Higher Level Programming on Parallel Computers: Sweetening the Deal for Programmers (and Making Compilers Work Harder)

Arun Chauhan, Indiana University

University of Rochester, Nov 30, 2009
Collaborators

Blake Barker
Torsten Hoefler
Eric Holk
William Holmes
Andrew Keep
Andrew Lumsdaine
Daniel McFarlin

Pushkar Ratnalikar
Koby Rubinstein
Sidney Shaw
Chun-Yu Shei
Jeremiah Willcock
Kevin Zumbrun
“A New Kind of Science”

Stephen Wolfram

“What our community should really aim for is the development of a curriculum that turns our subject into the fourth R—as in ’rogramming—of our education systems.

…

A form of mathematics can be used as a full-fledged programming language, just like Turing Machines.”

Matthias Felleisen and Shriram Krishnamurthy
Communications of the ACM, Jul 2009

“Computing is as fundamental as the physical, life, and social sciences.”

Peter J. Denning and Paul S. Rosenbloom
Communications of the ACM, Sep 2009
Programming
“Why can’t you be like the Math Department, which only needs a blackboard and wastepaper basket? Better still, like the Department of Philosophy. That doesn’t even need a wastepaper basket …”

Arthur C. Clarke
3001: The Final Odyssey
Computers are for Computing and ...

- Computers as general-purpose tools
  - communication, navigation, data collection, entertainment, etc.
- Computers as computing tools
  - problem solving
  - data processing and analysis
Overview

- Motivation
- Rethinking program analysis
- Rescuing parallel programmers
- Concluding remarks
Rethinking Program Analysis
Problem

- Nice programming languages
  - domain-specific
  - often dynamically typed and interpreted
- Poor performance
  - inefficient use of computing resources
  - inefficient use of energy
“It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.”

Sir Arthur Conan Doyle
A Scandal in Bohemia
Example 1: BLAS

\[ A + A \cdot B' + 2 \cdot (A+B)' \cdot A + (x+y) \cdot x' \]

```c
copy(A,tmp0);
gemm(1,A,B,1,tmp0);
copy(A,tmp1);
axpy(1,B,1,tmp1);
gemm(2,tmp1,A,1,tmp0);
copy(x,tmp1);
axpy(1,y,1,tmp1);
ger(1,tmp1,x,tmp0);
gemm(1,A,B,1,tmp0);
ger(1,x,x,tmp0);
ger(1,y,x,tmp0);
gemm(2,A,A,1,tmp0);
gemm(2,B,A,1,tmp0);
```

\[ A + A \cdot B' + 2 \cdot A' \cdot A + 2 \cdot B' \cdot A + x \cdot x' + y \cdot x' \]

```c
gemm(1,A,B,1,tmp0);
ger(1,x,x,tmp0);
ger(1,y,x,tmp0);
gemm(2,A,A,1,tmp0);
gemm(2,B,A,1,tmp0);
```
Example 1: BLAS

Implementing A Big Expression

- AMD Opteron
- PowerPC 970 (Apple G5)
- Intel Xeon
- Intel Itanium 2

Vector size (thousands of double elements) / time(parenthesized) / time(distributed)
Lessons

- Minimize buffer copies
- Combine as many simple operations as possible into a single BLAS call
- Work on data-flow graph
  - simple algorithm within basic blocks
  - expanded to work globally (intra-procedurally)
Example 1: BLAS

Implementing Scaled Vector Addition ($alpha \times x + beta \times y$)

- AMD Opteron
- PowerPC 970 (Apple G5)
- Intel Xeon
- Intel Itanium 2
Example 1: BLAS

Implementing Vector Outer Product ($x'y + A$)

- AMD Opteron
- PowerPC 970 (Apple G5)
- Intel Xeon
- Intel Itanium 2

Graph showing the time ratio ($t_{DGEMM} / t_{DGER}$) versus vector size (thousands of double elements) for different processors.
Results

POHLL (IPDPS) 2007, McFarlin and Chauhan

3.2 GHz dual Xeon, 1 GB RAM, Linux 2.6

- GS Arnoldi
- Gauss–Siedel
- Lanczos
- Arnoldi
- QR

![Graph showing performance comparison of different algorithms with data size on the x-axis and time ratio on the y-axis.](image-url)
Results

