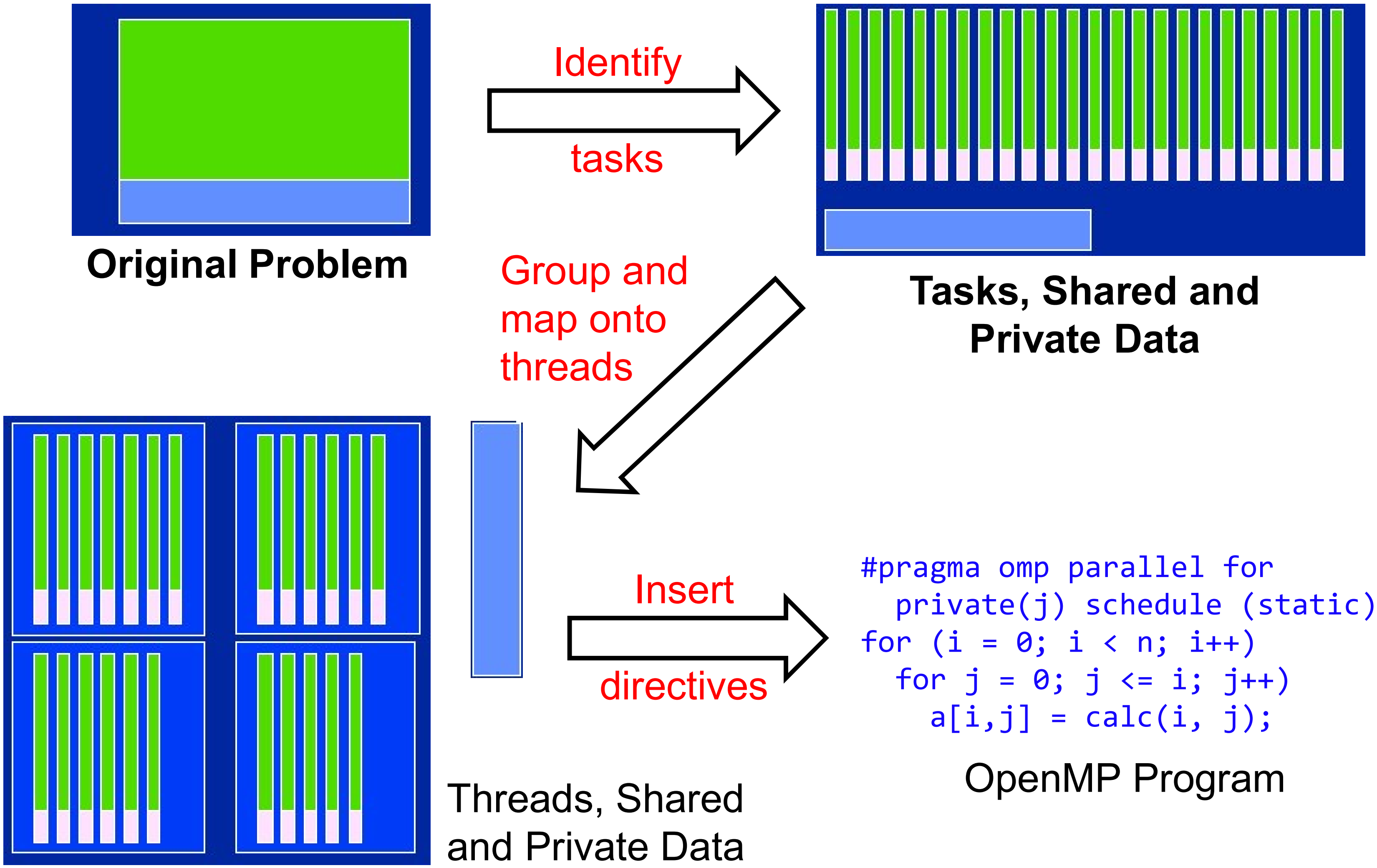
Loop Parallelism

CS121 Parallel Computing

Fall 2023

Shared memory algorithm design



Design considerations

- Break program into tasks, consisting of statements that must be executed in order.
  - Use data dependence analysis.
- Map independent tasks to different processors.
  - Mapping needs to consider load balancing, e.g. static vs dynamic, block vs cyclic work assignment.
- Variable specification
  - Shared vs. private vs. reduction
  - Shared variables cause cache coherence traffic and much lower performance.
  - Private and reduction variables don't need synchronization (except possibly at end of a loop).
- Dimension mapping, e.g. row-wise vs column-wise.
  - Matching mapping to access pattern improves cache locality.

