Effective (Parallel) Programming for the Masses
Optimizing High-level Languages

Arun Chauhan
School of Informatics and Computing
Indiana University, Bloomington

Purdue University
October 28, 2011
Computing as a Fundamental Science

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

Peter J. Denning and Paul S. Rosenbloom
Communications of the ACM, Sep 2009

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

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

“I would rather spend 10 minutes coding and letting the program run overnight, than spend weeks writing and debugging to be able to run the program in 10 minutes.”

A DSP researcher in EE
MATLAB
MATLAB: Ease of Use
MATLAB in a Nutshell

- C-like syntax
  - \( x = 1 + 1; \)
  - \( y = 2 \times x + 100; \)
- Array operations
  - \( C = A \times B; \)
  - \( C = A . \times B; \)
- IF, FOR, WHILE, SWITCH statements
Motivation: NASMG

```matlab
m = f(1).*(n(c,c,c))
  + f(2).*(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).*(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).*(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));
```
Optimization Potential

Effects of Memory Optimizations on NASMG

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

Speedup over MATLAB interpreter

- Core 2 Duo
- Pentium IV
MATLAB / Octave Compiler

Infrastructure written in Ruby
Uses our own embedded DSL called RubyWrite
BasicCompilationIssues

• Dynamic types
  • infer types to enable translation to lower-level (statically typed) language

• Dynamic dispatch
  • specialize for static dispatch
  • use types info. to specialize based on input types

• High-level operators
  • intelligently map to underlying libraries

Basic Compilation Issues

- **Dynamic types**
  - infer types to enable translation to lower-level (statically typed) language

- **Dynamic dispatch**
  - specialize for static dispatch
  - use types info. to specialize based on input types

- **High-level operators**
  - intelligently map to underlying libraries

MATLAB Type Inference: Past Efforts

• As a data flow problem
  • abstract interpretation to propagate types
  • can be combined with constant propagation
  • not easy to handle complex library functions

• As a set-theoretic problem
  • need external symbolic analysis tool (e.g., Mathematica)

• As constraint equations over sets
  • could be too loosely constrained
Leverage MATLAB Interpreter (1)

\[
x = 10; \\
y = 20; \\
z = x + y;
\]
Leverage MATLAB Interpreter (1)

\[ x = 10; \]
\[ y = 20; \]
\[ z = x + y; \]
Leverage MATLAB Interpreter (1)

\[
\begin{align*}
\text{BT}_x &= \text{‘i’}; \\
x &= 10; \\
y &= 20; \\
z &= x + y;
\end{align*}
\]
Leverage MATLAB Interpreter (1)

```matlab
BT_x = 'i';
x = 10;
BT_y = 'i';
y = 20;

z = x + y;
```
Leverage MATLAB Interpreter (1)

\[ \text{BT}_x = 'i'; \]
\[ x = 10; \]
\[ \text{BT}_y = 'i'; \]
\[ y = 20; \]
\[ \text{BT}_z = \text{BXF}_\text{sum}(\text{BT}_x, \text{BT}_y); \]
\[ z = x + y; \]
Leverage MATLAB Interpreter (2)

```matlab
x = 10.5;
y = [1, 2; 3, 4];
y = x*y + a;
```
Leverage MATLAB Interpreter (2)

```matlab
x = 10.5;
y = [1, 2; 3, 4];
t = x*y;
y = t + a;
```
Leverage MATLAB Interpreter (2)

\[
x \cdot 1 = 10.5;
y \cdot 1 = [1, 2; 3, 4];
t \cdot 1 = x \cdot 1 \cdot y \cdot 1;
y \cdot 2 = t \cdot 1 + a \cdot 1;
\]
Leverage MATAB Interpreter (2)

\[ x_1 = 10.5; \]

\[ y_1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}; \]

\[ t_1 = x_1*y_1; \]

\[ y_2 = t_1 + a_1; \]
Leverage MATLAB Interpreter (2)

```
BT_x1 = 'd';
x1 = 10.5;

y1 = [1, 2; 3, 4];

t1 = x1*y1;

y2 = t1 + a1;
```
Leverage MATLAB Interpreter (2)

```matlab
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];
t$1 = x$1*y$1;
y$2 = t$1 + a$1;
```
Leverage MATLAB Interpreter (2)

