Compiling for Coming Generations of Computer Architectures

Challenge: to develop compilation and tool technology needed to construct efficient object programs on future computer architectures.

1. Future architectures
   - petaflops/petaops architectures
   - heterogeneous distributed computing

2. Future programming
   - new languages, application composition

3. Implications for compiler technology
   - management of parallelism and memory hierarchy
   - support for heterogeneous distributed computing
   - whole-program analysis and beyond
Future Architectures I

• Petaflops/Petaops Architectures
  - Custom high-performance processors
    superconductivity, multithreading
    10,000 at 100 gigaflops
  - Commercial off-the-shelf (COTS)
    100,000 at 10 gigaflops
  - Processor-in-memory
    processor and memory on one chip
    1,000,000 at 1 gigaflops

• Implications
  - Deep memory hierarchy
    up to 10 levels
  - High levels of parallelism
    at least 10 million way parallelism must be found in the application
    used to exploit parallel processors and hide memory latency
Distributed Heterogeneous Systems

- geographically distributed high-end systems
- networks of PCs
- message passing across nodes, shared memory within
- nodes and links have varying performance
  number and power of processors, network bandwidth
  availability and performance may differ over time

Implications

- Program decomposition
  load balancing
  matching function to architecture
  minimizing distant communication

- Interaction with system and network
  system allocation and scheduling
  fault tolerance and migration
  quality of service
Future Applications

• Application Complexity
  - Irregular, adaptive computation
  - Multidisciplinary simulation and design
  - Commercial applications
    Java, C++, mixed languages
  - Data intensive computation

• Application Composability
  - Applications will involve many programs
  - MADIC study
    10,000 applications
    untrusting developers
  - Language interoperability
  - Application development tools
Key Technologies

• **High-End Performance**
  - Program transformation and parallelization
  - Management of memory hierarchy
  - Language extensions
  - Tuned libraries

• **Portability Support**
  - Standards
    - MPI, HPF, OpenMP
  - Portability layers
    - Java Virtual Machine

• **Compilation Management**
  - Data-driven compilation
  - Whole-program compilation

• **Tools**
  - Program construction and composition
  - Debugging and performance tuning
Memory Hierarchy Management: Key Ideas

• Program Reorganization
  - register and cache blocking
  - loop splitting

• Software Prefetching
  - prefetch selection and placement

• Memory Reorganization
  - variable grouping on cache lines
  - array storage reorganization
  - dynamic reorganization schemes

• Inclusion of I/O in Memory Hierarchy
  - extension of cache techniques
    reorganization, prefetching
  - improvement factors in the hundreds
Distributed Heterogeneous Computing

• Program Decomposition
  - Distributed objects
  - Distributed data structures
  - Adaptive distribution of standard data structures

• Scheduling
  - Static and dynamic performance estimation
  - System performance parameterization
  - Adaptive load matching

• Latency Management
  - Interaction with Quality-of-Service facilities
  - Fast translation of data formats
Portability: The HPF Story

• Goals:
  - machine independent parallel programming support
  - high level of abstraction
    single thread, shared memory, implicit communication
  - interoperability

• Strategy
  - distribution directives for memory layout
  - compiler implemented for each machine
tailors code and communications to target architecture
  - EXTRINSIC interface

• Problems
  - compilers slow to mature
everly performance unsatisfactory
  - incomplete application coverage
  - need for library support
  - lack of tools
HPF Portability Strategy

- Fortran 90 Program
- Data Distribution Directives
- HPF Program
- Portable HPF Compiler
- Native HPF Compiler
- Fortran 77 Plus MPI
- Target Machine 1
- Target Machine 2
- Target Machine 3
REAL A(1023,1023), B(1023,1023), APRIME(511,511)

!HPF$ TEMPLATE T(1024,1024)
!HPF$ ALIGN A(I,J) WITH T(I,J)
!HPF$ ALIGN B(I,J) WITH T(I,J)
!HPF$ ALIGN APRIME(I,J) WITH T(2*I-1,2*J-1)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK)

!HPF$ INDEPENDENT, NEW(I)
  DO J = 2, 1022  ! Multigrid Smoothing Pass (red-black relaxation)
    T(I,J) = 0.25*(A(I+1,J) + A(I+1,J) + A(I,J-1) + A(I,J+1)) + B(I,J)
  END DO
END DO

!HPF$ INDEPENDENT, NEW(I)
  DO J = 2, 510   ! Multigrid Restriction
    APRIME(I,J) = 0.05*(A(2*I-2,2*J-2) + 4*A(2*I-2,2*J-1) +
                          4*A(2*I-1,2*J-2) + 4*A(2*I-1,2*J) +
  END DO
END DO

