Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed-Memory

Roshan Dathathri Chandan Reddy
Thejas Ramashekar Uday Bondhugula

Department of Computer Science and Automation

Indian Institute of Science

{roshan,chandan.g,thejas,uday}@csa.iisc.ernet.in

September 11, 2013

Parallelizing code for distributed-memory architectures

OpenMP code for shared-memory systems:

```
for (i=1; i<=N; i++) {
#pragma omp parallel for
  for (j=1; j<=N; j++) {
    <computation>
  }
}
```

MPI code for distributed-memory systems:

```
for (i=1; i<=N; i++) {
   set_of_j_s = dist (1, N, processor_id);
   for each j in set_of_j_s {
     <computation>
   }
   <communication>
}
```

Explicit communication is required between:

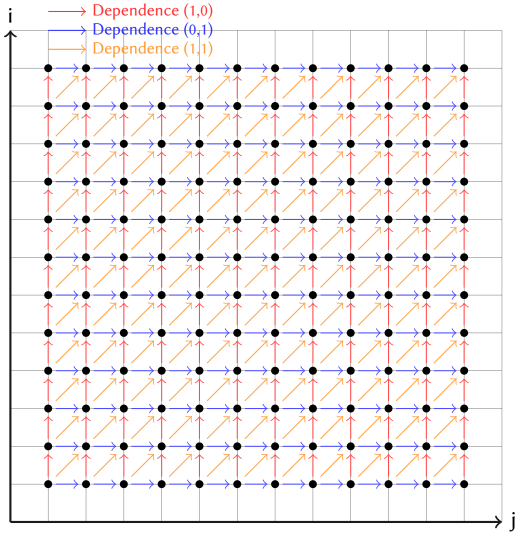
- devices in a heterogeneous system with CPUs and multiple GPUs.
- nodes in a distributed-memory cluster.

Hence, tedious to program.

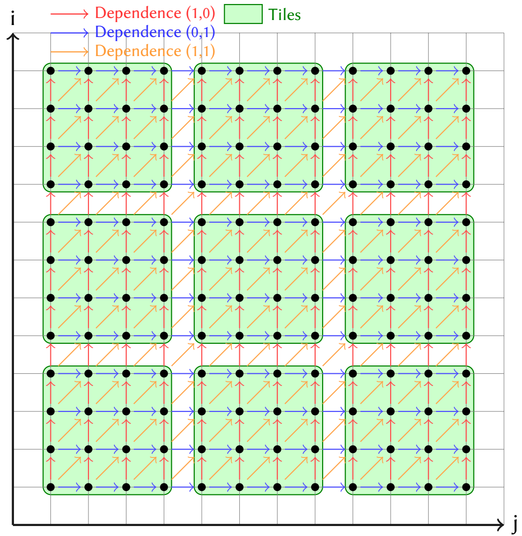
Affine loop nests

- Arbitrarily nested loops with affine bounds and affine accesses.
- Form the compute-intensive core of scientific computations like:
  - stencil style computations,
  - linear algebra kernels,
  - alternating direction implicit (ADI) integrations.
- Can be analyzed by the polyhedral model.

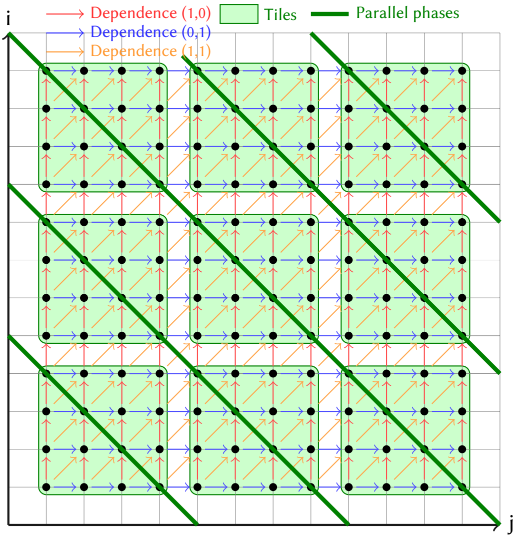
Example iteration space



Example iteration space



Example iteration space



Automatic Data Movement

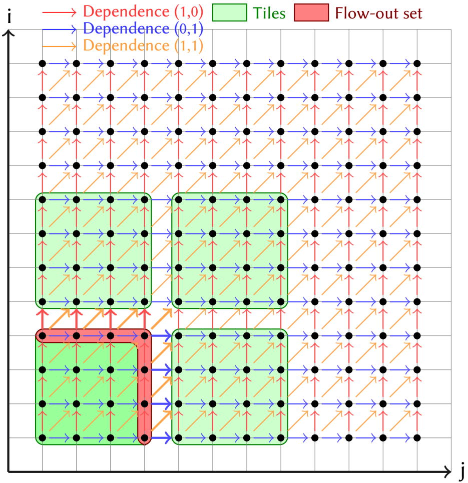
For affine loop nests:

- Statically determine data to be transferred between computes devices.
    - with a goal to move only those values that need to be moved to preserve program semantics.
- Generate data movement code that is:
    - parametric in problem size symbols and number of compute devices.
    - valid for any computation placement.

Communication is parameterized on a tile

- Tile represents an iteration of the innermost distributed loop.
- May or may not be the result of loop tiling.
- A tile is executed atomically by a compute device.

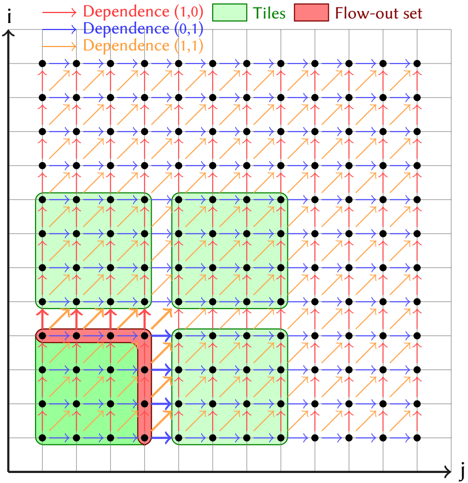
Existing flow-out (FO) scheme



Flow-out set:

- The values that need to be communicated to other tiles.
- Union of per-dependence flow-out sets of all RAW dependences.

