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

- In the code above:
  - Temporal locality: i, n, sum
  - Spatial locality: a[]

Exploiting Spatial Locality

- Rectangular matrix a[n][n]

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

C/C++ Memory Layout

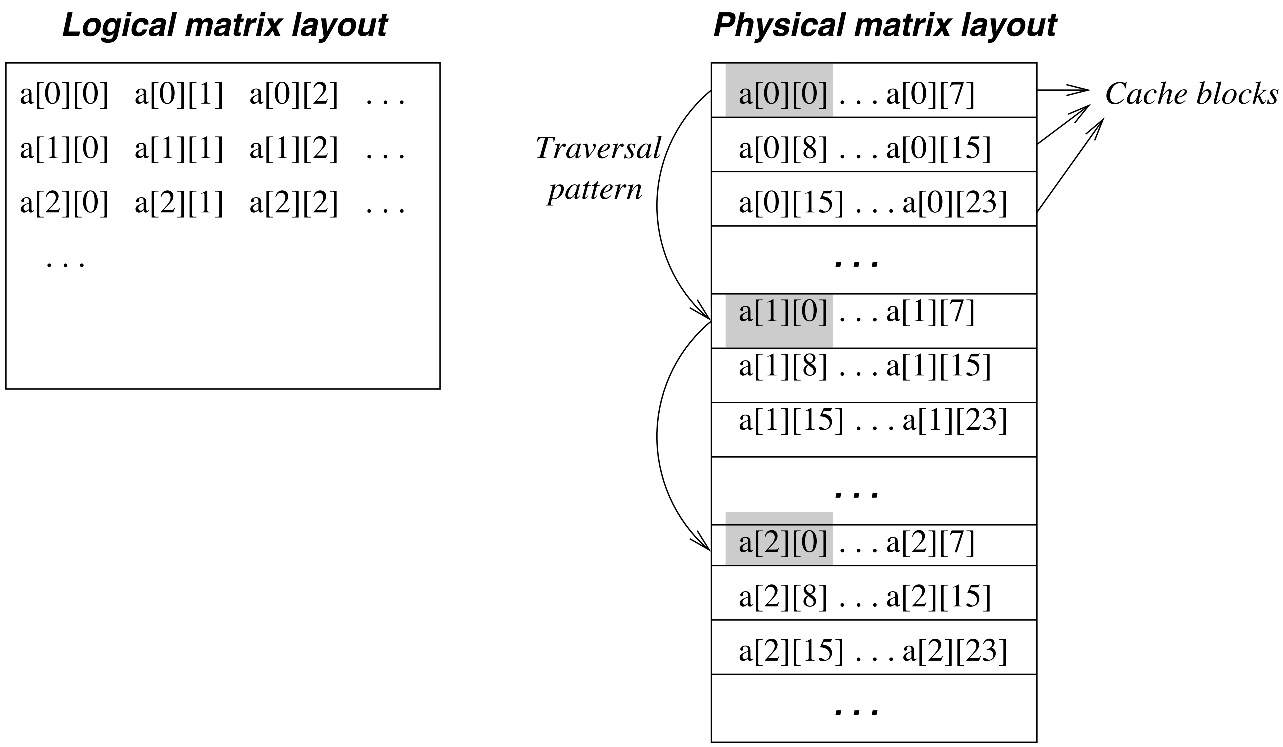


Figure 4.5: C/C++ memory layout of arrays, showing the traversal pattern for Code 4.14.

Exploiting Spatial Locality

- Rectangular matrix a[n][n]

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

Loop interchange →

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

- When is loop interchange safe? perfectly nested loop, i.e. when the body of a loop only contains one inner loop.
- Example of imperfectly nested loop:

```
sum = 0;
for (j=0; j<n; j++) {
  for (i=0; i<n; i++)
    sum = sum + a[i][j];
  a[0][j] = sum;
}
Print sum;
```