Teraflux, from the programming model to the execution model

Antoniu Pop, Feng Li and Albert Cohen

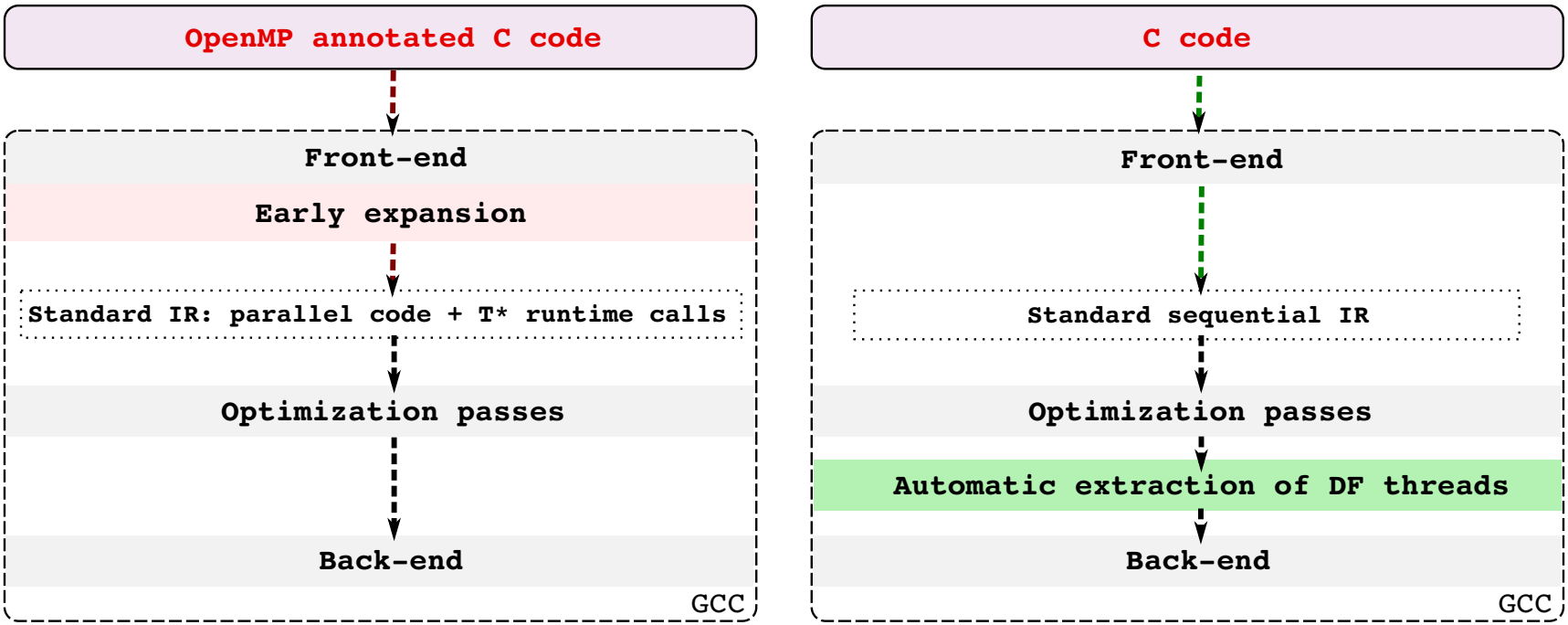
INRIA and École Normale Supérieure
http://www.di.ens.fr/ParkasTeam.html

CASTNESS'12 — Paris, January 26, 2012

Outline

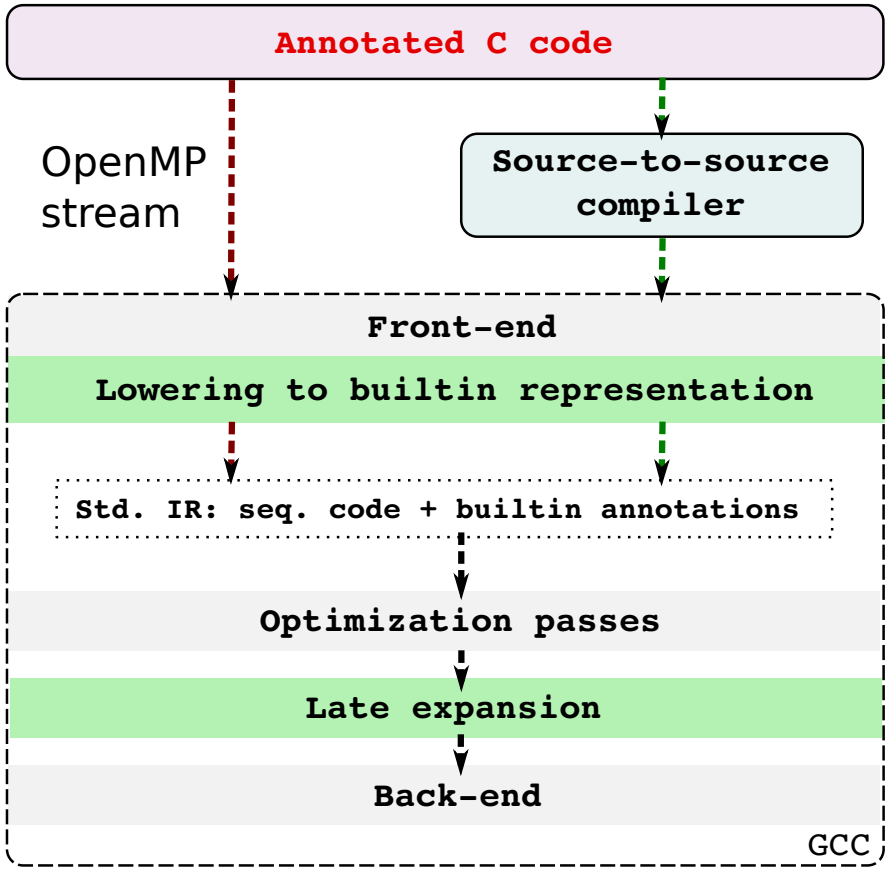
1. Teraflux Compilation Flow

Current compilation flow



- Automatic extraction handles control dependences and scalar data dependences
- Complementary OpenMP streaming annotations capture arbitrary DF
- Early expansion only allows to apply the automatic extraction to further refine granularity within larger annotated OpenMP tasks

Objective: integrate high-level DF information in DF analysis



- Merged compilation flows
- Common intermediate representation: OpenMP + stream-computing extension

2. Compilation of OpenMP Streaming programs to T*

T* runtime API

- `void *tcreate(void (*func)(), int sc, int size);`
  Thread creation and frame allocation.

- `void tdecrease(void *fp, int num);`
  Decrements the synchronization counter of the thread associated with frame pointer `fp` by `num`.

- `void tend();`
  Terminates the current thread and deallocates its frame.

- `void *tget_cfp();`
  Returns the frame pointer of the current thread.

