Automatic Parallelization: Parallelism and Tiling

Roshan Dathathri

Department of Computer Science and Automation
Indian Institute of Science

roshan@csa.iisc.ernet.in

June 25, 2013

Goals of program transformations/optimizations

Goals of program transformations/optimizations

- Increase performance
  - Execute lesser code - e.g., Loop Invariant Code Motion
  - Execute more efficient code - e.g., Algebraic Reassociation
  - Utilize memory efficiently - e.g., Loop Tiling
  - Parallelize execution

Goals of program transformations/optimizations

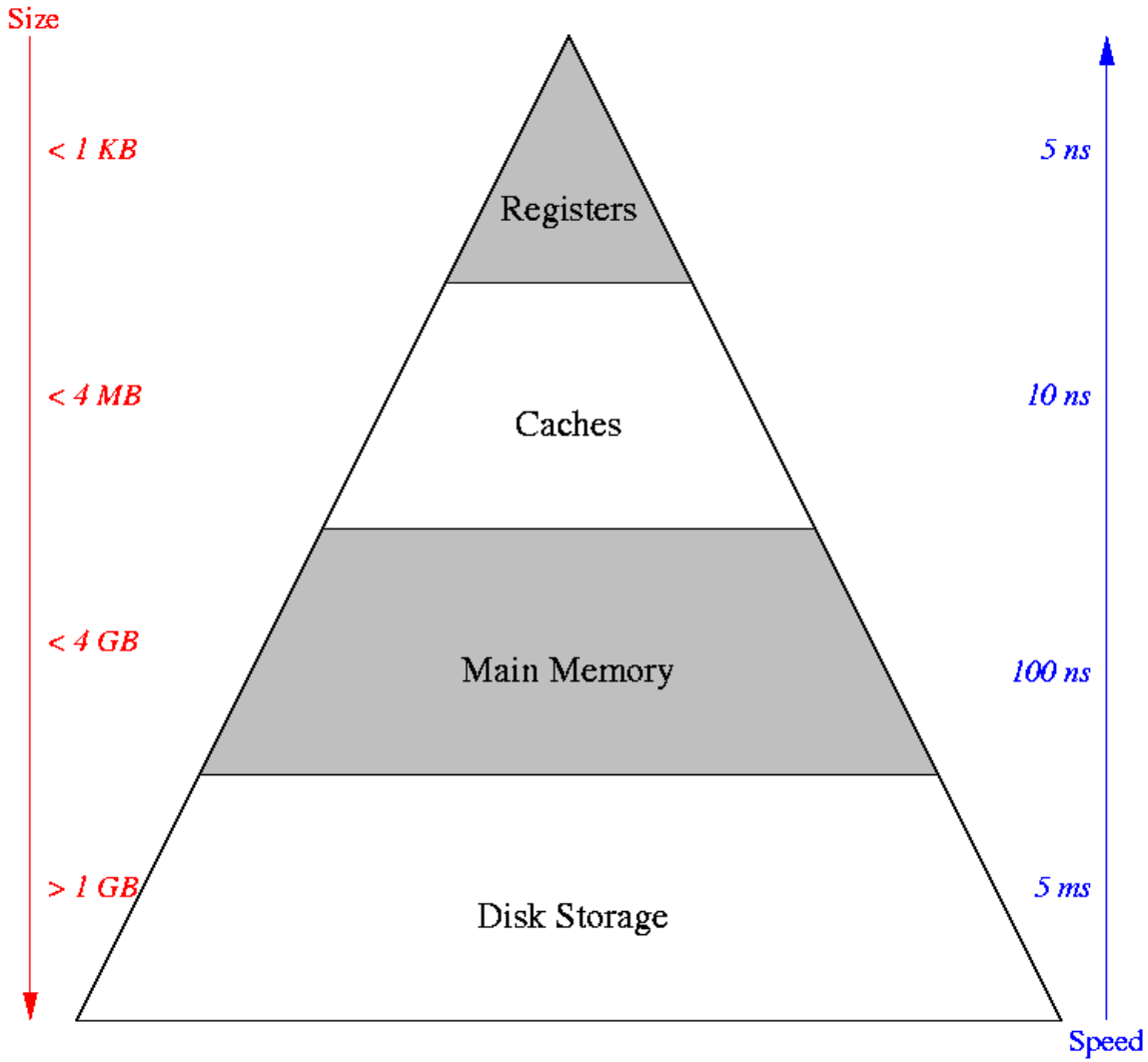
- Increase performance
  - Execute lesser code - e.g., Loop Invariant Code Motion
  - Execute more efficient code - e.g., Algebraic Reassociation
  - Utilize memory efficiently - e.g., Loop Tiling
  - Parallelize execution
- Reduce memory footprint
- Reduce energy usage

Goals of program transformations/optimizations

- Increase performance
  - Execute lesser code - e.g., Loop Invariant Code Motion
  - Execute more efficient code - e.g., Algebraic Reassociation
  - Utilize memory efficiently - e.g., Loop Tiling
  - Parallelize execution
- Reduce memory footprint
- Reduce energy usage

Today: Source code transformations

Memory Hierarchy



Data Locality

- Same memory location or related memory locations being frequently accessed
- Different classes of locality:
  - Spatial locality
  - Temporal locality
  - Group locality

Spatial locality

- Elements close-by (in space/memory) tend to be referenced soon
- e.g., c[i][j] in the code below

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Spatial locality

- Elements close-by (in space/memory) tend to be referenced soon
- e.g., c[i][j] in the code below

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

- Innermost dimension of the array should vary the fastest, by a constant

Which code exploits spatial reuse of c[i][j] better?