! Multigrid convergence test
ERR = MAXVAL( ABS(A(:,:)-B(:,:)) )
HPF: The Good News

• Compilers
  - Now available for every major platform
  - Language support more complete
  - Supported by all vendors

• Benchmarks and Applications
  - HPF within a factor of 2 of MPI on NAS benchmarks
  - Over 20 major applications in the CRPC database
    some over 100K lines
  - Standard migration path for CM-5 users
  - Performance record: 85 GF on 256 processor T3E

• Libraries and Tools
  - Portable interface to ScaLAPACK almost ready
  - Interfaces for other libraries underway
  - Pablo performance tool available, others under development

• Coverage
  - Much improved by extensions in HPF 2
Out-of-Core Support in HPF

• Support for Large-Memory Applications
  
  ![SIO$ IO-DISTRIBUTE A(BLOCK, *)](image)
  - I/O managed implicitly by compiler, run-time, and system

• Sources of Leverage
  - Program reorganization
  - Direct I/O
  - Prefetching

• Results
  - Up to factors of 100 or more improvement
  - Reorganization is the biggest win
The Role of Libraries

• Use of Libraries is Common
  - standard language functions
    HPF, Java
    collective communication
  - complex data structures
    e.g., adaptive arrays
  - standard scientific calculations
    linear algebra, FFT

• Portability
  - standard interface
  - callable from different language environments

• Performance
  - libraries tuned for each machine
  - compiler replacement of source by library calls
Programming in the Future

• Assertions
  - standard programming too hard
    large programs, legacy code
  - programs will be built from preexisting components
    different programming languages
    standard libraries
  - high performance will continue to be important

• Conclusions
  - future programming: two developer classes
    specialist and end user
  - future programming languages for end user:
    interfaces to component libraries
    problem-solving environments (PSEs)
    scripting languages
    languages may be graphical
The Challenge: Performance

• Performance Lost at Interfaces
  - languages increasingly emphasize modular organization
    object-oriented style, e.g., Java
  - implication: whole-program compilation is needed
    black-box approach to program subcomponents is too inefficient
    performance loss: factor of two or more

• Secondary Problems
  - components may be in different languages
    different run-time environment, data representation, ...
  - components may be from different companies
    who do not trust one another
Library-Based PSE

• Example: Matlab
  - interpreted control language
  - globally shared data structures
  - library calls (e.g. to LAPACK) for component operations

• Naive Implementation
  - straight interpretation

• Sophisticated implementation
  - compile control language to standard intermediate language (e.g., Fortran 90)
  - optimize control program and subroutines as a whole
  - prototype implementation by Padua et. al.
Toolkit for Library-Based PSEs

• Front End
  - GUI for interactively building language programs
  - parser and translator to standard language

• Interprocedural Optimization Framework
  - general and language independent
  - optimizations can be tailored

• Code Generation
  - standard highly-optimizing compiler

• Program Management System
  - manage library and control program
  - mechanism for feedback from actual run to optimizer and program constructor
Recompilation Analysis

- Source Importer
- Program Specifier
- Optimizing Compiler

- Source Changes
- Inlining Info
- Interprocedural Info Relied Upon
- File Changes
- Determine Files to Recompile

Rec ompilation Analysis
Compilation with Data

Program

Rarely Changing Data

Frequently Changing Data

Extended Optimizing Compiler

Object Program

Answers
New Technology Program Environments

- Program Definition
- Input Data Definition
- Run History
- Source Files
- Trace Files
- Whole-Program Compiler
- Machine Code
Source from Company A

Source from Company B

Library Source from Vendor C

Source encrypted using public key for <compiler,machine>

Validated to compile on machine x

Trusted Compiler

Trusted Loader

Target Machine
The Role of Tools

- **Source**
  - User
  - Compiler
  - Target Machine
  - Compiled Code

- **Execution Record**
  - Keyed to Source
  - Debugging and Analysis Tool

- **Info about Transformations**
  - From Compiler to Debugging and Analysis Tool
Composition of Tools

- **User**
  - HL Compiler
  - LL Compiler
  - Target Machine

- **Virtual Machine**

- **HL Debugger**
  - Execution Record

- **LL Debugger**

Flow of information:
- Source -> HL Compiler -> HL Debugger
- Source -> LL Compiler -> LL Debugger
- Compile Info
- Compile Info
Challenges for Compilation Research

• Complex Architectures
  - more parallelism
  - deeper memory hierarchies
  - heterogeneous distributed systems

• Compiler Technology
  - advanced analysis and transformations
  - whole-system compilation
  - tools developed with compiler

• New Challenges
  - compilation for distributed execution
    load matching, management of variable latencies
  - language interoperability with high performance
  - compiler security
  - compiler reliability