Data dependence analysis

- Let S1 and S2 be two statements in a sequential execution of a program.
- S1 and S2 are independent if running them in different orders produces the same result. Otherwise they're dependent.
- Suppose S1 occurs before S2. They can have the following dependencies.
  - S1 →T S2 denotes true dependence (RAW), i.e. S1 writes to a location that is read by S2.
  - S1 →A S2 denotes anti dependence (WAR), i.e. S1 reads a location written by S2.
  - S1 →O S2 denotes output dependence (WAW), i.e. S1 writes to the same location written by S2.

```
S1: x = 2;
S2: z = 3;
S3: y = x;
S4: y = x + z;
S5: z = 6;
```

- Dependences
  - S1 →T S3
  - S1 →T S4
  - S2 →T S4
  - S2 →O S5
  - S3 →O S4
  - S4 →A S5

Parallelizing loops

- When parallelizing a program, must ensure dependent statements run in the same order in the sequential and parallel programs.
  - Guarantees the parallel and sequential programs behave the same way.
  - A parallel program may run correctly without satisfying this condition, but it's not guaranteed.
  - The ordering requirement is transitive. I.e. if S1 $\rightarrow^*$ S2 and S2 $\rightarrow^*$ S3, then S1 must run before S3 in any parallelization.
- Dependent statements cannot on run on different processors, since we can't enforce the order of execution (interleaving) on different processors.
- Independent statements can run on different processors if they haven't been ordered by transitivity.

Parallelizing loops

- Goal is to identify all independent statements, to maximize parallelism.
- Focus on parallelizing loops, since these are common in shared memory programs and are the main performance hotspots.
- Notation
  - Let S denote a statement in the source program.
  - Given a nested loop with iteration variables i, j, ..., let S[i,j,…] denote a statement in loop iteration [i,j,…].

Loop dependence analysis

- Loop-carried dependence
  - Dependence exists across different iterations of loop.
- Loop-independent dependence
  - Dependence exists within the same iteration of loop.

```
for (i=1; i<=n; i++) {
  S1: a[i] = a[i-1] + 1;
  S2: b[i] = a[i];
}

for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S3: a[i][j] = a[i][j-1] + 1;

for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S4: a[i][j] = a[i-1][j] + 1;
```

S1[i] $\rightarrow$T S1[i+1]
- loop-carried dependence

S1[i] $\rightarrow$T S2[i]
- loop-independent dependence

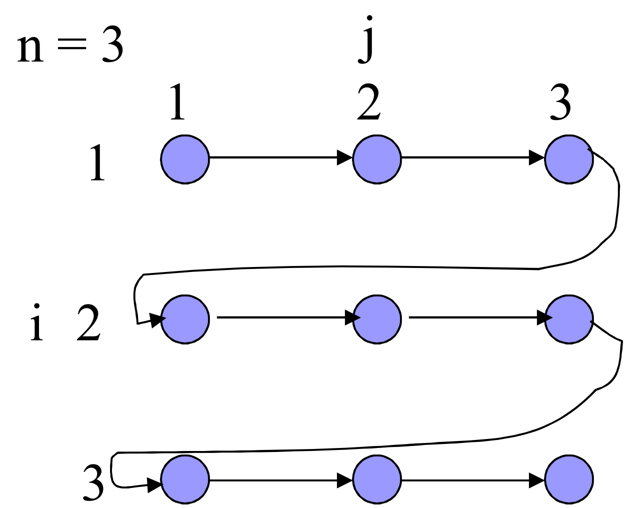
S3[i,j] $\rightarrow$T S3[i,j+1]
- loop-carried dependence in j loop
- no loop-carried dependence in i loop

S4[i,j] $\rightarrow$T S4[i+1,j]
- loop-carried dependence in i loop
- no loop-carried dependence in j loop

Iteration-space traversal graph

- Iteration-space traversal graph (ITG) is a line graph showing the order of traversal in the iteration space.
- Node in ITG is a point in the iteration space, i.e. a particular iteration.
- Directed edge in ITG gives the next iteration that will be executed after the current iteration.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

