On Improving the Execution of Distributed CnC Programs

Yuhan Peng[1], Martin Kong[1], Louis-Noel Pouchet[2], Vivek Sarkar[1]

[1]Department of Computer Science
Rice University

[2]Department of Computer Science
Colorado State University

CNC-2016 Workshop, September 2016

Concurrent Collections (CnC)

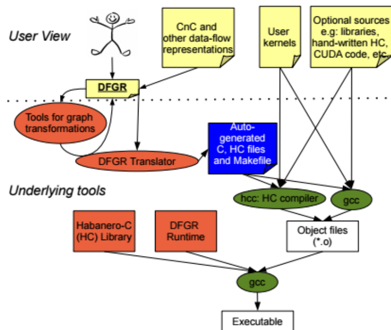
- Run-time and data-flow model for parallel programming.
- No direct specification of parallel operations.
  - The user specifies the semantics with data and control dependencies.
  - The runtime decides the schedule of parallel tasks.
- Applicable on both shared and distributed memory.
  - Can reach performance comparable to OpenMP/MPI applications.

Data-Flow Graph Language (DFGL)

- Intermediate graph representation for macro dataflow programs.
- Emphasizes the data dependencies between tasks.
- User-friendly, expressive language.
- DFGL provides great opportunities for performing high-level optimizations
  - Optimizations can be done through graph and loop transformations.
  - Especially good for polyhedral optimizations when program exhibits regularity.

Data-Flow Graph Language (DFGL)

- Automatic code generation tools can transform DFGL into CnC code for being executed.
- DFGL framework.[1]



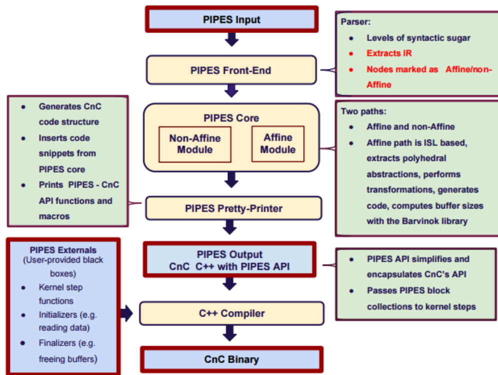
[1] Sbirlea, Alina, Louis-Noel Pouchet, and Vivek Sarkar. "Dfgr an intermediate graph representation for macro-dataflow programs." Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on. IEEE, 2014.

PIPES

- Programming language and compiler derived from DFGL.
  - Input: DFGL with producer and consumer relations, with other language abstractions.
  - Output: Intel CnC C++ compilable program.
- Concentrates on virtual topologies and task mappings.
- Automatically applying optimization transformations such as task coarsening and coalescing.
- Goal: better supporting task-based programming for shared and distributed memory.

PIPES

- Framework of PIPES.[2]
  - Great support for adding new optimization pass in PIPES core.



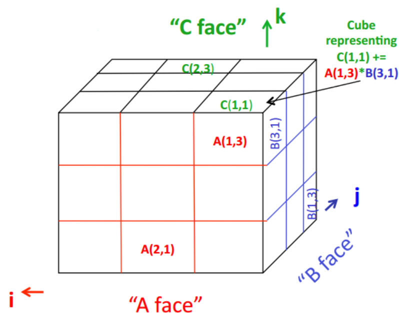
[2] M. Kong, L-N. Pouchet, P. Sadayappan, and V. Sarkar. "Pipes: A language and compiler for distributed memory task parallelism." SC 16. IEEE, 2016.

Motivation

- Managing and controlling the runtime overhead is crucial.
- In practice, such overhead depends on:
  - The total number of tasks created.
  - The number of tasks in flight at a give time point.
  - The total number of input dependencies.

Motivation

- Johnson 3D matrix multiply algorithm: our motivating example.
  - Introduced by Ramesh C. Agarwal et al. in 1995.
  - Parallelizable divide-and-conquer approach.
- To compute the product of A * B, Johnson 3D goes through two steps:
  - MMC: Divide A and B into small matrix pieces, and multiply the small matrix pieces in parallel.
  - MMR: Reduction to sum up the results of small matrix pieces.