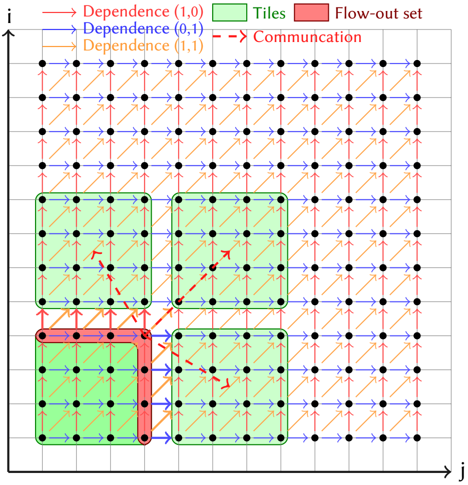
Existing flow-out (FO) scheme



Receiving tiles:

- The set of tiles that require the flow-out set.

Existing flow-out (FO) scheme



- All elements in the flow-out set might not be required by all its receiving tiles.
- Only ensures that the receiver requires at least one element in the communicated set.
- Could transfer unnecessary elements.

Our first scheme

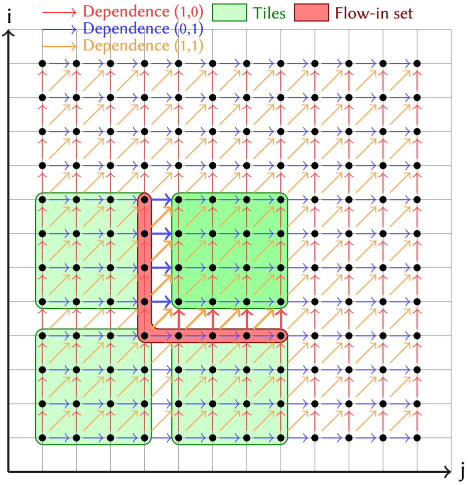
Motivation:

- All elements in the data communicated should be required by the receiver.

Key idea:

- Determine data that needs to be sent from one tile to another, parameterized on a sending tile and a receiving tile.

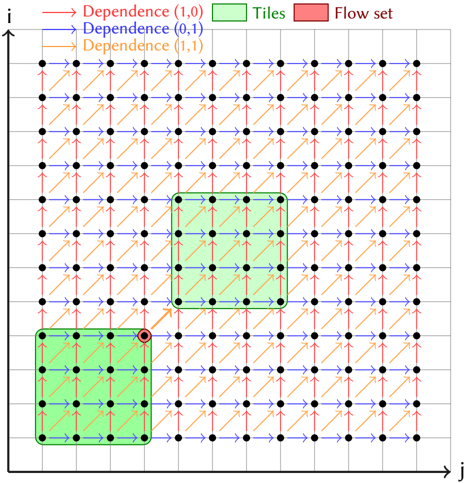
Flow-in (FI) set



Flow-in set:

- The values that need to be received from other tiles.
- Union of per-dependence flow-in sets of all RAW dependences.

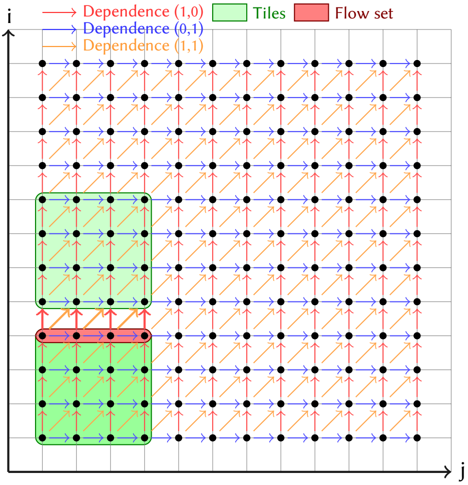
Flow-out intersection flow-in (FOIFI) scheme



Flow set:

- Parameterized on two tiles.
- The values that need to be communicated from a sending tile to a receiving tile.
- Intersection of the flow-out set of the sending tile and the flow-in set of the receiving tile.

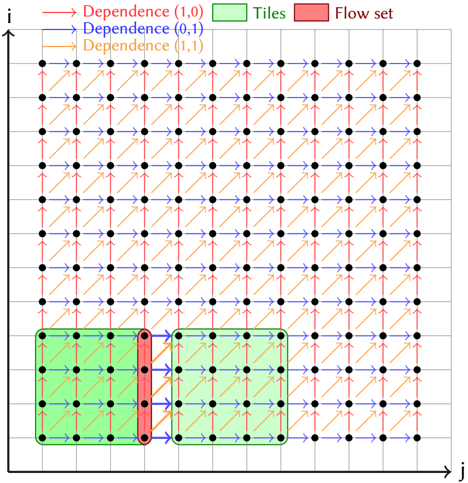
Flow-out intersection flow-in (FOIFI) scheme



Flow set:

- Parameterized on two tiles.
- The values that need to be communicated from a sending tile to a receiving tile.
- Intersection of the flow-out set of the sending tile and the flow-in set of the receiving tile.

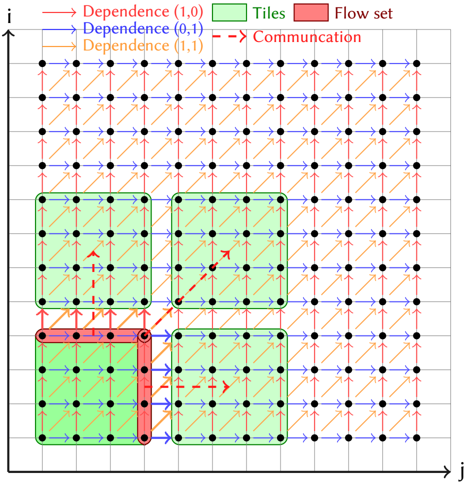
Flow-out intersection flow-in (FOIFI) scheme



Flow set:

- Parameterized on two tiles.
- The values that need to be communicated from a sending tile to a receiving tile.
- Intersection of the flow-out set of the sending tile and the flow-in set of the receiving tile.

