Automated Performance Analysis and Tuning through IBM HPCST

Guojing Cong

October, 2008
Background and motivation

- Quite some performance tools but few are widely accepted (observation at Watson and Poughkeepsie)
  - Profiling {Graham et al 82}
  - Performance counters
  - MPI tracing
- Addresses generic and pervasive problems with performance tuning; to improve productivity
  - Increasingly complex architectures
  - New science and applications
  - New programming models and languages
- Current practice is manual and labor intensive
  - Compiler has yet to catch up especially for news apps on new HPC platforms
  - Impossible to always produce optimal or near optimal solution
  - Many aspects of computing are evolving fast (at least at this time)
  - Domain specific knowledge
- Tuning is still a daunting task with different characteristics for different dimensions
Current efforts on analysis and tuning

- Feedback directed compiler optimization
  - limited, closed, compiler centric
- “Traditional” performance tools
  - Focuses mostly on instrumentation, tracing, profiling and presenting data
  - Not much tuning
  - Few interactions with the compiler
- Auto-tuners for libraries (e.g., ATLAS, SPIRAL)
  - Well-defined problems, small kernels, with limited parameter space
- Machine-learning based approaches
  - Training
  - Model unclear
Thousands of words
Our approach

- Automate or provide facilities to help semi-automate analysis and tuning
- Provide an open framework to draw on the expertise of the community
- Address multi dimensions of performance problems
  - CPU, memory, communication, I/O, etc
- Closer integration with compiler
- Knowledge-centric approach
- Support for PGAS languages
Key Questions

- What is a bottleneck
- Where is the bottleneck (hotspots)
- What are the possible solutions to a bottleneck
- What are the parameters to the solutions
- What are the impact of the solutions
Bottleneck detection

- Rely on expert knowledge
- Components
  - Rule-based database, rules appear as logical expressions
  - Performance metrics are the basic components to construct a rule on
  - Modules are responsible to collect the metrics
  - Scheduler for scheduling the collection of metrics
    - Accuracy
    - Time
Solution determination and implementation

- Key components
  - Solution Database
  - Solution parameters discovery
  - Legality check
  - Performance Estimation
    - Detecting better solutions
  - Criteria
    - Improvement
    - Code impact
  - Implementation engine
Bottleneck rules

- OpenMPOverhead; 0 0 0 1; too much runtime overhead, not worth parallelizing the loop; \( \text{MAX(PercentOverhead)} > 50 \)
- Latesender (wrong order from different source) waiting time detected; 0 0 0 1 0; Wrong order situation due to messages received from different sources; mpi_lswo_different > 1.0
- NotVectorized; 1 0 0 0 0; loop not vectorized and vectorization could help; ndivides > 0 and \( \text{PM_CMPPLU_STALL_FPU} > (0.1 \times \text{PM_RUN_CYC}) \) and compiler_vectorized < 1
- DcacheMiss; 1 0 0 0 0; cycles wasted due to stalls (D-cache miss, ERAT, FXU, FPU stalls); \( (\text{PM_RUN_CYC} - (\text{PM_GCT_EMPTY_CYC} + \text{PM_GRP_CMPL})) > 0.1 \times \text{PM_RUN_CYC} \)

\[
\begin{align*}
\frac{\text{PM_CMPPLU_STALL_LSU}}{\text{PM_CYC}} > \alpha \quad &\& \\
\text{SA\_STRIDE\_ONE\_ACCESS\_RATE} \leq \beta \quad &\& \\
\text{SA\_REGULAR\_ACCESS\_RATE}(n) > \text{SA\_STRIDE\_ONE\_ACCESS\_RATE} + \gamma \\
\# \text{divides} > 0 \quad &\& \\
\frac{\text{PM\_STALL\_FPU}}{\text{PM\_RUN\_CYC}} > t \quad &\& \text{vectorized} = 0
\end{align*}
\]
Sample bottlenecks discovered with integrated external tools