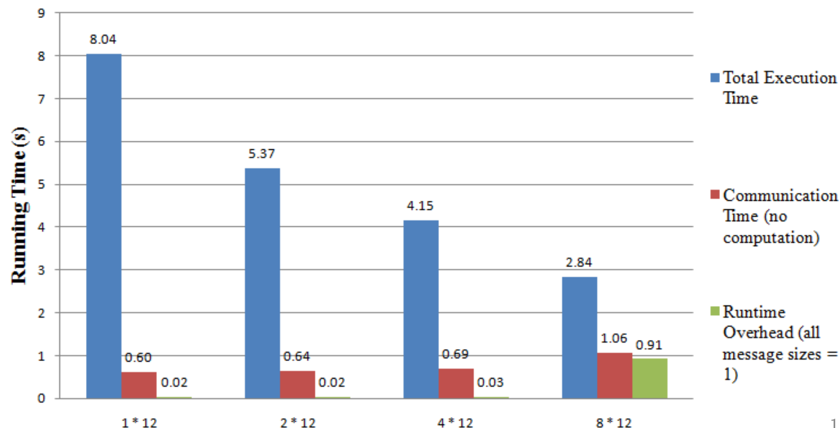
Source: Edgar Solomonik

Motivation

- We start by dissecting the program execution time of the Johnson 3D algorithm.
  - We tested the algorithm across different tile sizes.
  - We tested the algorithm across different number of nodes and task mappings.
- The program overhead is a non-negligible portion of the total time.
  - In distributed Johnson 3D algorithm, the overhead can take between 2% and 50% of the total execution time.

Motivation

- The overhead of the execution of Johnson 3D on different number of processors.
  - We use 1-8 nodes where each node has 12 processors.
  - More processors, larger overhead proportion.
  - The overhead grows superlinearly.



Our Approach

- Our goal: minimize the run-time overhead.
- We proposed two transformation techniques:
  - Dependency reduction.
  - Dynamic prescription.

Dependency Reduction

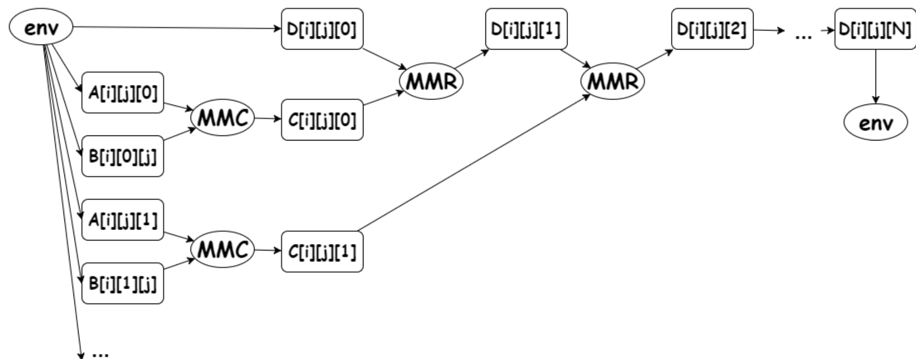
- Objective: minimizing the number of tasks and/or dependencies.
- Avoids needless polling of satisfied dependencies.
  - Depending on the runtime scheduler.
- Improves the program's progress.
  - Reduce the critical path length.
  - Minimize the number of task instances and block instances.
  - The processors will have fewer tasks to handle, and fewer dependencies to query.

Dependency Reduction

- The user may specify a reduction factor R.
- Then we transform the dataflow graph, so that
  - The semantic of the input DFGL does not change.
  - Minimize the total number of dependencies.
  - Essentially contracts one dimension by a factor of R.
    - If $R = 1$, no change.
    - If $R = 2$, every two instances are fused. i.e. N / 2 instances left.
    - If $R = N$, all instances are fused. i.e. dimension collapses.