Flow-out intersection flow-in (FOIFI) scheme



- Precise communication when each receiving tile is executed by a different compute device.
- Could lead to huge duplication when multiple receiving tiles are executed by the same compute device.

Comparison with virtual processor based schemes

- Some existing schemes use a virtual processor to physical mapping to handle symbolic problem sizes and number of compute devices.
- Tiles can be considered as virtual processors in FOIFI.
- Lesser redundant communication in FOIFI than prior works that use virtual processors since it:
  - uses exact-dataflow information.
  - combines data to be moved due to multiple dependences.

Our main scheme

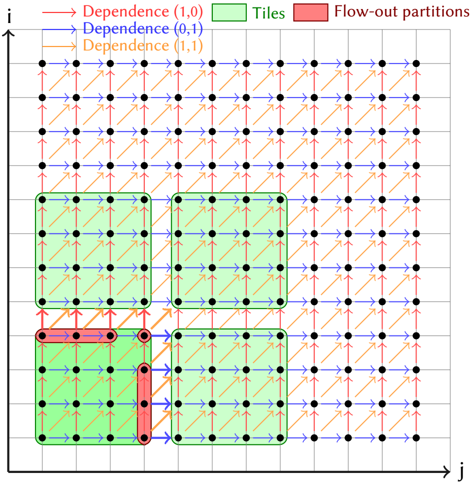
Motivation:

- Partitioning the communication set such that all elements within each partition is required by all receivers of that partition.

Key idea:

- Partition the dependences in a particular way, and determine communication sets and their receivers based on those partitions.

Flow-out partitioning (FOP) scheme

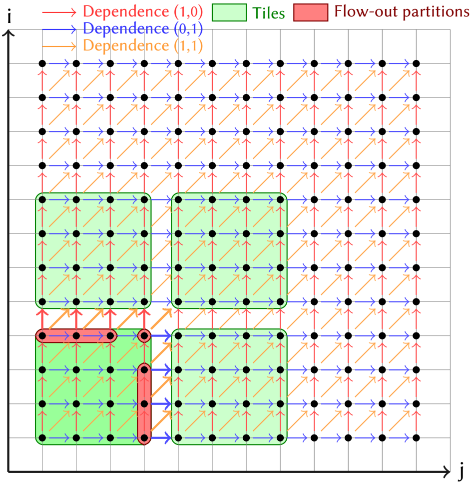


Source-distinct partitioning of dependences - partitions dependences such that:

- all dependences in a partition communicate the same set of values.
- any two dependences in different partitions communicate disjoint set of values.

Determine communication set and receiving tiles for each partition.

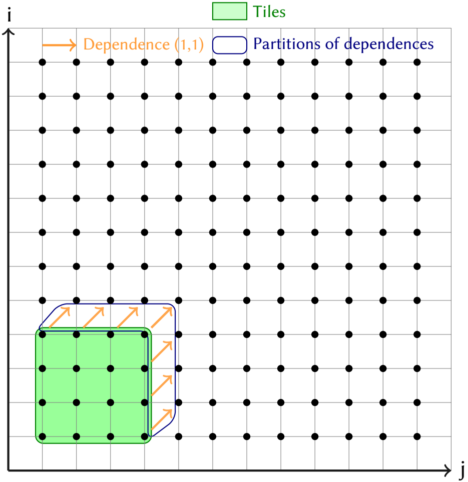
Flow-out partitioning (FOP) scheme



- Communication sets of different partitions are disjoint.
- Union of communication sets of all partitions yields the flow-out set.

Hence, the flow-out set of a tile is partitioned.

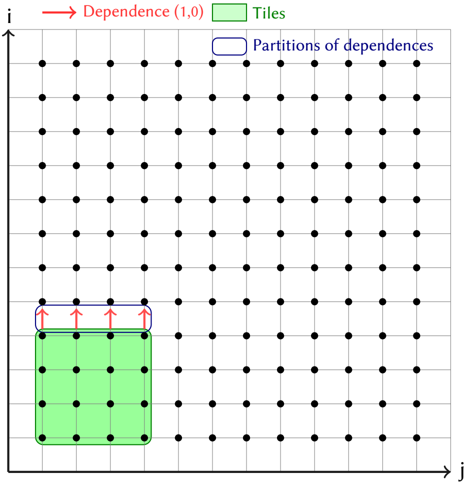
Source-distinct partitioning of dependences



Initially, each dependence is:

- restricted to those constraints which are inter-tile, and
- put in a separate partition.

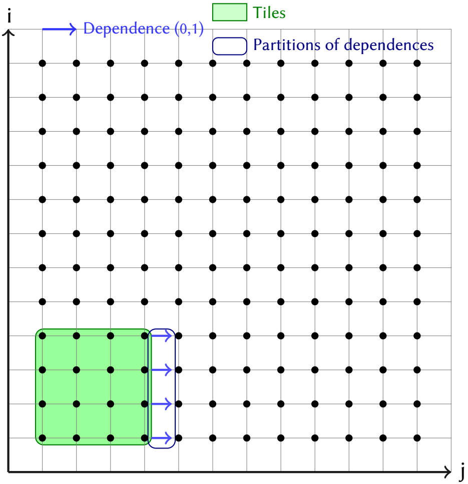
Source-distinct partitioning of dependences



Initially, each dependence is:

- restricted to those constraints which are inter-tile, and
- put in a separate partition.

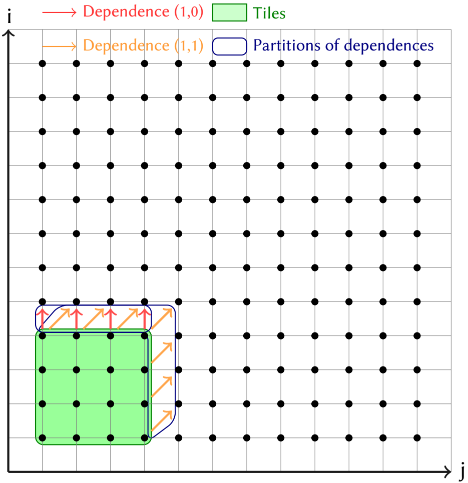
Source-distinct partitioning of dependences



Initially, each dependence is:

- restricted to those constraints which are inter-tile, and
- put in a separate partition.

