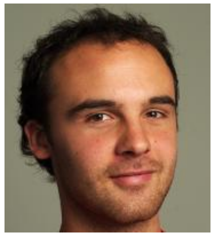
TORSTEN HOEFLER
DEPARTMENT OF COMPUTER SCIENCE, ETH ZÜRICH

A case for runtime recompilation in HPC (or: MPI+X; X=LLVM)

Setting the stage

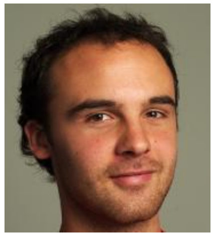
- **We use LLVM, but not like you may think!**
- **Runtime Recompilation and Specialization**
  - MPI optimizations [EuroMPI'13, LCPC'13]
- **Automatic Performance Model Generation**
  - Static and dynamic modeling [SPAA'14, PACT'14]
- **Compilation for Heterogeneous Systems**
  - Focused around Polyhedral techniques
- **We only compile test-applications!**
  - Mainly deal with IR and internal issues

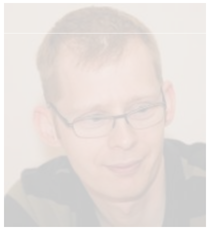


Platform for Advanced Scientific Computing

Topics Today (ask anything!)

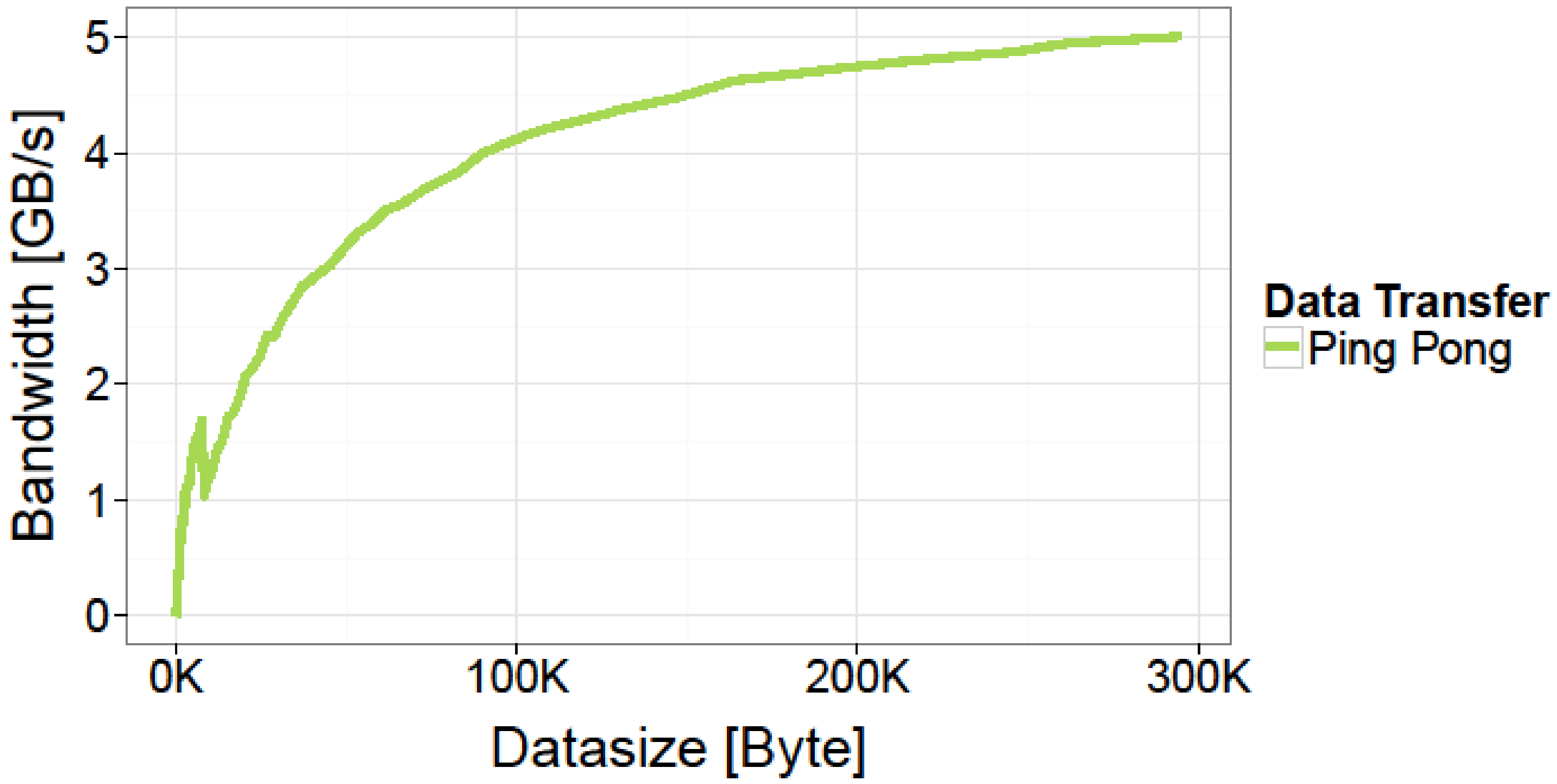
- We use LLVM, but not like you may think!
- **Runtime Recompilation and Specialization**
  - MPI optimizations [EuroMPI'13, LCPC'13]
- **Automatic Performance Model Generation**
  - Static and dynamic modeling [SPAA'14, PACT'14]
- Compilation for Heterogeneous Systems
  - Focused around Polyhedral techniques
- We only compile test-applications!
  - Mainly deal with IR and internal issues