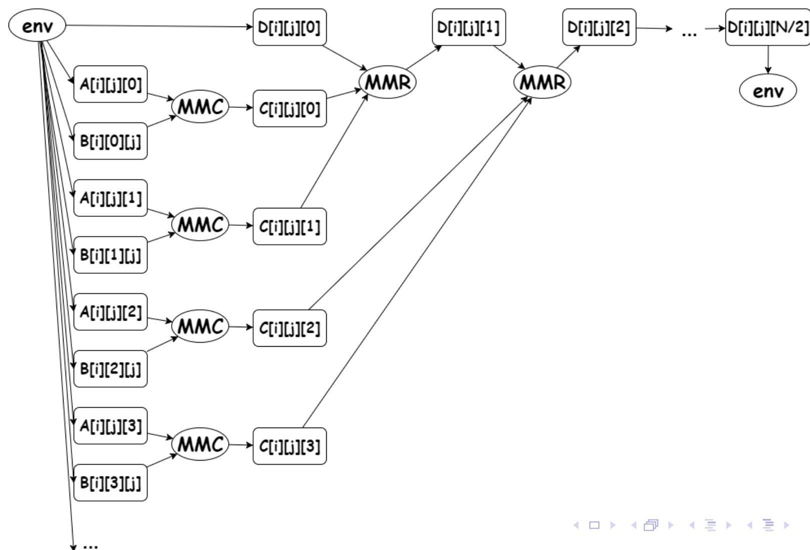
Dependency Reduction

- The dependency diagram of the original Johnson 3D algorithm.



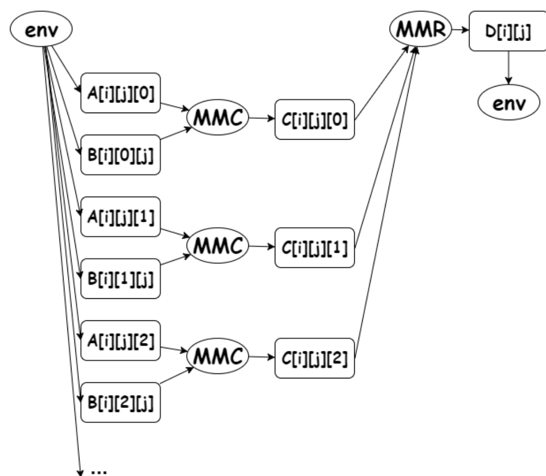
Dependency Reduction

- The dependency diagram of the Johnson 3D algorithm after dependency reduction with $R = 2$.



Dependency Reduction

- The dependency diagram of the Johnson 3D algorithm after dependency reduction with R = N.



Dependency Reduction

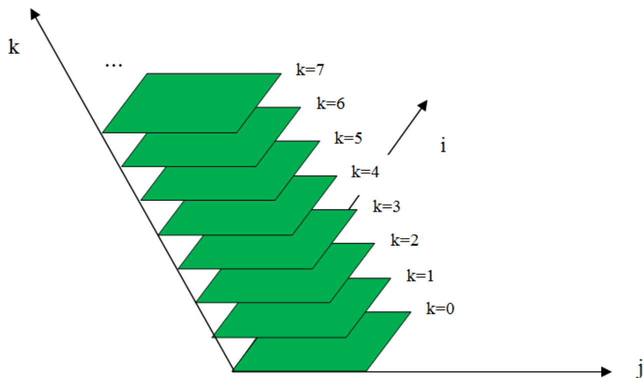
- Related work: OpenMP chunk_size.
  - Merge multiple loop bodies into one serial task, before being allocated to a thread.
  - Increase the work's granularity.
  - Improves program's scalability.
- The OpenMP chunk_size is similar to the reduction factor R.
- Difference between OpenMP chunk_size and PIPES dependency reduction.
  - OpenMP only performs data parallelism.
  - PIPES dependency reduction can support task parallelism.
  - PIPES dependency reduction can support the case when R = N, i.e. collapsing the entire dimension. Where OpenMP chunk_size must be a constant.
  - PIPES dependency reduction also removes intermediate results.

Dynamic Prescription

- Minimizes the number of tasks in flight by enforcing a dynamic prescription schedule, also known as creation and spawning schedule.
  - Determine when tasks are created and spawned.
  - Minimize the number of waiting tasks.
- Narrowing down the run-times scheduling options.
- Potentially improving the program's locality.
- Similarly, the user may specify a prescription factor K.
  - The size of the task set of each spawn.