Compilation of OpenMP streaming programs to T*

- OpenMP streaming programs rely on channels for communication

```
int x;

#pragma omp task output (x) // T1
  x = ...;

#pragma omp task input (x) output (y) // T2
  y = foo (x);
```

Compilation of OpenMP streaming programs to T*

- OpenMP streaming programs rely on channels for communication

```
int x;

#pragma omp task output (x) // T1
  x = ...;

#pragma omp task input (x) output (y) // T2
  y = foo (x);
```

- Use stream views to access data and link to owner frame for synchronization

```
struct view {
  void *data;
  // frame pointer to owner (consumer) thread
  void *owner;
} view_t, *view_p;
```

Compilation of OpenMP streaming programs to T*

- OpenMP streaming programs rely on channels for communication

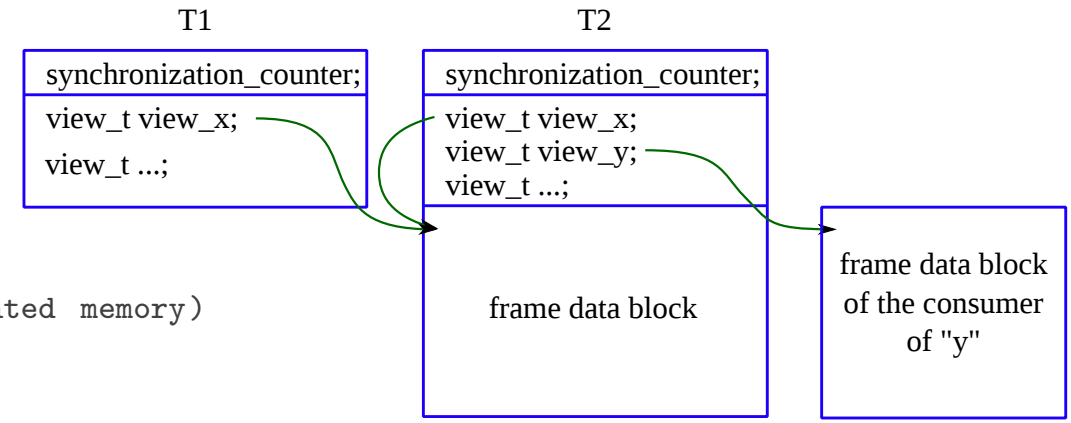
```
int x;

#pragma omp task output (x) // T1
  x = ...;

#pragma omp task input (x) output (y) // T2
  y = foo (x);
```

- Use stream views to access data and link to owner frame for synchronization

```
struct view {
  void *data;
  // frame pointer to owner (consumer) thread
  void *owner;
} view_t, *view_p;
```

- Replace communication channels by matching producer tasks to consumer tasks and giving producers access to write directly in the consumer's frame

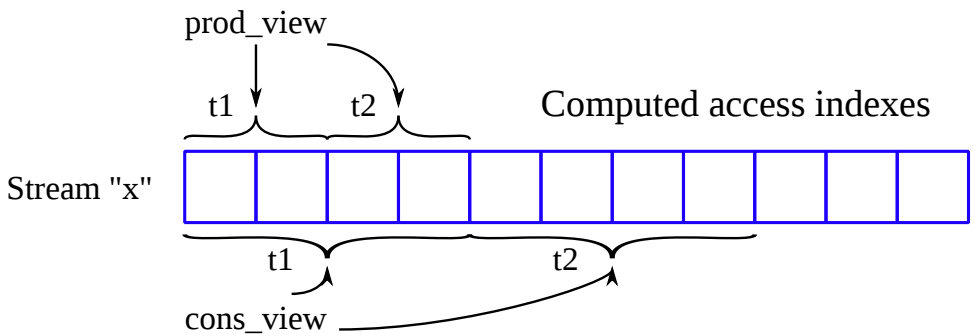
```
struct frame {
  int synchronization_counter;
  view_t view_x;
  view_t view_y;
  view_t ...
  // Data block starts here
  // (not a pointer to separately allocated memory)
  void *data_block;
} frame_t, *frame_p;
```

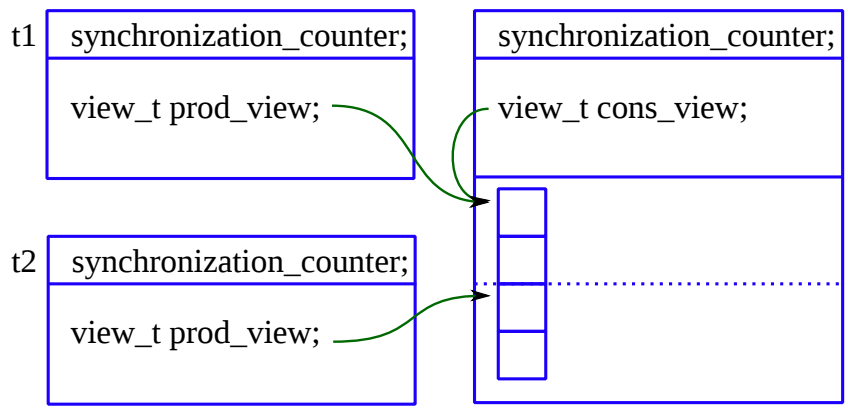


Stream communication and data-flow frames

```
int prod_burst = ..., cons_burst = ...;
int x, prod_view[prod_burst], cons_view[cons_burst];

#pragma omp task output (x >> prod_view[prod_burst])
    prod_view[0..prod_burst-1] = ...;

#pragma omp task input (x << cons_view[cons_burst])
    ... = cons_view[0..cons_burst-1];
```

↓ Dynamically resolve flow dependences between task activations ↓





Thread code generation: work function and synchronization

```
int X[x_burst], Y[y_burst];

#pragma omp task input (x >> X[x_burst]) output (y << Y[y_burst])
  {
    foo (X, Y);
  }
```

↓ work function code generation ↓

```
void work_function (void) {
  frame_type *fp = tget_cfp ();

  foo (fp->view_X.data, fp->view_Y.data); // Typically inlined (for a DF1 thread)

  tdecrease (fp->view_Y.owner, y_burst); // Owned by the consumer of stream ‘‘y’’
}
```

Control program code generation: resolve dependences

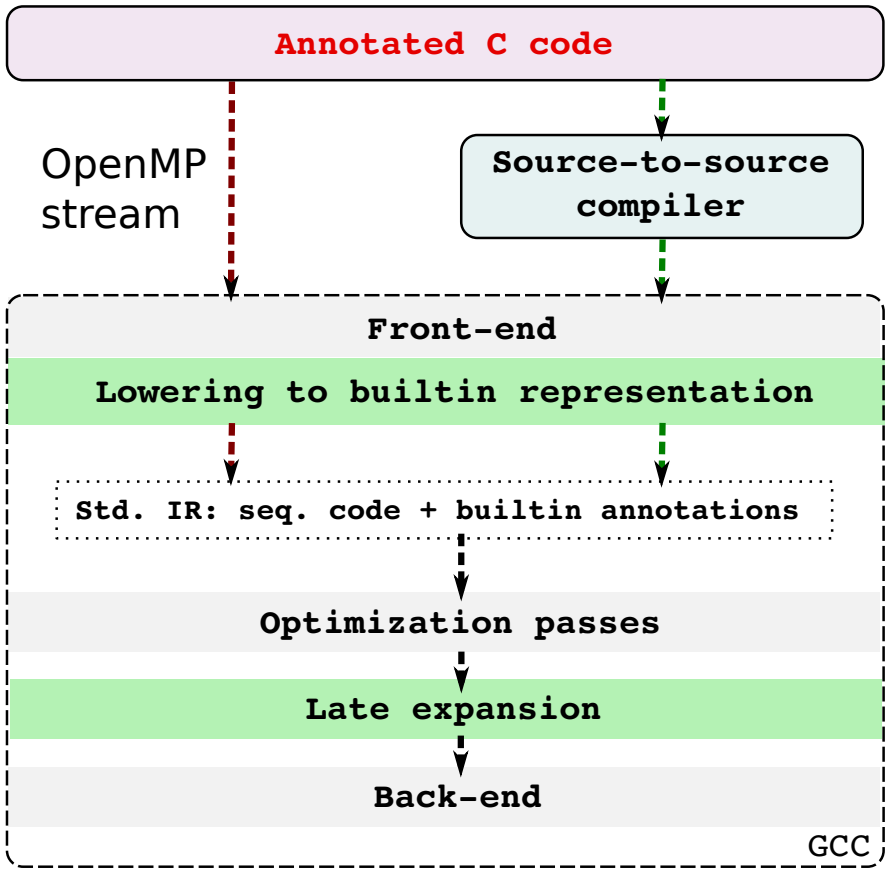
```
int X[x_burst], Y[y_burst];

#pragma omp task input (x >> X[x_burst]) output (y << Y[y_burst])
  {
    foo (X, Y);
  }
```