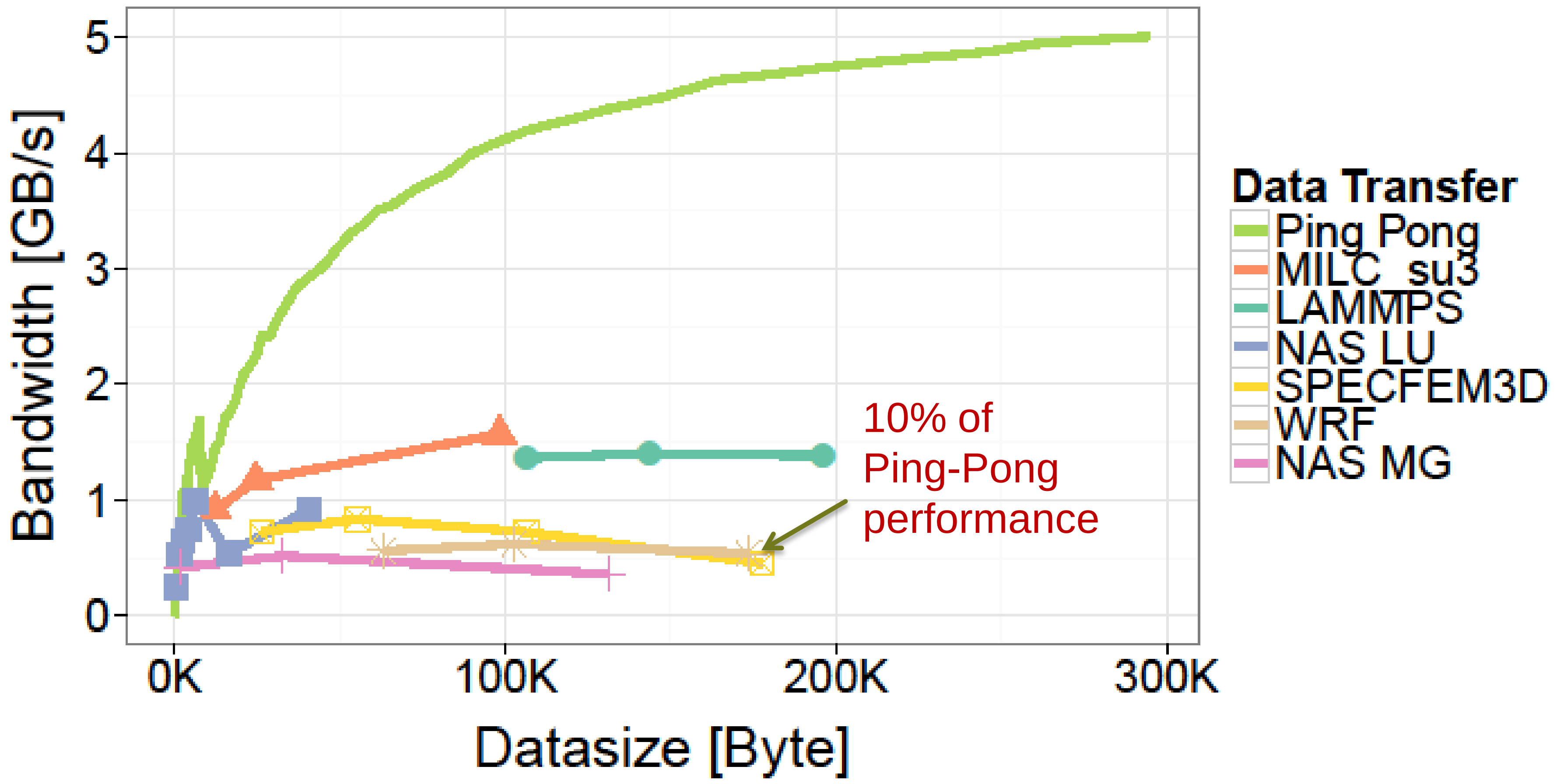


Platform for Advanced Scientific Computing

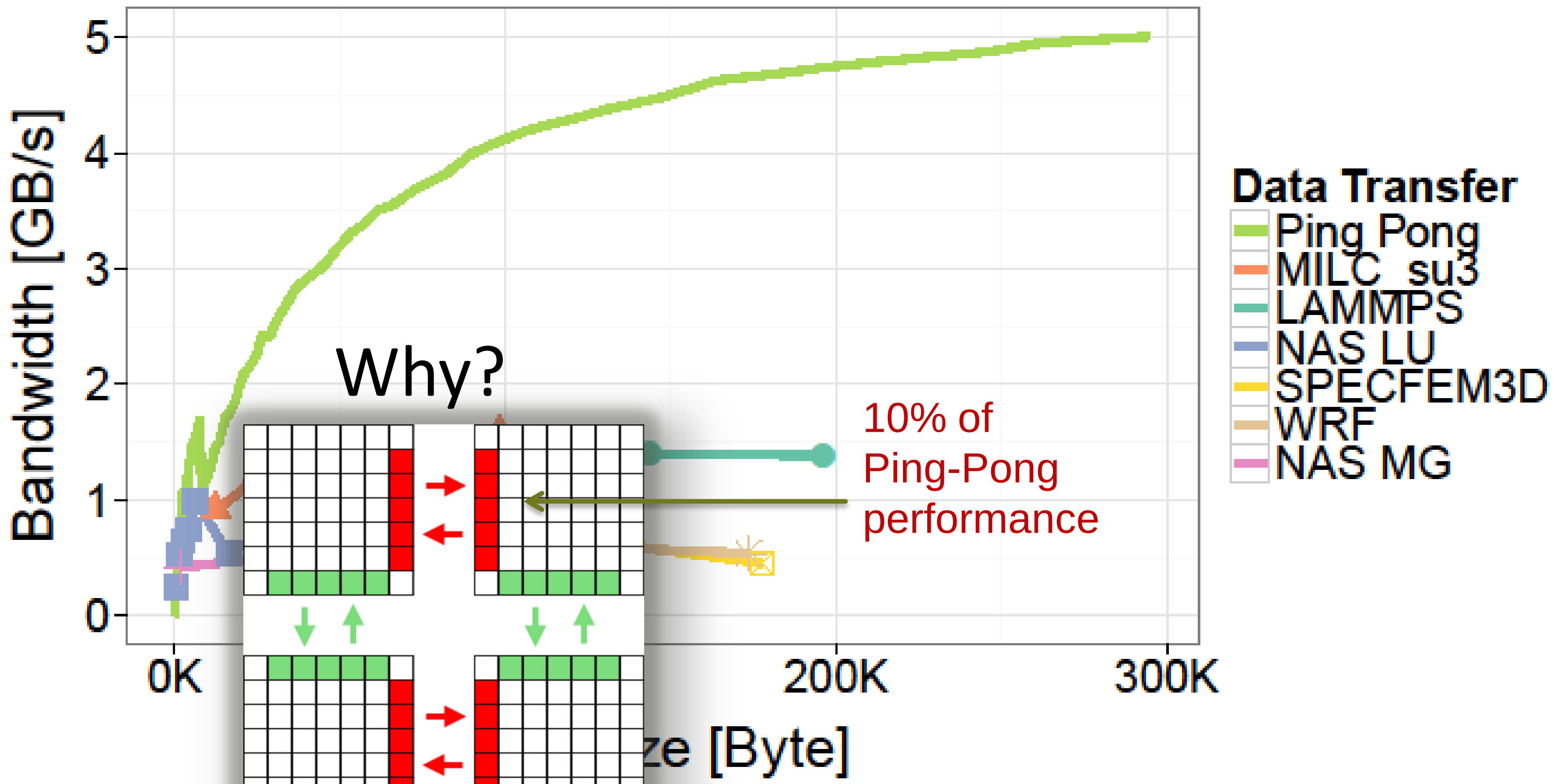
What your vendor sold you



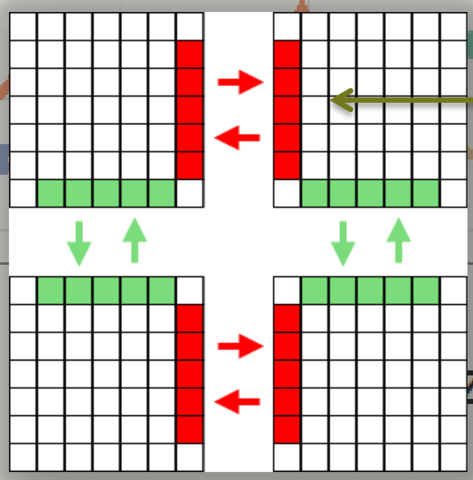
What your Applications get



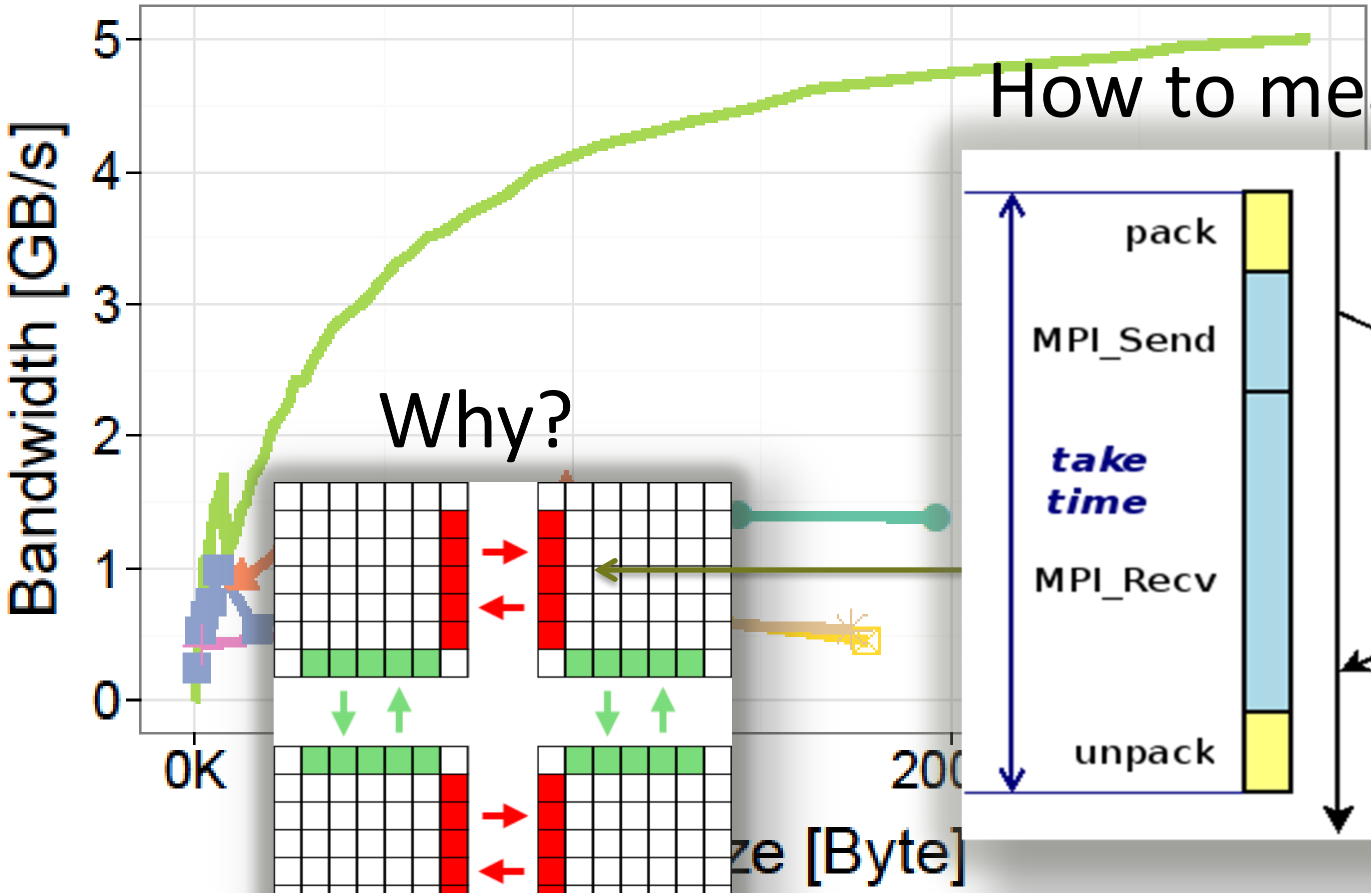
What your Applications get



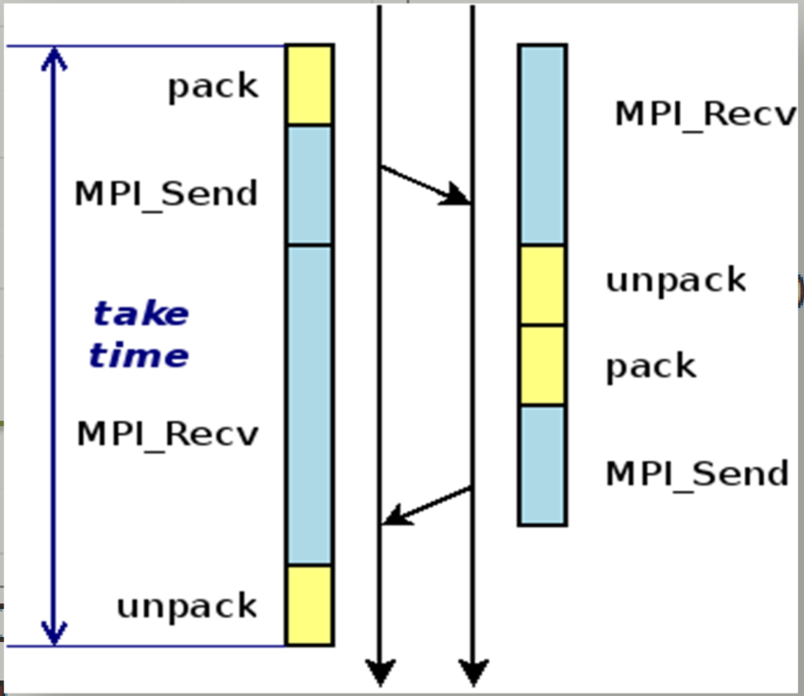
Why?



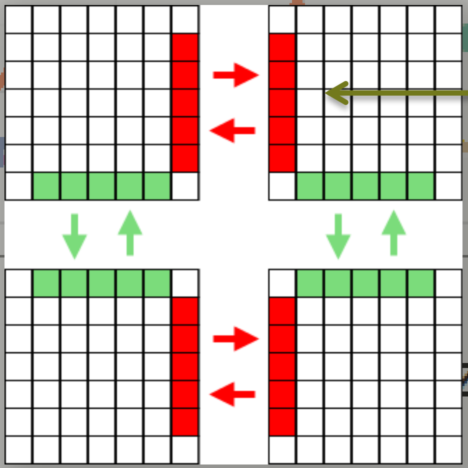
What your Applications get



How to measure?



Why?



What MPI offers

Manual packing

```
sbuf = malloc(N*sizeof(double))
rbuf = malloc(N*sizeof(double))
for (i=1; i<N-1; ++i)
   sbuf[i]=data[i*N+N-1]
MPI_Isend(sbuf, …)
MPI_Irecv(rbuf, …)
MPI_Waitall(…)
for (i=1; i<N-1; ++i)
   data[i*N]=rbuf[i]
free(sbuf)
free(rbuf)
```

MPI Datatypes

```
MPI_Datatype nt
MPI_Type_vector(N-2, 1, N,
                         MPI_DOUBLE, &nt)
MPI_Type_commit(&nt)
MPI_Isend(&data[N+N-1], 1, nt, …)
MPI_Irecv(&data[N], 1, nt, …)
MPI_Waitall(…)
MPI_Type_free(&nt)
```

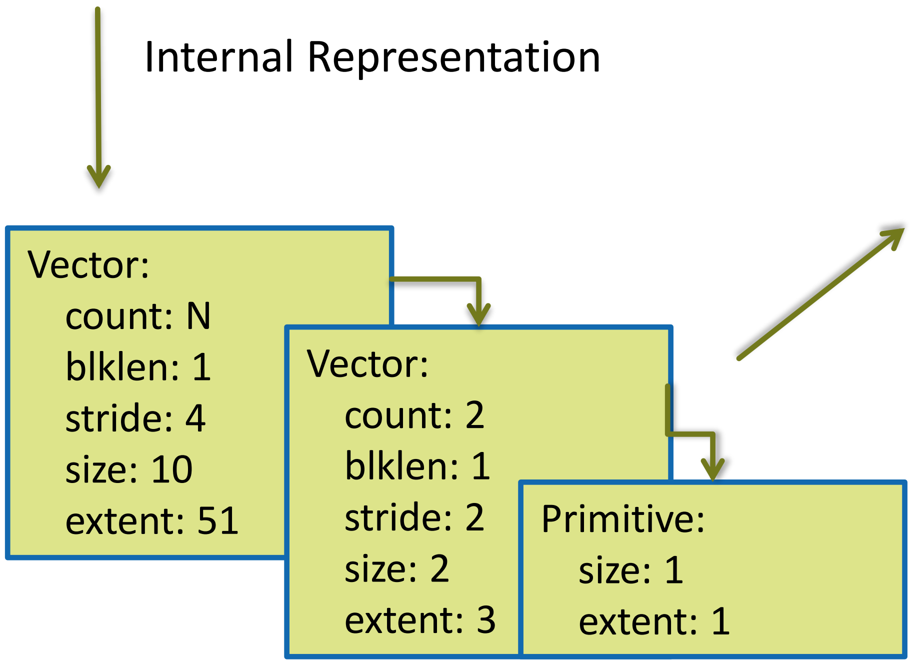
- No explicit copying
- Less code
- Often slower than manual packing (see [1])

[1] Schneider, Gerstenberger, TH: *Micro-Applications for Communication Data Access Patterns and MPI Datatypes*, EuroMPI’12

Interpretation vs. Compilation

- **MPI DDTs are interpreted at runtime, while manual pack loops are compiled**

```
bt = Vector(2, 1, 2, MPI_BYTE)
nt =Vector(N, 1, 4, bt)
```

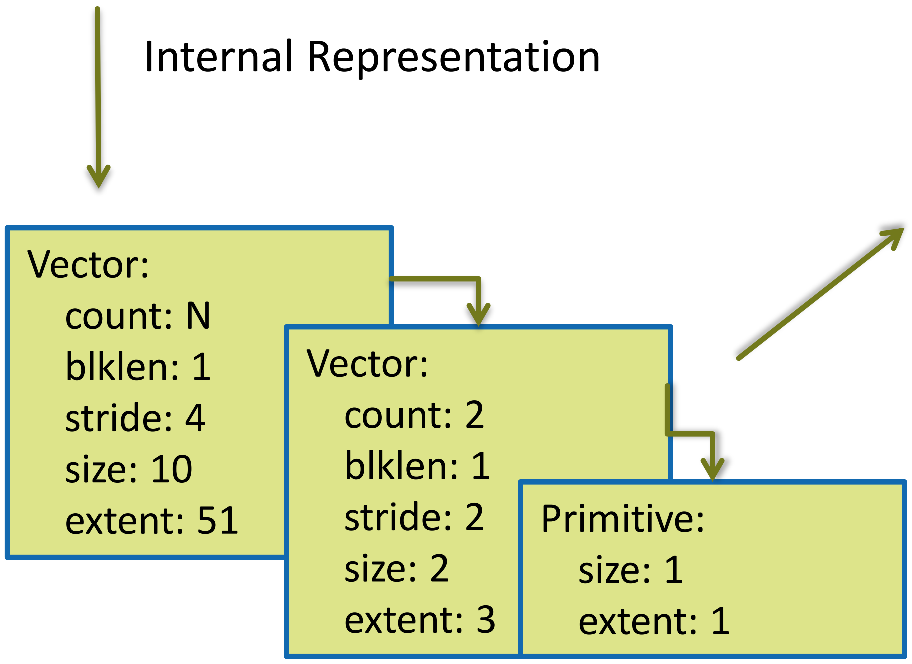


```
If (dt.type == VECTOR)
  for (int i=0; i<dt.count; i++) {
    tin = inbuf; tout=outbuf
    for (b=0; b<dt.blklen; d++) {
      interpret(dt.basetype, tin, tout)
    }
    tin +=  dt.stride * dt.base.extent
    tout = dt.blklen * dt.base.size
  }
  inbuf += dt.extent
  outbuf += dt.size
}
```