- Latesender waiting time detected; Time a receiving process is waiting for a message;
- Barrier completion waiting time detected; Time needed to finish an MPI barrier
- Barrier waiting time detected; Waiting time in front of MPI barriers
- Latesender (wrong order from same source) waiting time detected; Wrong order situation due to messages received from the same source;
- Latesender (wrong order from different source) waiting time detected; Wrong order situation due to messages received from different sources
- Late Receiver waiting time detected; Time a sending process is waiting for the receiver to become ready
- N-to-N completion waiting time detected; Time needed to finish a n-to-n collective operation
- Wait before N-to-N detected; Time due to inherent synchronization in MPI n-to-n operations
- Late broadcast waiting time detected; Waiting time due to a late sender in MPI 1-to-n operations
- Earlyscan waiting time detected; Waiting time due to an early receiver in an MPI scan operation
- Early Reduce waiting time detected; Waiting time due to an early receiver in MPI n-to-1 operations
- MPI Init Exit; Time spent in MPI initialization calls
Solution rules

- Loop not unrolled optimally; unroll; my criteria; this field is reserved
- TlbMiss; large_page; using default criteria; large tlbmisses might be alleviated through large page usage
- *; compiler_msg; my criteria; get the compiler hints and messages
Solution scenarios

- Source code modification
- Compiler directives
- Polyhedral framework with compiler to generate better binary
- Environment variables
- Suggestions
Cases:

- Finding the right unroll factor
- Performance tuning for LBMHD
- Communication/computation overlap
Unroll-and-jam

- Possibly the most frequently used optimizations
- Compiler oftentimes does not give the best factor
- Runtime metrics help pinpoint the loops to focus on
- Exhaustive search of the unroll space
  - Model-guided
  - Register allocation
  - Impact on prefetching
  - Generate “instructions” to “execute” on the machines
  - Select the best factor
Performance estimation
do J = 1, 1 + (((((NP1)-1)/1/2)-1)*1*2, 1*2  
do I = 1, 1 + (((((MP1)-1)/1/2)-1)*1*2, 1*2  
  C  J[0][0]  
  PSI(I,J) = A*SIN((I-.5D0)*DI)*SIN((J-.5D0)*DJ)  
P(I,J) = PCF*(COS(2.D0*(I-1)*DI)  
  +COS(2.D0*(J-1)*DJ)) + 50000.D0  
50 CONTINUE  
.....
  C  J[0][1]  
  PSI(I,J+1) = A*SIN((I+.5D0)*DI)*SIN((J-.5D0)*DJ)  
P(I,J) = PCF*(COS(2.D0*(I-1)*DI)  
  +COS(2.D0*(J-1)*DJ)) + 50000.D0  
CONTINUE  
  C  J[1][0]  
  PSI(I,J+1) = A*SIN((I-.5D0)*DI)*SIN((J+1-.5D0)*DJ)  
P(I,J) = PCF*(COS(2.D0*(I-1)*DI)  
  +COS(2.D0*(J-1)*DJ)) + 50000.D0  
CONTINUE  
  C  J[1][1]  
  PSI(I,J+1) = A*SIN((I+.5D0)*DI)*SIN((J+1-.5D0)*DJ)  
P(I,J) = PCF*(COS(2.D0*(I-1)*DI)  
  +COS(2.D0*(J-1)*DJ)) + 50000.D0  
CONTINUE  
edo  
do J = 1 + (((((NP1)-1)/1/2)-1)*1*2, 1*2  
do I = 1, 1 + (((((MP1)-1)/1/2)-1)*1*2, 1*2  
  C  J[0][0]  
  PSI(I,J) = A*SIN((I-.5D0)*DI)*SIN((J-.5D0)*DJ)  
P(I,J) = PCF*(COS(2.D0*(I-1)*DI)  
  +COS(2.D0*(J-1)*DJ)) + 50000.D0  
CONTINUE  
  C  J[0][1]  
  PSI(I,J) = A*SIN((I-.5D0)*DI)*SIN((J+.5D0)*DJ)  
P(I,J) = PCF*(COS(2.D0*(I-1)*DI)  
  +COS(2.D0*(J-1)*DJ)) + 50000.D0  
CONTINUE  
  C  J[1][0]  
  PSI(I,J+1) = A*SIN((I-.5D0)*DI)*SIN((J+1-.5D0)*DJ)  
P(I,J) = PCF*(COS(2.D0*(I-1)*DI)  
  +COS(2.D0*(J-1)*DJ)) + 50000.D0  
CONTINUE  
edo
  C  J[1][1]  
  PSI(I,J+1) = A*SIN((I+.5D0)*DI)*SIN((J+1-.5D0)*DJ)  
P(I,J) = PCF*(COS(2.D0*(I-1)*DI)  
  +COS(2.D0*(J-1)*DJ)) + 50000.D0  
CONTINUE
enddo
LBMHD

