Performance Programming for Scientific Computation

SIAM Short Course

V

Portable High Performance

Bowen Alpern and Larry Carter

13 March 1997

Expedient Portability

Goal

One (easy-to-write) program

Runs correctly (with ok performance)

On all **sequential** computers

Approach

High-level languages

Machine-specific compilers

Necessary social investment

To implement N applications on M machines

Costs

- $O(1)$, language design & compiler technology (enormous)
- $O(N)$, application development
- $O(M)$, compiler development
- $O(NM)$, makefile tweaking (tiny)

Performance Portability

Goal

One (easy-to-write) program

Runs correctly with **highest possible** performance

On all possible computers

Expeditious solution (first fallback)

One (easy-to-write) program

Runs correctly with reasonably good performance

On almost all computers

Comprehensive solution (second fallback)

One program

Runs correctly with **highest possible** performance

On a collection of computers

- First computer — no harder than hand tuning
- Additional computers — easier

Principle of Portable Performance

For near-peak performance, different computers will run different sequences of source-language statements.

Example: `DGEMV` (matrix-vector product)

- Scalar processors: `DDOT` based
  - Fewer `stores`
- Vector processors: `DAXPY` based
  - Independent `fmas`
- Superscalar processors: hybrid based
  - Some of both

How this is accomplished?

- Tuned libraries (LAPACK, ScaLAPACK, etc.)
- Optimizing compiler (FORTRAN90, HPF, etc.)
- Ad-hoc compiler directives and options
- Explicit program variants

Possible Approaches

Improve compiler technology

- Extends expedient portability
- Languages for parallelism (F90, HPF, ZPL, Java?)
- JIT and dynamic compilation

Kernel-based libraries (LAPACK/ScaLAPACK)

- Identify computationally intensive kernels
- Implement highly tuned kernels on every computer
  - Who implements the kernels? How??

Domain-specific libraries

- KeLP (structured, bulk-synchronous)
- Multipol (fine-grained, asynchronous)

Generic program

- Polyalgorithm (explicit program variants)
- Specialize for model of the target machine
- Machine-specific compilers

An On-going Debate

From Sabot's *High Performance Computing*

- "Don't stripmine or unroll loops."
- Hand optimizations inhibit portablility
  - Compilers do better on simple, clear code

Our viewpoint:

- Yes, old CRAY vector codes have "pessimizations"
- Yes, a few compilers do well on dense linear algebra
- Maybe by <this year>+3, compilers will be great
  - (for the machine you replaced two years ago)
- Stripmining and unrolling are sometimes needed.
  - When possible, write parameterized optimizations

More research needed

The Generic Program Approach

Generic program

- A family of *program variants*
  - **Pragmatically** equivalent semantics
  - Different performance characteristics
- Variation mechanisms
  - Overloading (alternative implementations)
  - Tuning parameters
  - Program transformations (semantics preserving)

Specialization

- Select the variant with best performance
  - On an **idealized model** of the target
- Discrete choices

Translation

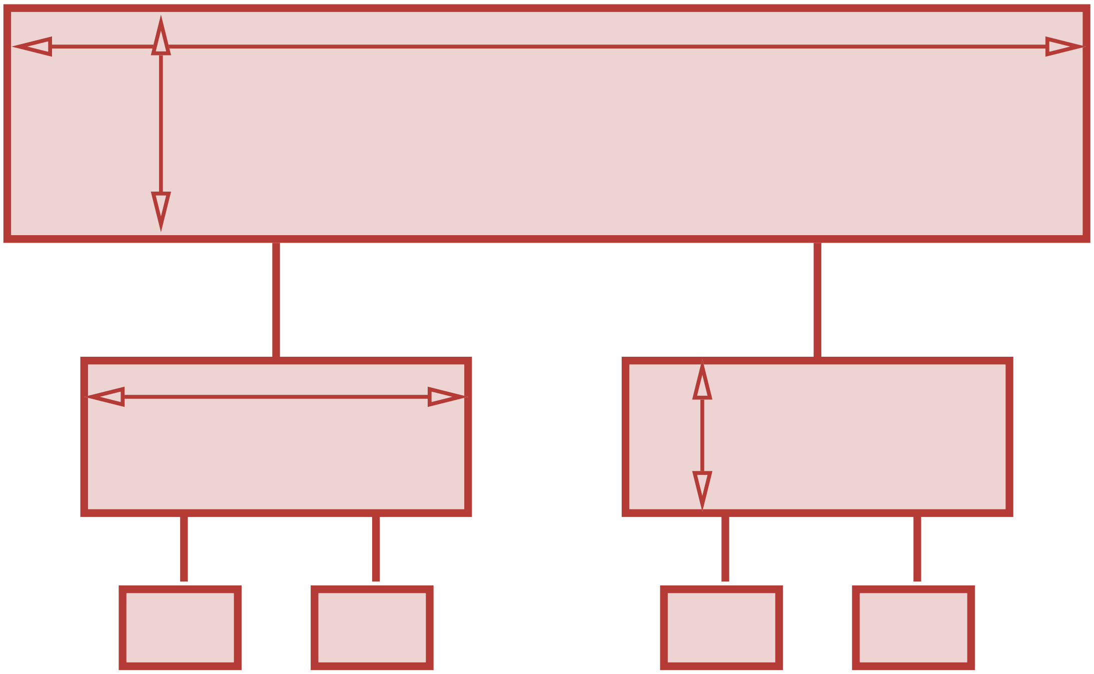
- From variant to executable code
- High-level target language