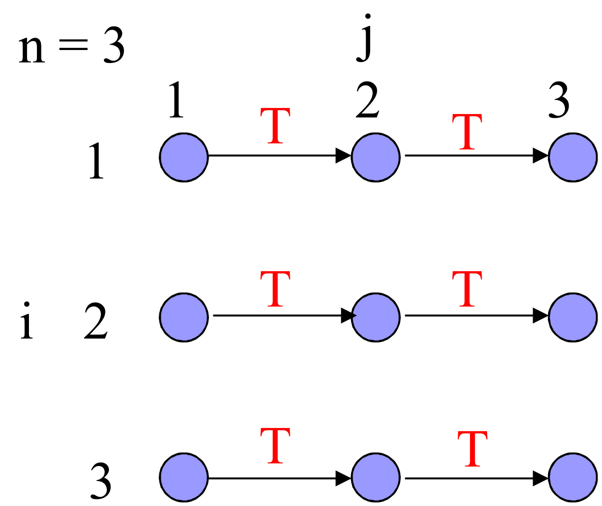


Loop-carried dependence graph

- Given the ITG, can determine the dependence between different loops.
- Loop-carried Dependence Graph (LDG) shows the loop-carried true/anti/output dependence relationships.
- Node in LDG is a point in the iteration space.
- Directed edge in LDG is the dependence.
- LDG helps identify parts of the loop that can be done in parallel.
  - Different connected components can be done in parallel.
  - Each connected component must be done sequentially.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

S3[i,j] $\rightarrow$T S3[i,j+1]
- loop-carried dependence in j loop
- no loop-carried dependence in i loop



Example 1

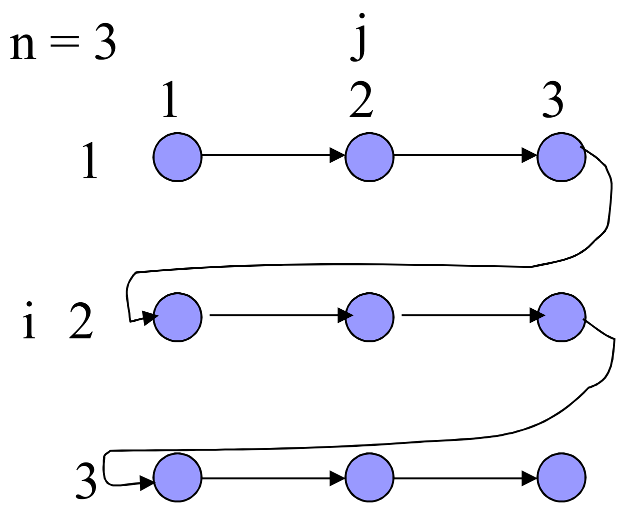
```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i][j-1] * d[i][j];
  }
```

True dependences
S1[i,j] $\rightarrow$ T S2[i,j+1]
- loop-carried dependence

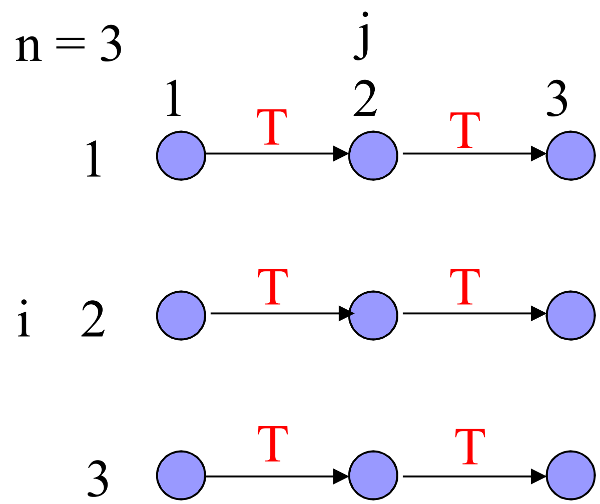
Anti dependences
S1[i,j] $\rightarrow$ A S2[i,j]
- loop-independent dependence

Output dependences
None

- ITG



- LDG



Example 1

- Task Identification
  - n parallel tasks - one task per iteration of i loop.
- Grouping / mapping
  - Static block as different iterations have same work.
- OpenMP

```
#pragma omp parallel for private(j) schedule(static)
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i][j-1] * d[i][j];
  }
```

Example 2

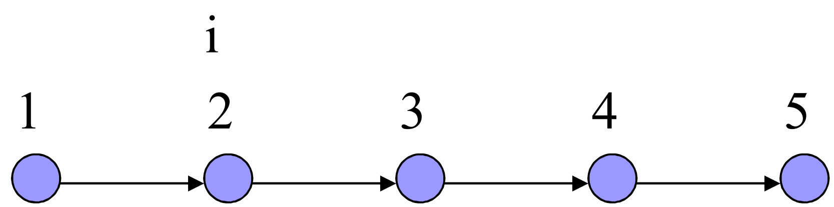
```
for (i=1; i<=n; i++)
  S1: a[i] = a[i-2];
```