Snippet 1

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Snippet 2

```
for (k=0; k<N; k++) {
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Table: Matrix multiplication code

Temporal locality

- Same element tends to be referenced soon
- e.g., c[i][j] in the code below

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Temporal locality

- Same element tends to be referenced soon
- e.g., c[i][j] in the code below

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

- Rank of an access function is less than the dimensionality of the loop nest

Which code exploits temporal reuse of c[i][j] better?

Snippet 1

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Snippet 2

```
for (k=0; k<N; k++) {
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Table: Matrix multiplication code

Group locality

- Multiple accesses of the same array tend to reference the same element soon
- e.g., $a[i+1], a[i], a[i-1]$ in the code below

```
for (t = 0; t < T-1; t++) {
  for (i = 1; i < N+1; i++) {
    temp[i] = 0.125 * (a[i+1] - 2.0 * a[i] + a[i-1]);
  }
  for (i = 1; i < N+1; i++) {
    a[i] = temp[i];
  }
}
```

Loop Tiling/Blocking

- Executing iteration space in blocks: block-after-block
- Most important of all loop transformations
- Crucial for locality and parallelism

Example – Tiling

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Original code

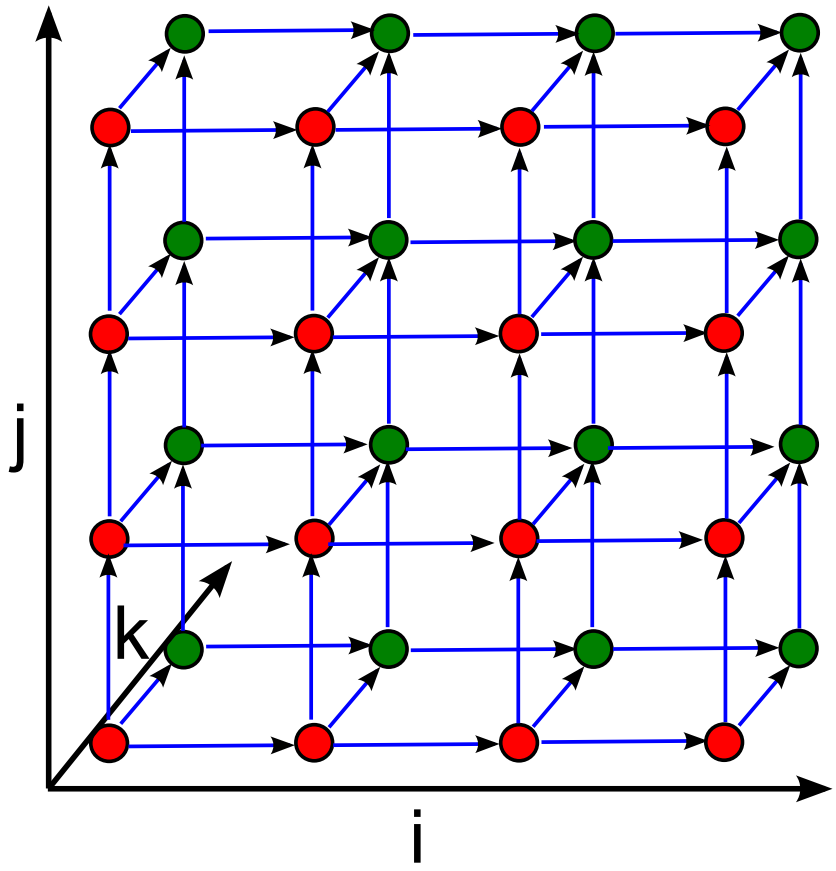


Figure: Locality in i, j, k dimensions

Example – Tiling

```
// inter-tile iterators
for (iT=0; iT<N; iT+=B) {
  for (jT=0; jT<N; jT+=B) {
    for (kT=0; kT<N; kT+=B) {
      // intra-tile iterators
      for (i=iT; i<iT+B; i++) {
        for (j=jT; j<jT+B; j++) {
          for (k=kT; k<kT+B; k++) {
            c[i][j] += a[i][k]*b[k][j];
          }
        }
      }
    }
  }
}
```

