An Overview of Static & Dynamic Techniques for Automatic Performance Modeling

in collaboration with Alexandru Calotoiu and Felix Wolf @ RWTH Aachen
with students Arnamoy Bhattacharyya and Grzegorz Kwasniewski @ SPCL
presented at ISC 2016, Frankfurt, July 2016
My sinful youth

- **Original findings:**
  - If carefully tuned, NBC speeds up a 3D solver
    - *Full code published*
  - $800^3$ domain – 4 GB array
    - 1 process per node, 8-96 nodes
    - *Opteron 246 (old even in 2006, retired now)*
  - Super-linear speedup for 96 nodes
    - ~5% better than linear
  - **9 years later: attempt to reproduce 😊!**
    - System A: 28 quad-core nodes, Xeon E5520
    - System B: 4 nodes, dual Opteron 6274

“Neither the experiment in A nor the one in B could reproduce the results presented in the original paper, where the usage of the NBC library resulted in a performance gain for practically all node counts, reaching a superlinear speedup for 96 cores (explained as being due to cache effects in the inner part of the matrix vector product).”
How to report a performance result?

Scientific Benchmarking of Parallel Computing Systems
Twelve ways to tell the masses when reporting performance results

Torsten Hoefler
Dept. of Computer Science
ETH Zurich
Zurich, Switzerland
htor@inf.ethz.ch

Robert Belli
Dept. of Computer Science
ETH Zurich
Zurich, Switzerland
bellir@inf.ethz.ch

@SC’15

ABSTRACT
Measuring and reporting performance of parallel computers constitutes the basis for scientific advancement of high-performance computing (HPC). Most scientific reports show performance improvements of new techniques and are thus obliged to ensure reproducibility or at least interpretability. Our investigation of a stratified sample of 120 papers across three top conferences in the field shows that the state of the practice is lacking. For example, it is often unclear if reported improvements are deterministic or observed by chance. In addition to distilling best practices from existing work, we propose statistically sound analysis and reporting techniques and simple guidelines for experimental design in parallel computing and codify them in a portable benchmarking library. We aim to improve the standards of reporting research results and initiate a discussion in the HPC field. A wide adoption of our minimal set of rules will lead to better interpretability of performance results and improve the scientific culture in HPC.

Reproducing experiments is one of the main principles of the scientific method. It is well known that the performance of a computer program depends on the application, the input, the compiler, the runtime environment, the machine, and the measurement methodology [20, 43]. If a single one of these aspects of experimental design is not appropriately motivated and described, presented results can hardly be reproduced and may even be misleading or incorrect.

The complexity and uniqueness of many supercomputers makes reproducibility a hard task. For example, it is practically impossible to recreate most hero-runs that utilize the world’s largest machines because these machines are often unique and their software configurations change regularly. We introduce the notion of interpretability, which is weaker than reproducibility. We call an experiment interpretable if it provides enough information to allow scientists to understand the experiment, draw own conclusions, assess their certainty, and possibly generalize results. In other words, interpretable experiments support sound conclusions and convey precise information among scientists. Obviously, every scientific
Analytical application performance modeling

- **Scalability bug prediction**
  - Find latent scalability bugs early on (before machine deployment)
  
  \[ SC13: \text{A. Calotoiu, TH, M. Poke, F. Wolf: Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes} \]

- **Automated performance testing**
  - Performance modeling as part of a software engineering discipline in HPC
  
  \[ ICS'15: \text{S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, F. Wolf: Exascalining Your Library: Will Your Implementation Meet Your Expectations?} \]

- **Hardware/Software co-design**
  - Decide how to architect systems

- **Making performance development intuitive**

\[ 1.5 \cdot 10^{-4} x^3 - 2.6 \cdot 10^{-2} x^2 \]
\[ R^2 = 0.93 \]
Manual analytical performance modeling

- Identify kernels
  - Parts of the program that dominate its performance at larger scales
  - Identified via small-scale tests and intuition

- Create models
  - Laborious process
  - Still confined to a small community of skilled experts

- Disadvantages
  - Time consuming
  - Error-prone, may overlook unscalable code

Our first step: scalability bug detector

main() {
    foo()
    bar()
    compute()
}

Performance measurements (profiles)

Instrumentation
- All functions

Input

Output

Weak scaling

Automated modeling

Ranking:
1. Asymptotic
2. Target scale $p_t$

1. foo
2. compute
3. main
4. bar
[...]
Primary focus on scaling trend

Our ranking

1. $F_1$
2. $F_3$
3. $F_2$

Common performance analysis chart in a not-so-great paper
Primary focus on scaling trend

