Assigning Blame

Nick Rutar
Jeffrey K. Hollingsworth

University of Maryland
Motivation

- Parallel languages becoming more mature
- Parallel frameworks becoming more common
  - PETSc, Cactus, POOMA, GrACE
- Frameworks provide
  - High level abstractions for mathematics
    - Matrices, Vectors, (Non)Linear Systems, PDEs
  - Masking of low level parallel constructs
- More levels of abstraction complicates
  - Profiling
  - Debugging
Parallel Framework Mapping

- **Traditional profiling represented as**
  - Functions, Basic Blocks, Statement
- **Frameworks have intuitive abstractions**
  - Direct ties with mathematical terms
- **Map profiling information to variables**
  - Maps to abstractions in case of frameworks
  - Also can be used for standard programs
    - Map Structs, Classes, Arrays, Scalars
Example PETSC Program

* - $PETSC_DIR/src/ksp/ksp/examples/tutorials/ex23.c

```c
int main(int argc,char **args) {
    Vec x, /* approx solution */
    b, /* right hand side */
    u; /* exact solution*/
    Mat A; /* linear system matrix */
    KSP ksp; /* linear solver context */
    PC pc; /* preconditioner context */
    VecCreate(PETSC_COMM_WORLD,&x);
    VecDuplicate(x,&b);
    VecDuplicate(x,&u);
    MatCreate(PETSC_COMM_WORLD,&A);
    MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
    /* Set exact solution */
    VecSet(u,one);
    MatMult(A,u,b);
    /* Create linear solver context */
    KSPCreate(PETSC_COMM_WORLD,&ksp);
    KSPGetPC(ksp,&pc);
    PCSetType(pc,PCJACOBI);
    /* Solve linear system */
    ierr = KSPSolve(ksp,b,x); }
```
Variable “Blame”

- Record activity in a function
- Build association tree of writes from ground up
- Use transfer function to filter information up
  - Up the call stack
  - Aggregate over distributed nodes
- Eventually reach high level abstractions
  - Example: Matrix abstraction
    - Allocated storage for actual data
      - Sparse or Dense
    - Storage for bookkeeping
- Augments traditional profiling approaches
Blame Calculation Components

Static Analysis
- LLVM IR Generation
- Implicit/Explicit Data Flow Relationships

Runtime (Instances)
- Transfer Functions
  - Generation
  - Application
- Mem Containers
  - Stubs
  - Allocation/Free
- Container Resolution
  - Static
  - Dynamic
- PAPI Hardware Counter Sampling
- Dynamic Instrumentation
- StackWalker API

Variable Blame
Data Flow Relationships for Blame

- Two kinds of relationships
  - Explicit
    - Small sample snippet
      ```
      int a, b, c;
      a = 7;
      b = 8;
      c = a + b;
      - Blame goes to c
      ```
  - Implicit
    - Control Flow Operations
      - Loop indices, conditional statements
Calculating Data Flow Relationships

- **Use LLVM for intermediate representation**
  - Allows same approach for all supported languages
    - C, C++, Fortran

- **Calculate explicit blame**
  - Create dependency graph based on data flow
  - Focus on nodes with no incoming edges

- **Calculate implicit blame**
  - Generate control flow graph & dominator tree
  - Calculate basic blocks affected by control flow
Transfer Functions

- Establish “Exit Variables” for each function
  - Those variables that are live after function ends
    - Parameters
    - Return Value
    - Modified Global/Static Variables
    - Generated Heap Variables
- Create transfer function in terms of exit variables
- Special transfer functions
  - Source is not available
    - Series of Heuristics used to calculate blame
      - Return value with no params, all blame to return
  - Well defined APIs
    - math.h
      - Know that all blame for sqrt(double) goes to return
Mem-Containers

- Representation of memory operations
  - Stack and Heap based
- Represents unique contiguous memory region
- Mappings handled with container resolution
  - Ultimately map up to program variables
  - Mem-containers can map to other mem-containers
- Discovered through static/dynamic analysis
  - Static determines allocation points
    - Stubs created at these points
  - Dynamic happens for each instantiation
    - Full path of allocation calculated
Instance Generation (Sampling)