- Plasma Physics
  Lattice Boltzmann approach for magneto-hydrodynamics
- Hot spot analysis results: Two functions consume 99.0% of time.
  - COLLISION: 59.602 sec (67.0%)
  - STREAM: 28.366 sec (32.0%)
  - Others: 0.896 sec (1.0%)

- Optimization steps
  **STEP1:** LARGEPAGES (16 M page)
  - Allocate data variables (f, feq, g, geq) on the data section (Originally allocated on the BSS section).
  **STEP2:** TRANSPOSE_A or TRANSPOSE_B (for COLLISION optimization)
  - Transpose index order of data variables (e.g. from g(i, j, k, l) to g(l, k, i, j)).
  - Interleave (f and feq) and (g and geq) into two arrays using "Type" (Originally four independent variables) (TRANSPOSE_B only)
  **STEP3:** LOOPINTERCHANGE (for STREAM optimization)
  - Change loop order of the STREAM function to mitigate effects of TRANSPOSE_A and TRANSPOSE_B.
**Observations from the source code**

- Memory access pattern is not stride-1.

```fortran
  do j = jsta, jend
    do i = ista, iend
      ...do k = 1, 4
        Bt2 = Bt2 + g(i, j, k, 1) * g(i, j, k+4, 2) ...
      enddo
    enddo
  enddo
```

- Loop nesting order cannot be changed because of dependency.

```fortran
  do k = 1, 4
    vt1 = vt1 + c(k,1)*f(i,j,k) + ...
  enddo
```

- Two pairs of arrays (f/feq and g/geq) are accessed with the same pattern.

**Optimization (Either TRANSPOSE_A or TRANSPOSE_B to be applied)**

- **TRANSPOSE_A:** Make memory access pattern stride-1 access by transposing dimension orders as follows:
  
  **Declaration Part** REAL, DIMENSION (-1:DX+2,-1:DY+2,9,2) :: g, geq  
  
  **Reference Part** g(i,j,k,1), g(i,j,k,2), g(i,j,k,3), g(i,j,k,4), g(i,j,k,5), g(i,j,k,6), g(i,j,k,7), g(i,j,k,8), g(i,j,k,9)  
  
- **TRANSPOSE_B:** TRANSPOSE_A and interleave two arrays (f/feq and g/geq) into one array using Fortran90’s Type as follows:
  
  **Declaration Part** REAL, DIMENSION (-1:DX+2,-1:DY+2) :: g, geq  
  
  **Reference Part** g(1,k,i,j), g(k,1,i,j), g(k,2,i,j), g(k,3,i,j), g(k,4,i,j), g(k,5,i,j), g(k,6,i,j), g(k,7,i,j), g(k,8,i,j), g(k,9,i,j)  