Our ranking

1. $F_1$
2. $F_3$
3. $F_2$

Actual measurement in laboratory conditions
Primary focus on scaling trend

Our ranking

1. $F_1$
2. $F_3$
3. $F_2$
Survey result: performance model normal form

\[ f(p) = \sum_{k=1}^{n} c_k \times p^{i_k} \times \log_{2}^{j_k}(p) \]

\[ n = 1 \]
\[ I = \{0, 1, 2\} \]
\[ J = \{0, 1\} \]

A. Calotoiu, T. Hoefler, M. Poke, F. Wolf: Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes, SC13
Survey result: performance model normal form

\[ f(p) = \sum_{k=1}^{n} c_k \times p^{i_k} \times \log^j_k(p) \]

- \( n = 2 \)
- \( I = \{0, 1, 2\} \)
- \( J = \{0, 1\} \)

\( c_1 \times p \)
- \( c_1 \cdot \log(p) + c_2 \cdot p \)
- \( c_1 \cdot \log(p) + c_2 \cdot p \cdot \log(p) \)
- \( c_1 \cdot \log(p) + c_2 \cdot p^2 \)

\( c_1 + c_2 \times p \)
- \( c_1 \cdot \log(p) + c_2 \cdot p^2 \cdot \log(p) \)

\( c_1 + c_2 \times p^2 \)
- \( c_1 \cdot p + c_2 \cdot p \cdot \log(p) \)
- \( c_1 \cdot p + c_2 \cdot p^2 \)

\( c_1 + c_2 \times \log(p) \)
- \( c_1 \cdot p + c_2 \cdot p^2 \cdot \log(p) \)

\( c_1 + c_2 \times p \times \log(p) \)
- \( c_1 \cdot \log(p) + c_2 \cdot p \)
- \( c_1 \cdot \log(p) + c_2 \cdot p \cdot \log(p) \)
- \( c_1 \cdot \log(p) + c_2 \cdot p^2 \)

\( c_1 + c_2 \times p^2 \times \log(p) \)
- \( c_1 \cdot p \cdot \log(p) + c_2 \cdot p^2 \)
- \( c_1 \cdot p \cdot \log(p) + c_2 \cdot p^2 \cdot \log(p) \)

\( c_1 + c_2 \times p^2 \times \log(p) \)
- \( c_1 \cdot p^2 + c_2 \cdot p^2 \cdot \log(p) \)
Our automated generation workflow

- Performance measurements
- Performance profiles
- Model generation
- Scaling models
- Performance extrapolation
- Ranking of kernels
- Model refinement
- Accuracy saturated?
- Yes
- No

Kernel refinement

Statistical quality assurance

A. Calotoiu, T. Hoefler, M. Poke, F. Wolf: Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes, SC13
**Model refinement**

1. **Input data**
2. **Hypothesis generation; hypothesis size \(n\)**
3. **Hypothesis evaluation via cross-validation**
4. **Computation of \(R^2\) for best hypothesis**

- \(n = 1; R_0 = \) \(\bar{R}^2\)
- \(c_1 \times p\)
- \(c_1 \times p^2\)
- \(c_1 \times \log(p)\)
- \(c_1 \times p \times \log(p)\)
- \(c_1 \times p^2 \times \log(p)\)

\[
R^2 = 1 - \frac{\text{residualSumSquares}}{\text{totalSumSquares}}
\]

\[
\bar{R}^2 = 1 - (1 - R^2) \cdot \frac{n}{6 - n - 1}
\]

\[
I = \{0, 1, 2\}; J = \{0, 1\}; n_{\text{max}} = 2
\]
Is this all? No, it’s just the beginning …

- We face several problems:
  - Multiparameter modeling – search space explosion
    *Interesting instance of the curse of dimensionality*
  - Modeling overheads
    *Cross validation (leave-one-out) is slow and*
    *Our current profiling requires a lot of storage (>TBs)*
Static analysis of explicitly parallel programs

- Structures that determine program scalability

  LOOPS

- Assumption: Other instructions do not influence it

- Example:

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

T. Hoefler, G. Kwasniewski: Automatic Complexity Analysis of Explicitly Parallel Programs, SPAA’14
Counting arbitrary affine loop nests