↓ control program code generation ↓

```
fp = tcreate (work_function, 1 + x_burst, sizeof (frame) + x_burst);

// input (x >> X[x_burst])
fp->X_view.data = &fp->data_block;
fp->X_view.owner = fp;
resolve_dependences (&fp->X_view, x, READ);

//output (y << Y[y_burst])
fp->Y_view.data = NULL; // Unknown for now as the data is stored within the consumer's frame
fp->Y_view.owner = NULL; // Unknown for now: dependence resolution will determine this
resolve_dependences (&fp->Y_view, y, WRITE);
```

3. Programming Model: OMPSs Compilation Flow

Compilation Flow for OMPSs Programs



Translation of OMPSs annotations to the OpenMP stream-computing extension

OMPSs annotations

- Task directives

```
#pragma omp task [clauses]
function-definition | function-declaration
```

OMPSs annotations

- Task directives

```
#pragma omp task [clauses]
function-definition | function-declaration
```

- Main clauses

```
input ([list of parameters])
output ([list of parameters])
inout ([list of parameters])
```

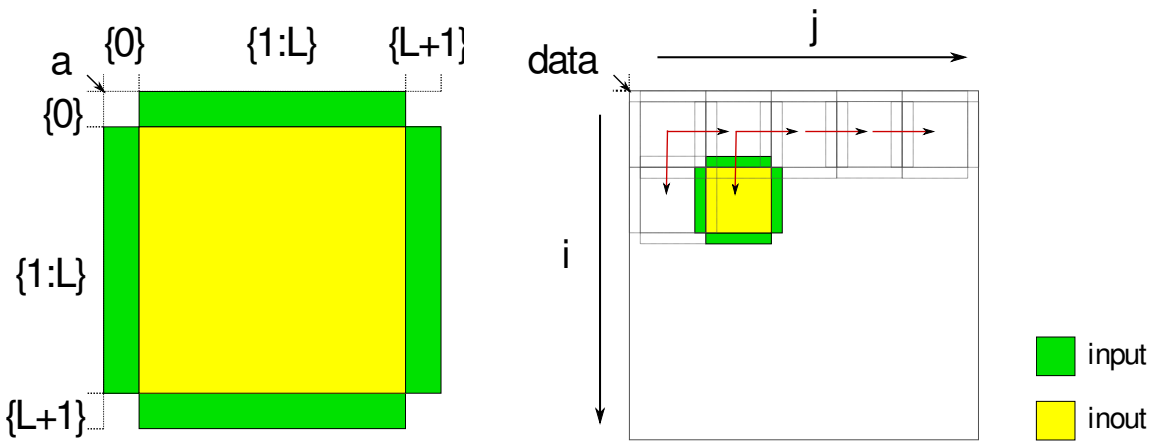
Running example: Gauss-Seidel[1]

```
#pragma css task input(a{0}{1:L}, a{L+1}{1:L}, a{1:L}{0}, a{1:L}{L+1}) inout(a{1:L}{1:L})
void gauss_seidel (double a[N][N]) {
  for (int i=1; i<=L; i++)
    for (int j=1; j<=L; j++)
      a[i][j] = 0.2 * (a[i][j] + a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
}

for (int it=0; it < NITERS; it++)
  for (int i=0; i<N-2; i+=L)
    for (int j=0; j<N-2; j+=L)
      gauss_seidel(&data[i][j]);
```

[1] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. 2010. *Handling task dependencies under strided and aliased references.* In Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10). ACM, New York, NY, USA, 263-274.

Running example: Gauss-Seidel[1]

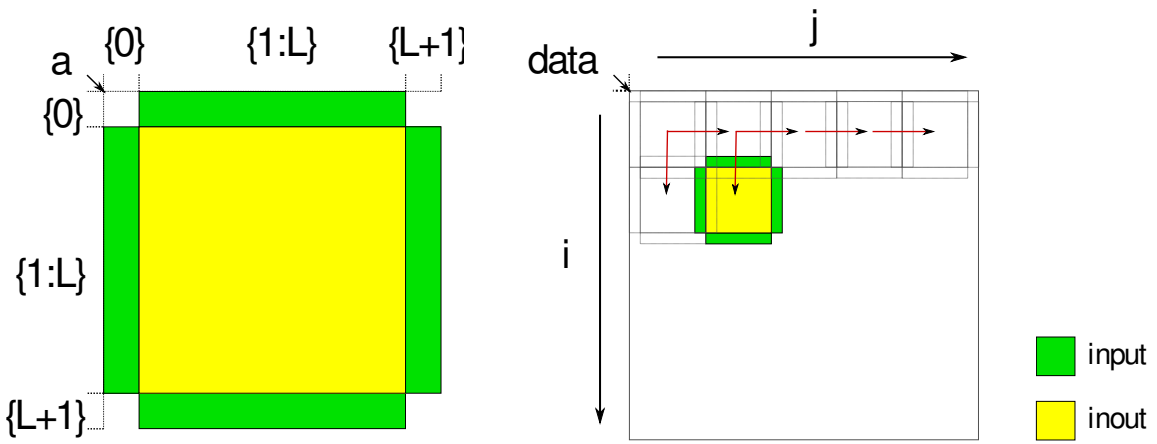
```
#pragma css task input(a{0}{1:L}, a{L+1}{1:L}, a{1:L}{0}, a{1:L}{L+1}) inout(a{1:L}{1:L})
void gauss_seidel (double a[N][N]) {
  for (int i=1; i<=L; i++)
    for (int j=1; j<=L; j++)
      a[i][j] = 0.2 * (a[i][j] + a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
}

for (int it=0; it < NITERS; it++)
  for (int i=0; i<N-2; i+=L)
    for (int j=0; j<N-2; j+=L)
      gauss_seidel(&data[i][j]);
```



[1] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. 2010. *Handling task dependencies under strided and aliased references.* In Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10). ACM, New York, NY, USA, 263-274.

Running example: Gauss-Seidel[1]

```
#pragma css task input(a{0}{1:L}, a{L+1}{1:L}, a{1:L}{0}, a{1:L}{L+1}) inout(a{1:L}{1:L})
void gauss_seidel (double a[N][N]) {
  for (int i=1; i<=L; i++)
    for (int j=1; j<=L; j++)
      a[i][j] = 0.2 * (a[i][j] + a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
}

for (int it=0; it < NITERS; it++)
  for (int i=0; i<N-2; i+=L)
    for (int j=0; j<N-2; j+=L)
      gauss_seidel(&data[i][j]);
```



- Resolve dependences at runtime by detecting collisions between region accesses
- Enforce dependences between tasks (no privatization)

[1] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. 2010. *Handling task dependencies under strided and aliased references.* In Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10). ACM, New York, NY, USA, 263-274.