**Description of STEP2 TRANSPOSE_A or _B (for COLLISION optimization)**

- Loop nesting order: k, i, j  
  
  Subscripts order: i, j, k

- "k" should be innermost to calculate the total (vt1)

- NOT MATCH

**Subroutine COLLISION**

```fortran
SUBROUTINE COLLISION
  USE mhd_vars
  IMPLICIT NONE
  REAL vt1, vt2, Bt1, Bt2, rho, rhoinv
  REAL vdotck(8), bdotck(8)
  do j = jsta, jend
    do i = ista, iend
      vt1 = 0    !tempv1 = 0
      vt2 = 0    !tempv2 = 0
      Bt1 = 0    !tempB1 = 0
      Bt2 = 0    !tempB2 = 0
      rho = rho(i,j)
      rhoinv = 1/rho
      do k = 1, 4
        vt1 = vt1 + c(k,1)*f(i,j,k) + c(k+4,1)*f(i,j,k+4)  
        vt2 = vt2 + c(k,2)*f(i,j,k) + c(k+4,2)*f(i,j,k+4)  
        Bt1 = Bt1 + g(i,j,k,1) + g(i,j,k+4,1)  
        Bt2 = Bt2 + g(i,j,k,2) + g(i,j,k+4,2)  
      enddo
      vt1 = vt1*rhoinv  
      vt2 = vt2*rhoinv  
      Bt1 = Bt1 + geq(i,j,9,1)  
      Bt2 = Bt2 + geq(i,j,9,2)  
      v2 = vt1**2 + vt2**2  
      B2 = Bt1**2 + Bt2**2  
      ! Used below in odd feq  
      temp1= .25*trho*cs2+ .125*(-trho*v2 + 3.0*B2)
      do k = 1, 8
        vdotc = c(k,1)*vt1 + c(k,2)*vt2
        Bdotc = c(k,1)*Bt1 + c(k,2)*Bt2
        feq(i,j,k) = vfac*f(i,j,k)+vtauinv*(temp1+trho*.25*vdotc+ &  
                                      .5*(trho*vdotc**2- Bdotc**2))  
        geq(i,j,k,1) = Bfac*g(i,j,k,1)+ Btauinv*.125*(theta*Bt1+ &  
                                      2.0*Bt1*vdotc- 2.0*vt1*Bdotc)  
        geq(i,j,k,2) = Bfac*g(i,j,k,2)+ Btauinv*.125*(theta*Bt2+ &  
                                      2.0*Bt2*vdotc- 2.0*vt2*Bdotc)
      enddo
    enddo
  enddo
END SUBROUTINE COLLISION
```
Description of STEP3: LOOPINTERCHANGE (for STREAM optimization)

- Observations from the source code
  - Assuming TRANSPOSE_A or B applied, memory access pattern is not stride-1.
    - Loop nesting order: i, j, k
    - Subscripts order: k, i, j

- Optimization
  - LOOPINTERCHANGE: Change the loop nesting order to fit with the new dimension order modified by TRANSPOSE_A (or B)
    - Loop nesting order: i, j, k
    - Subscripts order: k, i, j

Observations from the source code
- Assuming TRANSPOSE_A or B applied, memory access pattern is not stride-1.

Optimization
- LOOPINTERCHANGE: Change the loop nesting order to fit with the new dimension order modified by TRANSPOSE_A (or B)
Bottleneck detection