- **Affine loops**
  
  \[
  x = x_0; \quad \text{// Initial assignment}
  \]
  
  \[
  \text{while} (c^T x < g) \quad \text{// Loop guard}
  x = Ax + b; \quad \text{// Loop update}
  \]

- **Perfectly nested affine loops**
  
  \[
  \text{while} (c_1^T x < g_1) \{ \]
  \[
  x = A_1 x + b_1;
  \]
  \[
  \text{while} (c_2^T x < g_2) \{ \]
  \[
  \ldots
  x = A_{k-1} x + b_{k-1};
  \]
  \[
  \text{while} (c_k^T x < g_k) \{
  x = A_k x + b_k;
  \]
  \[
  \text{while} (c_{k+1}^T x < g_{k+1}) \{ \ldots \}
  x = U_k x + v_k; \}
  \]
  \[
  x = U_{k-1} x + v_{k-1};
  \]
  \[
  \ldots \}
  x = U_1 x + v_1; \}
  \]

\[
A_k, U_k \in \mathbb{R}^{m \times m}, b_k, v_k, c_k \in \mathbb{R}^m, g_k \in \mathbb{R} \text{ and } k = 1 \ldots r.
\]
Counting arbitrary affine loop nests

- Example

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

T. Hoefler, G. Kwasniewski: Automatic Complexity Analysis of Explicitly Parallel Programs, SPAA’14
Counting arbitrary affine loop nests

Example

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

```c
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

```plaintext
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\((c^T_1 x < g_1)\) \{
    \(x = A_1 x + b_1;\)
    while\((c^T_2 x < g_2)\) \{
        ... 
        \(x = A_{k-1} x + b_{k-1};\)
        while\((c^T_k x < g_k)\) \{
            \(x = A_k x + b_k;\)
            while\((c^T_{k+1} x < g_{k+1})\) \{
                ... 
                \(x = U_k x + v_k;\) 
            \}
            \(x = U_{k-1} x + v_{k-1};\)
        \}
        \(x = U_1 x + v_1;\)
    \}
```
Counting arbitrary affine loop nests

- **Example**

```plaintext
define veryComplicatedOperation(j, k):
    # very complex operation...
define A, b, x, j, k:

    # Initial setup
    x = A1 * x + b1
    while (c1^T x < g1):
        x = A1 * x + b1
        while (c2^T x < g2):
            # Inner loop
            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):
                    # Iterative process...
                    x = U_k * x + v_k
                x = U_k-1 * x + v_k-1
            x = U_1 * x + v_1
```

T. Hoefler, G. Kwasniewski: Automatic Complexity Analysis of Explicitly Parallel Programs, SPAA‘14
Counting arbitrary affine loop nests

- Example

```c
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

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

```latex
\begin{align*}
\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}; \\
\text{while}(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} < \frac{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}; \\
\text{while}(\begin{pmatrix} 0 & 1 \\ 1 & 0 \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}; \\
\end{align*}
```

T. Hoefler, G. Kwasniewski: Automatic Complexity Analysis of Explicitly Parallel Programs, SPAA’14
Counting arbitrary affine loop nests

- Example

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

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

while \((c_1^T x < g_1)\) {
    \[
x = A_1 x + b_1;
\]
while \((c_2^T x < g_2)\) {
    \[
x = A_{k-1} x + b_{k-1};
\]
    \[
    x = A_k x + b_k;
    \]
    \[
    while (c_{k+1}^T x < g_{k+1}) \{ \ldots \}
    \]
    \[
x = U_{k} x + v_k; \}
    \[
x = U_{k-1} x + v_{k-1};
    \]
\[
\ldots \}

\[
x = U_1 x + v_1;\}
\]

where \[
x = \begin{pmatrix} j \\ k \end{pmatrix}
\]
```

T. Hoefler, G. Kwasniewski: Automatic Complexity Analysis of Explicitly Parallel Programs, SPAA’14
Overview of the whole system

Parallel program

```
# Parall program

do i = 1, procs
   call mpi_irecv{ buff, .., dp_type, reduce_ench_procl, 
                      i, mpi_comm_world, request, ierr }
   call mpi_send{ buff2, .., dp_type, reduce_ench_procl, 
                     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
   enddo
```

Closed form representation

\[ x(i_1, ..., i_r) = A_{final}(i_1, ..., i_r) \cdot x_0 + b_{final}(i_1, ..., i_r) \]

with

\[ i_r = 0..n_i(x_{0,k}), k = 1..r \]

Affine loop synthesis