Enforcing dependences through streams

Naive approach

- Use one stream to synchronize each region
- Streams encode version chains

Enforcing dependences through streams

Naive approach

- Use one stream to synchronize each region
- Streams encode version chains

- Streams used as semaphores
- No streaming benefits

Enforcing dependences through streams

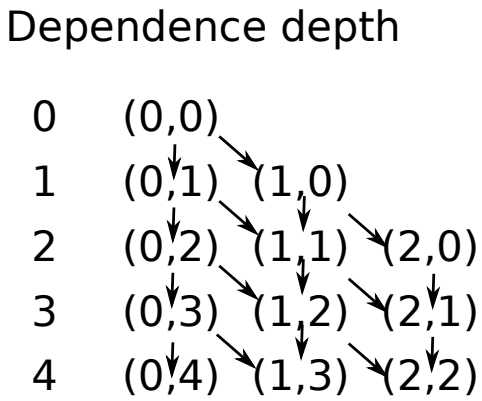
Naive approach

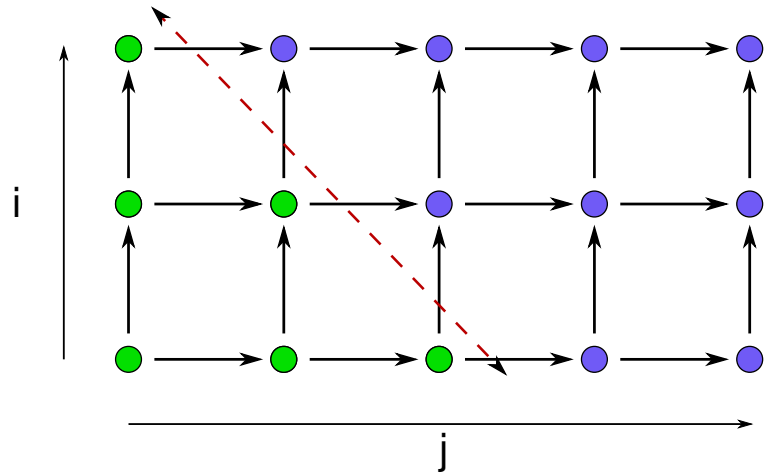
- Use one stream to synchronize each region
- Streams encode version chains