Tiled code with tile size $B * B * B$

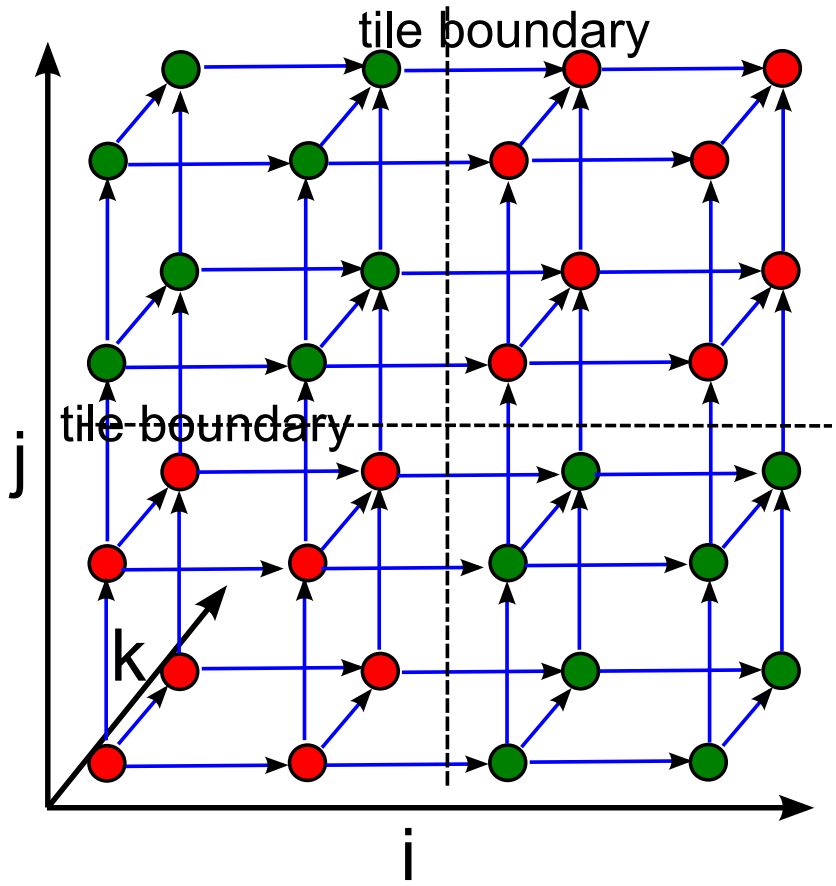


Figure: Exploiting reuse in i, j, k dimensions

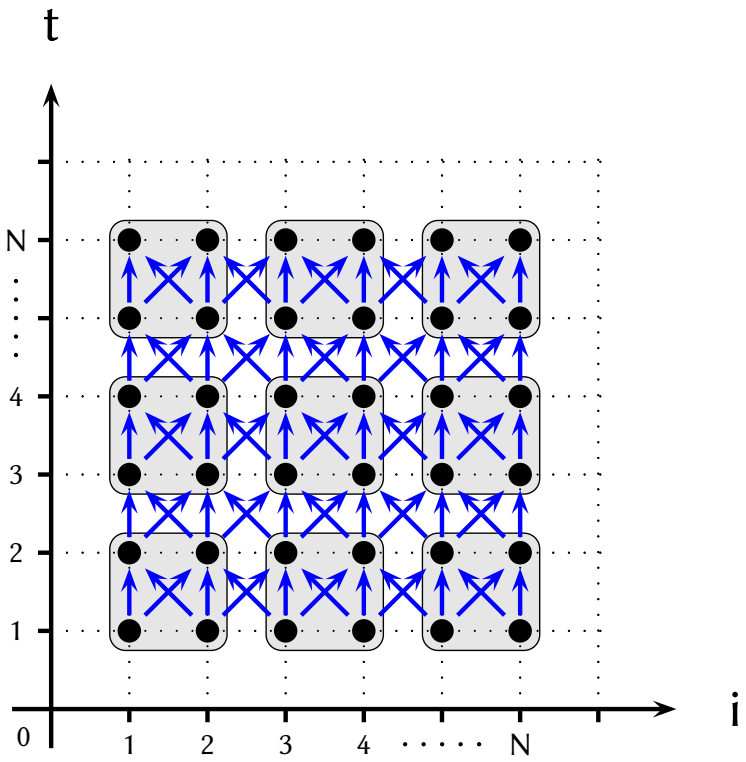
Tiling for Data Locality

- Tiling for caches
- Data touched by a tile should fit in faster memory
- Improves data reuse – allows reuse in multiple directions

Validity of Tiling

- A tile is a piece of computation that can execute atomically in its entirety
- Should be able to construct a total order on the set of all tiles

Example – Validity of Tiling




```
for (t=0; t<T; t++) {
  for (i=2; i<N-1; i++) {
    a[t][i] += 0.333*(a[t-1][i]+
      a[t-1][i-1]+a[t-1][i+1]);
  }
}
```

Original code

Figure: Dependences (1,0), (1,1), (1,-1)