Source-distinct partitioning of dependences

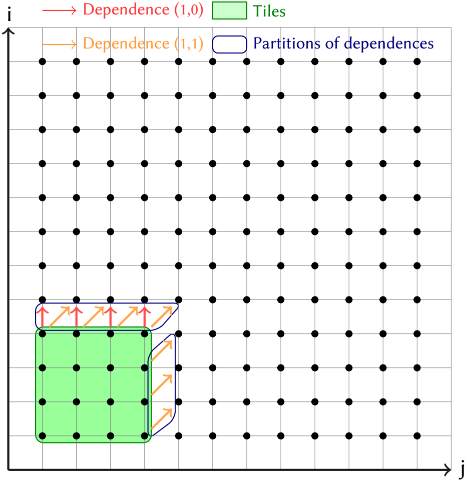


For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

The set of new dependences formed is a new partition.

Source-distinct partitioning of dependences

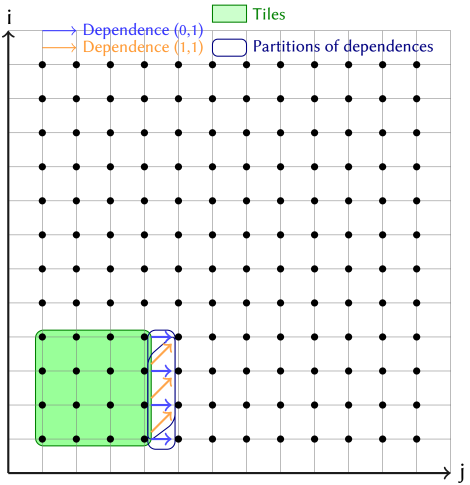


For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

The set of new dependences formed is a new partition.

Source-distinct partitioning of dependences

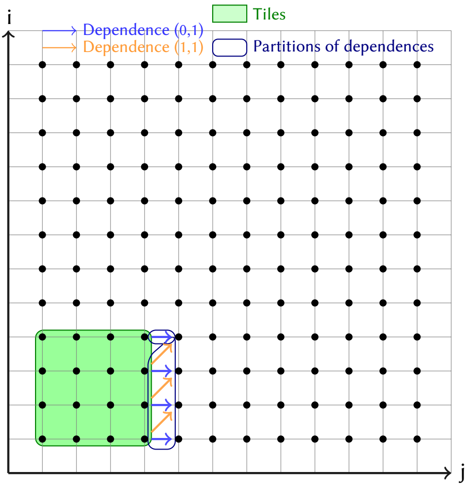


For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

The set of new dependences formed is a new partition.

Source-distinct partitioning of dependences

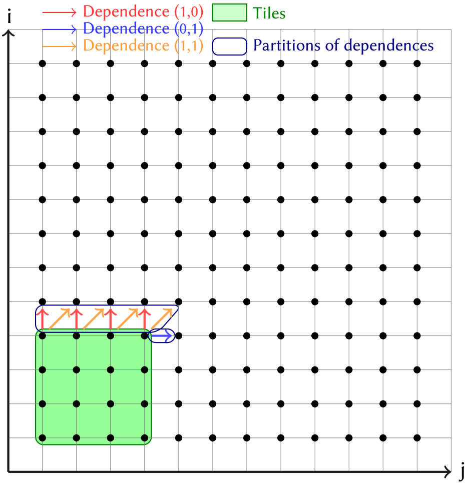


For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

The set of new dependences formed is a new partition.

Source-distinct partitioning of dependences

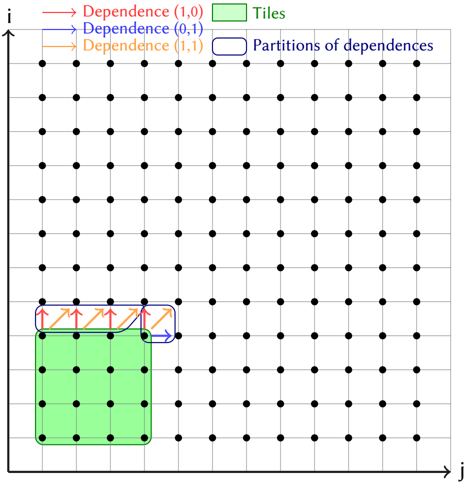


For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

The set of new dependences formed is a new partition.

Source-distinct partitioning of dependences

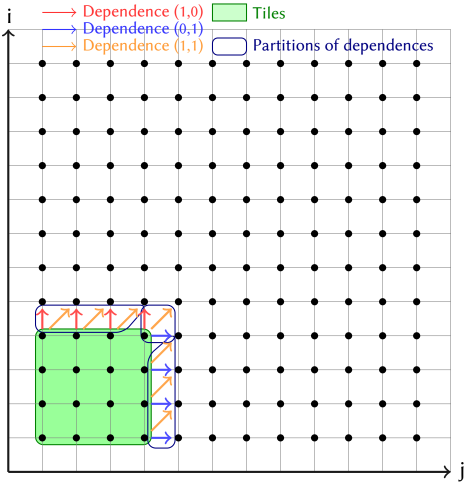


For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

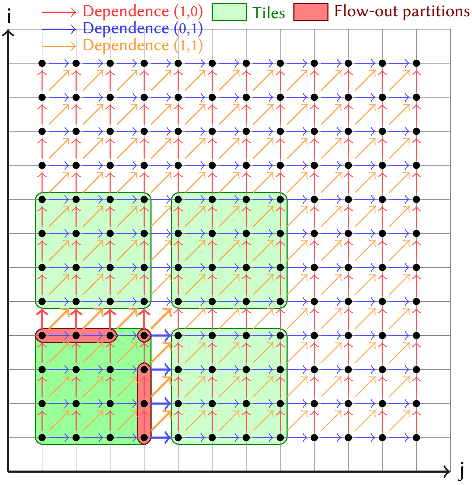
The set of new dependences formed is a new partition.

Source-distinct partitioning of dependences



Stop when no new partitions can be formed.