- Streams used as semaphores
- No streaming benefits

Wavefront approach

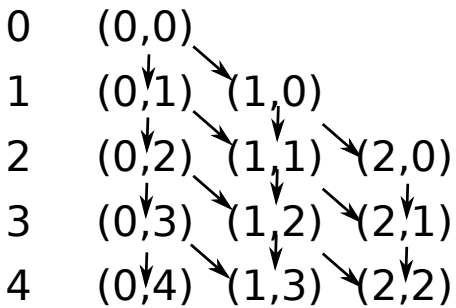
Topologically sorting task activations

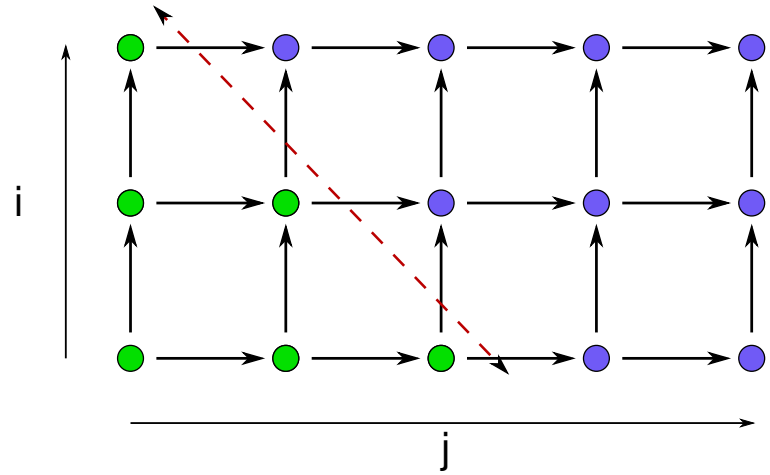




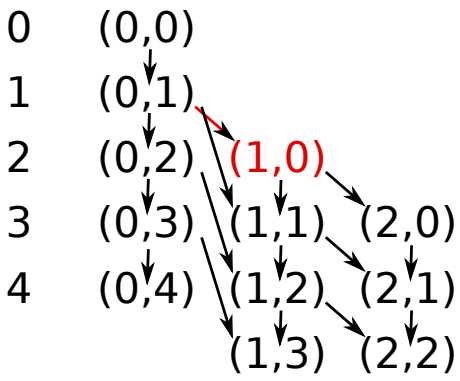
Topologically sorting task activations

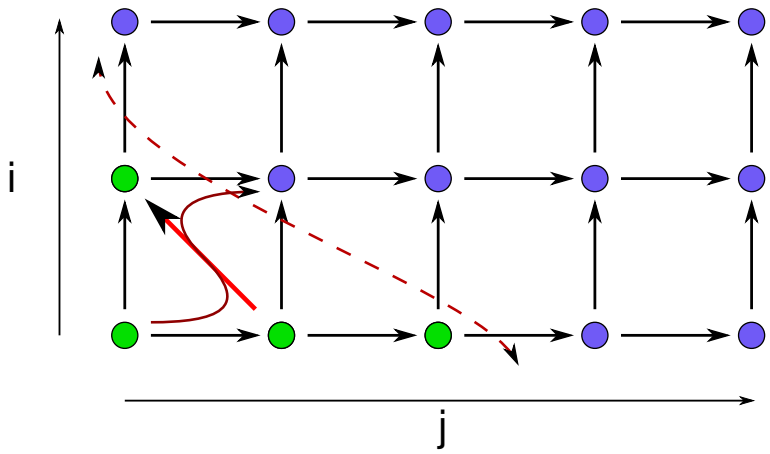
Dependence depth



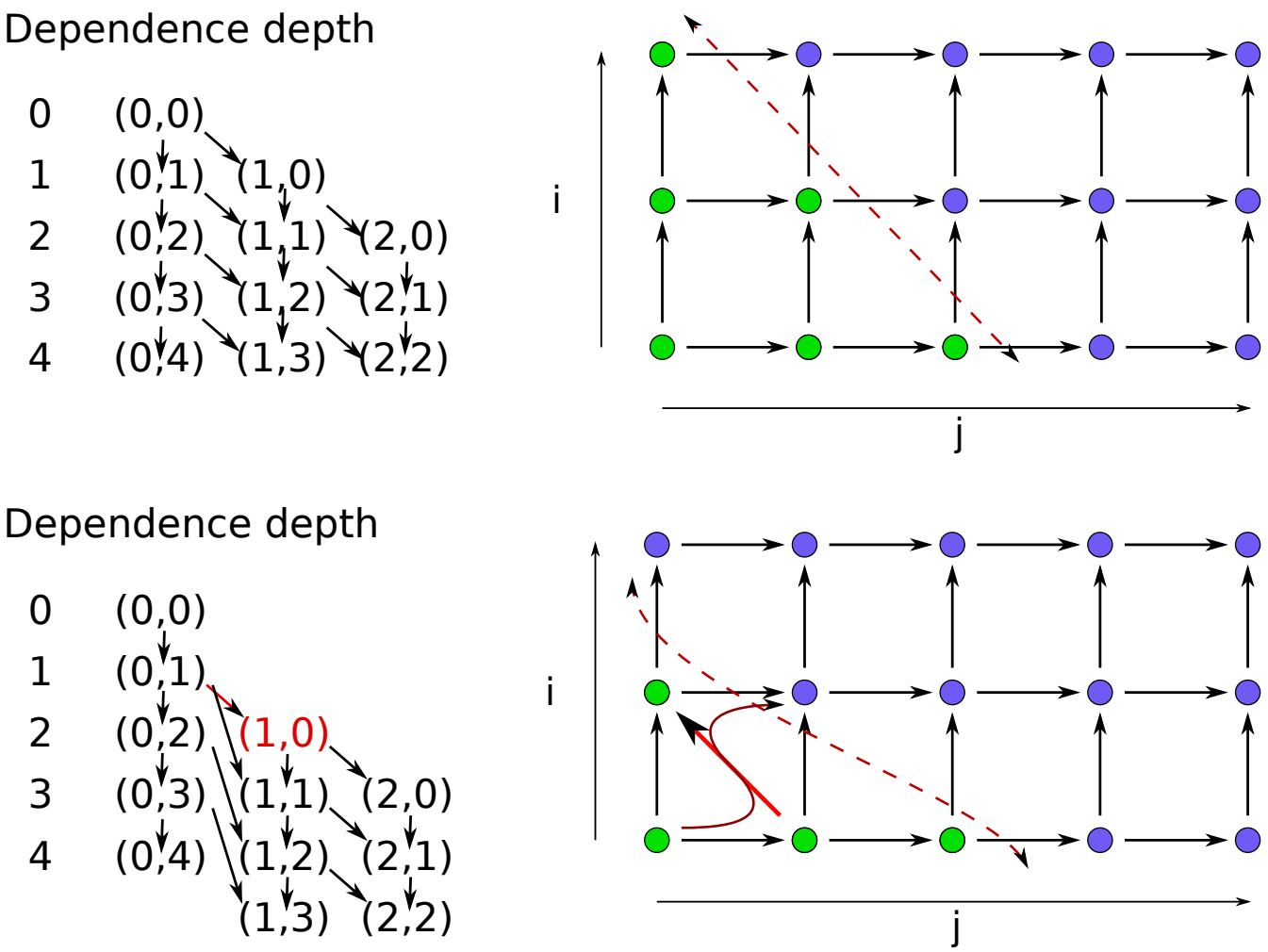


Dependence depth





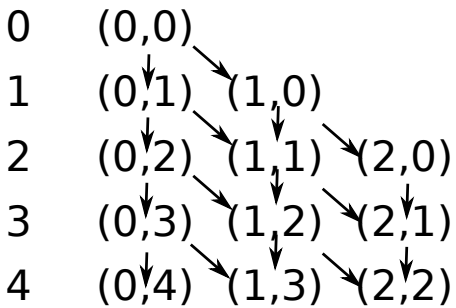
Topologically sorting task activations

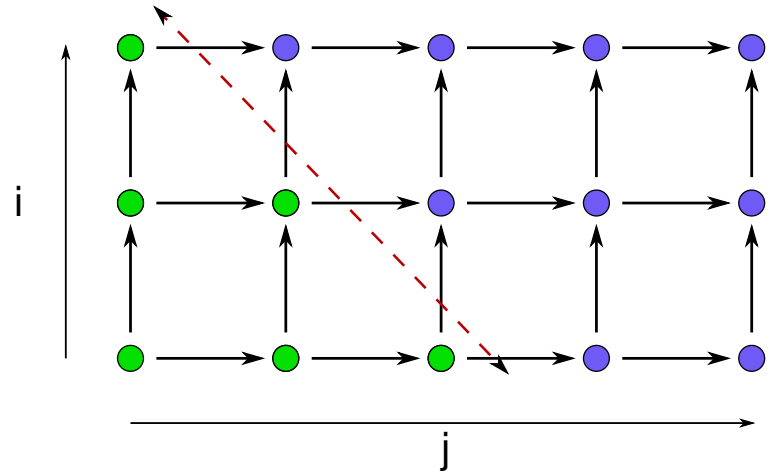


- Dynamically build a wavefront schedule

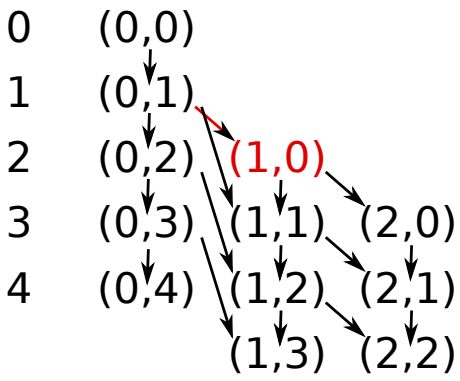
Topologically sorting task activations

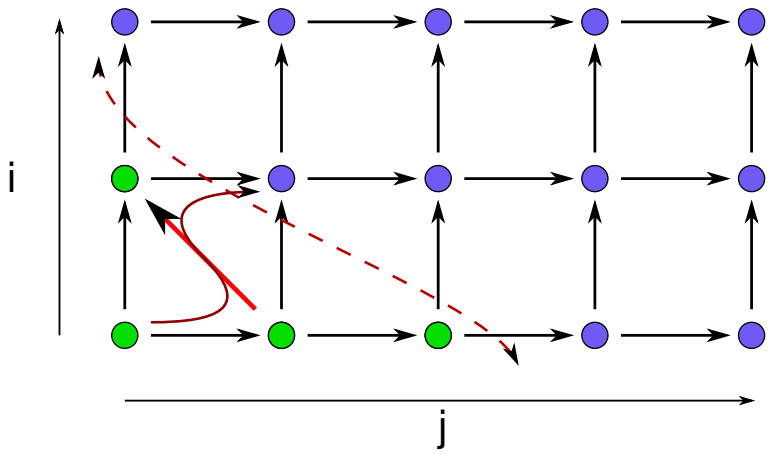
Dependence depth