\[ \frac{PM\_CMPLU\_STALL\_LSU}{PM\_CYC} > \alpha \land\land \]
\[ SA\_STRIDE\_ONE\_ACCESS\_RATE \leq \beta \land\land \]
\[ SA\_REGULAR\_ACCESS\_RATE(n) > SA\_STRIDE\_ONE\_ACCESS\_RATE + \gamma \]
Using directives as solutions

```
REAL, DIMENSION(-1:DX+2,-1:DY+2,9), TARGET :: f, feq
REAL, DIMENSION(-1:DX+2,-1:DY+2,9,2), TARGET :: g, geq

Subroutine stream
  ...
  do k = 1, 2
  do j = jsta, jend
  do i = ista, iend
    g(i,j,1,k) = geq(i-1,j,1,k)
    g(i,j,3,k) = geq(i,j-1,3,k)
    g(i,j,5,k) = geq(i+1,j,5,k)
    g(i,j,7,k) = geq(i,j+1,7,k)
  enddo
  Enddo
  ...
  Subroutine collision
  ...
  do i = ista, iend
  do j = jsta, jend
  do k = 1, 8
    feq(i,j,k) = vfac*f(i,j,k) + vtauinv*(temp1 + trho*0.25*vdotc + 0.5*(trho*vdotc**2 - Bdotc**2))
    geq(i,j,k,1) = Bfac*g(i,j,k,1) + Btauinv*1.25*(theta*Bt1 + 2.0*Bt1*vdotc - 2.0*vt1*Bdotc)
    geq(i,j,k,2) = Bfac*g(i,j,k,2) + Btauinv*1.25*(theta*Bt2 + 2.0*Bt2*vdotc - 2.0*vt2*Bdotc)
  enddo
```

```
Communication/Computation Overlapping
Two Optimization Patterns by Overlapping Communication

Policy: Optimize Behavior in each MPI Process without Changing its Communication Behavior

- **Pattern A**: Overlap Multiple Independent Communication Phases
  
  ![Diagram of Pattern A]

- **Pattern B**: Overlap Independent Communication and Calculation Phases
  
  ![Diagram of Pattern B]
**Sample Program including Two Overlap Patterns**

### Function

Calculate $X_{t,n}(k) =$ “value of $k$-th data in node $n$ at step $t$” for each node $n$ and step $t$

$$s = \sum_{k=1}^{X_{t,n}} (k) * \sin(X_{t,n}(k) * \theta), \text{ and } r = s / (1 + s)$$

$$X_{t+1,n}(k) = X_{t,n}(k) * 0.5 * r + (X_{t,n-1}(k) + X_{t,n+1}(k)) * 0.25 * (1 - r)$$

### Algorithm

1. **Step 1:** Receive $X_{t,n-1}$
2. **Step 2:** Receive $X_{t,n+1}$
3. **Step 3:** Calculate $r$
4. **Step 4:** Calculate $X_{t+1,n}$

### Data Flow

- **Node $n-1$:**
  - $x: X_{t,n-1}$
  - $x_{next}: X_{t,n}$

- **Node $n$:**
  - $x: X_{t,n}$
  - $x_{prev}: X_{t,n-1}$
  - $x_{next}: X_{t,n+1}$

- **Node $n+1$:**
  - $x_{prev}: X_{t,n}$
  - $x: X_{t,n+1}$

- $X_{t,n}(k) * 0.5 * r$
- $X_{t,n-1}(k) * 0.25 * (1 - r)$
- $X_{t,n+1}(k) * 0.25 * (1 - r)$
do t = 1, m
  call MPI_SENDRECV(x, n, MPI_DOUBLE_PRECISION, next, 0,
                 xprev, n, MPI_DOUBLE_PRECISION, prev, 0,
                 MPI_COMM_WORLD, status, ierr)
  if (ierr .ne. 0) stop 'ERROR IN MPI_SENDRECV (next/prev)'
  call MPI_SENDRECV(x, n, MPI_DOUBLE_PRECISION, prev, 0,
                 xnext, n, MPI_DOUBLE_PRECISION, next, 0,
                 MPI_COMM_WORLD, status, ierr)