- Represents operations at each sample
- Can use different metrics for sampling
  - L2 Cache misses, floating point ops, cycles
- Mapped to either mem-container or variable
- Information recorded per thread and node
- Gather context (stack) at each sample
- Implementation
  - PAPI to generate samples
  - Stackwalk API for context sensitive information
Final Variable Blame

- *Given to user at various “blame points”*
- **Blame point can be**
  - “main” function
  - Where blame cannot be propagated up
    - Set of exit variables is null
    - Function with void params & void return
  - Any function deemed interesting by user
  - Any function that matches defined criteria
    - Contains a variable that has threshold of blame
Experimental Results

- Chose three programs with similar properties to those found in parallel frameworks
- Blame metric is number of cycles
- For each sampling point (instance)
  - Instance gets blamed for set number of cycles
  - Variable that instance maps up to gets blame
**FFP_SPARSE**

- C++ program that solves Poisson’s Equation
  - Approximately 6,700 lines of code & 63 Functions
- Non-parallel program
- Uses Sparse Matrices
  - No specific data structure for representation
  - Composite of primitive pointers declared in ‘main’
- Recorded 101 samples from program run
## FFP_SPARSE Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Direct</th>
<th>Blame (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_u</td>
<td>double *</td>
<td>Solution vector</td>
<td>0</td>
<td>35 (34.7)</td>
</tr>
<tr>
<td>a</td>
<td>double *</td>
<td>Coefficient matrix</td>
<td>0</td>
<td>24.5 (24.3)</td>
</tr>
<tr>
<td>ia</td>
<td>int *</td>
<td>Non-zero row indices of a</td>
<td>1</td>
<td>5 (5.0)</td>
</tr>
<tr>
<td>ja</td>
<td>int *</td>
<td>Non-zero column indices of a</td>
<td>1</td>
<td>5 (5.0)</td>
</tr>
<tr>
<td>element_neighbor</td>
<td>int *</td>
<td>Estimate of non-zeroes</td>
<td>0</td>
<td>10 (9.9)</td>
</tr>
<tr>
<td>node_boundary</td>
<td>bool *</td>
<td>Bool vector for boundary</td>
<td>0</td>
<td>9 (8.9)</td>
</tr>
<tr>
<td>f</td>
<td>double *</td>
<td>Right hand side of vector</td>
<td>0</td>
<td>3.5 (3.5)</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td>99</td>
<td>9 (8.9)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>101</td>
<td>101 (100)</td>
</tr>
</tbody>
</table>
QUAD_MPI

- C++ MPI program
  - Approximates multidimensional integral
  - Approximately 2000 lines of code & 18 functions
- Program interesting to look at handling MPI
- Ran on 4 Red Hat Linux nodes
  - OpenMPI 1.28
  - Range of 94-108 samples per node
## QUAD_MPI Results

### Blame (per Node)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>N1 (%)</th>
<th>N2 (%)</th>
<th>N3 (%)</th>
<th>N4 (%)</th>
<th>Total (%)</th>
<th>Dominant MPI Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim_num</td>
<td>int</td>
<td>27(27.2)</td>
<td>90(95.7)</td>
<td>97(84.3)</td>
<td>102(94.4)</td>
<td>316 (76.0)</td>
<td>MPI_Bcast</td>
</tr>
<tr>
<td>quad</td>
<td>double</td>
<td>19(19.2)</td>
<td>1(1.1)</td>
<td>5(4.3)</td>
<td>5(4.6)</td>
<td>30 (7.2)</td>
<td>MPI_Reduce</td>
</tr>
<tr>
<td>task_proc</td>
<td>int</td>
<td>15(15.2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>15 (3.6)</td>
<td>MPI_Send</td>
</tr>
<tr>
<td>w</td>
<td>double *</td>
<td>9(9.1)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9 (2.1)</td>
<td>-</td>
</tr>
<tr>
<td>point_num</td>
<td>int</td>
<td>-</td>
<td>1(1.1)</td>
<td>7(6.1)</td>
<td>-</td>
<td>8 (1.9)</td>
<td>MPI_Recv</td>
</tr>
<tr>
<td>xProc</td>
<td>double *</td>
<td>-</td>
<td>2(2.1)</td>
<td>5(4.3)</td>
<td>-</td>
<td>6 (1.4)</td>
<td>MPI_Recv</td>
</tr>
<tr>
<td>Other</td>
<td>-</td>
<td>3(3.0)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3 (0.7)</td>
<td>-</td>
</tr>
<tr>
<td>Output</td>
<td>-</td>
<td>6(6.1)</td>
<td>-</td>
<td>1(0.9)</td>
<td>1(0.9)</td>
<td>8 (1.9)</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>99(100)</td>
<td>94(100)</td>
<td>115(100)</td>
<td>108(100)</td>
<td>416 (100)</td>
<td>-</td>
</tr>
</tbody>
</table>
HPL (Linpack)