Example – Validity of Tiling

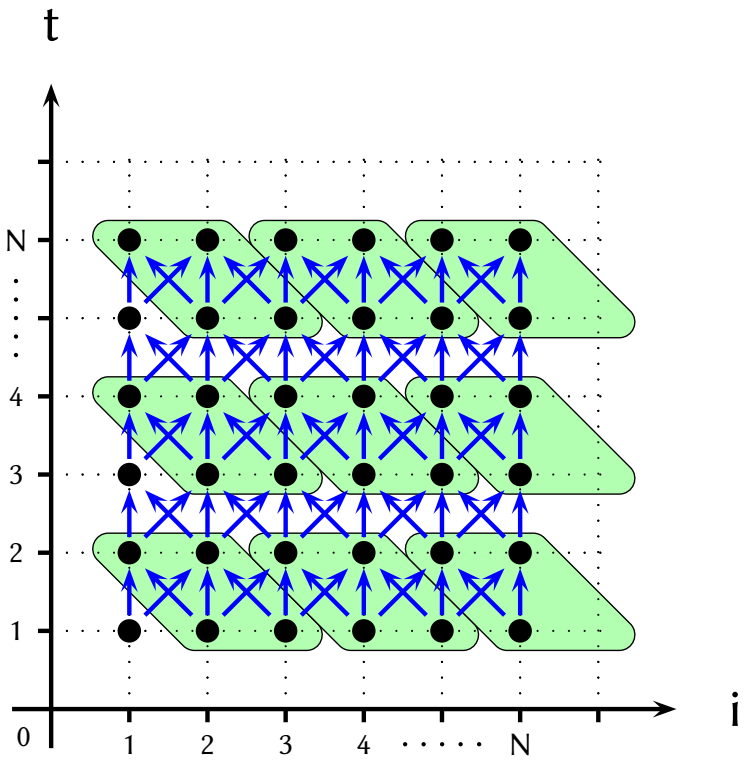
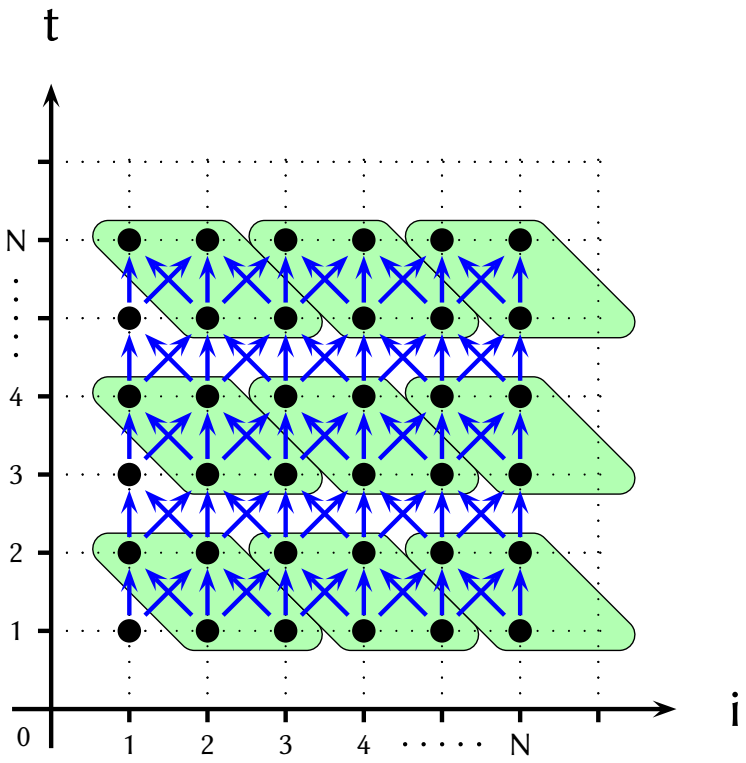


Figure: Dependences (1,0), (1,1), (1,-1)

Example – Validity of Tiling




```
for (t1=0; t1<=T-1;t1++) {
  for (t2=t1+2; t2<=t1+N-2;t2++) {
    a[t1][-t1+t2]+=0.333*(a[t1-1][-t1+t2]+
      a[t1-1][-t1+t2-1]+a[t1-1][-t1+t2+1]);
  }
}
```

Skewed code

Figure: Dependences (1,0), (1,1), (1,-1)

Validity of Tiling

- With distance vectors and tiling along original dimensions, all dependence components along dimensions being tiled should be non-negative

Validity of Tiling

- With distance vectors and tiling along original dimensions, all dependence components along dimensions being tiled should be non-negative
- With dependence polyhedron D, valid tiling hyperplanes, $\mathbf{h}$: $\mathbf{h}.\mathbf{D} \geq \mathbf{0}$

Validity of Tiling

- With distance vectors and tiling along original dimensions, all dependence components along dimensions being tiled should be non-negative
- With dependence polyhedron D, valid tiling hyperplanes, $\mathbf{h}$: $\mathbf{h}.\mathbf{D} \geq \mathbf{0}$

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 0 \end{pmatrix}$$

Validity of Tiling

- With distance vectors and tiling along original dimensions, all dependence components along dimensions being tiled should be non-negative
- With dependence polyhedron D, valid tiling hyperplanes, **h**: $\mathbf{h}.\mathbf{D} \geq \mathbf{0}$

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 0 \end{pmatrix}$$

- Consider dependences (1,0,1), (1, -2, 0), (0,1,0), (0,0,1): what kind of tiling is valid?

Validity of Tiling

- With distance vectors and tiling along original dimensions, all dependence components along dimensions being tiled should be non-negative
- With dependence polyhedron D, valid tiling hyperplanes, **h**: $\mathbf{h}.\mathbf{D} \geq \mathbf{0}$

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 0 \end{pmatrix}$$

- Consider dependences (1,0,1), (1, -2, 0), (0,1,0), (0,0,1): what kind of tiling is valid?