\[
\begin{align*}
BT_x$1 &= 'd'; \\
x$1 &= 10.5; \\
BT_y$1 &= \text{B XF}_\text{vertcat}( \ldots \\
& \quad \text{B XF}_\text{horzcat}( 'i', 'i' ), \ldots \\
& \quad \text{B XF}_\text{horzcat}( 'i', 'i' ), \ldots ) \\
y$1 &= [1, 2; 3, 4]; \\
BT_t$1 &= \text{B XF}_\text{product}(BT_x$1, BT_y$1); \\
t$1 &= x$1*y$1; \\
\end{align*}
\]

\[
y$2 = t$1 + a$1;
\]
Leverage MATLAB Interpreter (2)

\[
\begin{align*}
\text{BT}_x & = 'd'; \\
x & = 10.5; \\
\text{BT}_y & = \text{BXF}_\text{vertcat}( \ldots \\
& \quad \text{BXF}_\text{horzcat}( 'i', 'i' ), \ldots \\
& \quad \text{BXF}_\text{horzcat}( 'i', 'i' ) \ldots ) ; \\
y & = [1, 2; 3, 4]; \\
\text{BT}_t & = \text{BXF}_\text{product}(\text{BT}_x, \text{BT}_y) ; \\
t & = x \times y; \\
\text{BT}_y & = \text{BXF}_\text{sum}(\text{BT}_t, \text{BT}_a) ; \\
y & = t + a;
\end{align*}
\]
Leverage MATLAB Interpreter (2)

```matlab
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;
```
Leverage MATLAB Interpreter (3)

if x < 0
    y = 1.5;
else
    y = 2;
end
if $x_1 < 0$
  $y_1 = 1.5$;
else
  $y_2 = 2$;
end
$y_3 = \phi(y_1, y_2)$
Leverage MATLAB Interpreter (3)

if $x_1 < 0$

\[ y_1 = 1.5; \]
else

\[ y_2 = 2; \]
end

\[ y_3 = \varphi(y_1, y_2) \]
Leverage MATAB Interpreter (3)

if x$1 < 0
    BT_y$1 = ‘d’;
    y$1 = 1.5;
else
    BT_y$2 = ‘i’;
    y$2 = 2;
end

BT_y$2 = BTMAX(BT_y$1, BT_y$2);
y$3 = \phi(y$1, y$2)
Leverage MATAAB Interpreter (3)

```matlab
if x$1 < 0
    BT_y$1 = 'd';
y$1 = 1.5;
else
    BT_y$2 = 'i';
y$2 = 2;
end
BT_y$2 = BTMAX(BT_y$1,BT_y$2);
y$3 = \varphi(y$1,y$2)
```
Inference Steps

• For each statement of the form $\rho = f(\alpha)$, insert a statement $\rho_T = f_{\text{BXF}}(\alpha_T)$

• Perform concrete partial evaluation

• Perform dead-code elimination
  • leaves those type computations that are used for run time optimization
Inference Steps

- For each statement of the form \( \rho = f(\alpha) \), insert a statement \( \rho_T = f_{\text{BXF}}(\alpha_T) \)
- Perform concrete partial evaluation
- Perform dead-code elimination
  - leaves those type computations that are used for run time optimization

Need to do a bit more for loops (details in the paper)
Base Type Lattice

- Logical
- Integer
- Double
- Complex

- Char
- Cell
Other Type Inference Issues

• Struct types
  • each field can be considered a separate variable

• Procedures with side-effects
  • output types cannot be computed if that involves executing a slice of the original procedure with side-effects

• Recursive procedures
  • can be handled with a fixed-point evaluation
Evaluation: Precision (Base)
Evaluation: Precision (Size)
Evaluation: Static vs Dynamic

The diagram compares the percent of total variables that are statically inferred and dynamically inferred at runtime for various functions. The functions include `dlaplacian`, `arnoldi`, `v_hbmull`, `clean_image`, `reseat_points`, and `get_slopes`. The y-axis represents the percent of total variables, and the data shows a significant variation across the functions, indicating different levels of static and dynamic inference.
Observations

• Advantages of concrete interpretation
  • maintains semantic fidelity for languages defined by their interpreters
  • protects against language changes
  • avoids duplication of effort

• Solving other problems
  • can be seen as an alternative to traditional data flow analysis, for certain problems
The Free Lunch is Over

Herb Sutter, The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, Dr. Dobb's Journal, 30(3), March 2005
Exa-scale Challenge
Trends in Concurrency

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

Figure 4.17: Memory capacity in the Top 10 supercomputers.