Flow-out partitioning (FOP) scheme: at runtime

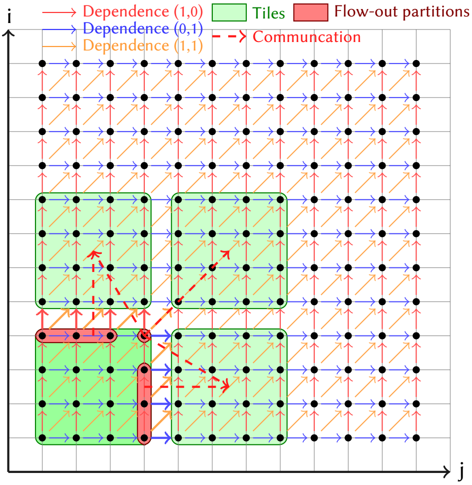


For each partition and tile executed, one of these is chosen:

- `multicast-pack`: the partioned communication set from this tile is copied to the buffer of its receivers.
- `unicast-pack`: the partioned communication set from this tile to a receiving tile is copied to the buffer of that receiver.

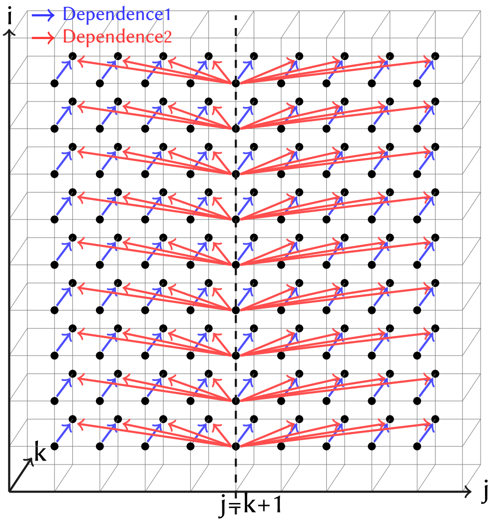
`unicast-pack` is chosen only if each receiving tile is executed by a different receiver.

Flow-out partitioning (FOP) scheme



- Reduces granularity at which receivers are determined.
- Reduces granularity at which the conditions to choose between `multicast-pack` and `unicast-pack` are applied.
- Minimizes communication of both duplicate and unnecessary elements.

Another example - dependences



Let:

(k, i, j) - source iteration

(k', i', j') - target iteration

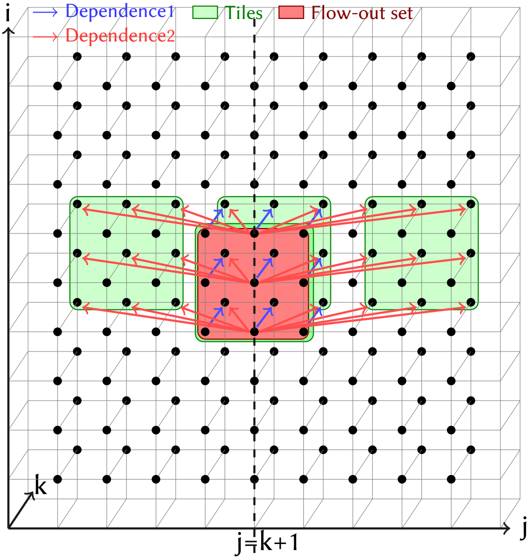
Dependence1:

$$k' = k + 1$$
$$i' = i$$
$$j' = j$$

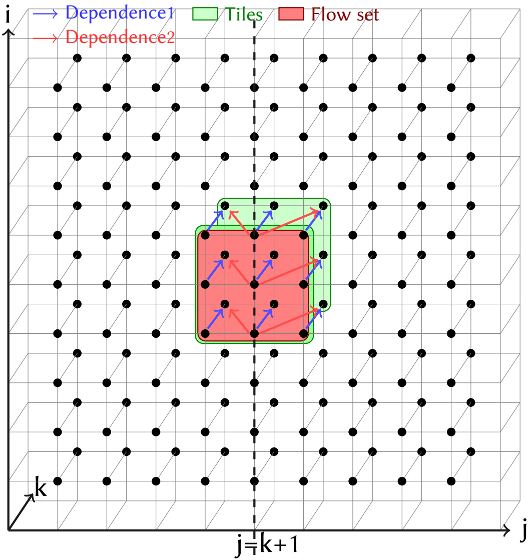
Dependence2:

$$k' = k + 1$$
$$i' = i$$
$$j = k + 1$$

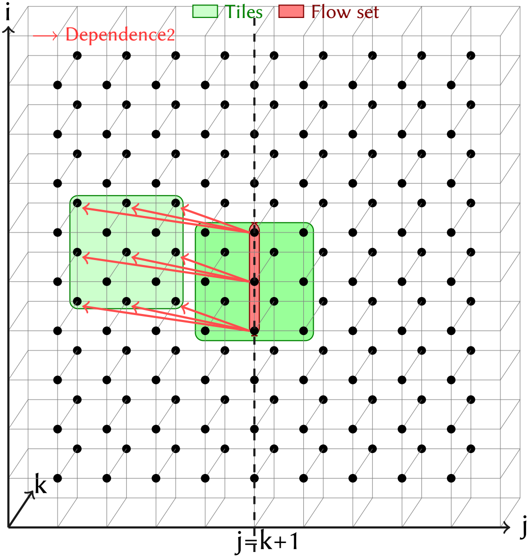
Another example - FO scheme



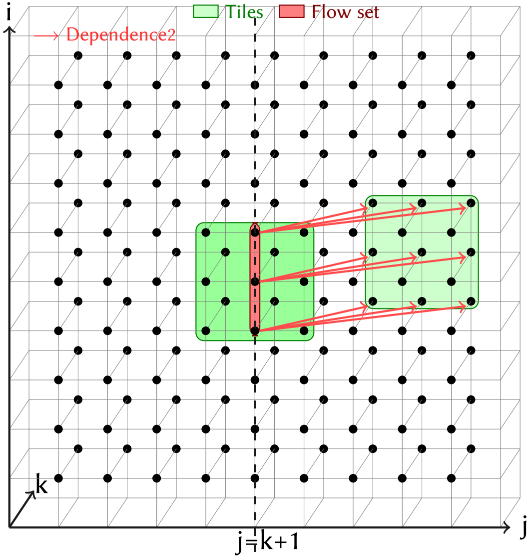
Another example - FOIFI scheme



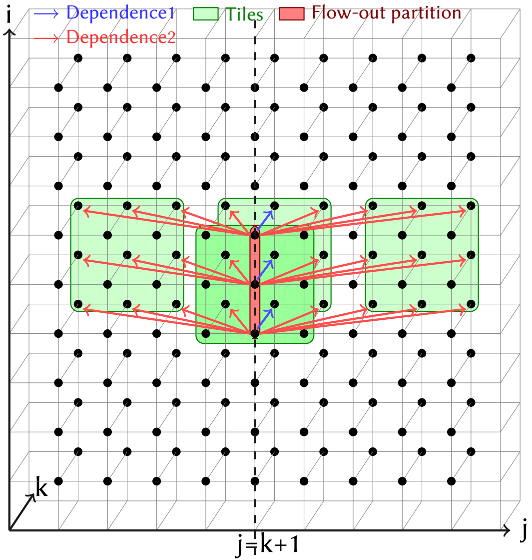
Another example - FOIFI scheme



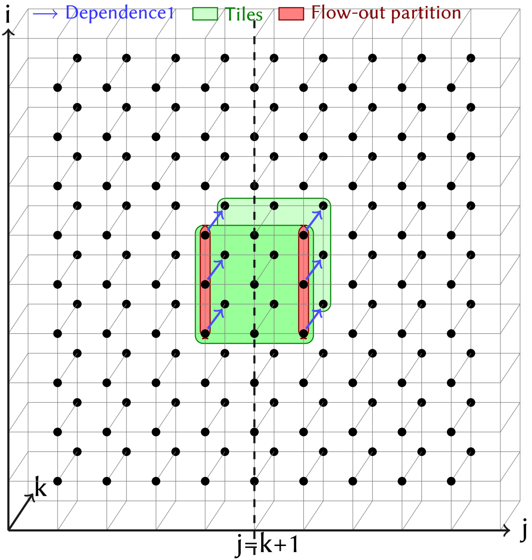
Another example - FOIFI scheme



Another example - FOP scheme



Another example - FOP scheme



Implementation

- As part of the PLUTO framework.
- Input is sequential C code which is tiled and parallelized using the PLUTO algorithm.
- Data movement code is automatically generated using our scheme.

Implementation - distributed-memory systems

- Code for distributed-memory systems using existing techniques is automatically generated.
- Asynchronous MPI primitives are used to communicate between nodes in a distributed-memory system.

Implementation - heterogeneous systems

- For heterogeneous systems, the host CPU acts both as a compute device and as the orchestrator of data movement between compute devices, while the GPU acts only as a compute device.
- OpenCL functions clEnqueueReadBufferRect() and clEnqueueWriteBufferRect() are used for data movement in heterogeneous systems.

Experimental evaluation: distributed-memory cluster

- 32-node InfiniBand cluster.
- Each node consists of two quad-core Intel Xeon E5430 2.66 GHz processors.
- The cluster uses MVAPICH2-1.8 as the MPI implementation.

Benchmarks

- Floyd Warshall (floyd).
- LU Decomposition (lu).
- Alternating Direction Implicit solver (adi).
- 2-D Finite Different Time Domain Kernel (fdtd-2d).
- Heat 2D equation (heat-2d).
- Heat 3D equation (heat-3d).

The first 4 are from Polybench/C 3.2 suite, while heat-2d and heat-3d are widely used stencil computations.

Comparison of FOP, FOIFI and FO

- Same parallelizing transformation -> same frequency of communication.
- Differ only in the communcation volume.
- Comparing execution times directly compares their efficiency.

Comparison of FOP with FO

- Communication volume reduced by a factor of $1.4\times$ to $63.5\times$.
- Communication volume reduction translates to significant speedup, except for heat-2d.
- Speedup of upto $15.9\times$.
- Mean speedup of $1.55\times$.

Comparison of FOP with FOIFI

- Similar behavior for stencil-style codes.
- For floyd and lu:
  - Communcation volume reduced by a factor of $1.5\times$ to $31.8\times$.
  - Speedup of upto $1.84\times$.
- Mean speedup of $1.11\times$.

OMPD - OpenMP to MPI

- Takes OpenMP code as input and generates MPI code.
- Primarily a runtime dataflow analysis technique.
- Handles only those affine loop nests which have a repetitive communication pattern.
    - Communication should not vary based on the outer sequential loop.
- Cannot handle floyd, lu and time-tiled (outer sequential dimension tiled) stencil style codes.

Comparison of FOP with OMPD

- For heat-2d and heat-3d, significant speedup over OMPD.
  - The computation time is much lesser.
  - Better load balance and locality due to advanced transformations.
  - OMPD cannot handle such transformed code.
- For adi: significant speedup over OMPD.
  - Same volume of communication.
  - Better performance due to loop tiling.
  - Lesser runtime overhead.
- Mean speedup of $3.06\times$.

Unified Parallel C (UPC)

- Unified programming model for both shared-memory and distributed-memory systems.
- All benchmarks were manually ported to UPC.
  - Sharing data only if it may be accessed remotely.
  - UPC-specific optimizations like localized array accesses, block copy, one-sided communication.

Comparison of FOP with UPC

- For lu, heat-2d and heat-3d, significant speedup over UPC.
  - Better load balance and locality due to advanced transformations.
  - Difficult to manually write such transformed code.
  - UPC model is not suitable when the same data element could be written by different nodes in different parallel phases.

Comparison of FOP with UPC

- For lu, heat-2d and heat-3d, significant speedup over UPC.
  - Better load balance and locality due to advanced transformations.
  - Difficult to manually write such transformed code.
  - UPC model is not suitable when the same data element could be written by different nodes in different parallel phases.
- For adi: significant speedup over UPC.
  - Same computation time and communication volume.
  - Data to be communicated is not contiguous in memory.
  - UPC incurs huge runtime overhead for such multiple shared memory requests to non-contiguous data.