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & -2 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

Example

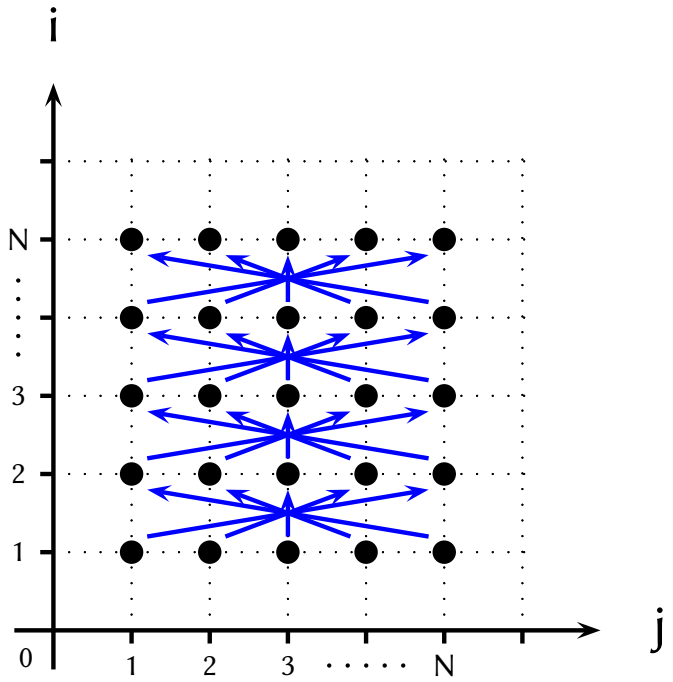


Figure: No 2-D tiling possible

Different kinds of parallelism

- Outer parallelism / communication-free parallelism
- Inner parallelism
- Pipelined parallelism

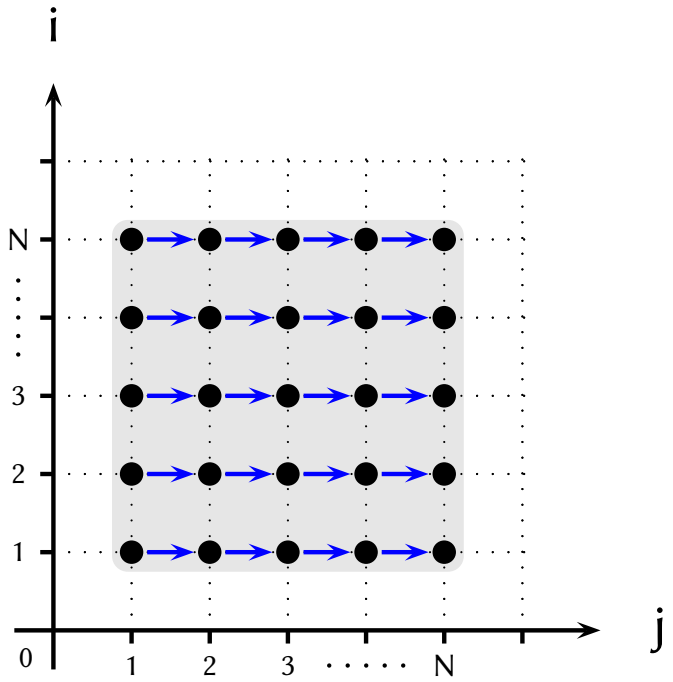
- Reduction parallelism
- SIMD (Singe Instruction Multiple Data) parallelism or vectorization

Different kinds of parallelism

- Outer parallelism / communication-free parallelism
- Inner parallelism
- Pipelined parallelism

- Reduction parallelism
- SIMD (Singe Instruction Multiple Data) parallelism or vectorization

Outer parallelism (loops)



Outer parallelism (loops)

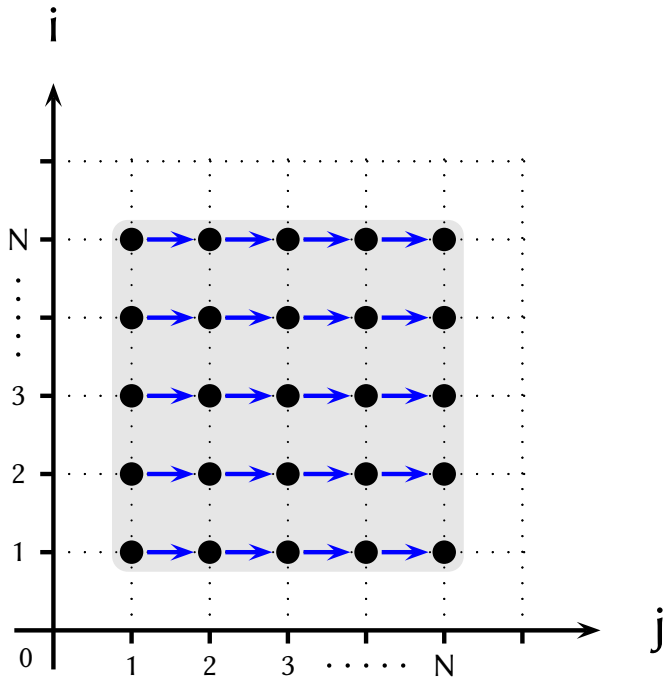
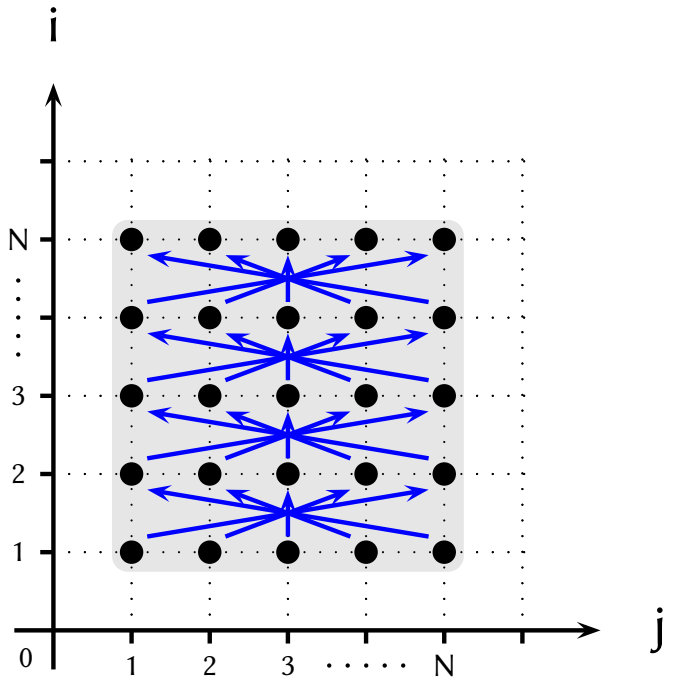