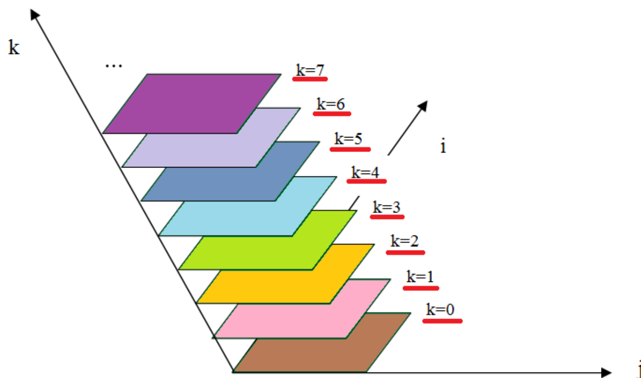
Dynamic Prescription

- Using the MMC in Johnson 3D as an example.
- Original version (K=N):
  - env::MMC(i,j,k) $0 \leq$ i, j, k $\leq$ n



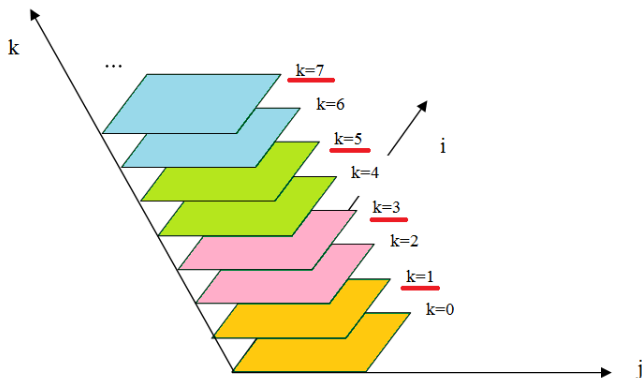
Dynamic Prescription

- Dynamic Prescription (K=1):
  - env::MMC(i,j,0) $0 \leq i, j \leq n$
  - MMC(i,j,k)::MMC(i,j,k+1) $0 \leq i, j \leq n$, $0 \leq k \leq n-1$



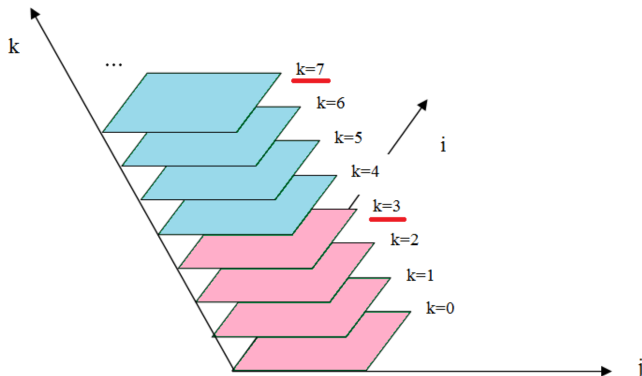
Dynamic Prescription

- Dynamic Prescription (K=2):
  - env::MMC(i,j,0), MMC(i,j,1) $0 \leq i, j \leq n$
  - MMC(i,j,k)::MMC(i,j,k+1), MMC(i,j,k+2) $0 \leq i, j \leq n$, $0 \leq k \leq n - 2$, $k \bmod 2 = 1$



Dynamic Prescription

- Dynamic Prescription (K=4):
  - env::MMC(i,j,0), MMC(i,j,1),MMC(i,j,2), MMC(i,j,3) $0 \leq i, j \leq n$
  - MMC(i,j,k)::MMC(i,j,k+1), MMC(i,j,k+2),MMC(i,j,k+3), MMC(i,j,k+4) $0 \leq i, j \leq n$, $0 \leq k \leq n - 4$, $k \bmod 4 = 3$



Dynamic Prescription

- Related work: cilk_for.
  - cilk_for can divide the loop into chunks.
  - Grain size: the maximum number of iterations in each chunk.
  - #pragma cilk grainsize = expression
- Grain size is similar to the prescription factor K.
- Difference between cilk_for and PIPES dynamic prescription.
  - cilk_for only performs data parallelism.
  - PIPES dynamic prescription can support task parallelism.
  - PIPES dynamic prescription can support prescription between different kernels.