Comparison of FOP with UPC

- For lu, heat-2d and heat-3d, significant speedup over UPC.
  - Better load balance and locality due to advanced transformations.
  - Difficult to manually write such transformed code.
  - UPC model is not suitable when the same data element could be written by different nodes in different parallel phases.
- For adi: significant speedup over UPC.
  - Same computation time and communication volume.
  - Data to be communicated is not contiguous in memory.
  - UPC incurs huge runtime overhead for such multiple shared memory requests to non-contiguous data.
- For fdtd-2d and floyd: UPC performs slightly better.
  - Same computation time and communication volume.
  - Data to be communicated is contiguous in memory.
  - UPC has no additional runtime overhead.

Comparison of FOP with UPC

- For lu, heat-2d and heat-3d, significant speedup over UPC.
  - Better load balance and locality due to advanced transformations.
  - Difficult to manually write such transformed code.
  - UPC model is not suitable when the same data element could be written by different nodes in different parallel phases.
- For adi: significant speedup over UPC.
  - Same computation time and communication volume.
  - Data to be communicated is not contiguous in memory.
  - UPC incurs huge runtime overhead for such multiple shared memory requests to non-contiguous data.
- For fdtd-2d and floyd: UPC performs slightly better.
  - Same computation time and communication volume.
  - Data to be communicated is contiguous in memory.
  - UPC has no additional runtime overhead.
- Mean speedup of $2.19\times$.

Results: distributed-memory cluster

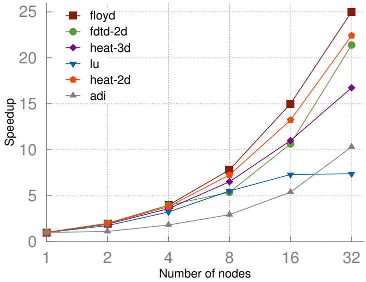


Figure: FOP – strong scaling on distributed-memory cluster

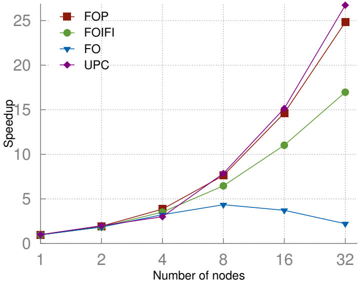


Figure: `floyd` – speedup of FOP, FOIFI, FO and hand-optimized UPC code over `seq` on distributed-memory cluster

For the transformations and computation placement chosen: FOP achieves the minimum communication volume.

Experimental evaluation: heterogeneous systems

Intel-NVIDIA system:

- Intel Xeon multicore server consisting of 12 Xeon E5645 cores.
- 4 NVIDIA Tesla C2050 graphics processors connected on the PCI express bus.
- NVIDIA driver version 304.64 supporting OpenCL 1.1.

Comparison of FOP with FO

- Communication volume reduced by a factor of $11\times$ to $83\times$.
- Communication volume reduction translates to significant speedup.
- Speedup of upto $3.47\times$.
- Mean speedup of $1.53\times$.

Results: heterogeneous systems

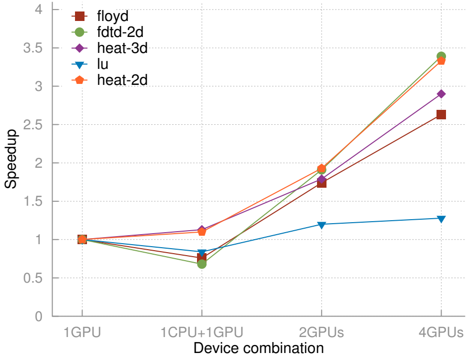


Figure: FOP – strong scaling on the Intel-NVIDIA system

For the transformations and computation placement chosen:
FOP achieves the minimum communication volume.

Acknowledgements

AMD

Department of Science & Technology
Ministry of Science & Technology

NVIDIA
CUDA RESEARCH CENTER

Conclusions

- The framework we propose frees programmers from the burden of moving data.
- Partitioning of dependences enables precise determination of data to be moved.
- Our tool is the first one to parallelize affine loop nests for a combination of CPUs and GPUs while providing precision of data movement at the granularity of array elements.
- Our techniques will be able to provide OpenMP-like programmer productivity for distributed-memory and heterogeneous architectures if implemented in compilers.

Publicly available: `http://pluto-compiler.sourceforge.net/`

Results: distributed-memory cluster

TABLE I: Total communication volume on distributed-memory cluster – FO and FOIFI normalized to FOP

Benchmark	Problem sizes	Tile sizes	4 nodes			8 nodes			16 nodes			32 nodes		
			FOP	FOIFI	FO	FOP	FOIFI	FO	FOP	FOIFI	FO	FOP	FOIFI	FO
floyd	8192^2	64^2	1.51GB	31.8×	63.5×	3.53GB	15.9×	63.5×	7.56GB	7.9×	63.5×	15.62GB	4.0×	63.5×
lu	4096^2	64^2	0.45GB	5.3×	1.4×	0.99GB	3.0×	1.4×	1.88GB	1.9×	1.4×	2.59GB	1.5×	1.5×
fdtd-2d	$1024 \text{x} 4096^2$	16^2	0.21GB	1.0×	14.3×	0.47GB	1.0×	15.1×	0.97GB	1.0×	15.5×	1.97GB	1.0×	15.7×
heat-2d	$1024 \text{x} 8192^2$	256^3	0.75GB	1.0×	2.0×	1.74GB	1.0×	2.0×	3.73GB	1.0×	2.0×	7.72GB	1.0×	2.0×
heat-3d	$256 \text{x} 512^3$	16^4	5.61GB	1.0×	2.0×	13.09GB	1.0×	2.0×	28.07GB	1.0×	2.0×	58.01GB	1.0×	2.0×
adi	$128 \text{x} 8192^2$	256^2	191.24GB	1.0×	4.0×	223.11GB	1.0×	8.0×	239.05GB	1.0×	16.0×	247.02GB	1.0×	32.0×

TABLE II: Total execution time on distributed-memory cluster – FOIFI, FO, OMPD and UPC normalized to FOP