Figure: Outer parallel loop i

Inner parallelism (loops)



Inner parallelism (loops)

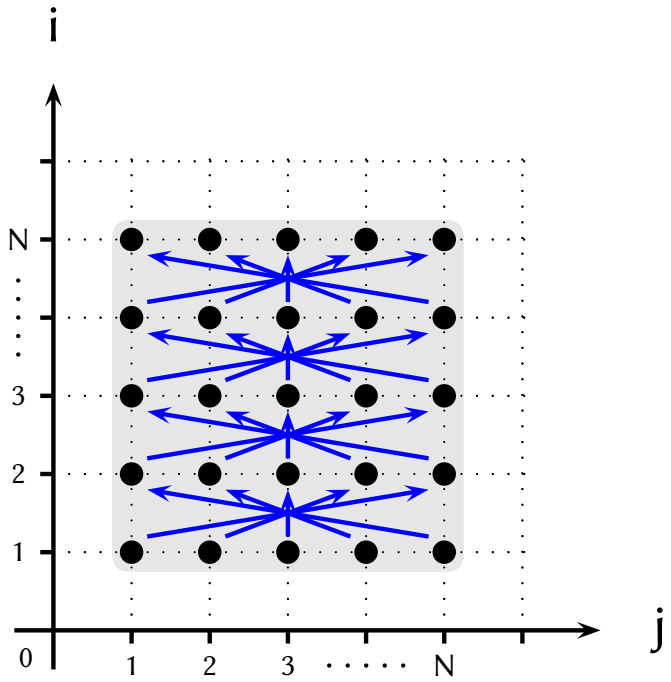
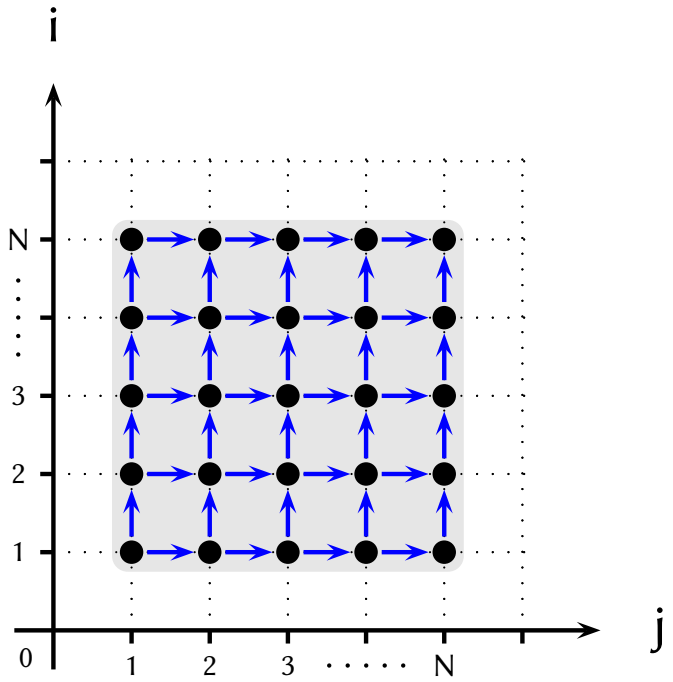


Figure: Inner parallel loop, j

Pipelined parallelism (loops)



Pipelined parallelism (loops)