What is the necessary social investment?

Example

```
   integer*4 class, Sample, ClassA, ClassB
   parameter ( Sample=1, ClassA=2, ClassB=3 )
   integer*4 cache, KB32, KB64, KB128, KB256
   parameter ( KB32=1, KB64=2, KB128=3, KB256=4 )
c
c  Specify the cache and problem sizes
c
   parameter (class = ClassA)
   parameter (cache = KB32)
c
c  Processor grid width for P processors
c  Three partially-conflicting goals:
c 1.  Shape roughly square to reduce communication
c 2.  Have enough columns to reduce cache misses
c 3.  Avoid overhead of too many columns
c
c  P =   1    2    4    8   16   32   64  128  256
  data (cols_array(LgProc, Sample, KB32),LgProc=0,8)
 $     / 1,   1,   2,   2,   4,   4,   8,   8,  16 /
  data (cols_array(LgProc, ClassA, KB32),LgProc=0,8)
 $     / 1,   2,   4,   4,   4,   4,   8,   8,  16 /
  data (cols_array(LgProc, ClassB, KB32),LgProc=0,8)
 $     / 1,   1,   2,   8,   8,   8,   8,  16,  16 /
  data (cols_array(LgProc, Sample, KB64),LgProc=0,8)
 $     / 1,   1,   2,   2,   4,   4,   8,   8,  16 /
  data (cols_array(LgProc, ClassA, KB64),LgProc=0,8)
 $     / 1,   2,   2,   2,   4,   4,   8,   8,  16 /
```

PMH Model



Sequential computer

- Sequence of *memory modules*
- Connected by *channels*
  - Channels can be active simultaneously

Parallel computer

- Tree of memory modules
- Processors at the leaves
  - Memory capacity concentrated toward the root

Space-Limited Procedures

Recursive procedures

- Recursive calls **must** use less space
  - Promotes locality

Ambiguous argument passing semantics

  - Even for arrays!
- call-by-reference
  - Allows aggressive inlining (within a memory module)
- call-by-value
  - Allows explicit data movement (between memory modules)

Procedure name overloading

- Interchangable *versions*

Explicit tuning parameters

- *Machine parameters* of the PMH model
- *Problem parameters* describe problem instances
- *Free parameters* are deferred tuning choices

Explicit parallelism

Specialization

- Series of discrete choices
- Select a version for each module
  - Inline procedures with big arguments
  - Surface-sharing
- Resolve all tuning parameters
  - Machine parameters from the specific PMH
  - Problem parameters by the application tuner
  - Free parameters
    - System supplied defaults
    - May be overridden by tuner
- Performance feedback
  - Variant cost-estimation
    - As a function of the free parameters?
  - Code instrumentation

Expeditious Portability

Divide-and-conquer!

- Recursively break problems into subproblems
- Leave number and size of subproblems free

General performance considerations

- Parallelism
  - Independent subproblems execute concurrently
- Memory hierarchy
  - Divide-and-conquer tends to maintain locality
- Processor utilization
  - Conventional compiler optimizations

Specific performance considerations

- Procedure call overhead inlined away
- Array arguments passed by value, only if ...
  - data movement entailed on target computer

Necessary Social Investment

To tune N applications for M machines

$O(1)$ costs

- Generic model of computation (PMH)
- Language for generic programs
  - Space-Limited Procedures
- An interactive specialization engine
- A translator archetype

$O(N)$ costs

- Generic programs for applications ($O(N \log M)$?)

$O(M)$ costs

- Translator development

$O(NM)$ costs

- Specialization
- Inline code (target-specific inner loops)