Interpretation vs. Compilation

- **MPI DDTs are interpreted at runtime, while manual pack loops are compiled**

```
bt = Vector(2, 1, 2, MPI_BYTE)
nt =Vector(N, 1, 4, bt)
```



```
If (dt.type == VECTOR)
  for (int i=0; i<dt.count; i++) {
    tin = inbuf; tout=outbuf;
    for (b=0; b<dt.blklen; d++) {
      interpret(dt.basetype, tin, tout)
    }
    tin += dt.stride * dt.base.extent
    tout = dt.blklen * dt.base.size
  }
  inbuf += dt.extent
  outbuf += dt.size
}
```

- None of these variables are known when this code is compiled
- Many nested loops and branches

Interpretation vs. Compilation

- **MPI DDTs are interpreted at runtime, while manual pack loops are compiled**

```
for (int i=0; i<N; ++i) {
   for(j=0;  j<2; ++j) {
      outbuf[j] = inbuf[j*2]
   }
   inbuf += 3*4
   outbuf  += 2
}
```

Interpretation vs. Compilation

- **MPI DDTs are interpreted at runtime, while manual pack loops are compiled**

```
for (int i=0; i<N; ++i) {
    for(j=0;  j<2; ++j) {
        outbuf[j] = inbuf[j*2]
    }
    inbuf += 3*4
    outbuf  += 2
}
```

- Loop unrolling

Interpretation vs. Compilation

- **MPI DDTs are interpreted at runtime, while manual pack loops are compiled**

```
for (int i=0; i<N; ++i) {
   int j = 0
   outbuf[j] = inbuf[j*2]
   outbuf[j+1] = inbuf[(j+1)*2]
   inbuf += 3*4
   outbuf  += 2
}
```

- Loop unrolling
- Constant Propagation

Interpretation vs. Compilation

- **MPI DDTs are interpreted at runtime, while manual pack loops are compiled**

```
for (int i=0; i<N; ++i) {
  outbuf[0] = inbuf[0]
  outbuf[1] = inbuf[2]
  inbuf += 12
  outbuf  += 2
}
```

- Loop unrolling
- Constant Propagation
- Strength reduction

Interpretation vs. Compilation

- **MPI DDTs are interpreted at runtime, while manual pack loops are compiled**

```
bound = outbuf + 2*N
while (outbuf<bound) {
   outbuf[0] = inbuf[0]
   outbuf[1] = inbuf[2]
   inbuf += 12
   outbuf  += 2
}
```

- Loop unrolling
- Constant Propagation
- Strength reduction

Interpretation vs. Compilation

- **MPI DDTs are interpreted at runtime, while manual pack loops are compiled**

```
bound = (outbuf + 2*N)/2
while (outbuf<bound) {
   outbuf[0] = inbuf[0]
   outbuf[1] = inbuf[2]
   outbuf[2] = inbuf[4]
   outbuf[3] = inbuf[6]
   inbuf += 24
   outbuf  += 4
}
…
```

- Loop unrolling
- Constant Propagation
- Strength reduction
- Unrolling of outer loop

Interpretation vs. Compilation

- **MPI DDTs are interpreted at runtime, while manual pack loops are compiled**

```
bound = (outbuf + 2*N)/2
while (outbuf<bound) {
   outbuf[0] = inbuf[0]
   outbuf[1] = inbuf[2]
   outbuf[2] = inbuf[4]
   outbuf[3] = inbuf[6]
   inbuf += 24
   outbuf  += 4
}
…
```

- Loop unrolling
- Constant Propagation
- Strength reduction
- Unrolling of outer loop
- SIMDization

Interpretation vs. Compilation

- **MPI DDTs are interpreted at runtime, while manual pack loops are compiled**

  - Loop unrolling
  - Constant Propagation
  - Strength reduction
  - Unrolling of outer loop
  - SIMDization

```
for (int i=0; i<N; ++i) {
   for(j=0;  j<2; ++j) {
      outbuf[j] = inbuf[j*2]
   }
   inbuf += 3*4
   outbuf  += 2
}
```

-O3

```
bound = (outbuf + 2*N)/2
while (outbuf<bound) {
   outbuf[0] = inbuf[0]
   outbuf[1] = inbuf[2]
   outbuf[2] = inbuf[4]
   outbuf[3] = inbuf[6]
   inbuf += 24
   outbuf  += 4
}
…
```

Runtime-Compiled pack functions

Declare

MPI_Type_vector(cnt, blklen, …)

Record arguments in internal representation (Tree of C++ objects)

Optimize

MPI_Type_commit(new_ddt)

Generate pack(*in, cnt, *out) function using LLVM IR. Compile to machine code. Store f-pointer.

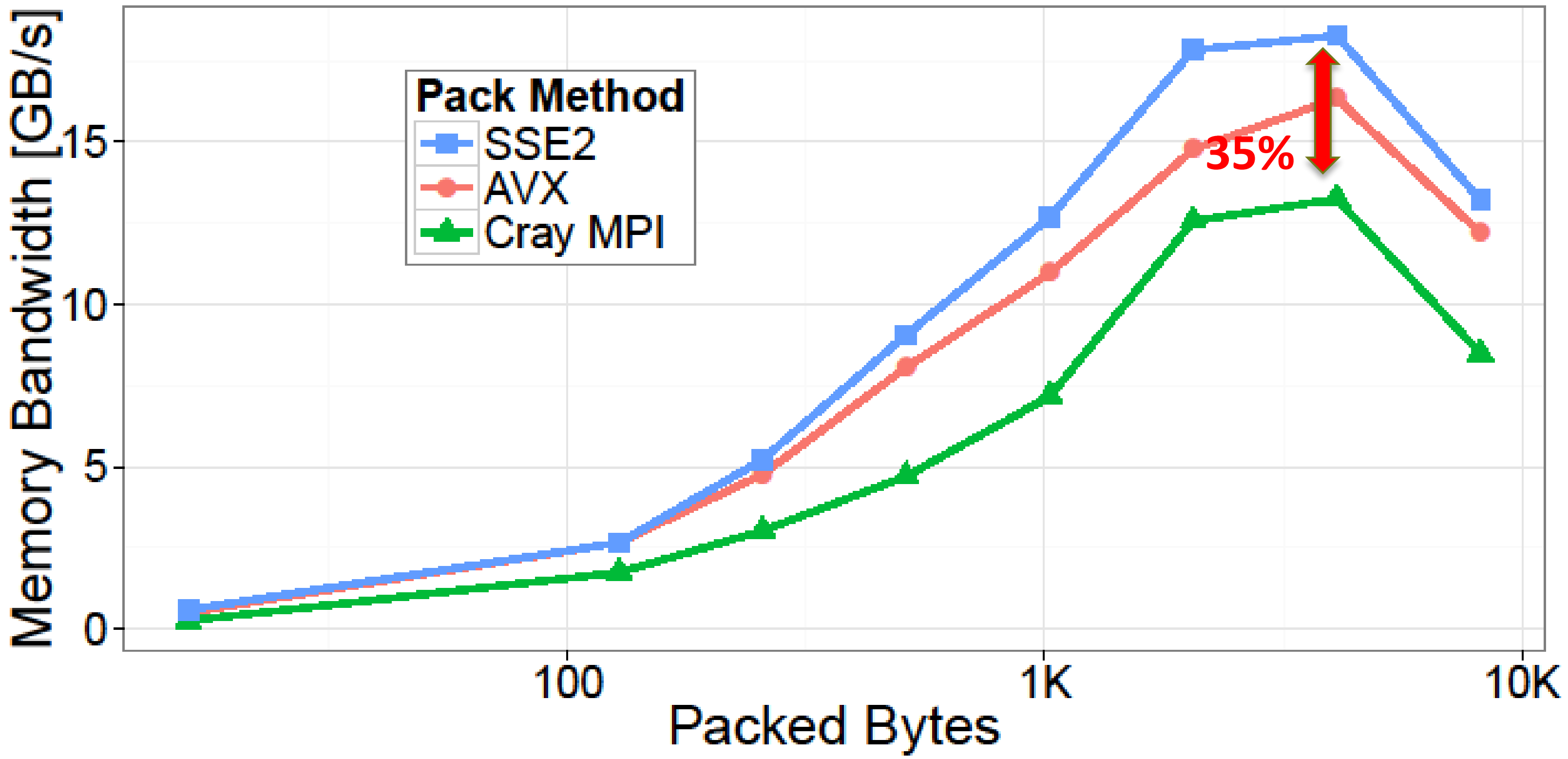
Use

MPI_Send(cnt, buf, new_ddt,…)

new_ddt.pack(buf, cnt tmpbuf)
PMPI_Send(…tmpbuf, MPI_BYTE)

Detour: Copying Data

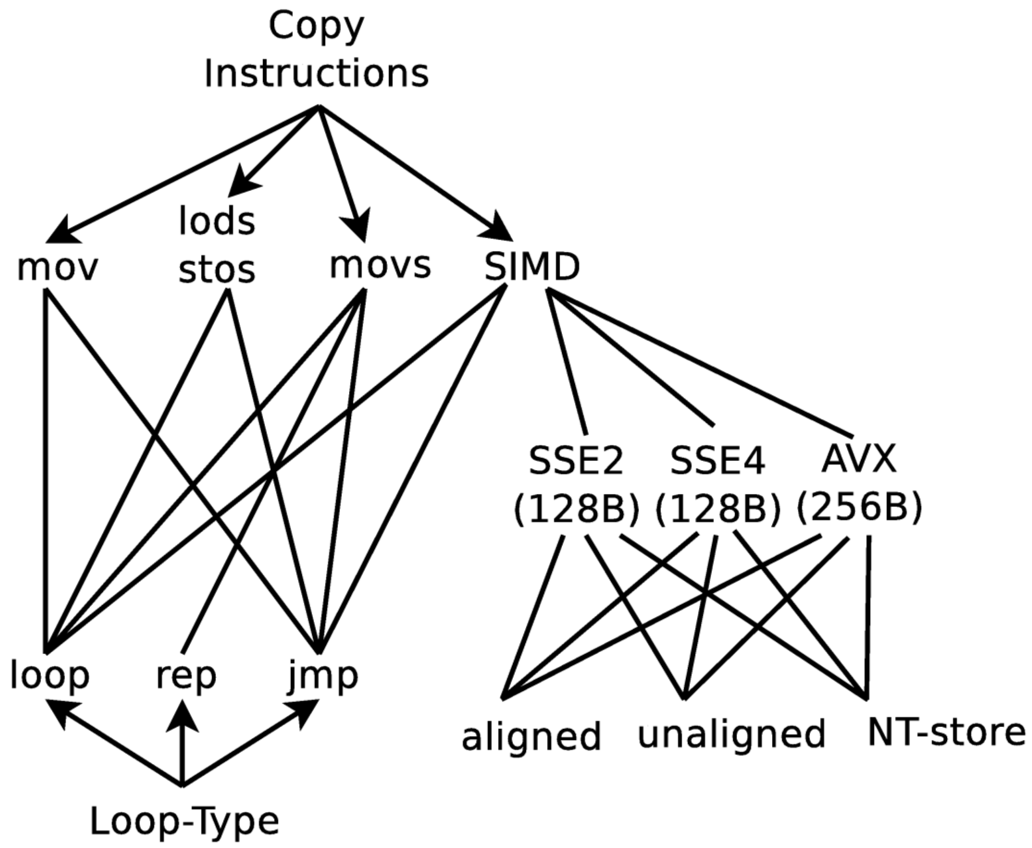
- **Basic elements of DDTs are always consecutive blocks**
- **If the size of the block is less the 256B we completely unroll the loop**
- **Otherwise: use fastest available instruction (SSE2 on our test system)**



In-cache measurement on AMD Interlagos CPU (Blue Waters test system)

Detour: How to Copy Fast on x86?