  if (ierr .ne. 0) stop 'ERROR IN MPI_SENDRECV (prev/next)'
  s = 0.0
  do i = 1, n
    s = s + x(i) * sin(x(i) * pi)
  enddo
  r = s / (1 + s)
  do i = 1, n
    x(i) = x(i) * 0.5 * r +
               (xprev(i) + xnext(i)) * 0.25 * (1 - r)
  check = check + x(i)
  enddo
enddo

Step 1: Receive $X_{t,n-1}$
Step 2: Receive $X_{t,n+1}$
Step 3: Calculate $r$
Step 4: Calculate $X_{t+1,n}$
Source Code of Optimization 1

do t = 1, m
   call MPI_IRECV(xprev, n, MPI_DOUBLE_PRECISION, prev, 0, MPI_COMM_WORLD, rreq1, ierr)
   if (ierr .ne. 0) stop 'ERROR IN MPI_IRECV (prev)'
   call MPI_IRECV(xnext, n, MPI_DOUBLE_PRECISION, next, 0, MPI_COMM_WORLD, rreq2, ierr)
   if (ierr .ne. 0) stop 'ERROR IN MPI_IRECV (next)'
   call MPI_ISEND(x, n, MPI_DOUBLE_PRECISION, next, 0, MPI_COMM_WORLD, sreq1, ierr)
   if (ierr .ne. 0) stop 'ERROR IN MPI_ISEND (next)'
   call MPI_ISEND(x, n, MPI_DOUBLE_PRECISION, prev, 0, MPI_COMM_WORLD, sreq2, ierr)
   if (ierr .ne. 0) stop 'ERROR IN MPI_ISEND (prev)'
   call MPI_BARRIER(MPI_COMM_WORLD, ierr)
   if (ierr .ne. 0) stop 'ERROR IN MPI_BARRIER'
   call MPI_WAIT(sreq1, status, ierr)
   if (ierr .ne. 0) stop 'ERROR IN MPI_WAIT (SEND prev)'
   call MPI_WAIT(sreq2, status, ierr)
   if (ierr .ne. 0) stop 'ERROR IN MPI_WAIT (SEND next)'
   call MPI_WAIT(rreq1, status, ierr)
   if (ierr .ne. 0) stop 'ERROR IN MPI_WAIT (RECV prev)'
   call MPI_WAIT(rreq2, status, ierr)
   if (ierr .ne. 0) stop 'ERROR IN MPI_WAIT (RECV next)'

   s = 0.0
   do i = 1, n
      s = s + x(i) * sin(x(i) * pi)
   enddo
   r = s / (1 + s)
   do i = 1, n
      x(i) = x(i) * 0.5 * r +
              (xprev(i) + xnext(i)) * 0.25 * (1 - r)
      check = check + x(i)
   enddo
enddo

Execution

Step 1: Receive $X_{t,n-1}$

Step 2: Receive $X_{t,n+1}$

Step 1 and 2 are Overlapped

Step 3: Calculate $r$

Step 4: Calculate $X_{t+1,n}$
do $t = 1, m$
  call MPI_IRECV(xprev, n, MPI_DOUBLE_PRECISION, prev, 0, MPI_COMM_WORLD, rreq1, ierr)
  if (ierr .ne. 0) stop 'ERROR IN MPI_IRECV (prev)'
  call MPI_IRECV(xnext, n, MPI_DOUBLE_PRECISION, next, 0, MPI_COMM_WORLD, rreq2, ierr)
  if (ierr .ne. 0) stop 'ERROR IN MPI_IRECV (next)'
  call MPI_ISEND(x, n, MPI_DOUBLE_PRECISION, next, 0, MPI_COMM_WORLD, sreq1, ierr)
  if (ierr .ne. 0) stop 'ERROR IN MPI_ISEND (next)'
  call MPI_ISEND(x, n, MPI_DOUBLE_PRECISION, prev, 0, MPI_COMM_WORLD, sreq2, ierr)
  if (ierr .ne. 0) stop 'ERROR IN MPI_ISEND (prev)'
  call MPI_BARRIER(MPI_COMM_WORLD, ierr)
  if (ierr .ne. 0) stop 'ERROR IN MPI_MPI_BARRIER'
  s = 0.0
  do $i = 1, n$
    s = s + x(i) * sin(x(i) * pi)
  enddo
  r = s / (1 + s)
  call MPI_WAIT(sreq1, status, ierr)