True dependences
S1[i] $\rightarrow$ T S1[i+2]
- loop-carried dependence

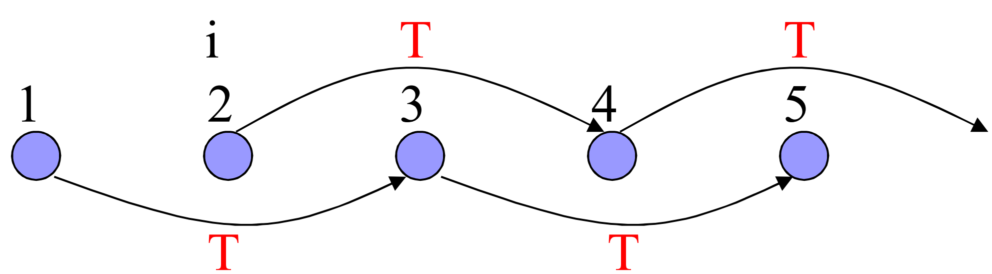
Anti dependences
None

Output dependences
None

- ITG



- LDG



Example 2

- Task Identification
  - There are opportunities when some dependences are missing.
  - Can divide the for loop into two parallel tasks, one with even iterations and another with odd iterations.
- Grouping / mapping
  - One task per thread
- OpenMP

```
#pragma omp parallel sections
  private(i)
{
  #pragma omp section
  for (i=1; i<=n; i+=2)
    a[i] = a[i-2];
  #pragma omp section
  for (i=2; i<=n; i+=2)
    a[i] = a[i-2];
}
```

Example 3

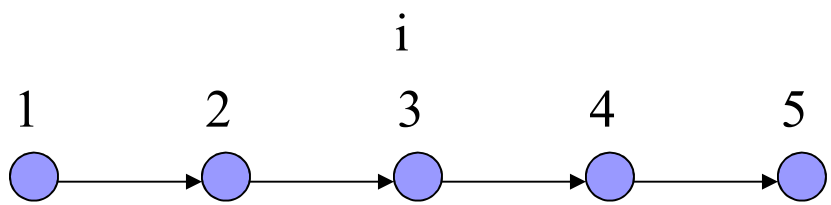
```
for (i=1; i<=n; i++)
  S1: a[i] = a[i-1] + b[i]*c[i] + d[i];
```

True dependences
S1[i] $\rightarrow$ T S1[i+1]
- loop-carried dependence

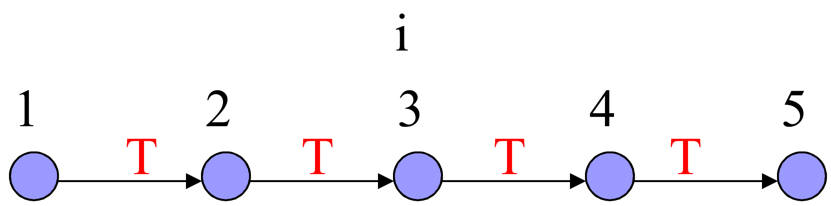
Anti dependences
None

Output dependences
None

- ITG



- LDG



Must respect loop-carried dependence

Example 3

- Task Identification
  - Any opportunities for parallelism? Loop-carried dependence S[i] $\rightarrow$T S[i+1] must be respected.
  - But no loop-carried dependence in `b[i]*c[i]+d[i]` part.
  - Loop fission
    - Distribute into two separate loops .
- Code after loop fission

```
for (i=1; i<=n; i++) {
  S1: temp[i] = b[i]*c[i] + d[i];
}

for (i=1; i<=n; i++) {
  S2: a[i] = a[i-1] + temp[i];
}
```

No dependences in first loop, so can be parallelized.

In second loop
True dependences:
S2[i] $\rightarrow$ T S2[i+1]
- loop-carried dependence
Anti dependences None
Output dependences None

Example 3

- OpenMP

```
#pragma omp parallel for schedule(static)
for (i=1; i<=n; i++) {
  temp[i] = b[i]*c[i] + d[i];
}
for (i=1; i<=n; i++) {
  a[i] = a[i-1] + temp[i];
}
```

  - Note array `temp[]` introduces storage overhead.