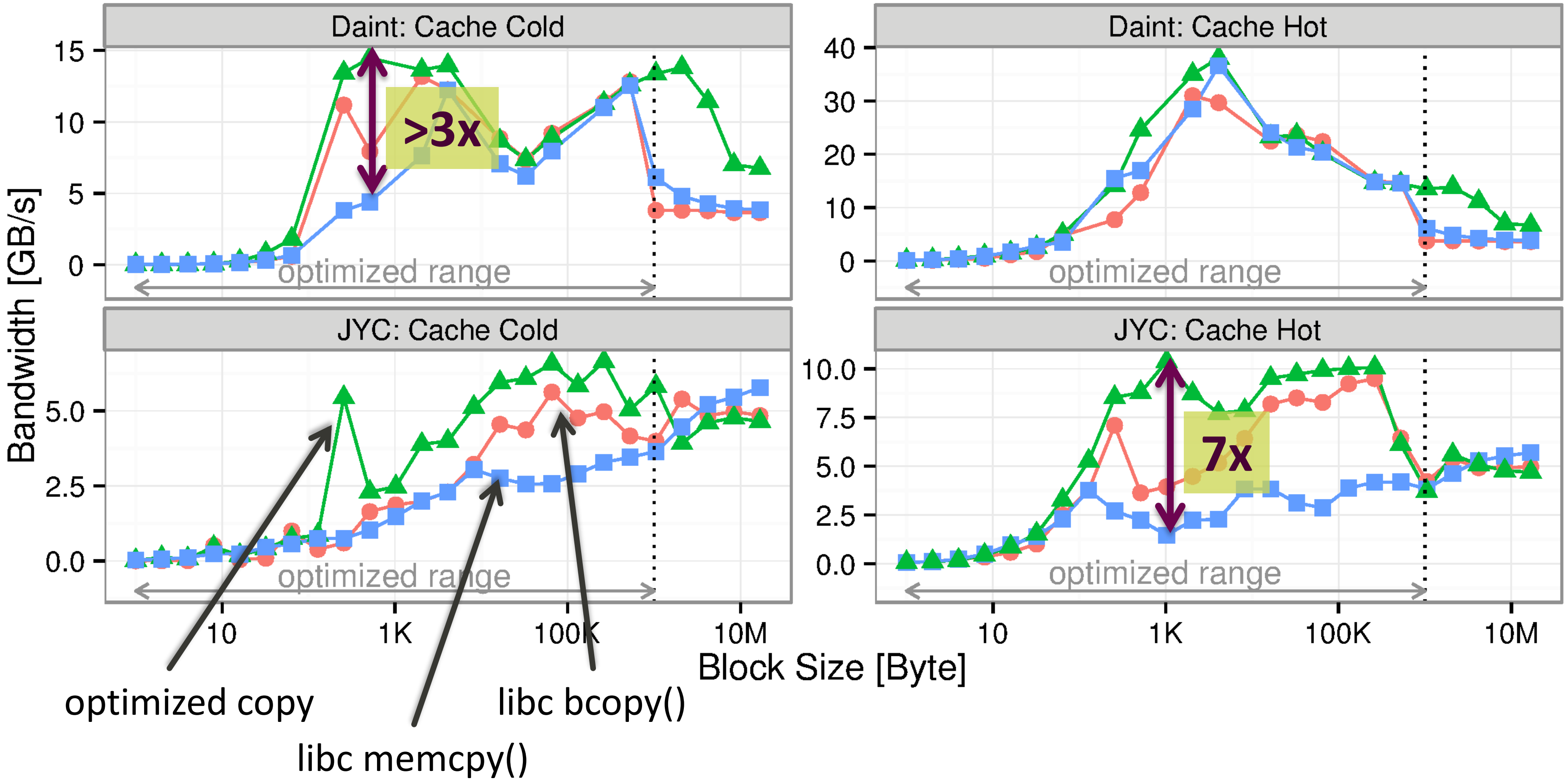
- **Lots of choice to move data!**
  - > 36 ways on x86
- **Restricted semantics allow for super-optimization [4]**
  - Exhaustive search
  - Runs ~1 day
  - Generates close-to-optimal copy sequences



Overview of data movement and loop-forming instructions on x86-64.

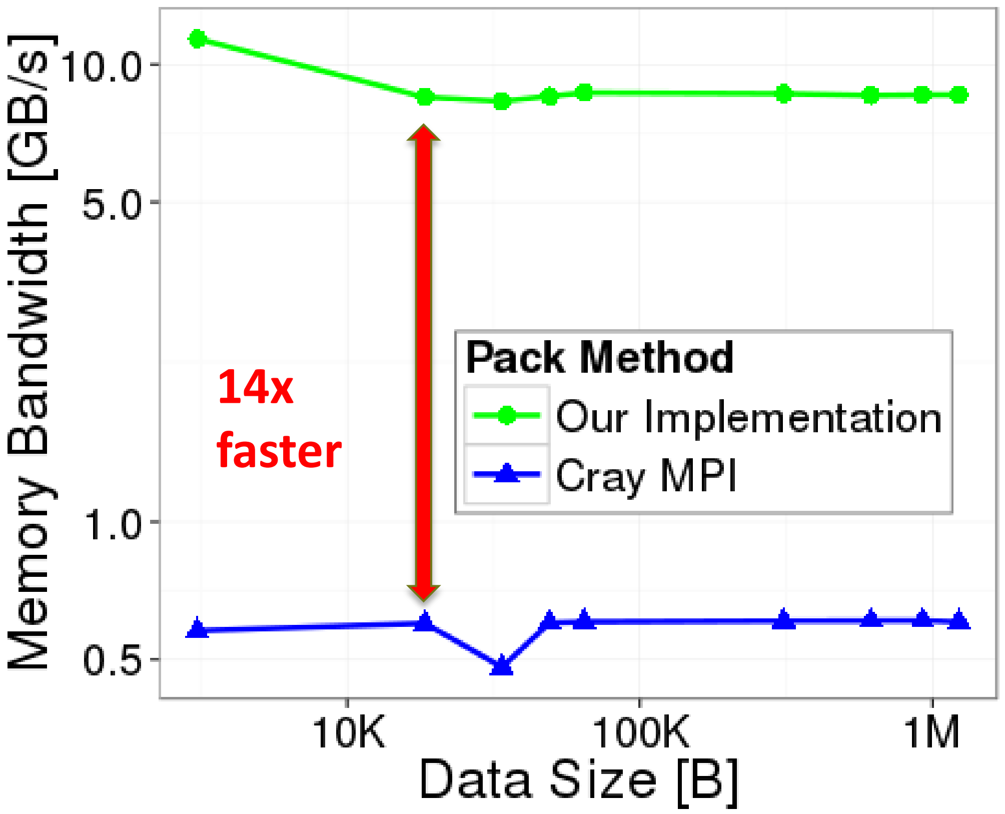
[4]: S. Bansal and A. Aiken: *"Automatic generation of peephole superoptimizers", SIGPLAN Notices 2006*

Detour: Optimized Local Copy Sequence



Datatype Example (1): Packing Vectors

- Vector count and size and extent of subtype are always known
- eliminate induction variables to reduce loop overhead
- Unroll loop for innermost loop 16 times



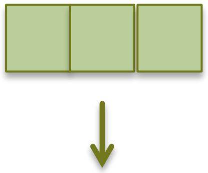
HVector(2,1,6144) of
Vector(8,8,32) of
Contig(6) of
MPI_FLOAT

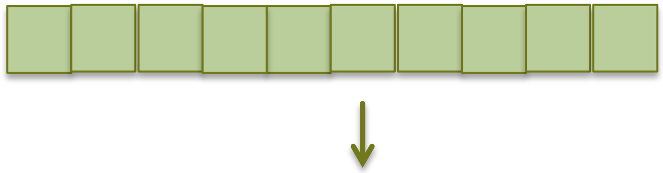
This datatype is used by the Quantum-Chromodynamics code MILC [2]

[2] Bernard, et al.: *Studying quarks and gluons on MIMD parallel computers*, Intl. Journal of Supercomputer Application, 1991

Datatype Example (2): Irregular Data

Depending on index list length:





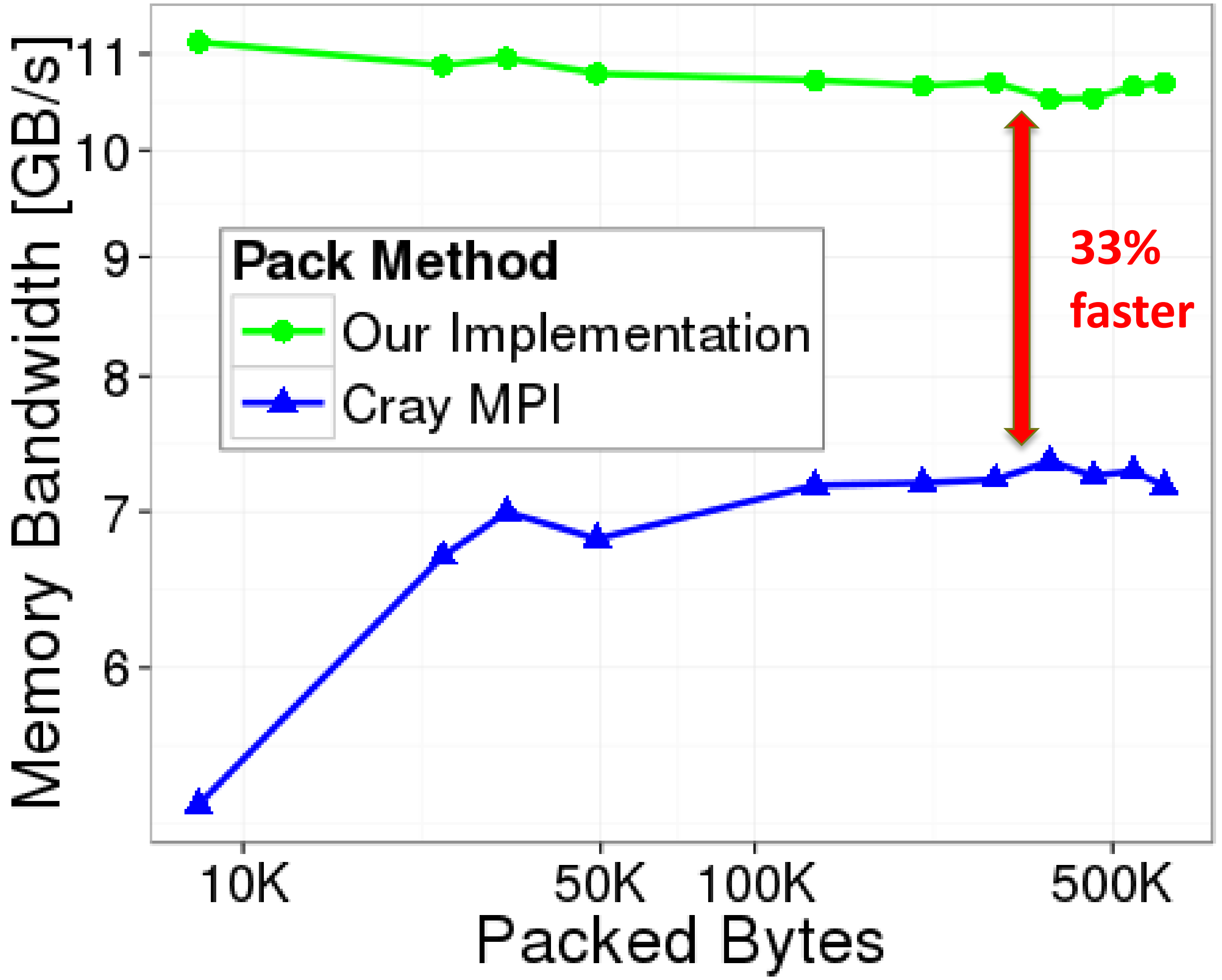
```
copy(inb+off[0],  outb+…, len[0])
copy(inb+off[1],  outb+…, len[1])
copy(inb+off[2],  outb+…, len[2])
```

Inline indices into code

```
for (i=0; i<idx.len; i+=3) {
  inb0=load(idx[i+0])+inb
  inb1=load(idx[i+1])+inb
  inb2=load(idx[i+2])+inb
  // load oub and len
  copy(inb0, outb0, len0)
  copy(inb1, outb1, len1)
  copy(inb2, outb2, len2)
}
```