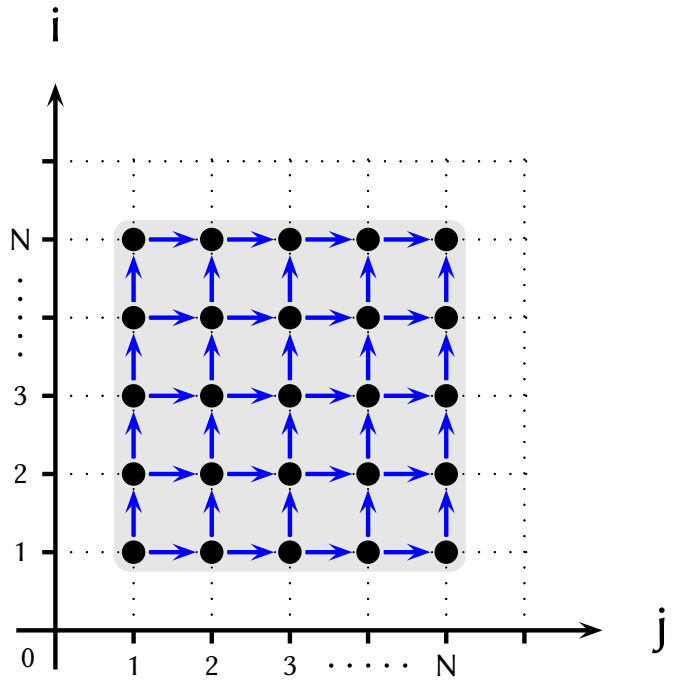


Figure: Pipelined parallel loop

Coarse-grained pipelined parallelism

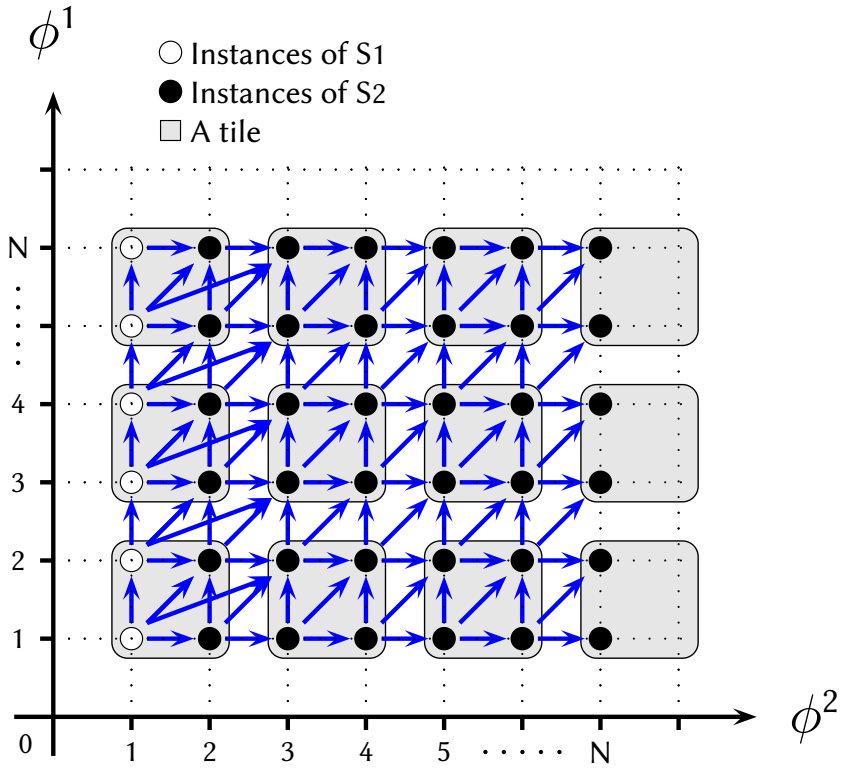


Figure: Tiling an iteration space

Tiling for Parallelism

- Achieves coarse-grained parallelization
- Reduces frequency of synchronization - improves computation to communication ratio
- Can reduce volume of communication
- How does the tile size affect parallelism?

Tiling for Parallelism

- Achieves coarse-grained parallelization
- Reduces frequency of synchronization - improves computation to communication ratio
- Can reduce volume of communication
- How does the tile size affect parallelism?
  - Larger -> lesser frequency of synchronization, more load-imbalance
  - Smaller -> more frequency of synchronization, reduces load-imbalance

Tiling is directly related to parallelism

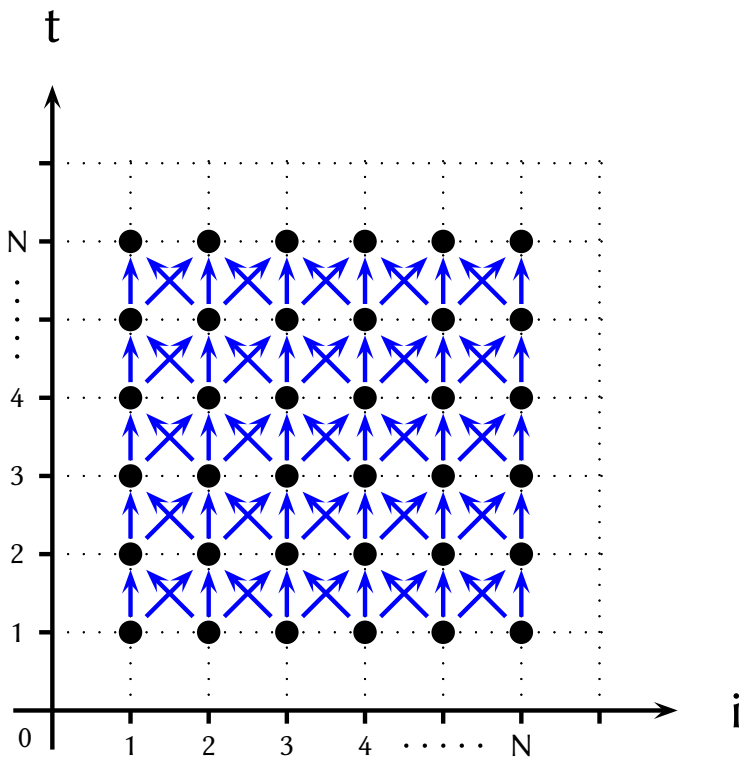


Figure: Dependences (1,0), (1,1), (1,-1)