- Better OpenMP solution

```
#pragma omp parallel for ordered private(t) schedule(static,1)
for (i=1; i<=n; i++) {
  t = b[i]*c[i] + d[i]; /* one copy of t per thread */
  #pragma omp ordered
  a[i] = a[i-1] + t; }
```

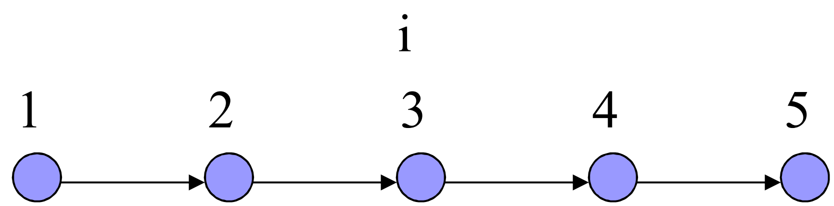
  - `ordered` statement enforces a[i] assignment ordering.
  - With k threads, uses k extra storage.
  - Typically k << n so we save space.

Example 4

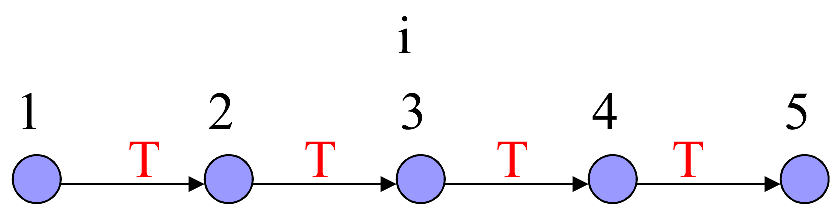
```
for (i=1; i<=n; i++) {
  S1: a[i] = b[i+1]*a[i];
  S2: b[i] = b[i]*coef;
  S3: c[i] = 0.5*(c[i] + a[i]);
  S4: d[i] = d[i-1] + d[i];
}
```

True dependences
S1[i] →T S3[i]
-loop-independent dependence
S4[i] →T S4[i+1]
-loop-carried dependence
Anti dependences
S1[i] →A S2[i+1]
-loop-carried dependence
Output dependences None

- ITG



- LDG



Must respect loop-carried dependence

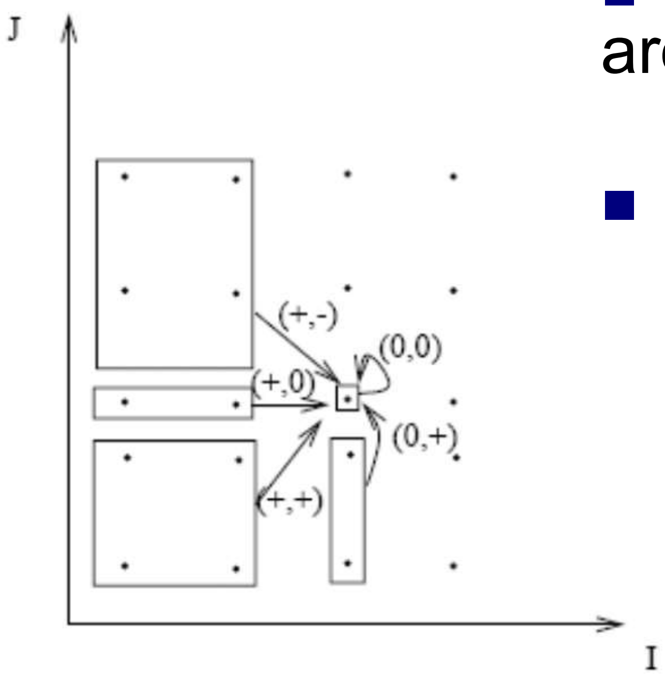
Example 4

- Task Identification
  - S4 has no dependences with other statements, so can distribute into two separate loops (loop fission).
  - This gives two parallel tasks, a loop containing S1, S2 and S3 and a loop containing S4.
- OpenMP

```
#pragma omp parallel sections
  private(i)
{
  #pragma omp section
  for (i=0; i<=n; i++) {
    S1: a[i] = b[i+1]*a[i];
    S2: b[i] = b[i]*coef;
    S3: c[i] = 0.5*(c[i] + a[i]);
  }
  #pragma omp section
  for (i=0; i<n; i++) {
    S4: d[i] = d[i-1] + d[i];
  }
}
```

Distance and direction vectors

- Let T_1 and T_2 be iterations s.t. $T_1 \rightarrow^* T_2$.
- Distance vector from T_1 to T_2 is $T_2 - T_1$.
- Direction vector from T_1 to T_2 is $\text{sign}(T_2 - T_1)$.
- Consider a nested loop over (i,j) (j is the inner loop).