  if (ierr .ne. 0) stop 'ERROR IN MPI_WAIT (SEND prev)'
  call MPI_WAIT(sreq2, status, ierr)
  if (ierr .ne. 0) stop 'ERROR IN MPI_WAIT (SEND next)'
  call MPI_WAIT(rreq1, status, ierr)
  if (ierr .ne. 0) stop 'ERROR IN MPI_WAIT (RECV prev)'
  call MPI_WAIT(rreq2, status, ierr)
  if (ierr .ne. 0) stop 'ERROR IN MPI_WAIT (RECV next)'
  do $i = 1, n$
    x(i) = x(i) * 0.5 * r + (xprev(i) + xnext(i)) * 0.25 * (1 - r)
    check = check + x(i)
  enddo
enddo
Step 1: Receive $X_{t,n-1}$

Performance Results (Blue Gene/P 32 node, SMP mode, 1 proc/node)

### Original Program

- **Step 1:** Receive $X_{t,n-1}$
- **Step 2:** Receive $X_{t+1}$
- **Step 3:** Calculate $r$

55.4 sec 102.5 sec 14.1 sec

177.3 sec

### Optimization 1: Parallelize Step 1 and Step 2 (Com1 and Com2)

- **Step 1:** Receive $X_{t,n-1}$
- **Step 2:** Receive $X_{t+1}$
- **Step 3:** Calculate $r$

43.0 sec 102.5 sec 14.1 sec -15.7 sec

161.6 sec

### Optimization 2: Parallelize Step 1, Step 2 and Step 3 (Com1, Com2 and Cal1)

- **Step 1:** Receive $X_{t,n-1}$
- **Step 2:** Receive $X_{t+1}$
- **Step 3:** Calculate $r$

37.0 sec 102.5 sec 14.1 sec -21.0 sec

156.3 sec

Wait

0.4 sec
Solution Implementation

Forward AFTER LINE 48:ST_ASSIGNMENT 2 [
  0:VAR(r),
  1:OP("/" 2[VAR(s),OP("+" 2[CONST(INT#1),VAR(s)]])),
]

Wait: InsertAfter [48:9:48:74] [
  call MPI_WAIT(NEW0_1, status, ierr)
  call MPI_WAIT(NEW0_2, status, ierr)
]

--------

Backward BEFORE LINE 36:ST_CALL 14 [
]
Forward AFTER LINE 48:ST_ASSIGNMENT 2 [
  0:VAR(r),
  1:OP("/" 2[VAR(s),OP("+" 2[CONST(INT#1),VAR(s)]])),
]

Wait: InsertAfter [48:9:48:74] [
  call MPI_WAIT(NEW1_1, status, ierr)
  call MPI_WAIT(NEW1_2, status, ierr)
]
Polyhedral for loop tiling

- No source code change
- After bottleneck analysis, in solution determination, hpcst automatically generates a script file that is fed back to the compiler
- Script provides command for optimization for the compiler to apply on the intermediate representation (W-code)
- Potential of composing transformations
- Can be used to implement many standard transformations

```
# Compute dependences syntax:
# PT_depcompute(bool RAR, bool ADA, bool summaryAll, bool summaryRAW)
#
PT_depcompute(1,1,0,0)
dumpprog erhs_initial

# Tile the i and j loops only
#PT_tile((0,0),1,2,55,55)
dumpcode lud_afterTiling

# Fuse loops together
#PT_fusion ((0,0),(1,0));
dumpcode main_afterFusion
```
Support for UPC

- Bottlenecks for UPC applications based on UPC profiling and tracing library
  - Load balancing issues
- Suggestions for removing bottlenecks
  - Communication aggregation