```
while(c^i_1 x < g^i_1) {
  x = A_i x + b_i;
  while(c^i_2 x < g^i_2) {
    ... 
    x = A_{k-1} x + b_{k-1};
  }
  while(c^i_{k+1} x < g^i_{k+1}) {...
    x = U_k x + v_k; 
    x = U_{k-1} x + v_{k-1};
    ...
  }
  x = U_1 x + v_1;
}
```

Number of iterations

\[ N = \sum_{i_1=0}^{n_1(x_{0,1})} \sum_{i_2=0}^{n_2(x_{0,2})} \ldots \sum_{i_{r-1}=0}^{n_{r-1}(x_{0,r-1})} n_r(x_{0,r}) \]

Loop extraction

```
LLVM
```

Program analysis

\[ W = N \bigg|_{p=1} \]

\[ D = N \bigg|_{p \to \infty} \]
What problems are remaining?

- Well, what about non-affine loops?
  - More general abstract interpretation (next step)
  - Not solvable → will always have undefined terms

- Back to PMNF?
  - Generalize to multiple input parameters
    - a) Bigger search-space 😞
    - b) Bigger trace files 😞

- Ad-hoc (partial) solution: online machine learning – PEMOGEN
  - Replace cross-validation with LASSO (regression with $L_1$ regularizer)
    - Much cheaper!
  - Replace LASSO with online LASSO [1]
    - No traces!

\[ N = \frac{na \cdot u}{\text{nprows}} \]

[1]: P. Garrigues and L. El Ghaoui. An homotopy algorithm for the Lasso with online observations. NIPS 2008
PEMOGEN – static 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

A. Bhattacharyya, T. Hoefler: PEMOGEN: Automatic Adaptive Performance Modeling during Program Runtime, PACT’14
Use-case A: automatic testing (Allreduce time)

- Divergence on Piz
- Daint is $O(p^{0.67})$, the highest of all three

---

S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, F. Wolf: Exascaling Your Library: Will Your Implementation Meet Your Expectations?, ICS'15
Use-case B: automatic testing (MPI memory size)

- Linear memory consumption on Juropa
- ParaStation MPI
- uses RC over IB

S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, F. Wolf: Exascaling Your Library: Will Your Implementation Meet Your Expectations?, ICS’15
Performance Analysis 2.0 – Automatic Models

- Is feasible
- Offers insight
- Requires low effort
- Improves code coverage


A. Bhattacharyya, T. Hoefler: PEMOGEN: Automatic Adaptive Performance Modeling during Program Runtime, *PACT 2014*

Backup
How to mechanize the expert? → Survey!

**Computation**
- LU: \( t(p) \sim c \)
- FFT: \( t(p) \sim \log_2(p) \)
- Naïve N-body: \( t(p) \sim p \)
- Samplesort: \( t(p) \sim p^2 \log_2(p) \)

**Communication**
- LU: \( t(p) \sim c \)
- FFT: \( t(p) \sim \log_2(p) \)
- Naïve N-body: \( t(p) \sim p \)
- Samplesort: \( t(p) \sim p^2 \)
Evaluation overview

$I = \{ \frac{0}{2}, \frac{1}{2}, \frac{2}{2}, \frac{3}{2}, \frac{4}{2}, \frac{5}{2}, \frac{6}{2} \}$

$J = \{0, 1, 2\}$

$n = 5$

Sweep3D ✔

MILC ✗

HOMME ✔ ✔ ✔

XNS ✔

Performance measurements → Model generation → Scaling models → Model refinement → Accuracy saturated?

Yes or No → Performance extrapolation → Ranking of kernels → Statistical quality assurance

Evaluation overview
Sweep3D communication performance

- Solves neutron transport problem
- 3D domain mapped onto 2D process grid
- Parallelism achieved through pipelined wave-front process

\[ t^{\text{comm}} = c \cdot \sqrt{p} \]

- LogGP model for communication developed by Hoisie et al.
  - We assume \( p = p_x \cdot p_y \) → Equation (6) in [1]

Sweep3D communication performance

<table>
<thead>
<tr>
<th>Kernel [2 of 40]</th>
<th>Runtime[%] $p_i=262k$</th>
<th>Model [s] $t = f(p)$</th>
<th>Predictive error [%] $p_i=262k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sweep $\rightarrow$ MPI_Recv</td>
<td>65.35</td>
<td>4.03$\sqrt{p}$</td>
<td>5.10</td>
</tr>
<tr>
<td>sweep</td>
<td>20.87</td>
<td>582.19</td>
<td>0.01</td>
</tr>
</tbody>
</table>

$p_i \leq 8k$

#bytes = const.
#msg = const.
MILC

- MILC/su3_rmd – from MILC suite of QCD codes with performance model manually created