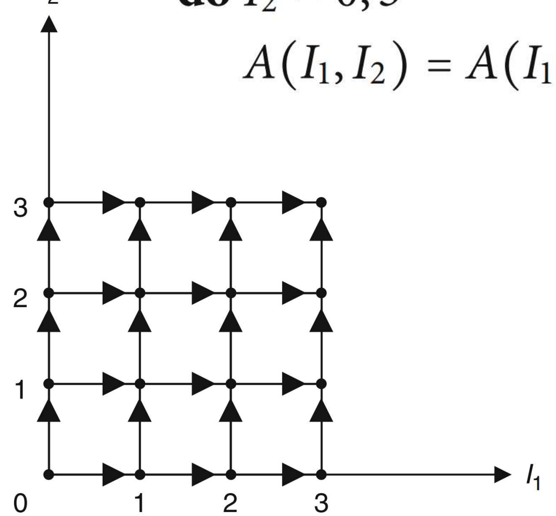


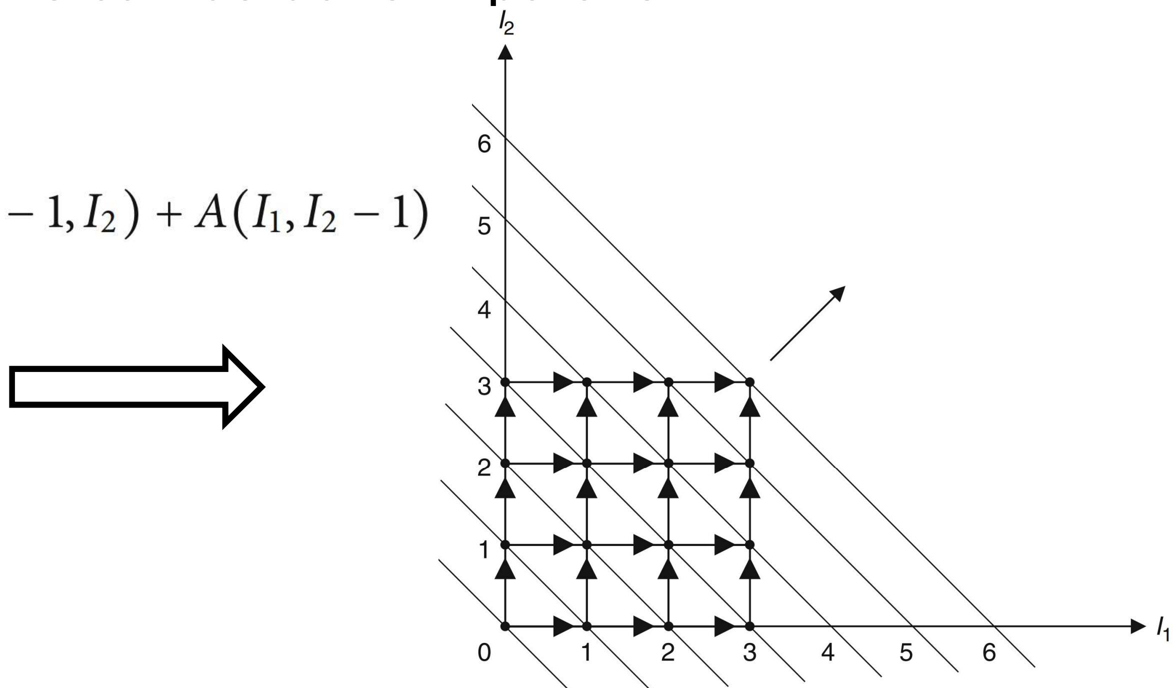
- The following direction vectors are possible
  - (+,+), (+,0), (+,-), (0,+), (0,0).
- The following directions are not possible.
  - (0,-), (-,+), (-,0), (-,-).
  - Ex Direction vector (-,+) would mean, e.g. iteration (i,j) depends on iteration (i+1, j-1). But (i,j) occurs before (i+1, j-1).

