Dependence Graphs and Compiler Optimizations

D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, M. Wolfe

The Authors

- David Kuck: currently an Intel Fellow. Sole software person on the Iliac IV project. Founded Kuck & Associates in 1979.
- Robert H. Kuhn: author of A Tutorial on Parallel Processing.
- David A. Padua: principal investigator for the Polaris Parallelizing compiler at UIUC.
- Bruce Leasure: current Race Walk Chair at US Track and Field.
- Michael Wolfe: PGI (The Portland Group) compiler engineer for NVIDIA.

Intel News Release

Intel To Acquire Kuck & Associates

Acquisition Expands Intel's Capabilities in Software Development Tools for Multiprocessor Computing

SANTA CLARA, Calif., April 6, 2000 - Intel Corporation today announced it has entered into a definitive agreement to acquire privately held Kuck & Associates, Inc. (KAI) for an undisclosed cash amount.

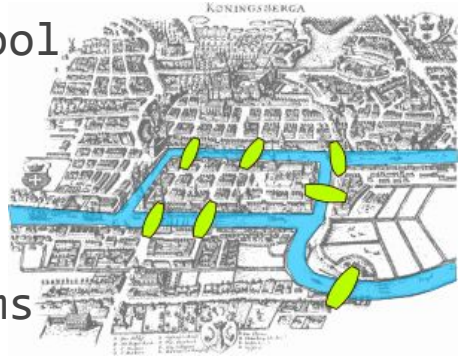
KAI is a leading provider of performance-oriented compilers and programming tools used in the development of multithreaded applications. The acquisition is designed to expand Intel's development tool offerings and accelerate the trend toward multiprocessor computing.

"Advances in Intel processor technology are broadening the opportunity for cost-effective multiprocessor computing," said Richard Wirt, Intel Fellow and general manager of Intel's Microcomputer Software Lab. "Increasingly, software developers are looking to harness multiprocessing as a way to improve application performance. This acquisition will provide developers with the tools they need to simplify the development of portable, multithreaded applications."

"The combination of Intel and Kuck will allow KAI's proven, cross-platform technology to expand in usefulness and reach more ISVs," said David Kuck, chairman and chief executive officer of KAI. "We are pleased to be joining Intel and taking a leading role in advancing parallelism and C++ tools for application developers."

Introduction

- We would like to be able to transform high-level language to promote parallelism at compile time.
- Graphs can provide both a visualization and a tool for optimization of language-independent algorithms.
- The graph serves as an abstraction of the algorithm in such a way that optimized algorithms become easily recognizable.

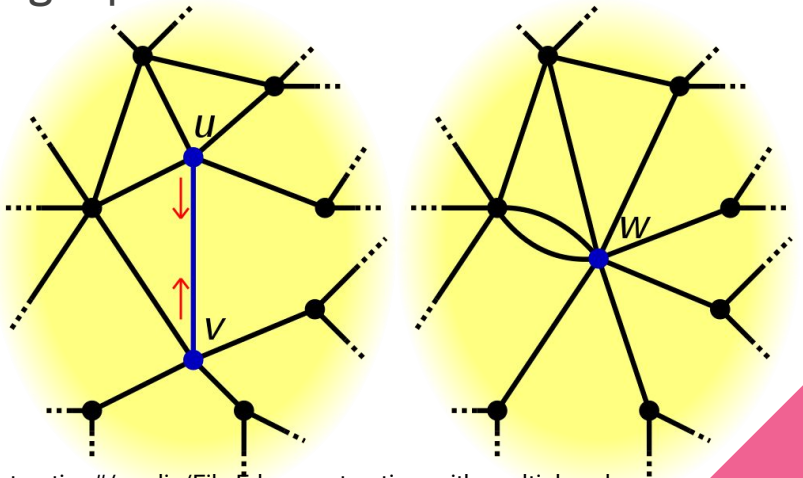


https://en.wikipedia.org/wiki/Graph_theory#/media/File:Konigsberg_bridges.png

A Bit About Graphs:

- Collection of edges and vertices.
- Directed graphs allow one-way connections.
- Functions acting on graphs alter either the set of edges or set of vertices.