- Time per process should remain constant except for a rather small logarithmic term caused by global convergence checks

<table>
<thead>
<tr>
<th>Kernel [3 of 479]</th>
<th>Model [s] t=f(p)</th>
<th>Predictive Error [%] p_t=64k</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute_gen_staple_field</td>
<td>$2.40 \times 10^2$</td>
<td>0.43</td>
</tr>
<tr>
<td>g_vecdoublesum → MPI_Allreduce</td>
<td>$6.30 \times 10^{-6} \times \log_2^2(p)$</td>
<td>0.01</td>
</tr>
<tr>
<td>mult_adj_su3_fieldlink_lathwec</td>
<td>$3.80 \times 10^3$</td>
<td>0.04</td>
</tr>
</tbody>
</table>

$p_i \leq 16k$
HOMME

- Core of the Community Atmospheric Model (CAM)
- Spectral element dynamical core on a cubed sphere grid

<table>
<thead>
<tr>
<th>Kernel [3 of 194]</th>
<th>Model [s] ( t = f(p) )</th>
<th>Predictive error [%] ( p_t = 130k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>box_rearrange → MPI_Reduce</td>
<td>( 0.026 + 2.53 \times 10^{-6} p \times \sqrt{p} + 1.24 \times 10^{-12} p^3 )</td>
<td>57.02</td>
</tr>
<tr>
<td>vlaplace_sphere_vk</td>
<td>49.53</td>
<td>99.32</td>
</tr>
<tr>
<td>compute_and_apply_rhs</td>
<td>48.68</td>
<td>1.65</td>
</tr>
</tbody>
</table>

\( p_i \quad \text{15k} \)
HOMME (2)

- Core of the Community Atmospheric Model (CAM)
- Spectral element dynamical core on a cubed sphere grid

<table>
<thead>
<tr>
<th>Kernel [3 of 194]</th>
<th>Model [s] $t = f(p)$</th>
<th>Predictive error [%] $p_t = 130k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>box_rearrange → MPI_Reduce</td>
<td>$3.63 \times 10^{-6} p \sqrt{p} + 7.21 \times 10^{-13} p^3$</td>
<td>30.34</td>
</tr>
<tr>
<td>vlaplace_sphere_vk</td>
<td>$24.44 + 2.26 \times 10^{-7} p^2$</td>
<td>4.28</td>
</tr>
<tr>
<td>compute_and_apply_rhs</td>
<td>49.09</td>
<td>0.83</td>
</tr>
</tbody>
</table>

$p_i 43k$
HOMME (3)
What about strong scaling?

- Wall-clock time not necessarily monotonically increasing – harder to capture model automatically
  - Different invariants require different reductions across processes

<table>
<thead>
<tr>
<th></th>
<th>Weak scaling</th>
<th>Strong scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invariant</td>
<td>Problem size per process</td>
<td>Overall problem size</td>
</tr>
<tr>
<td>Model target</td>
<td>Wall-clock time</td>
<td>Accumulated time</td>
</tr>
<tr>
<td>Reduction</td>
<td>Maximum / average</td>
<td>Sum</td>
</tr>
</tbody>
</table>

- Superlinear speedup through cache effects
  - Measure and model re-use distance?
XNS

- Finite element flow simulation program with numerous equations represented:
  - Advection diffusion
  - Navier-Stokes
  - Shallow water

- Strong scaling analysis
  - \( P = \{128; \ldots; 4,096\} \)
  - 5 measurements per \( p_i \)
  - Using accumulated time across processes as metric
XNS (2)

Accumulated time

Wallclock time

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Runtime[%] p=128</th>
<th>Runtime[%] p=4,096</th>
<th>Model [s] t = f(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ewdgennpm-&gt;MPI_Recv</td>
<td>0.46</td>
<td>51.46</td>
<td>$0.029 \times p^2$</td>
</tr>
<tr>
<td>ewddot</td>
<td>44.78</td>
<td>5.04</td>
<td>$p \times \log(p)$</td>
</tr>
</tbody>
</table>

#bytes = ~p
#msg = ~p
Step back – what do we really care about?

- Work
  \[ W = T_1 \]

- Depth
  \[ D = T_{\infty} \]

- Parallel efficiency
  \[ E_p = \frac{T_1}{pT_p} \]
Related work: counting loop iterations

- Polyhedral model

Related work: counting loop iterations

- Polyhedral model

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

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

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

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

\[
N = (n+1) \log_2 n - n + 2