Loop skewing

- If a loop has dependencies that prevent it being parallelized, we can try to transform (skew) the loop indices to create parallelizable loops.
- Ex Create new loop indices, the outer loop across the diagonal lines, the inner loop along the diagonal lines.
  - Inner loop iterations can be done in parallel.

$$
\begin{aligned}
&\textbf{do } I_1 = 0, 3\\
&\quad \textbf{do } I_2 = 0, 3\\
&\qquad A(I_1, I_2) = A(I_1 - 1, I_2) + A(I_1, I_2 - 1)
\end{aligned}
$$

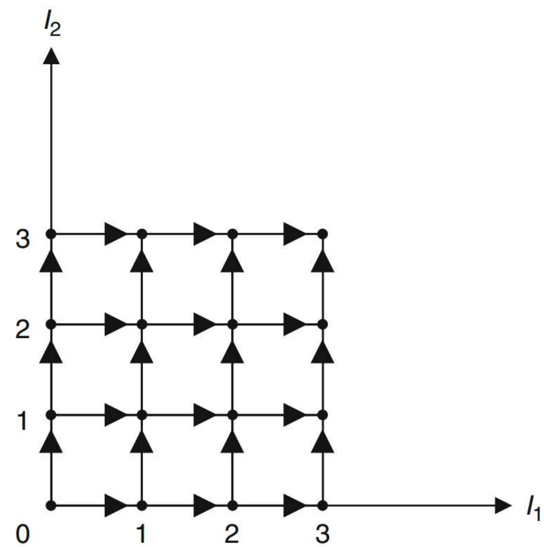


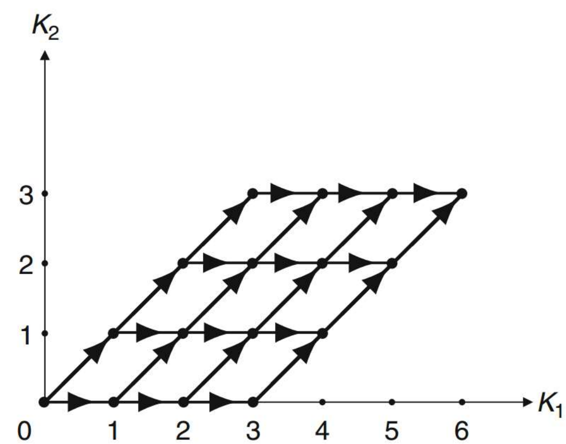


Source: Unimodular Transformations, Utpal Banerjee

Unimodular transformations

- Can define new indices using a linear transformation.
  - We use unimodular linear transformations, where the matrix has determinant -1 or 1.
  - Unimodularity ensures all iterations in original loop are performed in the transformed loop.
- Ex Let $U = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, and $(K_1, K_2) = (I_1, I_2)U = (I_1 + I_2, I_2)$.
  - Let K_1, K_2 be the outer and inner loops, resp. Then for each K_1 iteration, can run all the K_2 iterations in parallel.





Extracting parallelism

- Transformed loop must be legal. Also, we want it to be parallelizable.
- Thm Let U be a unimodular transformation. If vU is legal for all distance vectors v, then U is a legal transformation.
- Ex For $U = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, $(0,1)U = (1,1), (1,0)U = (1,0)$.
  - New direction vectors are (+,+) and (+,0), so the loop with the new loop indices is legal.
- Thm Suppose all the direction vectors for a loop are + in the i'th coordinate, for some i. Then all loops deeper than level i can be run in parallel.
- Ex For the above U, direction vectors are all + in K_1 coordinate, so the K_2 loop can be parallelized.

$$A[I_1,I_2] = A[I_1-1,I_2] + A[I_1,I_2-1]$$

$$\Downarrow$$

$$A[K_1,K_2] = A[K_1-1,K_2-1] + A[K_1-1,K_2]$$

Data accesses after skewing

- Since $(K_1, K_2) = (I_1, I_2)U$, then $(I_1, I_2) = (K_1, K_2)U^{-1}$.
- So in iteration (K_1, K_2) of the transformed loop, we access data from iteration $(K_1, K_2)U^{-1}$ of the original loop.
- Ex For $U = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, $U^{-1} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$, and $(K_1, K_2)U^{-1} = (K_1 - K_2, K_2) = (I_1, I_2)$.