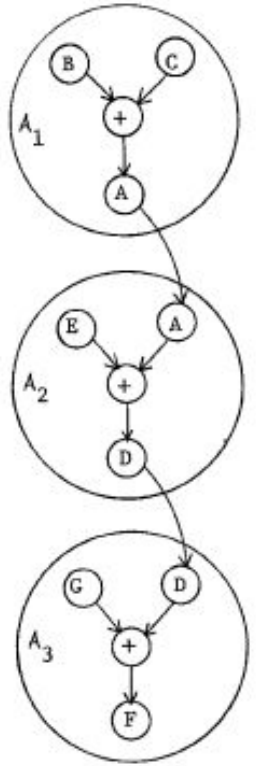
An example of edge contraction:



https://en.wikipedia.org/wiki/Edge_contraction#/media/File:Edge_contraction_with_multiple_edges.svg

Dependence Graphs:

- Represents elements (expressions or statements rather than functions) and their co-reliance on values within code.
- **Vertices (Elements):** each represents an operation composed of some atomic operands and operators.
- **Edges (Relations):** each represents a dependence between two vertices. Multiple edges are possible between any two vertices representing multiple kinds of dependency.
- Allowed variables are scalars and arrays.



Dependency Relationships:

01. **Loop Dependent:** component C is embedded in loop a loop with condition header L. **$L\ \delta^{L}\ C$**
02. **Output Dependent:** components A and B both have the same output variable. **$B\ \delta^{O}\ A$**
03. **Antidependent:** the output of A is an input of B. **$B\ \delta^{A}\ A$**
04. **Flow Dependent:** the value computed by B is used by A **$B\ \delta\ A$**
05. **Input Dependent:** the same variable name is an input to both A and B **$B\ \delta^{I}\ A$**

A and B can be the same node.

Depiction of Dependencies:

Fig. 1. Five types of dependence graph arcs

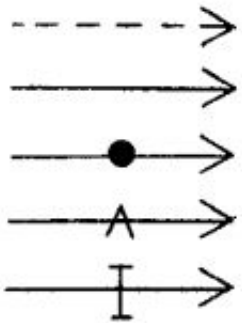
(a) loop dependence

(b) flow dependence

(c) output dependence

(d) antidependence

(e) input dependence



Arc Transformations:

- An “arc” is any relation represented in the dependence graph:

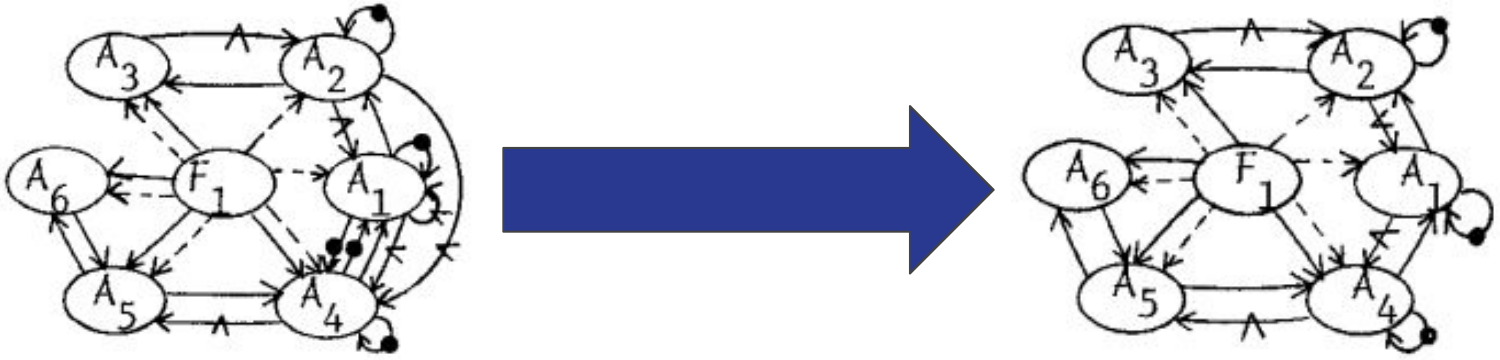
> A graph arc represents one of the five possible relations between the program components.