Long History of Parallelism

- Vector processors
- Symmetric multi-processors (SMPs)
- Nodes over inter-connection networks
- Instruction-level parallelism
- Multi-cores
- GPUs
- ...
Parallelism

Mainstream Parallelism-Oblivious Developers

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

Concurrency Experts

Parallelism-Aware Developers

(Stephanie)

(Doug)

(Joe)

Courtesy: Vivek Sarkar, Rice University
Parallelism
Parallelism
Thinking of Joe programmers
Automatic parallelization

“The reports of my death are highly exaggerated”

- MATLAB is the *lingua franca* of scientists and engineers
- Joe programmers would rather write in 10 minutes and let the program run for 24 hours, than vice versa
- They would still like their programs to run in 10 minutes!
- **We can leverage inferred types for automatic parallelization**
Parallelism in MATLAB

• Built-in `parallel-for` (with the parallel computing toolbox)

• Third party libraries to offload computations on clusters

• Third party and MathWorks libraries to offload computation on GPUs

• “declare” variables to be of GPU type

```matlab
A = GPUdouble(a);
B = GPUdouble(b);
C = A*B;
```
MATLAB: Empirical Study

Basic Block Sizes

Basic Block Counts
Automatic GPU Computation

- Model the computation
  - cost model for CPU times
  - cost model for GPU times
  - cost model for CPU-GPU data transfer
- Solve a binary integer linear programming problem

\[
\begin{align*}
\text{Minimize} \quad & \quad f^T \bar{x} \\
\text{such that} \quad & \quad A \bar{x} \leq \bar{b} \\
\text{and} \quad & \quad A_{eq} \bar{x} = \bar{b}_{eq}
\end{align*}
\]
Experimental Results

Heated Plate

N-body (3D)
Extending to other Languages

• Unique characteristics of MATLAB
  • simple basic data types
  • simple control flow
  • first-order functions
  • array language directly encodes data parallelism

• Ruby
  • object-oriented, with meta-programming support
  • closures, co-routines, higher-order functions
  • open classes
Ruby: Type Complications

class Foo
  def my_method
    ...
  end
end

... 
f = Foo.new

...

... 
g = Foo.new
Ruby: Type Complications

class Foo
    def my_method
        ...
    end
end
...
f = Foo.new
class Foo
    ...
end
...
g = Foo.new
Ruby: Type Complications

class Foo
  def my_method
    ...
  end
end

... 

f = Foo.new

... 

def bar
  class Foo
    ...
  end
end

g = Foo.new
Challenges

• *Reasonable* static type inference
• Identifying conditions under which the inference is correct
• Detecting and verifying those conditions at run-time
• Possibly *speculating* on types
What about Stephanie programmers?
High Performance Fortran

PROGRAM SUM
    REAL A(10000)
    READ (9) A
    SUM = 0.0
    DO I = 1, 10000
        SUM = SUM + A(I)
    ENDDO
    PRINT SUM
END
High Performance Fortran

PROGRAM SUM
    REAL A(10000)
    READ (9) A
    SUM = 0.0
    DO I = 1, 10000
        SUM = SUM + A(I)
    ENDDO
    PRINT SUM
END

PROGRAM PARALLEL_SUM
    REAL A(100), BUFF(100)
    IF (PID == 0) THEN
        DO IP = 0, 99
            READ (9) BUFF(1:100)
            IF (IP == 0) A(1:100) = BUFF(1:100)
            ELSE SEND(IP, BUFF, 100) ! 100 words to Proc 1
        ENDDO
    ELSE
        RECV(0, A, 100) ! 100 words from proc 0 into A
    ENDFI
    SUM = 0.0
    DO I = 1, 100
        SUM = SUM + A(I)
    ENDDO
    IF (PID == 0) THEN;
        RECV(99, SUM, 1); PRINT SUM; ENDIF
END
High Performance Fortran