Minimize loop overhead by unrolling the loop over the index list

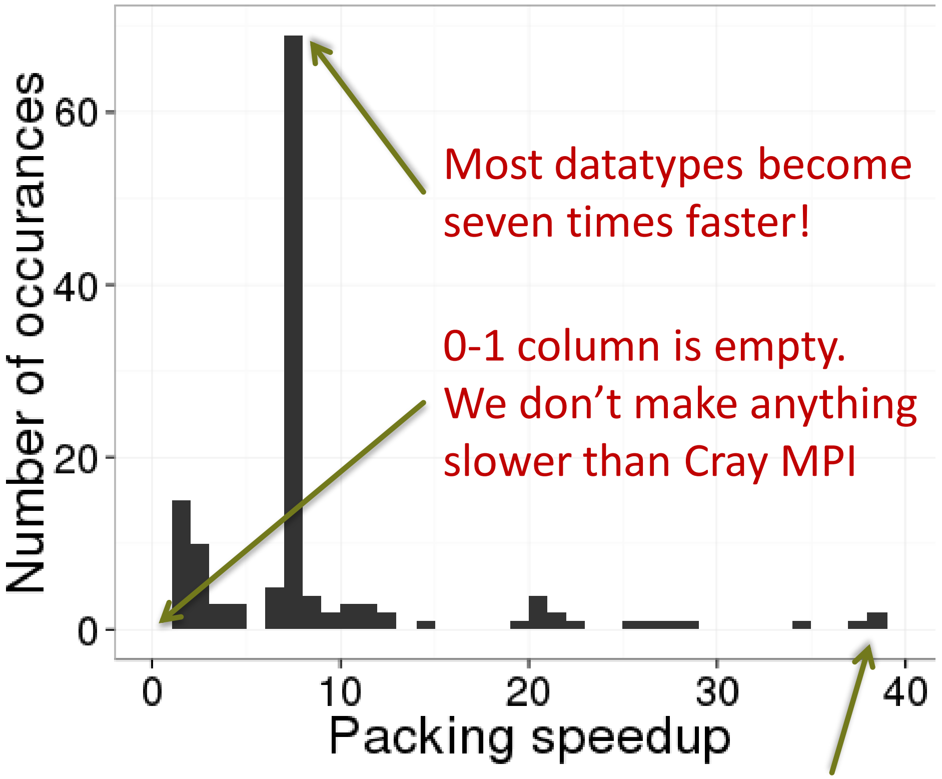
Datatype Example (2): Irregular Packing Performance



Hindexed DDT with random displacements

What’s the catch?

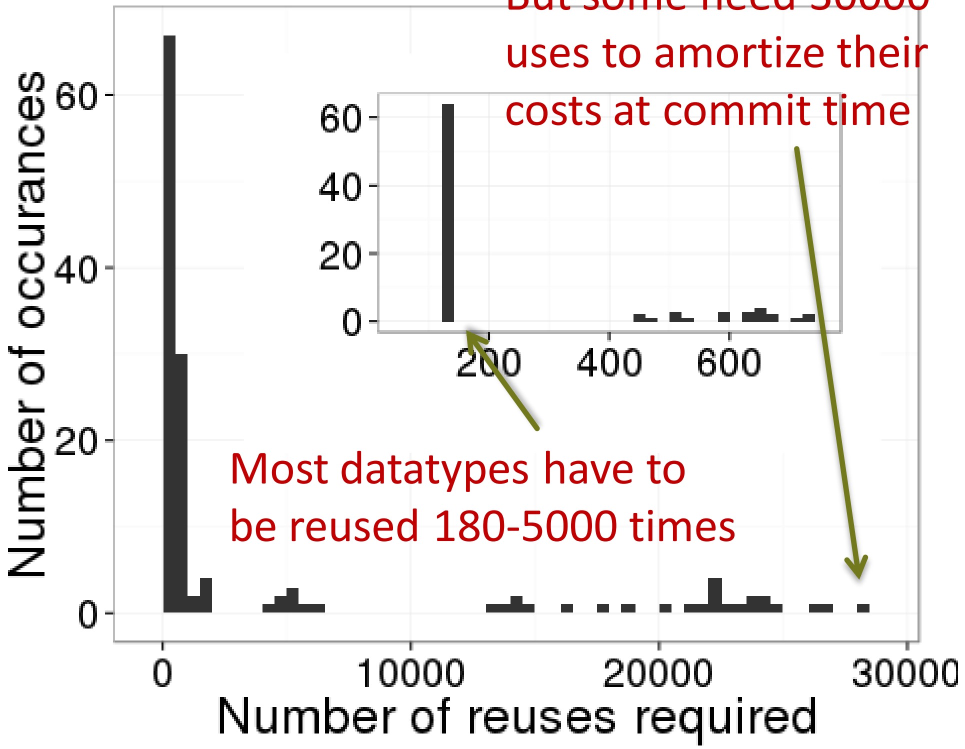
- Emitting and compiling IR is (too?) expensive!
- Commit should tune the DDT, but we do not know how often it will be used – how much tuning is ok?
- Case study: MIMD Lattice Computation (thanks to Steve Gottlieb)



Most datatypes become seven times faster!

0-1 column is empty. We don’t make anything slower than Cray MPI

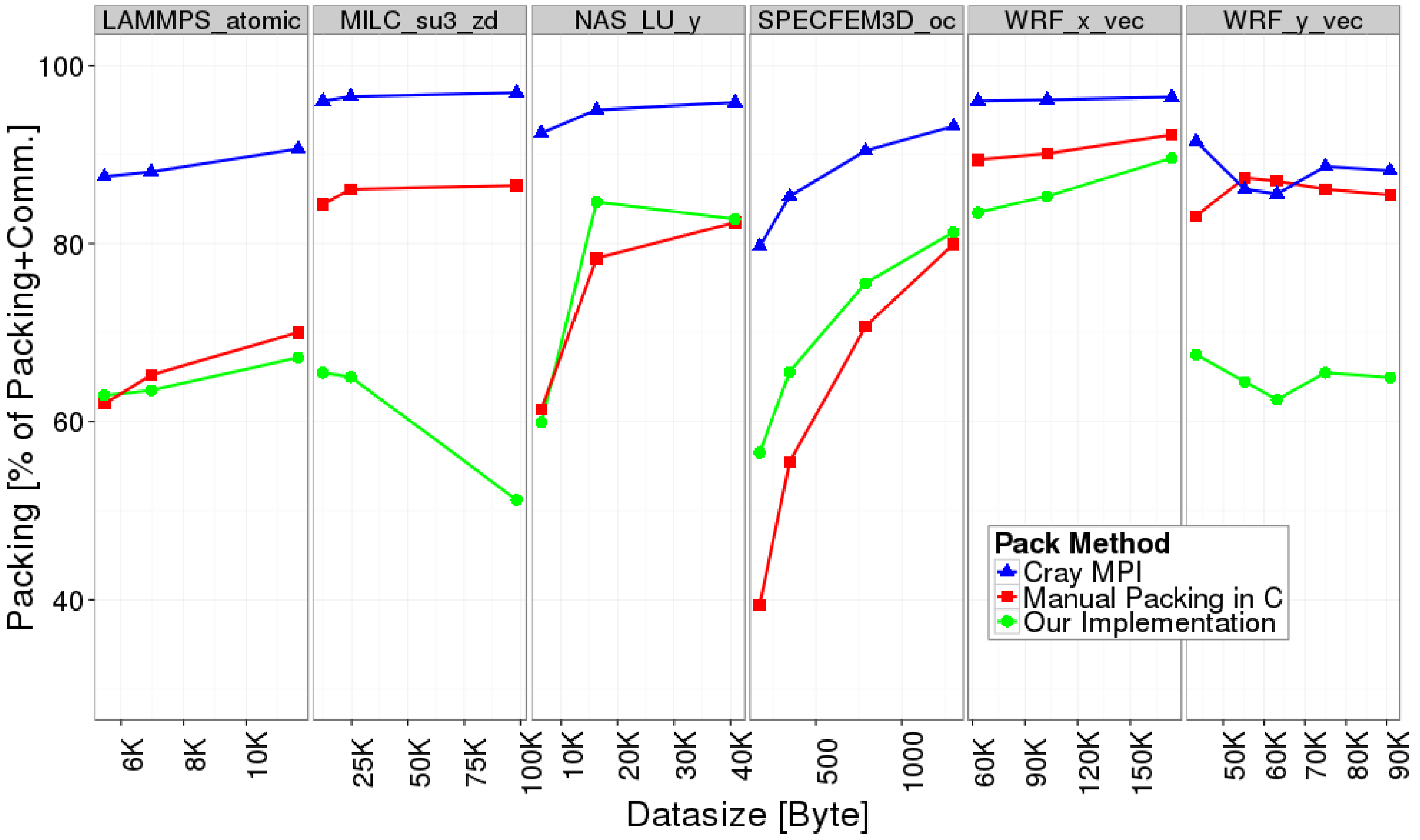
Some even 38 times



But some need 30000 uses to amortize their costs at commit time

Most datatypes have to be reused 180-5000 times

Can we beat manual packing?



The Runtime Recompilation for HPC Manifesto

- **Example demonstrator: MPI Datatypes (works great!)**
  - Has (limited) language interface
  - Missing information:

    How often will the DDT be reused?

    How will it be used (Send/Recv/Pack/Unpack)?

    Will the buffer argument be always the same?

    Will the data to pack be in cache or not?
- **How can we generalize this?**
  - Runtime-optimize everything!!
  - Two main problems:

    What to runtime-recompile?

    Idea: largest subgraph of CFG with constant variables (define "largest"!)

    When to runtime-recompile?

    Is it worth the recompilation overhead!?

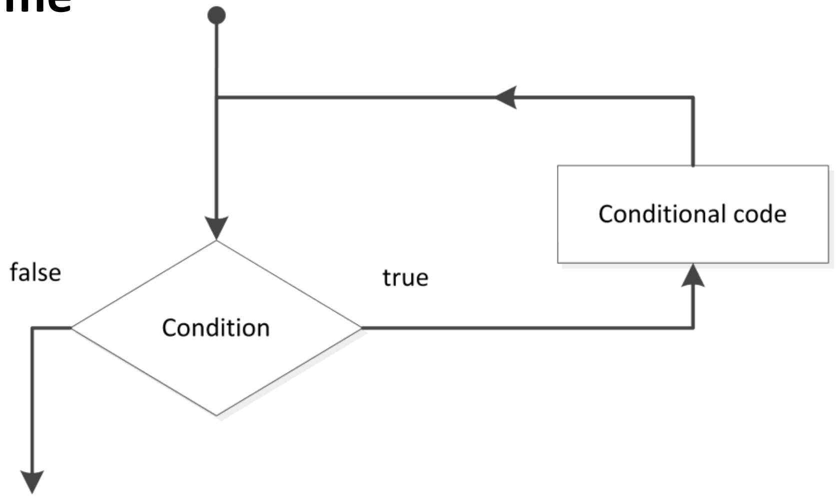
Is this a dead end?

http://spcl.inf.ethz.ch/Research/Parallel_Programming/MPI_Datatypes/libpack

Counting Loop Iterations!