- This set of transformations either remove arcs or break cycles.
- Possible transformations are:
  - Renaming
  - Expansion
  - Node Splitting
  - Forward Substitution

Renaming:

- Often the same variable is used for several purposes throughout a program.
- “[T]he use of the same memory location for different purposes could impose unnecessary sequentiality constraints on parallel programs.”
- Assigning new names to different uses of the same variable removes some **output** and **antidependence** arcs.
- Introduced in 1973 by AV Aho, JE Hopcraft, and JD Ullman.

Renamed Graph:



(a) loop dependence

(b) flow dependence

(c) output dependence

(d) antidependence

(e) input dependence

Expansion:

- Takes a variable that is used inside of a loop and change it into a suitable data aggregate.
- Some definitions:
  - **Directly for loop dependent:** a component is in the top level of a loop. C depends on F and there's no F' such that C δ F' δ F.
  - **Chain:** nested loops. A series of loop header F_1 to F_m such that F_i always directly depends on F_{i+1}.
  - **Forwards to the next iteration:** the value given to a variable by a component in a loop is the same and still valid at the next iteration of the loop.
- Expand forwarded scalar values into arrays.

Expanded Graph:

(a) loop dependence

(b) flow dependence

(c) output dependence

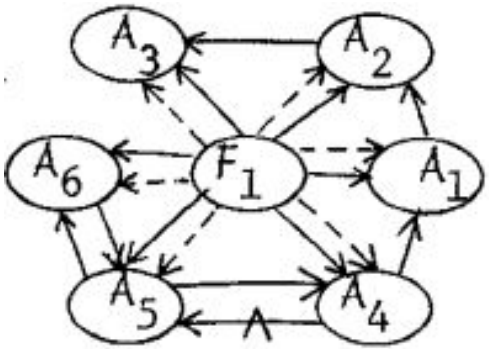
(d) antidependence

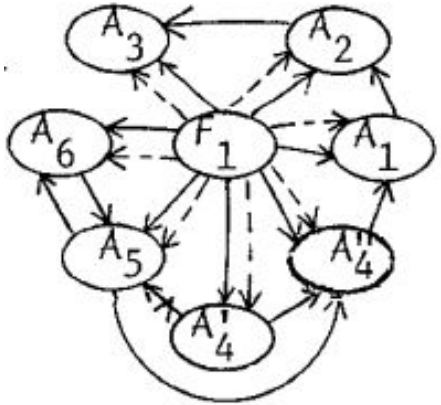
(e) input dependence

Node Splitting:

- Finds cycles in the graph that disappear at an atomic level, which must include an antidependence arc.
- Identifies the atomic element within the anti-dependence arc that the arc “emanates” from.
- Introduces a new variable and replaces all occurrences of the atomic element with the new variable.
- Applies the expansion transformation to the new variable.
- These goals can be achieved by architectural means on some platforms.