PROGRAM SUM
    REAL A(10000)
    READ (9) A
    SUM = 0.0
    DO I = 1, 10000
        SUM = SUM + A(I)
    ENDDO
    PRINT SUM
END

PROGRAM HPF_SUM
    REAL A(10000)
    !HPF$ DISTRIBUTE A(BLOCK)
    READ (9) A
    SUM = 0.0
    DO I = 1, 10000
        SUM = SUM + A(I)
    ENDDO
    PRINT SUM
END
HPF: Victim of its own Success?

- No prior compiler technology to learn from
- Limited number of data distribution primitives
  - not user expandable
- Paucity of good HPF libraries
- Lack of performance-tuning tools
- Lack of patience of user community!

HPF: Victim of its own Success?

- No prior compiler technology to learn from
- Limited number of data distribution primitives
  - not user expandable
- Paucity of good HPF libraries
- Lack of performance-tuning tools
- Lack of patience of user community!

Does not motivate users to think in parallel

Design Principles

• Users must think in parallel (creativity)
  • but not be encumbered with optimizations that can be automated, or proving synchronization correctness

• Compiler focuses on what it can do (mechanics)
  • not creative tasks, such as determining data distributions, or creating new parallel algorithms

• Incremental deployment
  • not a new programming language
  • more of a coordination language (DSL)

• Formal semantics
  • provable correctness
Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
  - alternate between computation and communication
  - communication optimization breaks the structure
Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
- alternate between computation and communication
- communication optimization breaks the structure
Declarative Approach

• Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication

• alternate between computation and communication

• communication optimization breaks the structure
Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
  - alternate between computation and communication
  - communication optimization breaks the structure

- Extend to non BSP-style applications
Kanor for Clusters

@communicate { b@recv_rank <<= a@send_rank }
Kanor for Clusters

@communicate { b@recv_rank <<= a@send_rank }

e_0 @ e_1 <<= op <<= e_2 @ e_3 where e_4

---


Arun Chauhan, Programming for the Masses, Purdue, Oct 28, 2011
Kanor for Clusters

@communicate { b@recv_rank <= a@send_rank }

e_0 @ e_1 << op << e_2 @ e_3 where e_4

e_0 @ e_1 <<= e_2 @ e_3 where e_4
Kanor for Clusters

```latex
@communicate \{ b@recv\_rank <= a@send\_rank \}

\[ e_0 @ e_1 << op << e_2 @ e_3 \text{ where } e_4 \]

\[ e_0 @ e_1 <= e_2 @ e_3 \text{ where } e_4 \]
```

- **A[j]**: Storage location
- **@i**: Receiver rank
- **<<=**: Reduction operator
- **B[i]**: Data
- **@j**: Sender rank
- **where i in world**, **j in {0...i}**, **i % 2 == 0**: Generator, generator, filter

---


Arun Chauhan, Programming for the Masses, Purdue, Oct 28, 2011
Kanor for Clusters

@communicate { b@recv_rank <<= a@send_rank }

e_0 @ e_1 <<= op <<= e_2 @ e_3 where e_4

e_0 @ e_1 <<= e_2 @ e_3 where e_4

A[j] @ i <= B[i] @ j where i in world, j in {0...i}, i % 2 == 0

Source-level compiler (using ROSE)

standard C++ code


Arun Chauhan, Programming for the Masses, Purdue, Oct 28, 2011
Distributed Memory Targets

• Generate MPI

• Recognize collectives that map to MPI collectives

• Optimize communication
  • computation-communication overlap
  • communication coalescing
Software Pipelining

for (int i = 0; i < OCTANTS; i++) {
    for (int j = 0; j < ANGLES; j++) {
        // loop through the diagonals, N is the number of processors
        for (int diag = 0; diag < 2 * N + 1; diag++) {
            if ((myid.x + myid.y) == diag) { compute(); } /* wave front */
            @communicate { temp_s@(x, y+1) <<= A[lastrow]@(x, y)
                where x, y in {0...N-1} and x + y = diag; }
            @communicate { temp_e@(x + 1, y) <<= A[lastcol]@(x, y)
                where x, y in {0...N-1} and x + y = diag; }
        }
    }
}