(a) floyd – `seq` time is 2012s

Nodes	FOP	FOIFI	FO	UPC
1	2065.2s	1.01×	1.00×	0.97×
4	521.4s	1.10×	1.20×	0.97×
8	263.9s	1.18×	1.75×	0.97×
16	137.6s	1.33×	3.93×	0.97×
32	81.1s	1.46×	11.18×	0.93×

(b) lu – `seq` time is 82.9s

Nodes	FOP	FOIFI	FO	UPC
1	29.5s	1.00×	1.00×	2.86×
4	9.1s	1.42×	1.02×	2.42×
8	5.4s	1.70×	1.05×	2.30×
16	4.1s	1.84×	1.05×	1.50×
32	3.9s	1.58×	1.00×	1.25×

(c) fdtd-2d – `seq` time is 351.7s

Nodes	FOP	FOIFI	FO	UPC
1	359.5s	1.00×	1.00×	0.98×
4	90.8s	1.00×	1.03×	1.26×
8	66.9s	1.00×	1.04×	1.01×
16	33.8s	1.00×	1.09×	1.01×
32	16.8s	1.00×	1.24×	0.99×

(d) heat-2d – `seq` time is 796.4s

Nodes	FOP	FOIFI	FO	OMPD	UPC
1	228.3s	1.00×	1.00×	3.42×	5.33×
4	59.8s	1.00×	1.01×	3.29×	5.11×
8	31.4s	1.00×	1.02×	3.92×	5.47×
16	17.3s	1.00×	1.03×	3.58×	5.00×
32	10.2s	1.00×	1.04×	3.06×	4.25×

(e) heat-3d – `seq` time is 590.6s

Nodes	FOP	FOIFI	FO	OMPD	UPC
1	235.5s	1.00×	1.00×	2.51×	2.68×
4	65.4s	1.00×	1.05×	2.39×	2.46×
8	36.1s	1.00×	1.15×	2.82×	2.54×
16	21.4s	1.00×	1.23×	2.58×	2.21×
32	14.1s	1.00×	1.33×	2.29×	1.78×

(f) adi – `seq` time is 2717s

Nodes	FOP	FOIFI	FO	OMPD	UPC
1	422.7s	1.00×	0.95×	6.27×	7.90×
4	231.7s	1.00×	2.11×	3.55×	4.68×
8	143.6s	1.00×	4.00×	3.43×	4.29×
16	78.6s	1.00×	7.87×	2.88×	4.47×
32	41.0s	1.00×	15.9×	2.95×	5.22×

- Mean speedup of FOP over FO is 1.55x
- Mean speedup of FOP over OMPD is 3.06x
- Mean speedup of FOP over UPC is 2.19x

Results: heterogeneous systems

TABLE III: Results on the Intel-NVIDIA system

Benchmark	Problem sizes	Tile sizes	Device combination	Total execution time				Total communication volume		
				-	FOP	FO	Speedup	FOP	FO	Reduction
floyd	10240x10240	32x32	1 CPU (12 cores)	890s	–	–	–	–	–	–
			1 GPU	113s	–	–	–	–	–	–
			1 CPU + 1 GPU	–	148s	180s	1.22	0.8 GB	25.0 GB	32
			2 GPUs	–	65s	104s	1.60	1.6 GB	51.0 GB	32
			4 GPUs	–	43s	107s	2.49	3.1 GB	102.0 GB	32
lu	11264x11264	256x256	1 CPU (12 cores)	412s	–	–	–	–	–	–
			1 GPU	77s	–	–	–	–	–	–
			1 CPU + 1 GPU	–	92s	132s	1.43	0.9 GB	63 GB	70
			2 GPUs	–	64s	147s	2.30	0.7 GB	62.0 GB	83
			4 GPUs	–	60s	208s	3.47	1.2 GB	63.0 GB	51
fdtd-2d	4096x10240x10240	32x32	1 CPU (12 cores)	1915s	–	–	–	–	–	–
			1 GPU	397s	–	–	–	–	–	–
			1 CPU + 1 GPU	–	580s	603s	1.03	0.9 GB	11.0 GB	11
			2 GPUs	–	207s	236s	1.14	0.9 GB	22.0 GB	22
			4 GPUs	–	117s	164s	1.40	2.2 GB	62.0 GB	28
heat-2d	4096x10240x10240	32x32	1 CPU (12 cores)	1112s	–	–	–	–	–	–
			1 GPU	266s	–	–	–	–	–	–
			1 CPU + 1 GPU	–	242s	255s	1.05	0.6 GB	21.0 GB	32
			2 GPUs	–	138s	157s	1.14	0.6 GB	21.0 GB	32
			4 GPUs	–	80s	124s	1.55	1.9 GB	62.0 GB	32
heat-3d	4096x512x512x512	32x32x32	1 CPU (12 cores)	3080s	–	–	–	–	–	–
			1 GPU	1932s	–	–	–	–	–	–
			1 CPU + 1 GPU	–	1718s	2018s	1.17	16.0 GB	512.0 GB	32
			2 GPUs	–	1086s	1379s	1.26	16.0 GB	512.0 GB	32
			4 GPUs	–	670s	1658s	2.47	49.0 GB	1535.4 GB	32

Mean speedup of FOP over FO is 1.53x

Results: heterogeneous systems

TABLE IV: Results on the AMD system

Benchmark	Problem sizes	Tile sizes	Device combination	Total execution time				Total communication volume		
				-	FOP	FO	Speedup	FOP	FO	Reduction
floyd	10240x10240	32x32	1 CPU (4 cores)	1084s	–	–	–	–	–	–
			1 GPU	512s	–	–	–	–	–	–
			2 GPUs	–	286s	305s	1.07	0.8 GB	25.0 GB	32
fdtd-2d	4096x5120x5120	32x32	1 CPU (4 cores)	1529s	–	–	–	–	–	–
			1 GPU	241s	–	–	–	–	–	–
			2 GPUs	–	133s	242s	1.82	0.2 GB	2.15 GB	17
heat-2d	4096x8192x8192	32x32	1 CPU (4 cores)	3654s	–	–	–	–	–	–
			1 GPU	256s	–	–	–	–	–	–
			2 GPUs	–	142s	353s	2.49	0.25 GB	8.0 GB	32