Split Graph:





(a) loop dependence

(b) flow dependence

(c) output dependence

(d) antidependence

(e) input dependence

Forward Substitution:

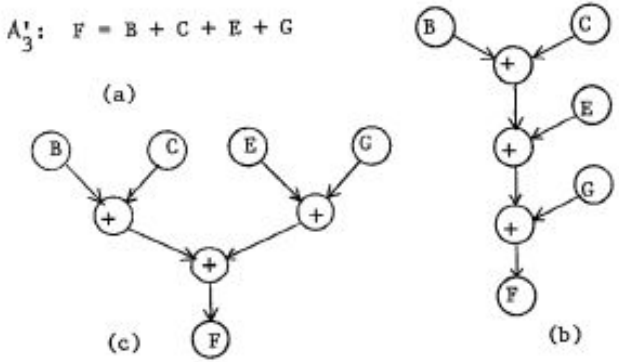
- Eliminates **flow** dependence arcs.
- Substitutes the right hand side of assignments into the right hand side of other assignment statements.
- Employs dead code elimination as described by D. Gries in 1971.
- Note: is there a statement missing between the commas in 3.4?

```
//So:
S = 1;
A[2] = S;

//Could be reduced to:
A[2] = 1;
```

Graph Abstraction:

- Merges nodes (vertices or elements) into compound nodes.
- Used here to, “isolate sets of statements that can be translated into high quality machine code only when taken as an ensemble.”
- Distinct from the transformations listed earlier.



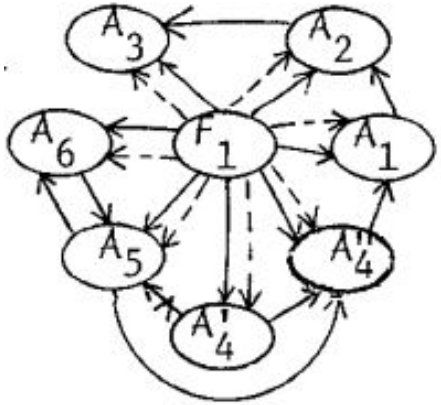
(I’m misusing this figure to show that the visualization of a node can change for a single programmatic statement.)

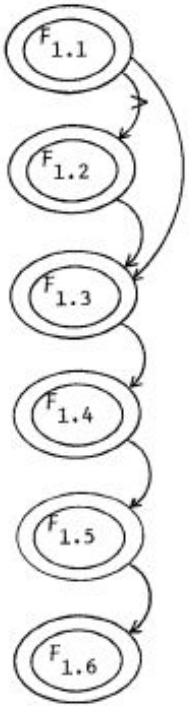
Loop Distribution:

- Applies Graph Abstraction to merge strongly connected components along with the loop header node.
- Independent nodes are merged with the loop header node in a separate compound node.
- Different applications of the Loop Distribution algorithm can optimize for vector processors or memory management.

```
//So:
for (i = 0; i<10; i++){
  S = 2*i;
  A[i] += S;
  F--;
}
//Could be distributed to:
for (i = 0; i<10; i++){
  S = 2*i;
  A[i] += S;
}
for (i = 0; i<10; i++)
  F--;
```

Distributed Graph:





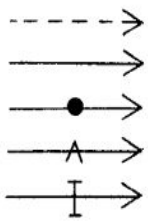
(a) loop dependence

(b) flow dependence

(c) output dependence

(d) antidependence

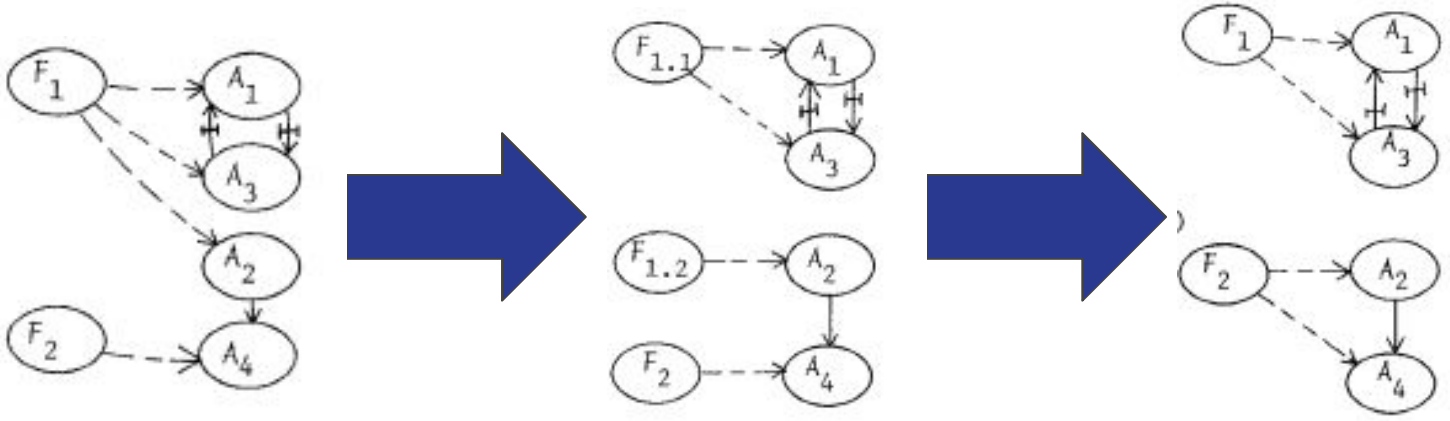
(e) input dependence



Loop Fusion:

- Selectively merges two compound loop nodes.
- The “opposite” of loop distribution.
- Creates a new loop that contains all of the statements from the two merged loops.
- Can optimize for memory management or vector register processors.

Distributed and Fused Graph:



(a) loop dependence

(b) flow dependence

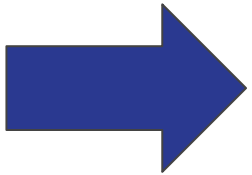
(c) output dependence

(d) antidependence

(e) input dependence

Transformed and Abstracted Code:

```
F1:  for I = 1 to N
A1:    A = A + 1
A2:    Y = A + 2
A3:    Z(I) = Y + V(I)
A4:    A = X(I+1) + X(I-1)
A5:    X(I) = W(I) + 1
A6:    W(I+1) = X(I) + 1
       end for
```



```
F1.1:  for I = 1 to N
A4' :    T(I-1) = X(I+1)
       end for
F1.2:  for I = 1 to N
A5  :    X(I) = W(I) + 1
A6  :    W(I+1) = X(I) + 1
       end for
F1.3:  for I = 1 to N
A4" :    A(2)(I) = T(I-1)+X(I-1)
       end for
F1.4:  for I = 1 to N
A1  :    A(1)(I-1) = A(2)(I-1)+1
       end for
F1.5:  for I = 1 to N
A2  :    Y(I-1) = A(1)(I-1)+2
       end for
F1.6:  for I = 1 to N
A3  :    Z(I) = Y(I-1) + V(I)
       end for
```

- Scalars expanded to arrays.
- Variable A renamed to A^1 and A^2
- For loop distributed.
- Assignment 4 was split into 2 operations, introducing T.

Conclusion:

- These techniques have been implemented for compilers across several architectures.
- Transformations increase the independence of nodes.
- Abstraction creates blocks which are more convenient for code generation.
- Specific applications for parallelization are discussed in “High-Speed Multiprocessors and Compilation Techniques.”, DA Padua, DJ Kuck, and DH Lawrie, IEEE Transactions on Computers, Sept. 1980

Final Reflections:

- Roses:
  - Strong mathematical foundation.
  - Thorough descriptions of algorithms.
  - Strongly portable.
- Thorns:
  - Results are declared rather than presented.
  - The set of transformed programs are not released so results are irreplicable.
  - Same for implementation.
  - Commercial ties of the authors are obfuscated.
  - It was weird to have to flip back to figures.

Questions?

- Section 1.2 identifies four occasions for dependence reduction: language selection, expression of the algorithm, compile time, and execution. This paper focused on compile time but do you think that these techniques are applicable at other times?
- The dependence graphs presented in this paper are all planar (i.e. no arcs cross). Is this necessarily implied by the condition of “sharpness”?