Software Pipelining

```c
for (int i = 0; i < OCTANTS; i++) {
    for (int j = 0; j < ANGLES; j++) {
        // loop through the diagonals, N is the number of processors
        for (int diag = 0; diag < 2 * N + 1; diag++) {
            if ((myid.x + myid.y) == diag) { compute(); } /* wave front */
            @communicate { temp_s@(x, y+1) <<= A[lastrow]@(x, y) }
            where x, y in {0...N-1} and x + y = diag;
            @communicate { temp_e@(x + 1, y) <<= A[lastcol]@(x, y) }
            where x, y in {0...N-1} and x + y = diag;
        }
    }
}
```


Arun Chauhan, Programming for the Masses, Purdue, Oct 28, 2011
### Harlan for GPUs

Harlan for GPUs

```c
__global__ void add_kernel(int size, float *X, float *Y, float *Z)
{
    int i = threadIdx.x;
    if (i < size) { Z[i] = X[i] + Y[i]; }
}

void vector_add(int size, float *X, float *Y, float *Z)
{
    float *dX, *dY, *dZ;
    cudaMalloc(&dX, size * sizeof(float));
    cudaMalloc(&dY, size * sizeof(float));
    cudaMalloc(&dZ, size * sizeof(float));

    cudaMemcpy(dX, X, size * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dY, Y, size * sizeof(float), cudaMemcpyHostToDevice);

    add_kernel<<<1, size>>>(size, dX, dY, dZ);

    cudaMemcpy(Z, dZ, size * sizeof(float), cudaMemcpyDeviceToHost);
}
```

Figure 1. CUDA code for adding two vectors.

```c
__global__ void add_kernel(int size, float *X, float *Y, float *Z)
{
    int i = threadIdx.x;
    if (i < size) { Z[i] = X[i] + Y[i]; }
}

void vector_add(int size, float *X, float *Y, float *Z)
{
    float *dX, *dY, *dZ;
    cudaMalloc(&dX, size * sizeof(float));
    cudaMalloc(&dY, size * sizeof(float));
    cudaMalloc(&dZ, size * sizeof(float));

    cudaMemcpy(dX, X, size * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dY, Y, size * sizeof(float), cudaMemcpyHostToDevice);

    add_kernel<<<1, size>>>(size, dX, dY, dZ);

    cudaMemcpy(Z, dZ, size * sizeof(float), cudaMemcpyDeviceToHost);
}
```

Figure 2. Harlan code for adding two vectors.
Harlan for GPUs

```cpp
__global__ void add_kernel(int size, float *X, float *Y, float *Z)
{
    int i = threadIdx.x;
    if (i < size) { Z[i] = X[i] + Y[i]; }
}

void vector_add(int size, float *X, float *Y, float *Z)
{
    float *dX, *dY, *dZ;
    cudaMalloc(&dX, size * sizeof(float));
    cudaMalloc(&dY, size * sizeof(float));
    cudaMalloc(&dZ, size * sizeof(float));

    cudaMemcpy(dX, X, size * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dY, Y, size * sizeof(float), cudaMemcpyHostToDevice);

    add_kernel <<<1, size >>>(size, dX, dY, dZ);

    cudaMemcpy(Z, dZ, size * sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(dX);
    cudaFree(dY);
    cudaFree(dZ);
}
```

```cpp
void vector_add (vector<float> X, vector<float> Y, vector<float> Z)
{
    kernel (x : X, y : Y, z : Z) { z = x + y; }
}
```
Harlan Features

Reductions

\[ z = +/-\text{kernel} \ (x : X, \ y : Y) \ \{ \ x * \ y \ \}; \]
Harlan Features

Reductions

\[ z = \texttt{+/kernel} (x : X, y : Y) \{ x * y \}; \]

Asynchronous kernels

\[
\text{handle} = \texttt{async kernel} (x : X, y : Y) \{ x * y \};
\]

// other concurrent kernels of program code here
\[ z = \texttt{+/wait} (\text{handle}); \]
Harlan Features