Complexity Analysis

- No dependency reduction on MMR.
- Dynamic prescription on MMC. (K = 1, 2, 4)

	Original	K = 1	K = 2	K = 4
env::MMC	N^3	N^2	$2N^2$	$4N^2$
env::MMR	N^3	N^3	N^3	N^3
MMC::MMC	0	$N^3 - N^2$	$N^3 - 2N^2$	$N^3 - 4N^2$
MMC::MMR	0	0	0	0
MMR::MMR	0	0	0	0
Theoretical CPL	N + 1	N + 1	N + 1	N + 1

Complexity Analysis

- Dependency reduction on MMR. (R = N)
- Dynamic prescription on MMC. (K = 1, 2, 4)

	Original	K = 1	K = 2	K = 4
env::MMC env::MMR	N^3 N^2	N^2 N^2	$2N^2$ N^2	$4N^2$ N^2
MMC::MMC MMC::MMR	0 0	$N^3 - N^2$ 0	$N^3 - 2N^2$ 0	$N^3 - 4N^2$ 0
MMR::MMR	0	0	0	0
Theoretical CPL	2	2	2	2

Experimental Setup

- All experiments were performed on Davinci Cluster at Rice University.
  - The following table shows the detailed configuration.

Parameters	Value
Nodes	1-8
Processor	Intel Xeon X5660 @ 2.80 GHz
Sockets per node	2
Cores per socket	6
InfiniBand QDR bandwidth	40 GB/s
L1 Cache	32 KB per core
L2 Cache	256 KB per core
L3 Cache	12 MB per socket
CnC	1.01
MPI run-time	Intel MPI 5.0
Compiler	Intel ICPC 13
Slurm	2.6.5

Experimental Setup

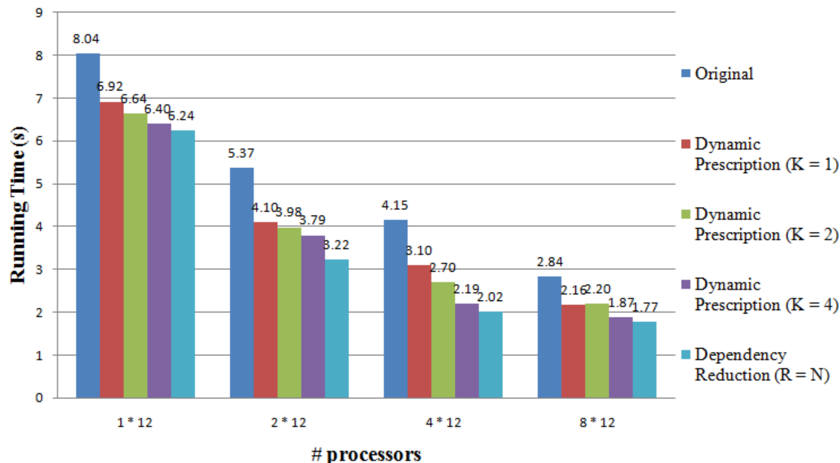
- Matrix Size: 8000 * 8000.
- 1, 2, 4, 8 nodes * 12 processors per node
- Tile Size: 400, 500, 800, 1000, 1600, 2000.
- All transformations were manually implemented.
- CnC tuners were used.
  - Dependency consumer.
  - computed_on
  - consumed_on

Performance Result

- We applied our proposed transformations on the Johnson 3D algorithm.
  - Dependency reduction on MMR ($R = N$)
  - Dynamic prescription on MMC ($K = 1, 2, 4$)
  - Dynamic prescription on both MMC and MMR ($K = 1, 2, 4$)
    - Adding MMC(i,j,k)::MMR(i,j,k) $0 \leq i, j, k \leq n$
  - Dynamic reduction ($K = 1, 2, 4$)
    - Dependency reduction on MMR ($R = N$), plus dynamic prescription on MMC ($K = 1, 2, 4$)
- We obtained 30% speedup when combining the proposed transformations comparing to the base version.