- **Structures that determine program runtime**

LOOPS



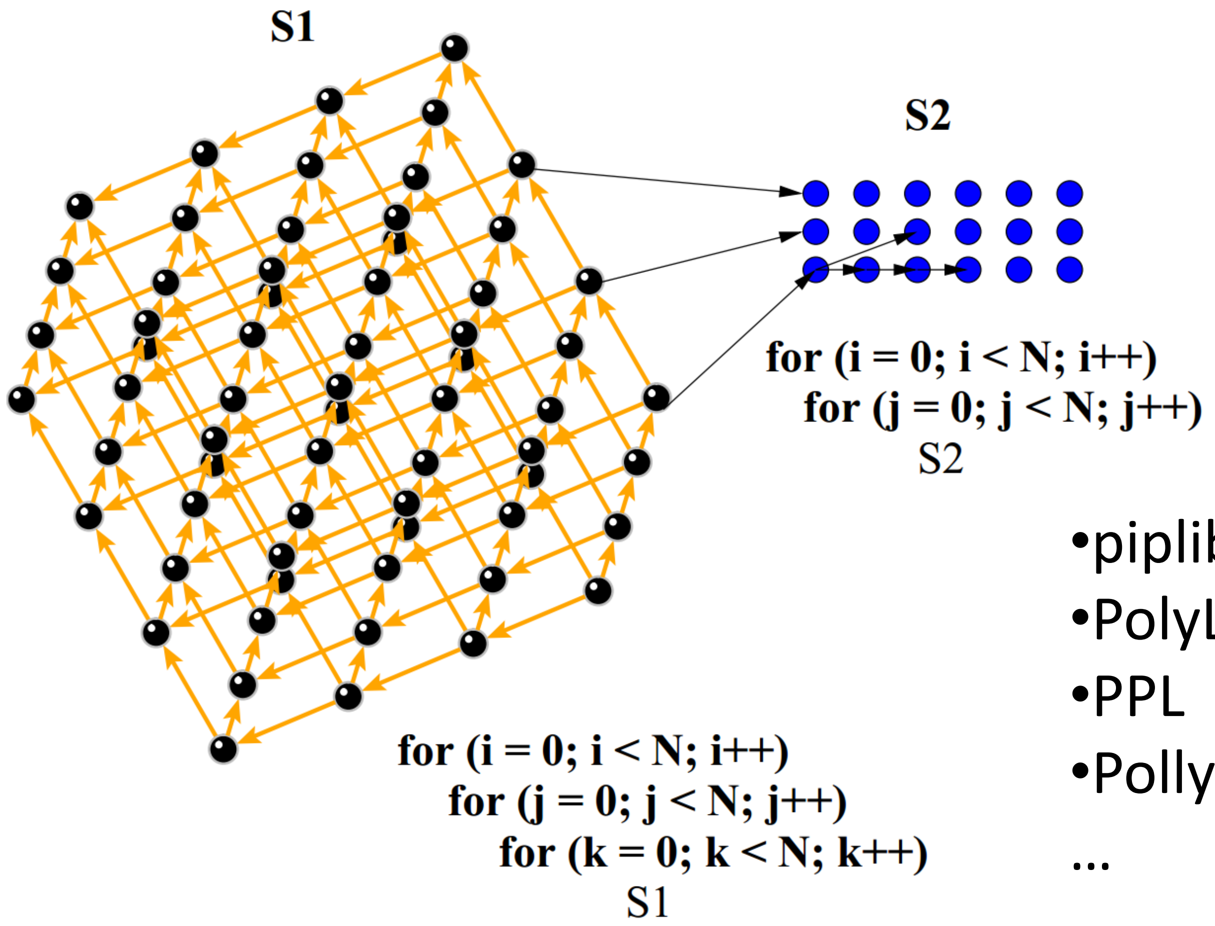
- **Assumption:**
  Other instructions do not influence it

- **Example:**

```
for (x=0; x < n/p; x++)
        for (y=1; y < n; y=2*y )
                veryComplicatedOperation(x,y);
```

Related Work: Counting Loop Iterations

- **In the Polyhedral model:**



```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      S2
```

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
         S1
```

- piplib
- PolyLib
- PPL
- Polly

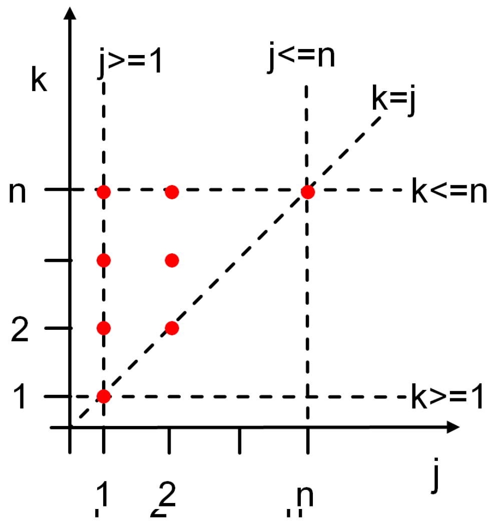
…

R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. J. ACM, 14(3):563–590, July 1967.

Related Work: Counting Loop Iterations

- **When the polyhedral model fails**

```
for (j = 1; j <= n; j = j*2)
        for (k = j; k <= n; k = k++)
                veryComplicatedOperation(j,k);
```



$$j \in [1, n]$$

$$k \in [j, n]$$

$$N = (n+1)\log_2 n - n + 2$$

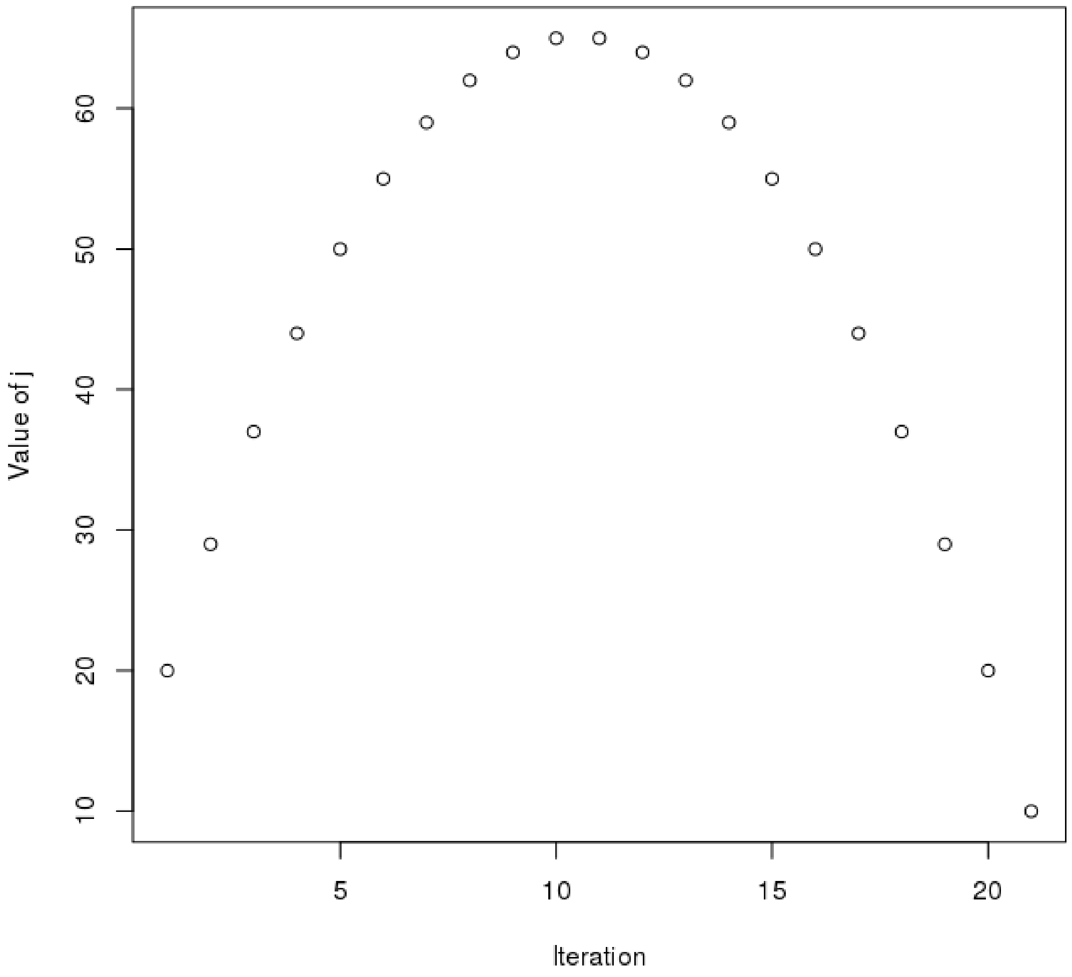
$$N = \frac{n(n+1)}{2}$$

Related Work: Counting Loop Iterations

- **When the polyhedral model cannot handle it**

```
j=10;
k=10;
while (j>0){
   j=j+k;
   k--;
}
```

?



Counting Arbitrary Affine Loop Nests

- **Affine loops**

```
$x$=$x_0$;                // Initial assignment
while($c^T x < g$)        // Loop guard
    $x$=$Ax + b$;         // Loop update
```

- **Perfectly nested affine loops**

```
while($c_1^T x < g_1$) {
   $x = A_1 x + b_1$;
   while($c_2^T x < g_2$) {
      ...
       $x = A_{k-1} x + b_{k-1}$;
       while($c_k^T x < g_k$) {
           $x = A_k x + b_k$;
           while($c_{k+1}^T x < g_{k+1}$) {... }
           $x = U_k x + v_k$; }
       $x = U_{k-1} x + v_{k-1}$;
      ...}
   $x = U_1 x + v_1$;}
```

$$A_k, U_k \in \mathbb{R}^{m \times m},\ b_k, v_k, c_k \in \mathbb{R}^m,\ g_k \in \mathbb{R} \text{ and } k = 1 \ldots r.$$

Counting Arbitrary Affine Loop Nests

- **Example**

```
for (j=1; j < n/p + 1; j= j*2)
        for (k=j; k < m; k = k + j )
                veryComplicatedOperation(j,k);
```

Counting Arbitrary Affine Loop Nests

- **Example**

```
for (j=1; j < n/p + 1; j= j*2)
        for (k=j; k < m; k = k + j )
                veryComplicatedOperation(j,k);
```

```
while($c_1^T x < g_1$) {
  $x = A_1 x + b_1$;
  while($c_2^T x < g_2$) {
    ...
     $x = A_{k-1} x + b_{k-1}$;
     while($c_k^T x < g_k$) {
        $x = A_k x + b_k$;
        while($c_{k+1}^T x < g_{k+1}$) {... }
        $x = U_k x + v_k$; }
     $x = U_{k-1} x + v_{k-1}$;
    ...}
  $x = U_1 x + v_1$;}
```

Counting Arbitrary Affine Loop Nests

- **Example**

```
for (j=1; j < n/p + 1; j= j*2)
        for (k=j; k < m; k = k + j )
                veryComplicatedOperation(j,k);
```

$$\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} j \\ k \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix};$$

```
while($c_1^T x < g_1$) {
  $x = A_1 x + b_1$;
  while($c_2^T x < g_2$) {
    ...
     $x = A_{k-1} x + b_{k-1}$;
     while($c_k^T x < g_k$) {
        $x = A_k x + b_k$;
        while($c_{k+1}^T x < g_{k+1}$) {... }
        $x = U_k x + v_k$; }
     $x = U_{k-1} x + v_{k-1}$;
    ...}
  $x = U_1 x + v_1$;}
```

Counting Arbitrary Affine Loop Nests

- **Example**

```
for (j=1; j < n/p + 1; j= j*2)
        for (k=j; k < m; k = k + j )
                veryComplicatedOperation(j,k);
```

$$\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix};$$

$$while\left(\begin{pmatrix} 1 & 0 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} < {}^{n}\!/_{p} + 1\right)\{$$

$$\}$$

```
while($c_1^T x < g_1$) {
  $x = A_1 x + b_1;$
  while($c_2^T x < g_2$) {
    ...
     $x = A_{k-1} x + b_{k-1};$
     while($c_k^T x < g_k$) {
        $x = A_k x + b_k;$
        while($c_{k+1}^T x < g_{k+1}$) {... }
        $x = U_k x + v_k;$ }
     $x = U_{k-1} x + v_{k-1};$
    ...}
  $x = U_1 x + v_1;$}
```

Counting Arbitrary Affine Loop Nests

- **Example**

```
for (j=1; j < n/p + 1; j= j*2)
        for (k=j; k < m; k = k + j )
                veryComplicatedOperation(j,k);
```

```
while ($c_1^T x < g_1$) {
  $x = A_1 x + b_1$;
  while ($c_2^T x < g_2$) {
    ...
     $x = A_{k-1} x + b_{k-1}$;
     while ($c_k^T x < g_k$) {
        $x = A_k x + b_k$;
        while ($c_{k+1}^T x < g_{k+1}$) { ... }
        $x = U_k x + v_k$; }
     $x = U_{k-1} x + v_{k-1}$;
    ...}
  $x = U_1 x + v_1$;}
```

```
$\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}$;
$while(\begin{pmatrix} 1 & 0 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} < n/p + 1)\{$
    $\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$;
    $while(\begin{pmatrix} 0 & 1 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} < m)\{$

    }
}
```

Counting Arbitrary Affine Loop Nests

- **Example**

```
for (j=1; j < n/p + 1; j= j*2)
        for (k=j; k < m; k = k + j )
                veryComplicatedOperation(j,k);
```