Reductions

\[
  z = +/\text{kernel} (x : X, y : Y) \{ x \ast y \};
\]

Asynchronous kernels

\[
  \text{handle} = \text{async \ kernel} (x : X, y : Y) \{ x \ast y \}; \quad \text{// other concurrent kernels of program code here}\n\]
\[
  z = +/\text{wait} (\text{handle});
\]

Nested kernels

\[
  \text{total} = +/\text{kernel} (\text{row} : \text{Rows}) \{ +/\text{kernel} (x : \text{row}); \};
\]

Serious Joe programmer?
Scalable Speculative Parallelism on Clusters

```c
// safe code

// code where speculation possible (code region A)
// safe code

// code where speculation possible (code regions B)

FF_init();

// safe code

if (FF_fork() == FF_VERIFIER) {
    // safe version of the code region A
} else { // FF_SPECULATOR
    // unsafe version of the code region A
}

FF_create_validation_thread();

// safe code

if (FF_fork() == FF_VERIFIER) {
    // safe version of the code region B
} else { // FF_SPECULATOR
    // unsafe version of the code region B
}

FF_create_validation_thread();
```

The actual validation may be performed on either end, i.e., either on the thread created by the verifier or the speculator, based on a configuration parameter passed to \texttt{FFfork}. All the details of forking, validator thread creation, and data transfer are hidden inside the API. Notice that in order to gain any real performance over the original code there must be at least two regions of code that can be executed speculatively, or at least a speculative region followed by a non-speculative (safe) region. This lets the latency of the verification process be hidden by concurrently executing the verifier for first speculative region with the second speculative region or the non-speculated region.

The validation is performed by comparing results of the region that is computed speculatively. A simple method to automate the validation step is to compare only the set of live variables at the end of the speculated region. In our current implementation this can be done either through our source-level compiler or through a callback function written by the user, for complicated cases that compiler may not be able to handle. Using an explicit comparison callback function offers a high level of flexibility in implementing the comparison operation. Thus, recursive pointer-based data structures could be easily compared using the knowledge of the data structures. For example, if a dynamically allocated data structure was created within the code region that is being speculated on then a simple byte comparison for validation may be misleading since pointer addresses are unlikely to match across the verifier and the speculator. Similarly, the comparison operation could be made to tolerate small differences. If speculation involves changing the order of floating point operations the results might vary within an acceptable margin of error. This requires algorithmic knowledge that a compiler is unlikely to have. In such cases a special callback function could be specified by the user for particular data structures or variables.

2.2. The Runtime System

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure2}
\caption{Intra-node and inter-node implementations of FastForward. Validation threads are not shown in the inter-node case for the sake of clarity. Also omitted are the components of the checkpoint/restart library, DMTCP. In reality, the processes must be started through the DMTCP proxies and a DMTCP coordinator must run on each node where speculators could be launched. Figure 2 shows the architecture of the runtime system of FastForward. Two different strategies are used for intra-node and inter-node implementations. When speculating within a node new processes are created using standard \texttt{fork} system call with copy-on-write semantics, as shown in Figure 2(a). The \texttt{fork} system call turns out to have a low overhead. For inter-node, or distributed speculation, we use the DMTCP library to checkpoint the running process and then transfer the checkpointed image to the remote node to start the duplicate process. In effect, this implements a remote fork. FastForward makes use of a helper MPI application to implement the distributed speculation, as Figure 2(b) shows. The checkpoint data are transferred using NFS that operates over GigE on our system. The MPI helper process listens on a pipe. As soon as it receives an indication that the checkpoint data is being written, it starts the process of contacting the directory server and requesting for an available node. Thus, three activities progress concurrently: (a) writing of the checkpoint; (b) execution of the next speculative region; and (c) protocol for requesting an available remote node for remote-fork. As soon as the checkpoint is ready, the MPI helper sends a message to its peer on an available node to restart the process, thus finishing the remote-fork. The same \texttt{DMTCP} checkpoints to the filesystem, which is only sharable over GigE. In future, we plan to checkpoint to memory, thus bypassing NFS.}
\end{figure}
Implementing Inter-Node Speculation