- **C program that solves a linear system**
  - Utilizes MPI and BLAS
  - Has wrappers for functions from both libraries
  - Operations done on dense matrices
  - Approximately 18,000 lines of code
  - 149 source files

- **32 Red Hat nodes connected via Myrinet**
  - OpenMPI 1.2.8
  - Range of 149-159 samples over the nodes
## HPL Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Mean (%</th>
<th>Node St. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Instances</td>
<td>-</td>
<td>154.7</td>
<td>(100)</td>
</tr>
<tr>
<td>mat</td>
<td>HPL_T_pmat</td>
<td>139.3</td>
<td>(90.0)</td>
</tr>
<tr>
<td>Anorm1</td>
<td>double</td>
<td>1.4</td>
<td>(0.9)</td>
</tr>
<tr>
<td>AnormI</td>
<td>double</td>
<td>1.1</td>
<td>(0.7)</td>
</tr>
<tr>
<td>XnormI</td>
<td>double</td>
<td>0.5</td>
<td>(0.3)</td>
</tr>
<tr>
<td>Xnorm1</td>
<td>double</td>
<td>0.2</td>
<td>(0.1)</td>
</tr>
<tr>
<td>A</td>
<td>HPL_T_pmat *</td>
<td>136.6</td>
<td>(88.3)</td>
</tr>
<tr>
<td>PANEL→L2</td>
<td>HPL_T_pmat</td>
<td>112.8</td>
<td>(72.9)</td>
</tr>
<tr>
<td>PANEL→A</td>
<td>double</td>
<td>12.8</td>
<td>(8.3)</td>
</tr>
<tr>
<td>PANEL→U</td>
<td>double</td>
<td>10.2</td>
<td>(6.6)</td>
</tr>
<tr>
<td>PANEL→HPL_pdtest</td>
<td>HPL_T_grid</td>
<td>2.2</td>
<td>(1.4)</td>
</tr>
<tr>
<td>PANEL→HPL_pdtest→HPL_pdgesv</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PANEL→HPL_pdtest→HPL_pdgesv→HPL_pdgesv0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Blame Points**

- main
  - grid: HPL_T_grid
  - HPL_T_pmat: 139.3 (90.0)
  - double: 1.4 (0.9)
  - double: 1.1 (0.7)
  - double: 0.5 (0.3)
  - double: 0.2 (0.1)
  - HPL_T_pmat *: 136.6 (88.3)

- main→HPL_pdtest
  - mat: HPL_T_pmat
  - double: 112.8 (72.9)
  - double: 12.8 (8.3)
  - double: 10.2 (6.6)

- main→HPL_pdtest→HPL_pdgesv
  - double: 10.2 (6.6)

- main→HPL_pdtest→HPL_pdgesv→HPL_pdgesv0

University of Maryland
Conclusion

● Variable “blame” mapping
  - Switch analysis from delimited regions to variables
  - Used to represent abstractions in parallel frameworks, standard programs as well
  - Application Programmer Centric Analysis

● Target applications are large and parallel
  - Many levels of abstraction
  - Data structures map to mathematical constructs

● Future work
  - Create corpus of shared library transfer functions
  - Create GUI for data presentation
  - Evaluate system on larger programs
Further Information

- **Europar 2009 paper**
  - [www.dyninst.org/papers/euroPar09.pdf](http://www.dyninst.org/papers/euroPar09.pdf)

- **SC’09 Demo**
  - Booth 2449 (Dyninst and Paradyn Projects)