On Mapping Parallel Algorithms into Parallel Architectures

By Francine Berman and Lawrence Snyder

Presented by Chris Eldred
DOE Computational Science Graduate Fellow
Department of Atmospheric Science, University of Utah

Caveats

- I am not a computer science major- graph theory, formal language theory and algorithm analysis is way past what I know
- Feel free to interrupt me at any times to ask questions or tell me that what I am saying doesn't make sense/is just plain wrong
- My focus here is going to be on the big picture issues and the application of this technique- not the details

Outline

- Mapping Problem
  - Topological Variation
  - Cardinality Variation
- Solution
  - Contraction
  - Embedding
  - Multiplexing
- Conclusions

Mapping Problem

- The mapping problem is the issue that arises when the communication structure of a parallel algorithm is different from the interconnection structure of a parallel machine (topological variation) or the number of processes used by the algorithm is greater than the number of processing units present (cardinality variation)
- The authors present a solution for a class of common parallel interconnections structures that include: shuffle-exchange networks, hypercubes, square meshes, linear systolic arrays, cube-connected cycles, and complete binary trees

Topological Variation

- Topological variation is when the communications structure of an algorithm differs from the interconnection structure of a parallel machine
- The problem then is how to map the communication in the algorithm to the machine communication structure
  - Efficiency is the key metric here
  - We want to minimize the amount of additional communication overhead incurred by the mapping

Cardinality Variation

- Cardinality variation occurs when the number of processes in the parallel algorithm is greater than the number of processing units (however those are defined) in the parallel machine
- The problem then becomes how to map these processes to processing units
  - Again efficiency is the key metric
  - We want to minimize the multiplexing overhead

Solution

- Parallel algorithms are represented as communication graphs
  - It is assumed that all communication paths have the same length and bandwidth- BIG ASSUMPTION
- The authors outline a three-part solution to the mapping problem (solving both topological and cardinality variation)
  - Contraction
  - Embedding
  - Multiplexing

Contraction

- Gets rid of cardinality variation in the absence of topological variation
- Embeds Gn into Gk- where the graphs are from the same family and Gk is smaller than Gn
- Done using edge grammars and k-truncation
- Evaluated using weighting functions to measure the distribution of nodes and communication paths on the contracted graph
  - Example in the paper compares 2 different edges grammars that produce different contractions

Embedding

- Eliminates topological variation in the absence of cardinality variation
- Maps Gk to H, where H is the parallel interconnection architecture, with 1 process per processing unit
- Done using either known layout results or the approximation algorithm from the paper
- Evaluated using cost functions that measure how far a communication path must be stretched

Multiplexing

- Combines the previous two steps to resolve both cardinality and topological variation
- Implement Gn on image of Gk in H using multiplexing
- Not really discussed in the paper

Bottom-Line

- The process outlined by the authors provides a way to automate the mapping of parallel algorithms to a wide class of parallel interconnect architectures
  - There are assumptions (same length and bandwidth) about communication paths that do not hold in modern parallel architectures, especially in the new petascale and exascale machines

Conclusion

- Mappings add overhead at every step
- Algorithm designs should pay attention to architecture- this will be very important as machines get larger since the number of independent processing units that need to communicate is rapidly growing
- Good algorithms for one architecture are not necessarily good algorithms for another
  - Example: solving linear systems- good shared-memory single processor algorithms can make very poor distributed memory multiple processor algorithms