Tiling is directly related to parallelism

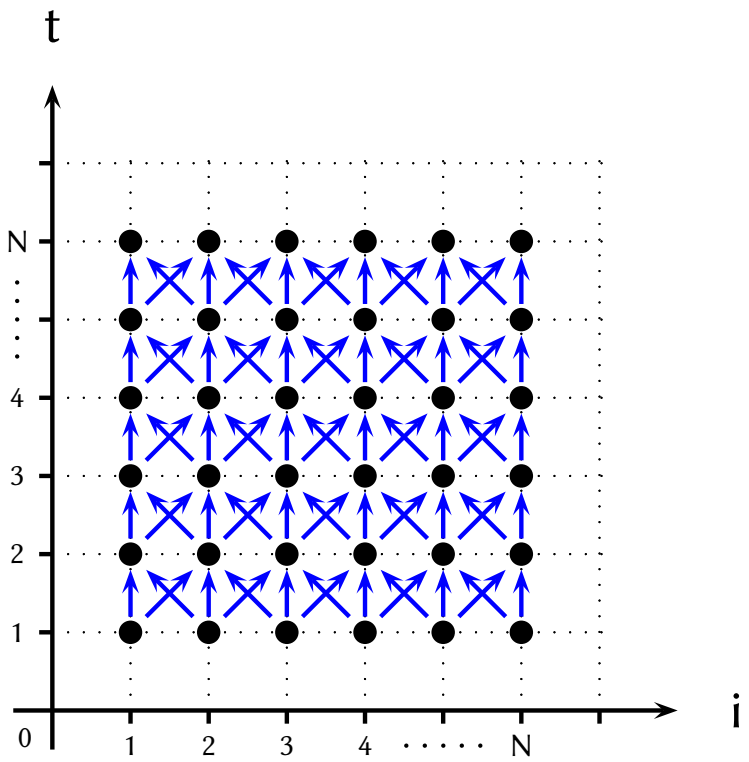


Figure: Dependences (1,0), (1,1), (1,-1)

- Tiling is valid -> Parallelism (at least pipelined parallelism) exists

Tiling is directly related to parallelism

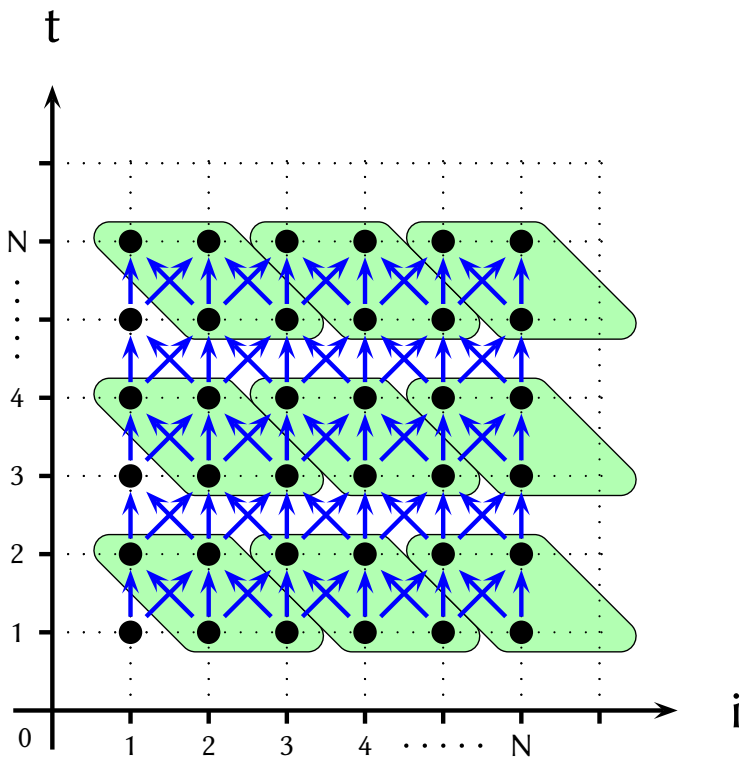


Figure: Dependences (1,0), (1,1), (1,-1)

- Tiling is valid -> Parallelism (at least pipelined parallelism) exists

References

- **PLUTO** - An automatic parallelizer and locality optimizer for multicores
  `http://pluto-compiler.sourceforge.net/`
- **PoCC** - The Polyhedral Compiler Collection
  `http://www.cse.ohio-state.edu/~pouchet/software/pocc/`

Complexity of architectures and input code