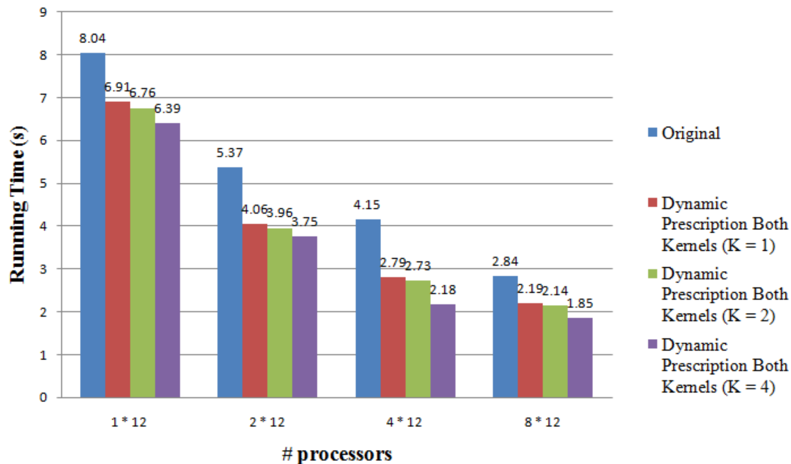
Performance Result

- Dependency reduction on MMR ($R = N$)
- Dynamic prescription on MMC ($K = 1, 2, 4$)



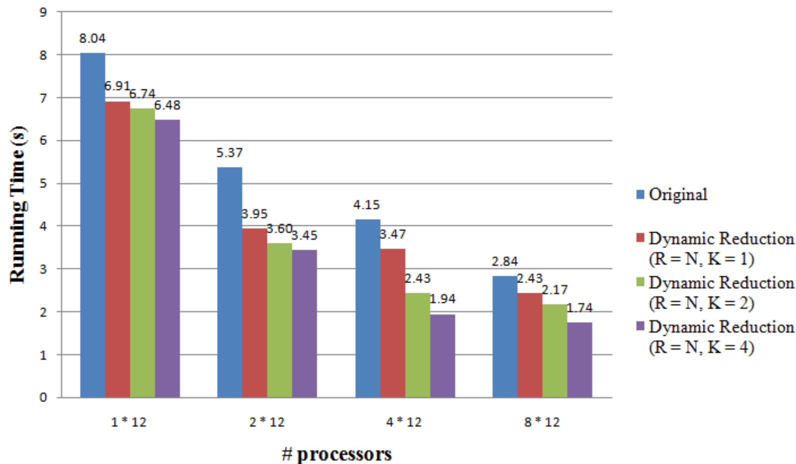
Performance Result

- Dynamic prescription on both MMC and MMR (K = 1, 2, 4)



Performance Result

- Dynamic reduction (R = N, K = 1, 2, 4)



Conclusion

- The overhead of task scheduling in distributed CnC programs is non-negligible.
- We proposed two transformations for overhead reduction:
  - Dependency reduction.
  - Dynamic prescription.
- Our preliminary results obtaining 30% speedup by applying our proposed transformations on Johnson distributed matrix-multiply algorithm.

Ongoing Work

- Currently we are focusing on dynamic prescription.
- Degree of freedom (dof): a property of task scheduling.
  - We have identified several dofs.
    - Manipulator: concentrate the prescription on as few tasks as possible.
    - Balanced: try to have more tasks being in charge of prescription operations.
    - Phased: all tasks of A should finish before any task of B starts.
    - Interleaved: some task of A should finish before starting some task of B.
  - More dofs to discover.
- Policies: combinations of dofs.
  - Policies determine runtime behavior.
  - Policies are applicable program-wide, or a subset of tasks.

References

- R. Agarwal et al., "A Three-dimensional Approach to Parallel Matrix Multiplication," IBM Journal of Research and Development, vol. 39, no. 5, pp. 575582, Sept 1995.
- Chandramowlishwaran, Knobe, and Vuduc. "Performance evaluation of concurrent collections on high-performance multicore computing systems." IPDPS, 2010.
- Sbirlea, Alina, Louis-Noel Pouchet, and Vivek Sarkar. "Dfgr an intermediate graph representation for macro-dataflow programs." Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on. IEEE, 2014.
- Sbrlea, Alina, et al. "Polyhedral Optimizations for a Data-Flow Graph Language." International Workshop on Languages and Compilers for Parallel Computing. Springer International Publishing, 2015.
- M. Kong, L-N. Pouchet, P. Sadayappan, and V. Sarkar. "Pipes: A language and compiler for distributed memory task parallelism." SC 16. IEEE, 2016.