POHLL (IPDPS) 2007, McFarlin and Chauhan

1.5 GHz dual Intel Itanium 2, 4GB RAM, Linux 2.6
Example 2: Subscripts

\[ m = f(1) \cdot (n(c,c,c)) + f(2) \cdot (n(c,c,u) + n(c,c,d) + n(c,u,c) + n(c,d,c) + n(u,c,c) + n(d,c,c)) + f(3) \cdot (n(c,u,u) + n(c,u,d) + n(c,d,u) + n(c,d,d) + n(u,c,u) + n(u,c,d) + n(d,c,u) + n(d,c,d) + n(u,u,c) + n(u,d,c) + n(d,u,c) + n(d,d,c)) + f(4) \cdot (n(u,u,u) + n(u,u,d) + n(u,d,u) + n(u,d,d) + n(d,u,u) + n(d,u,d) + n(d,d,u) + n(d,d,d)); \]

Effects of Memory Optimizations on NASMG

- Direct translation to C
- Loop fusion
- Loop fusion + subscript opt.

HiPC 2009, Shei, Chauhan, and Shaw
Enabling Technology

- **Type Inference**
  - infer base types, and array sizes

- **Leverage MATLAB / Octave interpreter**
  - “concretely interpreted partial evaluation” to combine type inference and constant propagation+folding
  - type transfer functions encoded within MATLAB

- **Potential for spectacular improvements**
  - 100x on biology code (electron µ-scope image-proc.)
  - 1.5x on math code (ODE solver)
Type Inference Through Concrete Interpretation

Example Code

```
x = 10.5;
y = [1, 2; 3, 4];
y = x * y + a;
```

After SSA & flattening

```
x$1 = 10.5;
y$1 = [1, 2; 3, 4];
t$1 = x$1 * y$1;
y$2 = t$1 + a$1;
```

With type disambiguation code

```
BT_x$1 = 'd';
x$1 = 10.5;
BT_y$1 = BXF_vertcat(...
  BXF_horzcat('i', 'i'), ...;
  BXF_horzcat('i', 'i') ...)
);
y$1 = [1, 2; 3, 4];
BT_t$1 = ...;
  BXF_product(BT_x$1, BT_y$1);
t$1 = x$1 * y$1;
BT_y$2 = ...;
  BXF_sum(BT_t$1, BT_a$1);
y$2 = t$1 + a$1;
```

After partial evaluation

```
x$1 = 10.5;
y$1 = [1, 2; 3, 4];
t$1 = x$1 * y$1;
BT_y$2 = ...;
  BXF_sum(BT_t$1, BT_a$1);
y$2 = t$1 + a$1;
```
Inferring Base Types

- dlaplacian
- arnoldi
- v_hbmult
- clean_image
- reseat_points
- get_slopes

**Graph:**
- x-axis: base type annotations
- y-axis: percent of total variables
- Categories: No Annotations, Base Type Annotations, Base Type + Size Annotations, Base Type + Size + Value Annotations

Values:
- diaplacian: 100%
- arnoldi: 100%
- v_hbmult: 100%
- clean_image: 70%
- reseat_points: 40%
- get_slopes: 20%
Inferring Array Sizes

The diagram shows a bar chart comparing different levels of annotations in terms of the percent of total variables. The categories include:

- No Annotations
- Base Type Annotations
- Base Type + Size Annotations
- Base Type + Size + Value Annotations

The bars represent different functions, with 'diplacian' leading significantly in terms of annotations compared to 'arnoldi', 'v_hbmult', 'clean_image', 'reseat_points', and 'get_slopes'.
Static vs Dynamic Inference

![Bar chart showing the percentage of total variables that are statically inferred versus dynamically inferred at runtime for various functions: `diplacian`, `arnoldi`, `v_hbmult`, `clean_image`, `reseat_points`, and `get_slopes`. The chart indicates the distribution of variables across different functions, with a comparison between static and dynamic inference methods.]
Observations

- Memory seems to play a key role in performance of high-level dynamically typed languages (studied MATLAB and Ruby).

- Lack of general-purpose analytical models to guide the compiler toward generating programs with better memory locality:
  - Need inter-procedural methods.
  - Need a way to incorporate separately-compiled libraries.
Static Reuse Distances

\[
x = a + b; \\
c = a + d[i]*100; \\
y = x * 10;
\]

Static Reuse Distance = 6 (a, b, c, d, i, 100)