  So in iteration (K_1, K_2), we do

$$A[I_1, I_2] = A[K_1 - K_2, K_2] =$$
$$A[K_1 - K_2 - 1, K_2] + A[K_1 - K_2, K_2 - 1]$$

Loop bounds after skewing

- We have $I_1 = K_1 - K_2$, $I_2 = K_2$. Also, $0 \le I_1 \le 3$ and $0 \le I_2 \le 3$.
- So $0 \le K_1 - K_2 \le 3$ and $0 \le K_2 \le 3$.
- Since $0 \le K_1 - 3 \le 3$, then $0 \le K_1 \le 6$.
- Since $K_1 - K_2 \le 3$, then $K_1 - 3 \le K_2$. Also, $0 \le K_2$. So $K_2 \ge \max(0, K_1 - 3)$.
- Likewise, $K_2 \le \min(3, K_1)$.
- In general, the bounds can be computed using the Fourier-Motzkin method.
- Altogether, we have the following. The K_1 loop is sequential, but the K_2 loop can be run in parallel.

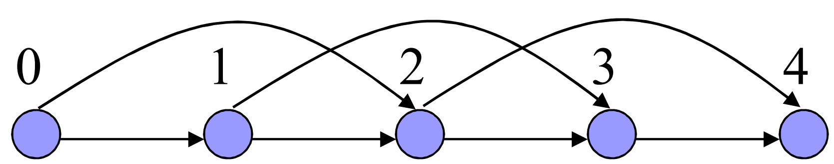
$$
\begin{aligned}
&\textbf{do } I_1 = 0, 3 \\
&\quad \textbf{do } I_2 = 0, 3 \\
&\qquad A(I_1, I_2) = A(I_1 - 1, I_2) + A(I_1, I_2 - 1)
\end{aligned}
\quad \Longrightarrow \quad
\begin{aligned}
&\textbf{do } K_1 = 0, 6 \\
&\quad \textbf{do } K_2 = \max(0, K_1 - 3),\ \min(3, K_1) \\
&\qquad A(K_1 - K_2, K_2) = A(K_1 - K_2 - 1, K_2) \\
&\qquad\qquad + A(K_1 - K_2, K_2 - 1)
\end{aligned}
$$

Algorithmic analysis

- Sometimes there is no way to restructure a loop to increase parallelism.
- We can try to restructure the algorithm to eliminate dependences and improve parallelism.
- Need to understand the purpose of the algorithm and how it is used.
- For example, some algorithms are nondeterministic or calculate an approximation (e.g., Gauss-Seidel iteration).
  - In this case, restructuring the algorithm or ignoring some dependences may still give a valid result.

Fibonacci numbers

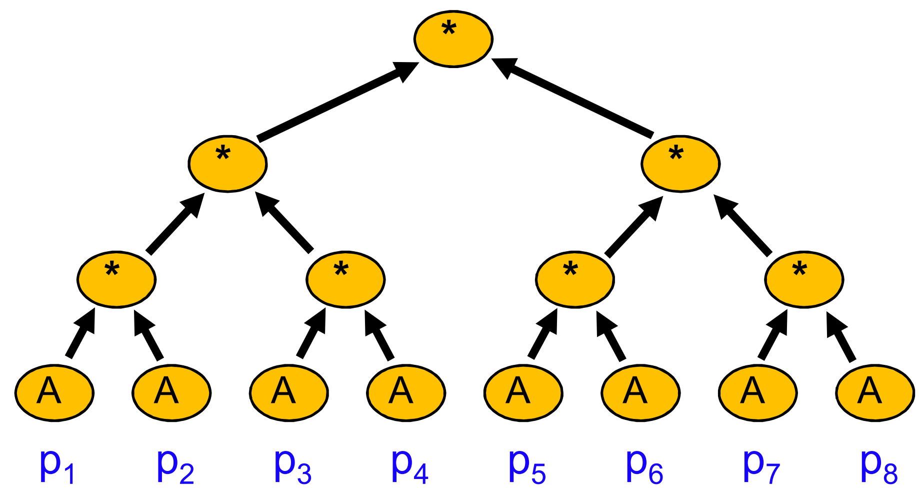
- Fibonacci numbers
  - $F_1=F_2=1$.
  - $F_n=F_{n-1}+F_{n-2}$, for n>2.
  - 1,1,2,3,5,8,13,21,34,...
- Computing F_n sequentially takes O(n) time.
- Can we compute F_1, F_2, ..., F_n in parallel?
  - Looking at the LDG, it seems there's no parallelism available.



Fibonacci numbers in parallel

- A simple identity.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$$



- Let A= $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
- By repeatedly applying the identity, we get

$$A^n \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix}$$

- So if we can quickly compute A^n in parallel, we can compute F_{n+2}.
  - Can compute F_n in $O(\log n)$ time with n processors.