A node can be in one of three states: idle, speculating, or verifying. Figure 4 shows the state transition diagram. Additionally, the directory server maintains a record of which nodes (MPI ranks) are currently available for computing. It uses a scheme to keep track of the available nodes. Thus, the protocol effectively delays the speculation as long as possible, thus minimizing the bookkeeping overheads. The protocol also allows for multiple processes on each node, which would be useful to allow the main computation to proceed concurrently. The helper process will checkpoint and other bookkeeping can often be completely overlapped, thus minimizing the bookkeeping overheads. The protocol also allows for multiple processes on each node, which would be useful to allow the main computation to proceed concurrently. The helper process will checkpoint and other bookkeeping can often be completely overlapped, thus minimizing the bookkeeping overheads.

An important aspect of FastForward is its support of multi-level speculation. We believe that the ability to continue speculating without waiting for the last verification results is critical in obtaining efficiency in speculative parallelism, as Section 4 explains. For this purpose, FastForward implements a protocol to keep track of which speculative computing and computing done for verifying speculative computation—a node might be available on a high-speed cluster and leverage MPI optimizations for data transfer. Second, it solves a practical problem on batch allocated clusters by enabling controlled remote process creation through the MPI helper.

The comparison of results could be performed either on the speculator or the verifier. If the speculation is expected to succeed in most cases then it would be more useful to perform the comparison of results could be performed either on the speculator or the verifier. If the speculation is expected to succeed in most cases then it would be more useful to perform.

Figure 3 illustrates the progression of the protocol with an example. Node 1 speculates twice. For each speculation it forks a child process and sends a message to the idle node informing it of the node that it expects checkpoint and other communications to arrive from the requester, this allows us to use MPI's efficient optimization for message passing. If the speculator's child process fails, the speculator is notified. The speculator then contacts the directory server, which responds with the number of the idle node. The directory service keeps track of the available nodes. Thus, the protocol effectively delays the speculation as long as possible, thus minimizing the bookkeeping overheads. The protocol also allows for multiple processes on each node, which would be useful to allow the main computation to proceed concurrently. The helper process will checkpoint and other bookkeeping can often be completely overlapped, thus minimizing the bookkeeping overheads. The protocol also allows for multiple processes on each node, which would be useful to allow the main computation to proceed concurrently. The helper process will checkpoint and other bookkeeping can often be completely overlapped, thus minimizing the bookkeeping overheads.

The helper process will checkpoint and other bookkeeping can often be completely overlapped, thus minimizing the bookkeeping overheads. The protocol also allows for multiple processes on each node, which would be useful to allow the main computation to proceed concurrently. The helper process will checkpoint and other bookkeeping can often be completely overlapped, thus minimizing the bookkeeping overheads.
Analysis

\[ T = \text{time of execution of original program} \]
\[ p = \text{probability that speculation succeeds} \]
\[ k = \text{number of simultaneous speculations} \]
\[ s = \text{speedup of speculatively parallelized code over the original sequential code} \]
\[ S = \text{overall speedup of the program} \]

Running time of code, with speculation = \[ T + pk \frac{T}{s} + (1 - p)kT \]

Overall speedup, \[ S = \frac{T(k + 1)}{T + pk \frac{T}{s} + (1 - p)kT} = \frac{k + 1}{k + 1 + pk(\frac{1}{s} - 1)} \]

\[ S \leq k + 1 \text{ (strict upper bound, as } s \to \infty) \]
What next?
The Maze of Parallel Programming

- client-server vs p-to-p / master-slave vs symm.
- threads vs producer-consumer
- tightly vs loosely coupled
- shared mem vs msg passing
- data vs task parallel
- recursive vs iterative
- course vs fine grained
- SPMD vs MPMD
- parallel programs
Concluding Remarks

• Effectively programming modern computers requires leveraging parallelism at multiple levels

• There is no silver bullet of parallel programming (and there may never be)

• Tool (compiler developers, OS developers, architects) need to recognize the different needs of (parallel) programmers

• Parallel programming needs to become an integrated core of computer science education

• every future programmer is a parallel programmer
Questions?
http://www.cs.indiana.edu/~achauhan/
Google: arun indiana