**Definition:** A reference point, \( p \), is the unique syntactic reference that is either an lvalue or an rvalue. When the point is inside a loop nest a superscripted reference point \( p^i \) refers to the dynamic instance of \( p \) at the iteration vector \( i \).
SRD Across Function Calls

\[ [\beta_1, \beta_2, \ldots, \beta_n] = f(\alpha_1, \alpha_2, \ldots, \alpha_m) \]

\[ x_f : A \rightarrow Z \]
\[ x_f(a) \text{ is the volume of data accessed within } f. \]

\[ I_f : P \times A \rightarrow Z \]
\[ I_f(p, a) \text{ is the volume of data accessed within } f \text{ before the first access to } \rho. \]

\[ O_f : P \times A \rightarrow Z \]
\[ O_f(p, a) \text{ is the volume of data accessed within } f \text{ after the last access to } \rho. \]

\[ P_f : P \times P \times A \rightarrow Z \]
\[ P_f(p_1, p_2, a) \text{ is the volume of data accessed between the last use of } \rho_1 \text{ and the first use of } \rho_2 \text{ within } f. \text{ It is 0 if } a(\rho_1, \rho_2) \text{ is true.} \]
SRD For Regions of Code

\( \mathcal{X}_f : A \rightarrow Z \)
\( \mathcal{X}_f(a) \) is the volume of data accessed within \( f \).

\( \mathcal{I}_f : P \times A \rightarrow Z \)
\( \mathcal{I}_f(p,a) \) is the volume of data accessed within \( f \) before the first access to \( p \).

\( \mathcal{O}_f : P \times A \rightarrow Z \)
\( \mathcal{O}_f(p,a) \) is the volume of data accessed within \( f \) after the last access to \( p \).

\( \mathcal{P}_f : P \times P \times A \rightarrow Z \)
\( \mathcal{P}_f(p_1,p_2,a) \) is the volume of data accessed between the last use of \( p_1 \) and the first use of \( p_2 \) within \( f \). It is 0 if \( \alpha(p_1,p_2) \) is true.

\( \mathcal{X}_c : A \rightarrow Z \)
\( \mathcal{X}_c(a) \) is the volume of data accessed within \( c \).

\( \mathcal{I}_c : P \times A \rightarrow Z \)
\( \mathcal{I}_c(p,a) \) is the volume of data accessed within \( c \) before the first execution of \( p \).

\( \mathcal{O}_c : P \times A \rightarrow Z \)
\( \mathcal{O}_c(p,a) \) is the volume of data accessed within \( c \) after the last execution of \( p \).
Algorithm to Compute $I_c$

1 \textbf{Algorithm: Compute} $I_c$
2 \textbf{Input:} code region $c$; reference point $p$; alias function $a$ that is valid over $c$
3 \textbf{Output:} $I_c(p, a)$

4 if $c = [c_1; c_2]$ then
5 \hspace{1em} if $p \in [c_1]$ then
6 \hspace{2em} return $I_{c_1}(p, a)$
7 \hspace{1em} else
8 \hspace{2em} return $X_{c_1}(a) + I_{c_2}(p, a)$
9 else if $c = [\text{if } e \text{ then } c_1 \text{ else } c_2]$ then
10 \hspace{1em} if $p = c$ then
11 \hspace{2em} return 0
12 \hspace{1em} else if $p \in [c_1]$ then
13 \hspace{2em} return $|e| + I_{c_1}(p, a)$
14 \hspace{1em} else
15 \hspace{2em} return $|e| + I_{c_2}(p, a)$
16 else if $c = \text{for } i = L : S : U \text{ begin } c_1 \text{ end}$ then
17 \hspace{1em} if $p \in [\text{for } i = L : S : U]$ then
18 \hspace{2em} return 0
19 else
20 \hspace{1em} $\vec{k} \leftarrow$ first iteration vector in which $p$ is reached
21 \hspace{1em} $\vec{k}' \leftarrow$ largest iteration vector smaller than $\vec{k}$
22 \hspace{1em} $r \leftarrow \Sigma_{p_1, p_2 \in c_1} |p_1 \xrightarrow{c} p_2|$, $p_1 \xrightarrow{c} p_2 \in \text{polytope}(c, \vec{k}')$
23 \hspace{1em} /* $\xrightarrow{c}$ denotes loop carried dependence */
24 \hspace{1em} return $\Sigma_{i < \vec{k}} X_{c_1}^i(a) + I_{c_1}^{\vec{k}}(p, a) - r$
25 else
26 \hspace{1em} ERROR
Algorithm to Compute $X_c$