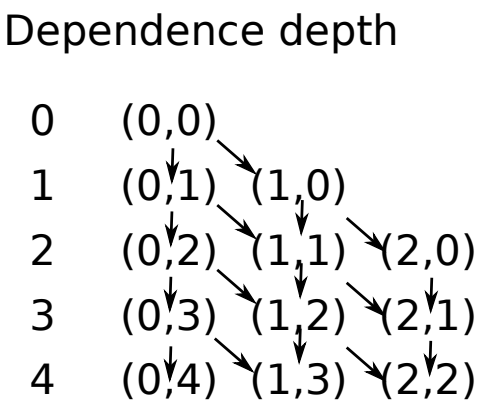
Dependence depth

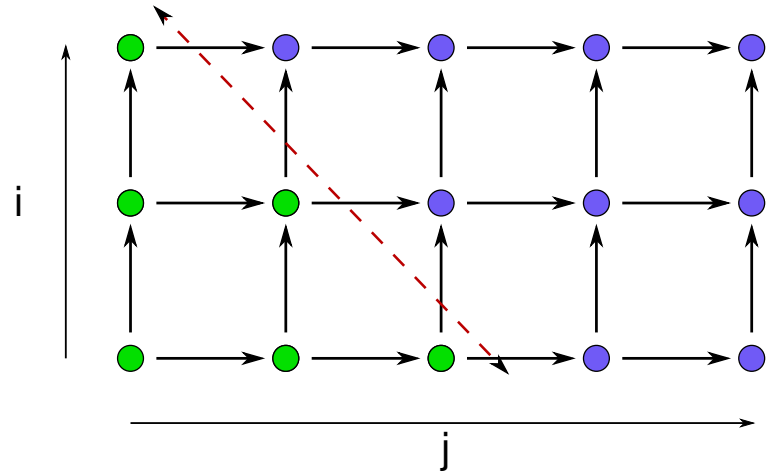


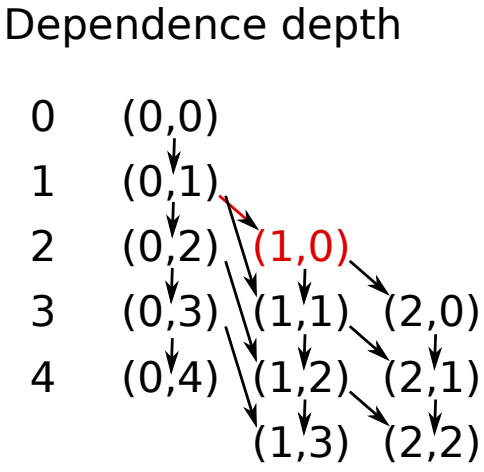


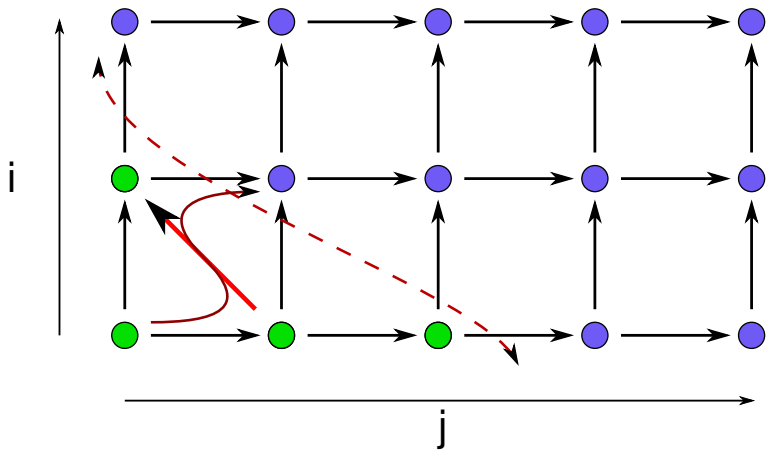
- Dynamically build a wavefront schedule
- Only downward dependences

Topologically sorting task activations



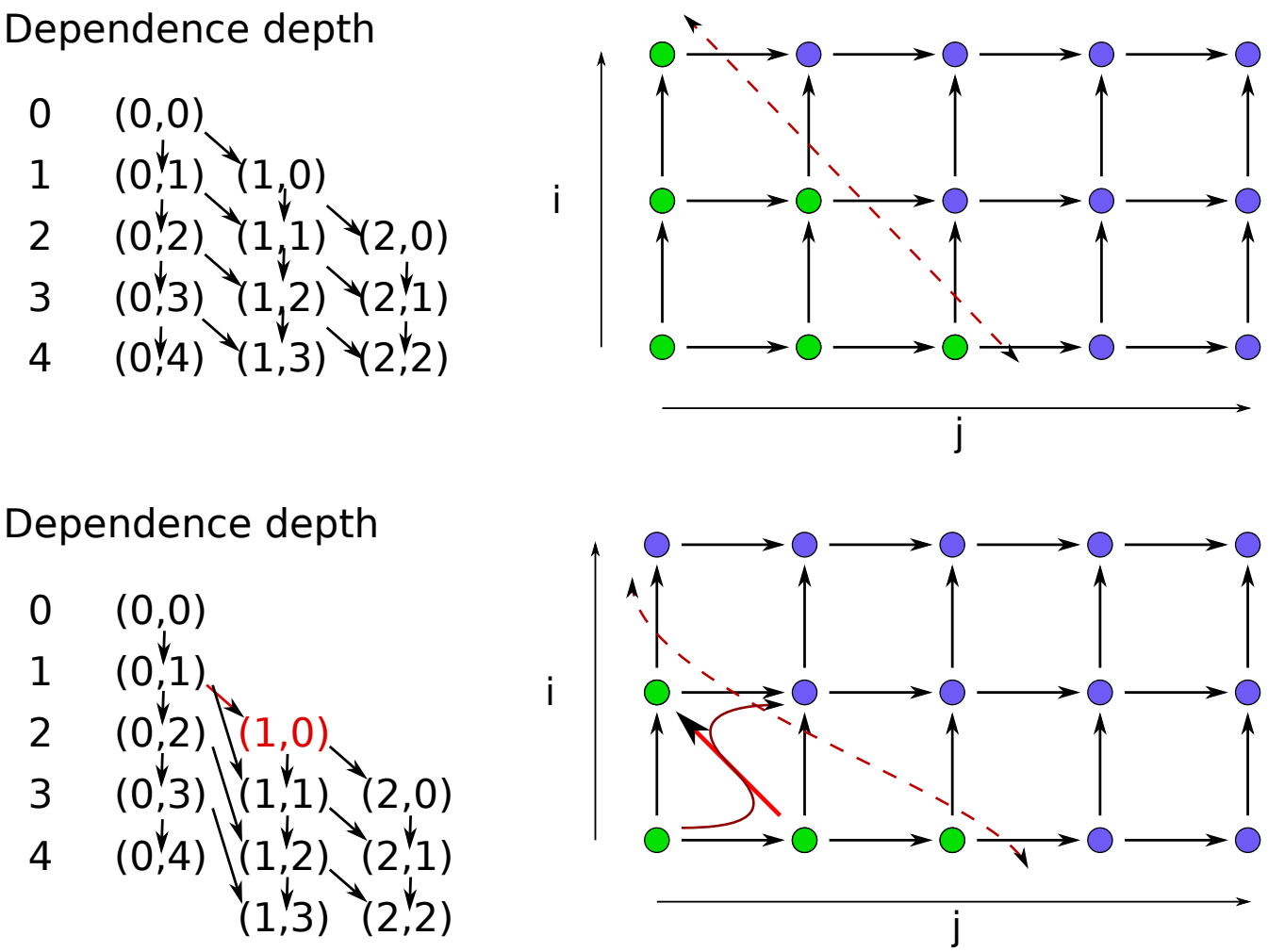






- Dynamically build a wavefront schedule
- Only downward dependences $\Rightarrow$ no deadlocks possible

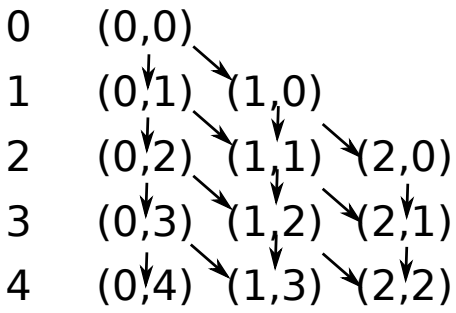
Topologically sorting task activations

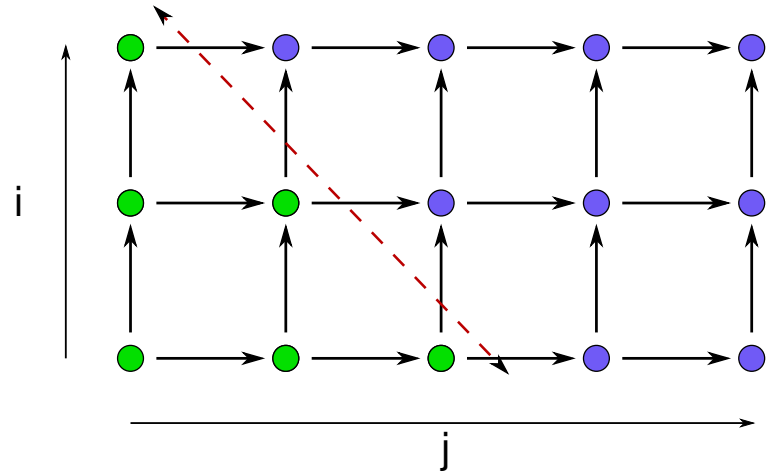


- Dynamically build a wavefront schedule
- Only downward dependences $\Rightarrow$ no deadlocks possible
- No horizontal dependences