```
while ($c_1^T x < g_1$) {
  $x = A_1 x + b_1$;
  while ($c_2^T x < g_2$) {
    ...
     $x = A_{k-1} x + b_{k-1}$;
     while ($c_k^T x < g_k$) {
        $x = A_k x + b_k$;
        while ($c_{k+1}^T x < g_{k+1}$) {... }
        $x = U_k x + v_k$; }
     $x = U_{k-1} x + v_{k-1}$;
    ...}
  $x = U_1 x + v_1$;}
```

$$\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix};$$

$$while(\begin{pmatrix} 1 & 0 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} < n/p + 1)\{$$

$$\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$$

$$while(\begin{pmatrix} 0 & 1 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} < m)\{$$

$$\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$$

$$\}\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$$

$$\}$$

Counting Arbitrary Affine Loop Nests

- **Example**

```
for (j=1; j < n/p + 1; j= j*2)
        for (k=j; k < m; k = k + j )
                veryComplicatedOperation(j,k);
```

```
while($c_1^T x < g_1$) {
  $x = A_1 x + b_1$;
  while($c_2^T x < g_2$) {
    ...
     $x = A_{k-1} x + b_{k-1}$;
     while($c_k^T x < g_k$) {
        $x = A_k x + b_k$;
        while($c_{k+1}^T x < g_{k+1}$) {... }
        $x = U_k x + v_k$; }
     $x = U_{k-1} x + v_{k-1}$;
    ...}
  $x = U_1 x + v_1$;}
```

$$x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix};$$

$$while\left(\begin{pmatrix} 1 & 0 \end{pmatrix} x < n/p + 1\right)\{$$

$$x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$$

$$while\left(\begin{pmatrix} 0 & 1 \end{pmatrix} x < m\right)\{$$

$$x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\} x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$$

$$\}$$

where $x = \begin{pmatrix} j \\ k \end{pmatrix}$

The Workflow

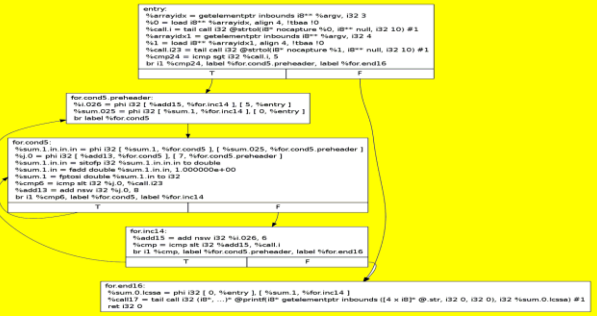
Parallel program

```
    do i = 1, procCols
       call mpi_irecv( buff, 2, dp_type, reduce_exch_proc(i),
>                      i, mpi_comm_world, request, ierr )
       call mpi_send( buff2, 2, dp_type, reduce_exch_proc(i),
>                      i, mpi_comm_world, ierr )
       call mpi_wait( request, status, ierr )
    enddo

    do i = id *n/p, ( id +1)* n/p
       do j = 1, nSize
            call compute
```

LLVM

Loop extraction



```
entry:
 %arrayidx = getelementptr inbounds i8** %argv, i32 3
 %0 = load i8** %arrayidx, align 4, !tbaa !0
 %call.i = tail call i32 @strtol(i8* nocapture %0, i8** null, i32 10) #1
 %arrayidx1 = getelementptr inbounds i8** %argv, i32 4
 %1 = load i8** %arrayidx1, align 4, !tbaa !0
 %call.i23 = tail call i32 @strtol(i8* nocapture %1, i8** null, i32 10) #1
 %cmp24 = icmp sgt i32 %call.i, 5
 br i1 %cmp24, label %for.cond5.preheader, label %for.end16

for.cond5.preheader:
 %i.026 = phi i32 [ %add15, %for.inc14 ], [ 5, %entry ]
 %sum.025 = phi i32 [ %sum.1, %for.inc14 ], [ 0, %entry ]
 br label %for.cond5

for.cond5:
 %sum.1.in.in.in = phi i32 [ %sum.1, %for.cond5 ], [ %sum.025, %for.cond5.preheader ]
 %j.0 = phi i32 [ %add13, %for.cond5 ], [ 7, %for.cond5.preheader ]
 %sum.1.in.in = sitofp i32 %sum.1.in.in.in to double
 %sum.1.in = fadd double %sum.1.in.in, 1.000000e+00
 %sum.1 = fptosi double %sum.1.in to i32
 %cmp6 = icmp slt i32 %j.0, %call.i23
 %add13 = add nsw i32 %j.0, 8
 br i1 %cmp6, label %for.cond5, label %for.inc14

for.inc14:
 %add15 = add nsw i32 %i.026, 6
 %cmp = icmp slt i32 %add15, %call.i
 br i1 %cmp, label %for.cond5.preheader, label %for.end16

for.end16:
 %sum.0.lcssa = phi i32 [ 0, %entry ], [ %sum.1, %for.inc14 ]
 %call17 = tail call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str, i32 0, i32 0), i32 %sum.0.lcssa) #1
 ret i32 0
```

Affine loop synthesis

```
while($c_1^T x < g_1$) {
  $x = A_1 x + b_1$;
  while($c_2^T x < g_2$) {
    ...
     $x = A_{k-1} x + b_{k-1}$;
     while($c_k^T x < g_k$) {
        $x = A_k x + b_k$;
        while($c_{k+1}^T x < g_{k+1}$) {... }
        $x = U_k x + v_k$; }
     $x = U_{k-1} x + v_{k-1}$;
    ...}
  $x = U_1 x + v_1$;}
```

Closed form representation

$$x(i_1,\ldots,i_r) = A_{final}(i_1,\ldots,i_r)\cdot x_0 + b_{final}(i_1,\ldots,i_r)$$

with

$$i_r = 0\ldots n_k(x_{0,k}),\ k = 1\ldots r$$

Number of iterations

$$N = \sum_{i_1=0}^{n_1(x_{0,1})} \sum_{i_2=0}^{n_2(x_{0,2})} \cdots \sum_{i_{r-1}=0}^{n_{r-1}(x_{0,r-1})} n_r(x_{0,r}).$$

Program analysis

$$W = N\Big|_{p=1}$$

$$D = N\Big|_{p\to\infty}$$

Algorithm details

Closed form representation of a loop

- Single affine statement

$$x = Lx + p$$

- Counting function

$$n(x_0)$$

$$x = x_0;$$

$$while\ (c^T x < g)$$

$$x = Ax + b;$$

$$x(i, x_0) = A^i x_0 + \sum_{j=0}^{i-1} A^j \cdot b$$

- **Example**

$$x(i, x_0) = L(i) \cdot x_0 + p(i)$$

$$x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$$

$$n(x_0, c, g) = \arg\min_i (c^T \cdot x(i, x_0) \geq g)$$

$$while\ ((1\ \ 0)\, x < m)\{$$

$$x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$$

$$\}$$

$$\Longrightarrow$$

$$x(i, x_0) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}^i x_0 + \sum_{j=0}^{i-1} \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}^j \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ i & 1 \end{pmatrix} x_0 + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$n(x_0) = \left\lceil \frac{m - k_0}{j_0} \right\rceil$$

Algorithm in details

Folding the loops

```
$x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix};$
while ($\begin{pmatrix} 1 & 0 \end{pmatrix} x < n/p$){
    $x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
    while ($\begin{pmatrix} 0 & 1 \end{pmatrix} x < m$){
        $x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
    }$x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
}
```

Algorithm in details

Folding the loops

```
$x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix};$
while ($\begin{pmatrix} 1 & 0 \end{pmatrix} x < n/p$){
    $x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
    while ($\begin{pmatrix} 0 & 1 \end{pmatrix} x < m$){
        $x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
    }$x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
}
```

```
$x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix};$
while ($\begin{pmatrix} 1 & 0 \end{pmatrix} x < n/p$){
    $x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
    $x = \begin{pmatrix} 1 & 0 \\ i & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
    $x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
}
```

Algorithm in details

Folding the loops

```
$x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix};$
while ($(1\ 0)\, x < n/p$){
    $x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
    while ($(0\ 1)\, x < m$){
        $x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
    }$x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
}
```

$\Rightarrow$

```
$x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix};$
while ($(1\ 0)\, x < n/p$){
        $x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
        $x = \begin{pmatrix} 1 & 0 \\ i & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
        $x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
}
```

$\Rightarrow$

```
$x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix};$
while ($(1\ 0)\, x < n/p$){
        $x = \begin{pmatrix} 2 & 0 \\ i+1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};$
}
```

Algorithm in details

Starting conditions

```
$x_{0,1} \longrightarrow$  $x = x_0;$
                           $while\ (c_1^T x < g_1)\{$
$x_{0,2} \longrightarrow$      $x = A_1 x + b_1;$
                               $while\ (c_2^T x < g_2)\{$
$x_{0,3} \longrightarrow$          $x = A_2 x + b_2;$
                                   $while\ (c_3^T x < g_3)\{$
                                       $x = A_3 x + b_3;$
                                   $\}x = U_2 x + v_2;$
                               $\}x = U_1 x + v_1;$
                           $\}$
```

Algorithm in details

Counting the number of iterations

We have:

Algorithm in details

Counting the number of iterations

We have:

- The closed form for each loop:
  - *Single affine statement*
  - *Counting function*
- Starting condition for each loop

Algorithm in details

Counting the number of iterations

We have:

- The closed form for each loop:
  - *Single affine statement*
  - *Counting function*
- Starting condition for each loop

Number of iterations:

$$N = \sum_{i_1=0}^{n_1(x_{0,1})} \sum_{i_2=0}^{n_2(x_{0,2})} \dots \sum_{i_{r-1}=0}^{n_{r-1}(x_{0,r-1})} n_r(x_{0,r}).$$

Algorithm in details

Counting the number of iterations

- The equation gives precise number of iterations

$$N = \sum_{i_1=0}^{n_1(x_{0,1})} \sum_{i_2=0}^{n_2(x_{0,2})} \dots \sum_{i_{r-1}=0}^{n_{r-1}(x_{0,r-1})} n_r(x_{0,r}).$$

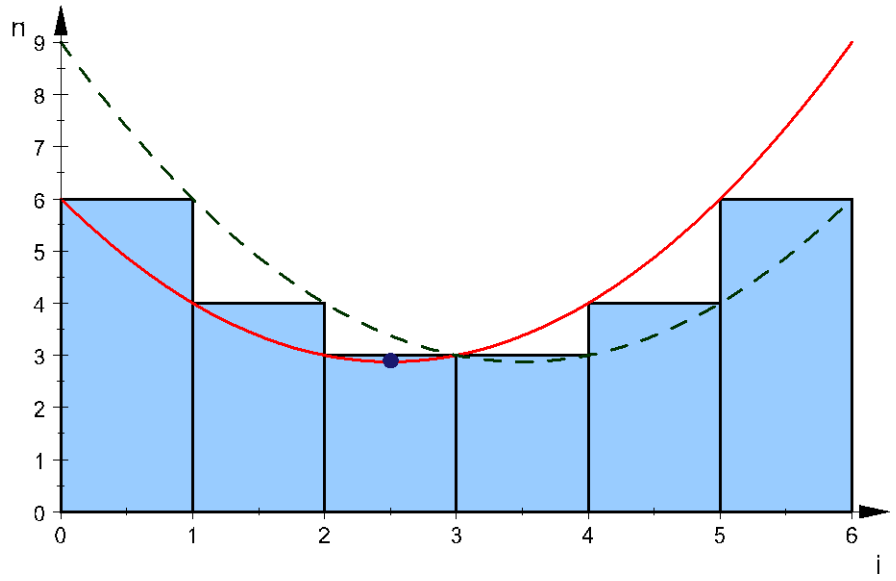
Algorithm in details

Counting the number of iterations

- The equation gives precise number of iterations

$$N = \sum_{i_1=0}^{n_1(x_{0,1})} \sum_{i_2=0}^{n_2(x_{0,2})} \cdots \sum_{i_{r-1}=0}^{n_{r-1}(x_{0,r-1})} n_r(x_{0,r}).$$

- But simplification may fail → Sum approximation
  - *Approximate sums by integrals*
    → lower and upper bounds



Solving more general problems

- Multipath loops
- Conditional statements
- Non-affine loops

```
do j=1,lastrow-firstrow+1
   sum = 0.d0
   do k=rowstr(j),rowstr(j+1)-1
      sum = sum + a(k)*p(colidx(k))
   enddo
   w(j) = sum
enddo
```

$\text{lastrow-firstrow+1} = \texttt{row\_size} = \frac{\texttt{na}}{\texttt{nprows}}$

rowstr(j+1)-1-rowstr(j)=*u*

$$\boxed{N = \frac{\texttt{na} \cdot u}{\texttt{nprows}}}$$

Case studies

- **NAS Parallel Benchmarks: EP**

$$N(m,p) = \left\lceil \frac{2^{m-16} \cdot (u + 2^{16})}{p} \right\rceil$$