1. **Algorithm**: Compute $X_c$
2. **Input**: code region $c$; alias function $a$ that is valid over $c$; probability weights, $\pi$, on CFG edges
3. **Output**: $X_c(a)$

```plaintext
1 if $c = [c_1; c_2]$ then
2 | return $X_{c_1}(a) + X_{c_2}(a)$
3 else if $c = [\text{if } e \text{ then } c_1 \text{ else } c_2]$ then
4 | return $|e| + \pi(\text{true}) \times X_{c_1}(a) + \pi(\text{false}) \times X_{c_2}(a)$
5 else if $c = \text{for } i = L : S : U \text{ begin } c_1 \text{ end}$ then
6 | $\vec{n} \leftarrow$ iteration vector for the last iteration
7 | $r = \sum_{p_1, p_2 \in c_1} |p_1 \overset{c}{\rightarrow} p_2|$, $p_1 \overset{c}{\rightarrow} p_2 \in \text{polytope}(c, \vec{n})$
8 | /* $\overset{c}{\rightarrow}$ denotes loop carried dependence */
9 | return $\sum_{i \leq \vec{n}} X^{\vec{i}}_{c_1}(a) - r$
10 else
11 | ERROR
```
Accuracy

Accuracy

L1 Hits
L2 Hits
L2 Misses
0
0.5
1
1.5
2
2.5
3
\(x \times 10^9\)

Arnoldi (computed)
Arnoldi (measured)
NASCG (computed)
NASCG (measured)
Empirical Data for Library Functions

\[ \frac{\beta_1, \beta_2, \ldots, \beta_n}{P_r} = f(\alpha_1, \alpha_2, \ldots, \alpha_m) \]

\[ X_f \approx L2 \text{ cache misses} \]

\[ I_f \approx O_f \approx X_f \]
Challenges Remain: FFT
Rescuing Parallel Programmers
Concurrency Trends
(ExaScale Computing Study, Peter Kogge et al.)

Figure 4.16: Total hardware concurrency in the Top 10 supercomputers.

Figure 4.17: Memory capacity in the Top 10 supercomputers.
Types of (Parallel) Programmers

- **Mainstream Parallelism-Oblivious Developers**
  - Joe

- **Parallelism-Aware Developers**
  - Stephanie

- **Concurrency Experts**
  - Doug

Joe needs high level Programming Models designed for Domain Experts

Stephanie needs simple Parallel Programming Models with safety nets

Focus of today’s Parallel Programming Models

**Courtesy: Vivek Sarkar, Rice University**
Parallelism Oblivious Users

- Programming languages-driven
  - implicit parallelism, compiler support
- OS-driven
  - innovative solutions to leverage extra cores
- Architecture-driven
  - ILP, hyper-threading
Observations for Parallelism-Aware and Expert Users

- Completely automatic parallelization has had limited success
- Writing parallel programs is hard; optimizing and maintaining them is harder!
- Compilation technology has worked well in communication optimization
Declarative Parallel Programming

- Let users write parallel programs
- Let compilers optimize parallel programs
- Separate computation and communication specification, using a domain-specific language to specify communication
- Key insight: most parallel applications have predictable (but not necessarily static) communication patterns
Declarative Specification of Communication

Compiler converts collectives to MPI calls and optimizes communication by coalescing and overlapping with computation.

```
@collective cshift (A)
{
    foreach processor i
    {
        A@i := A@(i+1)
    }
}
```

```
@collective stencil (A, B)
{
    foreach processor i in Mesh2D
    {
        B@i := 0.25*(A@i.N + A@i.S + A@i.W + A@i.E)
    }
}
```
Concluding Remarks

- Computing is a core technique in an increasing number of fields
  - programming is no longer restricted to scientists and engineers
  - conventional programming models are inadequate
- Parallelism is no longer restricted to scientific and engineering applications
  - need to address the needs of different types of users and applications
- Traditional program analysis is inadequate on modern machines
Other Interests

- High-level Languages
  - Ruby
- Heterogeneous parallel computing
- Large memory-footprint applications
- Automatic parallelization
Scratch
http://scratch.mit.edu/
http://www.cs.indiana.edu/~achauhan

http://phi.cs.indiana.edu/
Bonus Material
Ruby Class Hierarchy

```
class B < Object
  ...
end

class D < B
  ...
end

d = D.new

def d.newMeth
  ...
end
```

Objects store values
Classes store methods
Ruby Classes as Objects

Class → Class’ → B → B’ → B” → B’” → nil

Module → Module’ → Obj. → Obj’ → Obj” → Obj’” → nil

undefined

M → M’ → S_d → S_d’ → S_d” → S_d’”

d → d

Superclass of the meta-class is meta-class of the superclass.