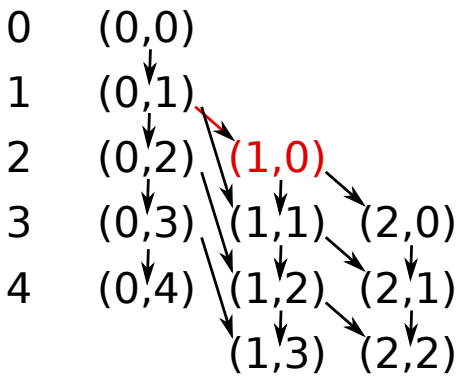
Topologically sorting task activations

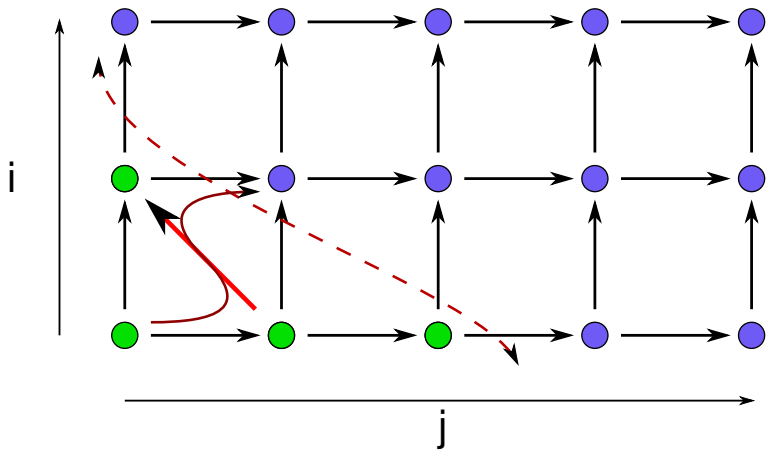
Dependence depth





Dependence depth

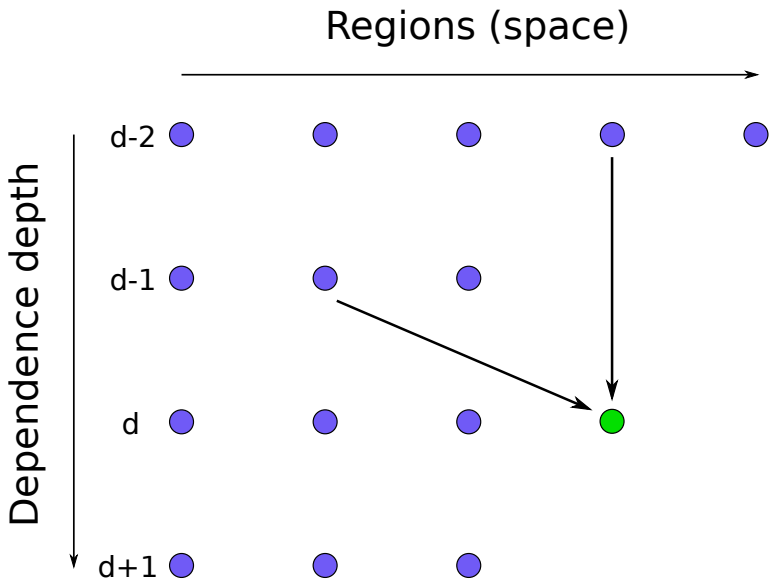




- Dynamically build a wavefront schedule
- Only downward dependences $\Rightarrow$ no deadlocks possible
- No horizontal dependences $\Rightarrow$ data parallelism available within each depth level

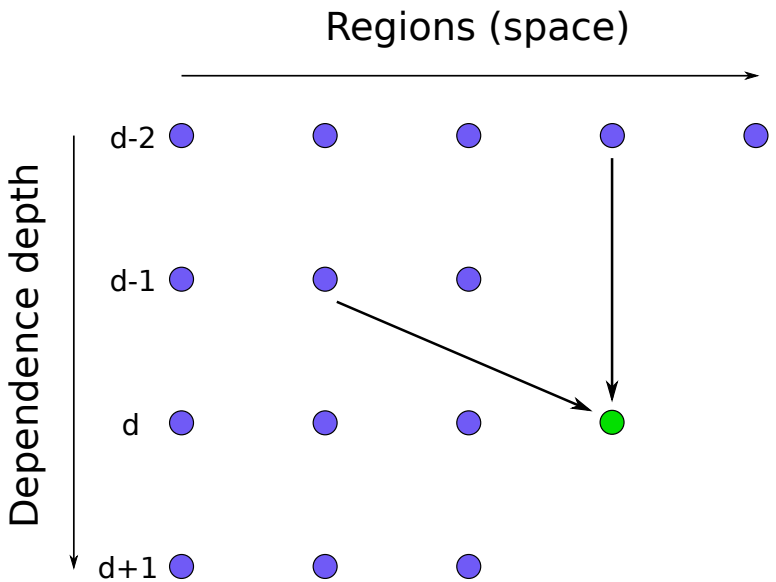
“Deep” stream synchronization

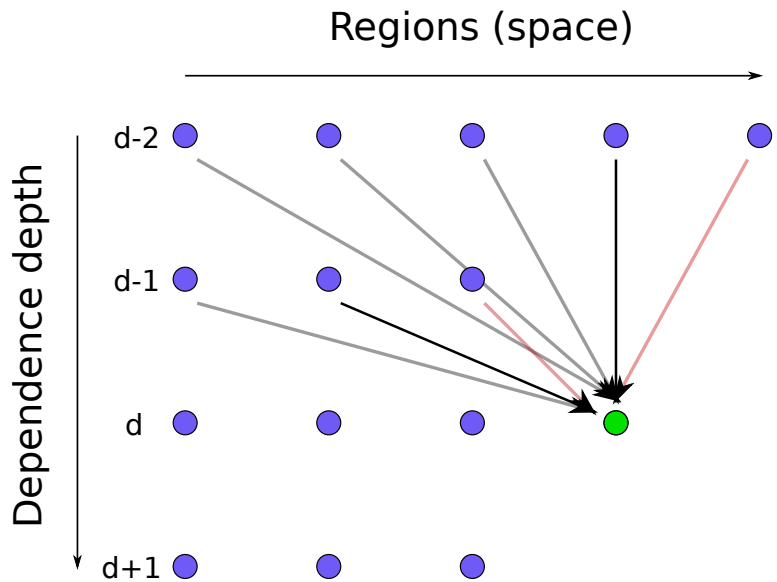
Stream dependences between each level of dependence depth