```
u :    do i=1,100
         ik =kk/2
         if (ik .eq. 0) goto 130
         kk=ik
       continue
```

Case studies

- **NAS Parallel Benchmarks: EP**

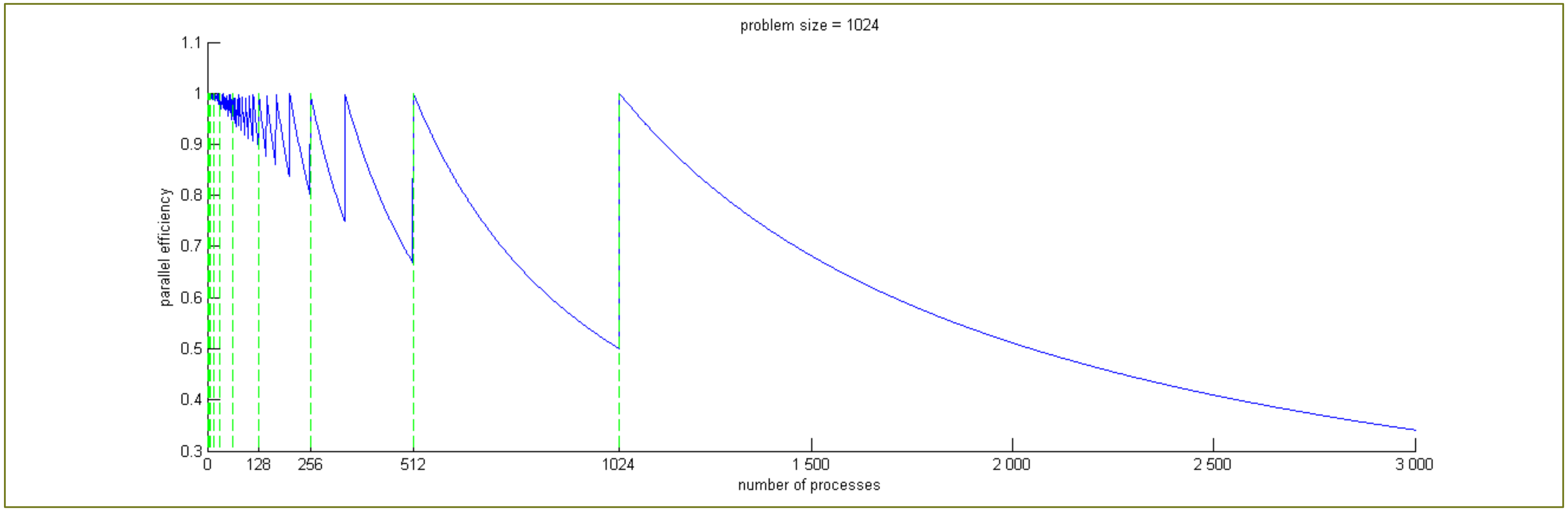
$$N(m,p) = \left\lceil \frac{2^{m-16} \cdot (u + 2^{16})}{p} \right\rceil$$

```
u :     do i=1,100
          ik =kk/2
          if (ik .eq. 0) goto 130
          kk=ik
        continue
```

$$W = T_1 \approx 2^m$$
$$D = T_\infty \approx 1$$

$$E_P = \frac{2^m}{p\left\lceil \frac{2^m}{p} \right\rceil}$$

$$E_P \approx 1 \text{ if } p \leq 2^m$$
$$E_P \approx 2^m/p \text{ if } p > 2^m$$



Case studies

CG – conjugate gradient

$$N \approx k_1 \left\lceil \frac{m}{p} \right\rceil + k_2 \sqrt{\left\lceil \frac{m}{p} \right\rceil} + k_3 \log_2\left(\sqrt{p}\right)$$

$$D = T_\infty = \infty$$

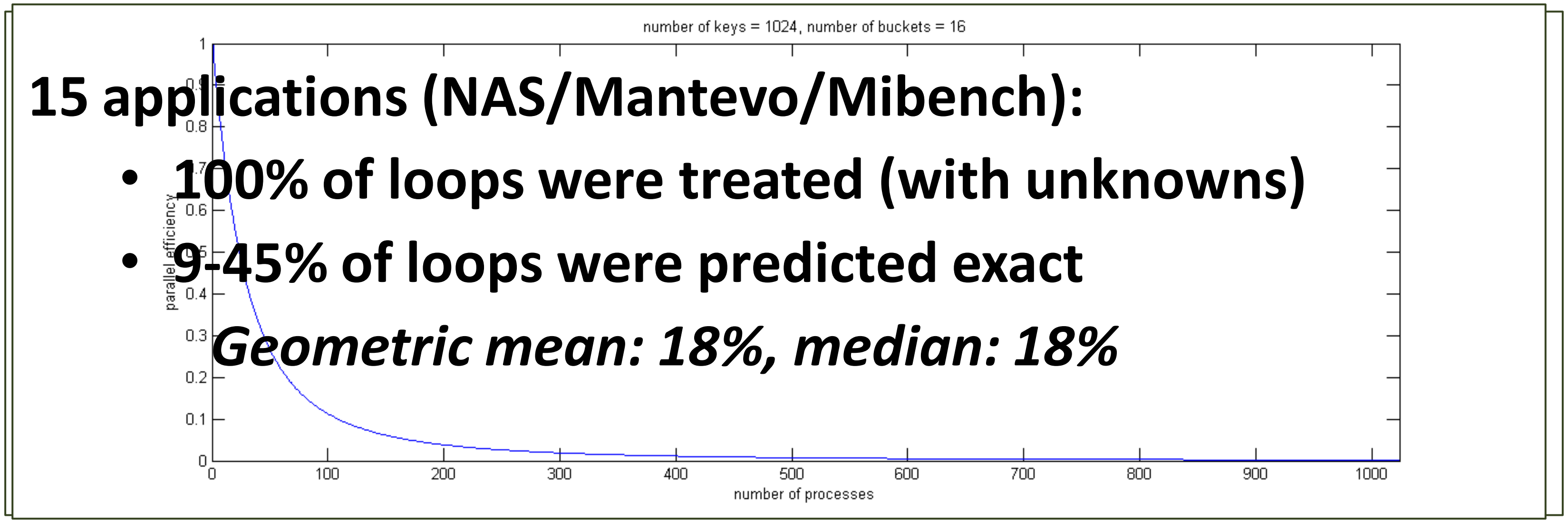
$$N \approx n\left(3(b+t) + 2\left\lceil \frac{m}{p} \right\rceil + p + u_1 + u_2\right)$$

IS – integer sort

$$D = T_\infty = \infty$$

$$E_p = \frac{k_4}{p\left(k_1 \left\lceil \frac{m}{p} \right\rceil + k_2 \sqrt{\left\lceil \frac{m}{p} \right\rceil} + k_3 \log_2\left(\sqrt{p}\right)\right)}$$

$$E_p = \frac{k_1}{\frac{m}{p}\left(p + k_2 \sqrt{\left\lceil \frac{m}{p} \right\rceil} + k_3\right)}$$



15 applications (NAS/Mantevo/Mibench):

- **100% of loops were treated (with unknowns)**
- **9-45% of loops were predicted exact**

Geometric mean: 18%, median: 18%

What problems are remaining?

- **Well, what about non-affine loops?**
  - More general abstract interpretation (next step)
    Any ideas for a more general algebra?
  - In general not solvable
    → *will always have undefined terms*

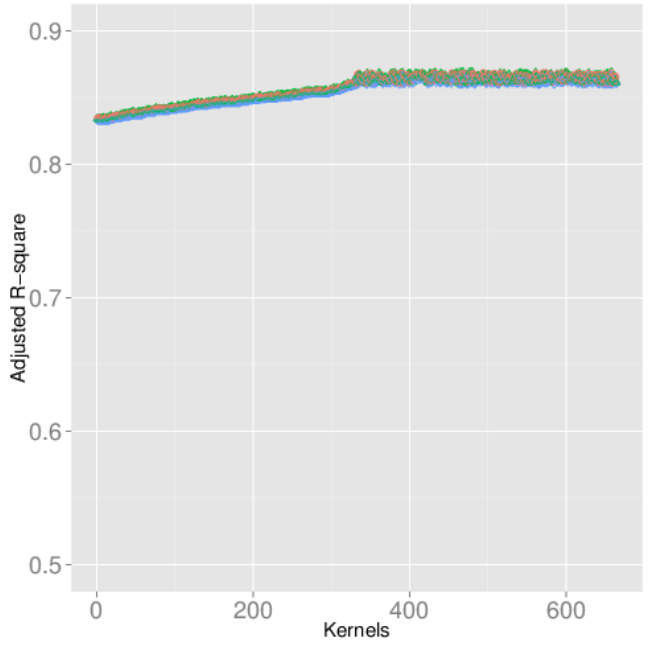
$$\boxed{N = \frac{\mathtt{na} \cdot u}{\mathtt{nprows}}}$$

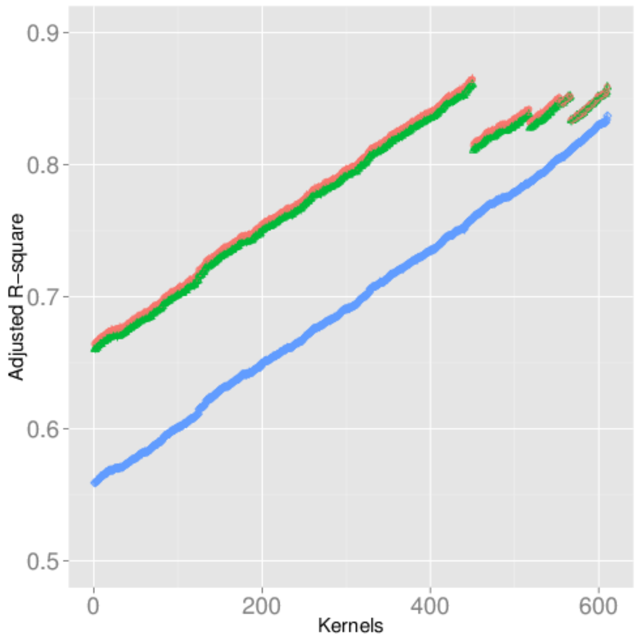
- **Ad-hoc (partial) solution: online machine learning – PEMOGEN**
  - Replace cross-validation with LASSO (regression with L_1 regularizer)
    Much cheaper! (some issues with accuracy – RIP?)
  - Replace LASSO with online LASSO [1]
    No traces! O(1) memory overhead!

P. Garrigues and L. El Ghaoui. An homotopy algorithm for the Lasso with online observations. NIPS 2008

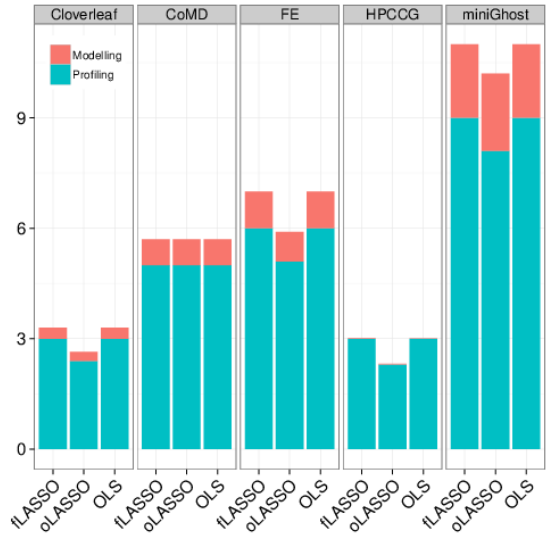
PEMOGEN – static+dynamic analysis

- **Also integrated into LLVM compiler**
  - Automatic kernel detection and instrumentation (Loop Call Graph)
  - Static dataflow analysis reduces parameter space for each kernel





Quality: NAS UA and Mantevo MiniFE



Overhead: Mantevo

The Dragon's Wishlist

- **Faster online JIT support**
  - Optimize LLVM itself
  - Performance expectations for passes
  - Analyze benefits of passes (and orders)
- **Superoptimization**
  - Would be nice, works well, offline!
  - Some approaches exist
- **Specific passes**
  - Better alias analysis
  - Abstract interpretation (cf., PAGAI, e.g., for MPI matching)
  - More in tomorrow's LLVM BoF!