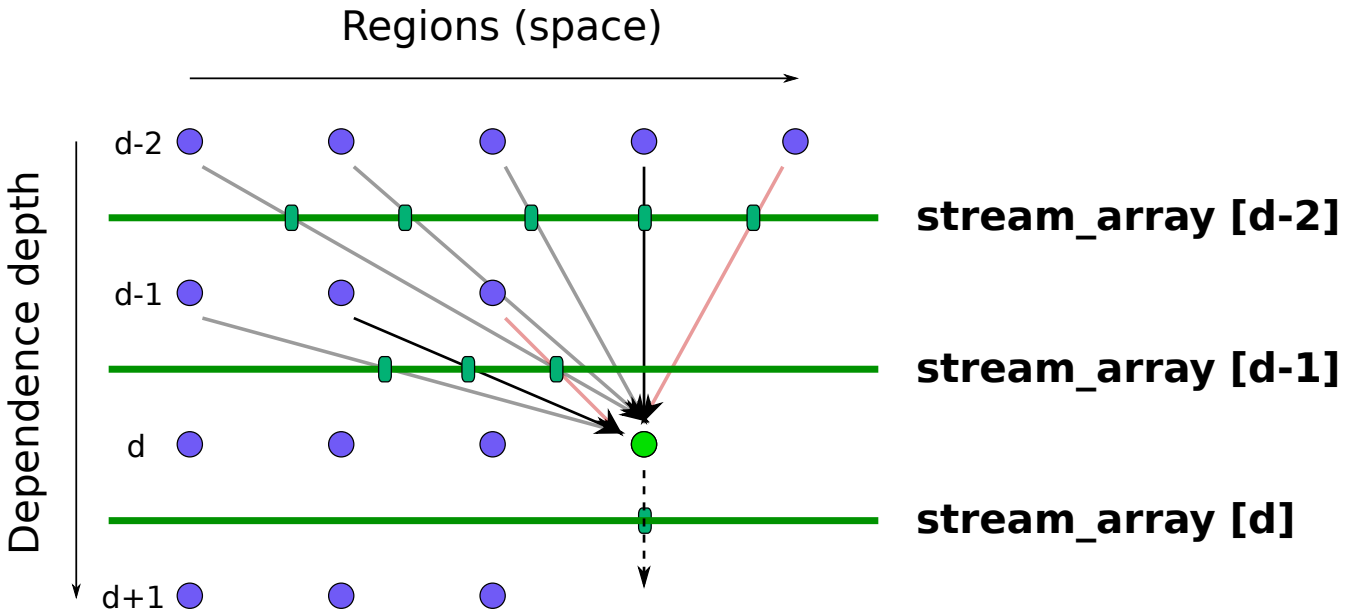
“Deep” stream synchronization

Stream dependences between each level of dependence depth

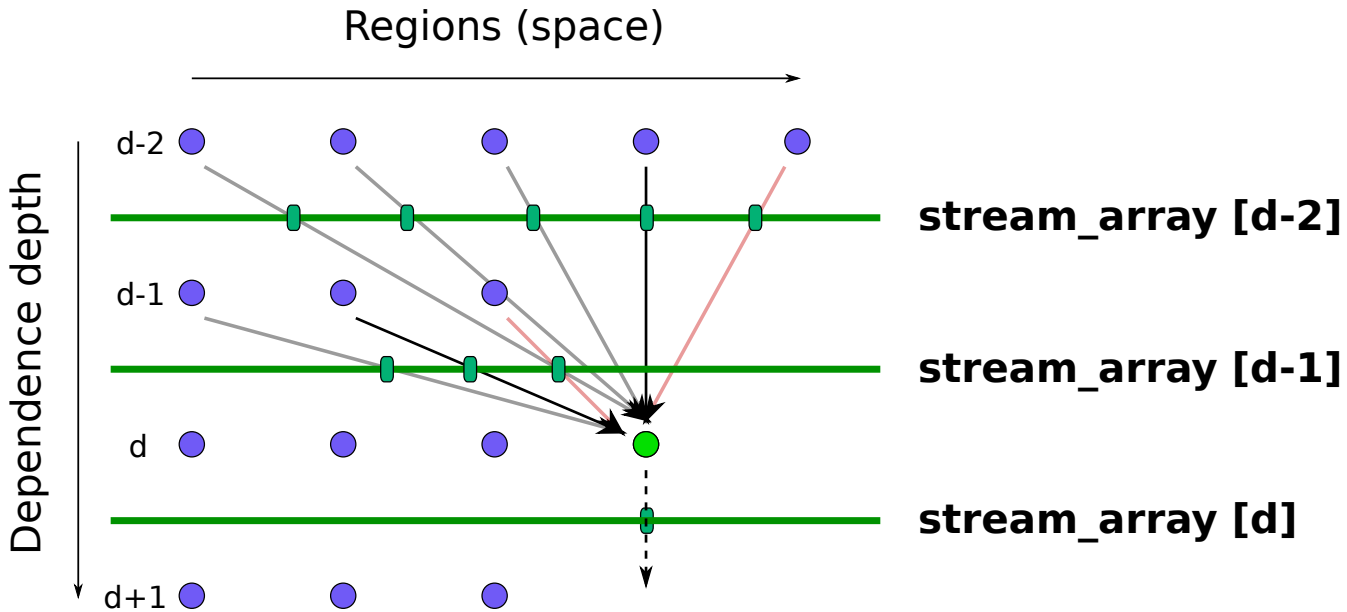




“Deep” stream synchronization



“Deep” stream synchronization



- Closed prefix stream synchronization (monotonicity)
  - over-synchronization
  - efficient synchronization algorithm
  - one single dependence enforced per depth level

Runtime support required for dependence depth resolution

Required information

- Set of unique dependence depths on which a task depends
- No need for precise information on dependence origin (producer task)
  - Easy bookkeeping in runtime
  - Update region depth counter when the version changes

Runtime support required for dependence depth resolution

Required information

- Set of unique dependence depths on which a task depends
- No need for precise information on dependence origin (producer task)
  - Easy bookkeeping in runtime
  - Update region depth counter when the version changes

Runtime API

```
void get_depth_vector (region_descriptors, int **depth_vector, int *size, int *max_depth);
```

Gauss-Seidel code generation objective

```
//#pragma  css task input(a{0}{1:L}, a{L+1}{1:L}, a{1:L}{0}, a{1:L}{L+1}) inout(a{1:L}{1:L})
void gauss_seidel (double a[N][N]) {
  for (int i=1; i<=L; i++)
    for (int j=1; j<=L; j++)
      a[i][j] = 0.2 * (a[i][j] + a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
}

bool stream_array[MAX_DEPTH];

for (int it=0; it < NITERS; it++)
  for (int i=0; i<N-2; i+=L)
    for (int j=0; j<N-2; j+=L)
    {
      int *depth_vector, size, max_depth;

      get_depth_vector (region_descriptors, &depth_vector, &size, &max_depth);

      {
        stream s[size];
        bool in_view[size][1], out_view;

        for (i = 0; i < size; ++i)
          s[i] = stream_array[depth_vector[i]];

#pragma omp task input (s >> in_view[size][0]) output (stream_array[max_depth+1] << out_view)
        gauss_seidel(&data[i][j]);

#pragma omp tick (stream_array[max_depth+1] >> 1)
      }
    }
```

Conclusion – Perspectives

- Unified compilation flow
- Compilation to T*

Conclusion – Perspectives

- Unified compilation flow
- Compilation to T*

- Current scheme
  - No spatial locality improvement (no data in streams)
  - Control over temporal locality (stream buffer size)

Conclusion – Perspectives

- Unified compilation flow
- Compilation to T*

- Current scheme
  - No spatial locality improvement (no data in streams)
  - Control over temporal locality (stream buffer size)
- Future directions
  - Build dynamic task graph by fast-forwarding the control program
    - construct a precise wavefront schedule
    - avoid over-synchronization

Conclusion – Perspectives

- Unified compilation flow
- Compilation to T*

- Current scheme
  - No spatial locality improvement (no data in streams)
  - Control over temporal locality (stream buffer size)
- Future directions
  - Build dynamic task graph by fast-forwarding the control program
    - construct a precise wavefront schedule
    - avoid over-synchronization
  - Generation of streaming code (privatizing semantics)
    - improve spatial locality