\]

Related work: counting loop iterations

- When the polyhedral model cannot handle it

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

![Graph showing iteration vs. value of j]
Algorithm in details

Closed form representation of a loop

- Single affine statement
  \[ x = Lx + p \]
  \[ x = x_0; \]

- Counting function
  \[ n(x_0) \]
  \[ \text{Example} \]
  \[ x(i, x_0) = L(i) \cdot x_0 + p(i) \]
  \[ x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \]
  \[ x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \]
  \[ x(i, x_0) = A^i x_0 + \sum_{j=0}^{i-1} A^j \cdot b \]
  \[ x(i, x_0) = \begin{pmatrix} 1 & 0 \end{pmatrix}^i x_0 + \sum_{j=0}^{i-1} \begin{pmatrix} 1 & 0 \end{pmatrix}^j \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \end{pmatrix} x_0 + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \]
  \[ n(x_0) = \frac{m - k_0}{j_0} \]

while \((c^T x < g)\)
\[ x = A x + b; \]
\[ \text{while} \left( c^T x < g \right) \]
Algorithm in details

Folding the loops

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

\[
\text{while}((1 \ 0)x < n/p)\{
\]
\[
x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};
\]

\[
\text{while}((0 \ 1)x < m)\{
\]
\[
x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};
\]

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

\}
Algorithm in details

Folding the loops

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

\[
\text{while}((1 \ 0)x < n/p)\{
    x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};
}\]

\[
\text{while}((0 \ 1)x < m)\{
    x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};
}\]

\[
} x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};
\]
Algorithm in details

Folding the loops

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

while \((1 \ 0)x < \frac{n}{p}\){

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

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};
\]
Algorithm in details

Starting conditions

\begin{align*}
    x_{0,1} & \rightarrow & x &= x_0; \\
    & \quad & \text{while}(c_1^T x < g_1)\{ \\
    \end{align*}

\begin{align*}
    x_{0,2} & \rightarrow & x &= A_1 x + b_1; \\
    & \quad & \text{while}(c_2^T x < g_2)\{ \\
    \end{align*}

\begin{align*}
    x_{0,3} & \rightarrow & x &= A_2 x + b_2; \\
    & \quad & \text{while}(c_3^T x < g_3)\{ \\
    & \quad & \quad & x &= A_3 x + b_3; \\
    & \quad & \quad & } x &= U_2 x + v_2; \\
    & \quad & \quad & } x &= U_1 x + v_1; \\
    \end{align*}
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

T. Hoefler, G. Kwasniewski: Automatic Complexity Analysis of Explicitly Parallel Programs, SPAA’14
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:**

\[
N = \sum_{i_1=0}^{n_1(x_0,1)} \sum_{i_2=0}^{n_2(x_0,2)} \cdots \sum_{i_{r-1}=0}^{n_{r-1}(x_0, r-1)} n_r(x_0, r).
\]
Algorithm in details

Counting the number of iterations

- The equation computes the precise number of iterations

\[ N = \sum_{i_1=0}^{n_1(x_0,1)} \sum_{i_2=0}^{n_2(x_0,2)} \ldots \sum_{i_{r-1}=0}^{n_{r-1}(x_0,r-1)} n_r(x_0,r). \]
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

T. Hoefler, G. Kwasniewski: Automatic Complexity Analysis of Explicitly Parallel Programs, SPAA’14
Solving more general problems
Solving more general problems

- Multipath loops
Solving more general problems

- Multipath loops
- Conditional statements
Solving more general problems

- Multipath loops
- Conditional statements
- Non-affine loops

```fortran
  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
  lastrow-firstrow+1 = row_size = na / nprows
  rowstr(j+1)-1-rowstr(j) = U
```

\[ N = \frac{na \cdot U}{nprows} \]
Case studies

- NAS Parallel Benchmarks: EP

\[ N(m, p) = \left\lfloor \frac{2^{m-16} \cdot (u + 2^{16})}{p} \right\rfloor \]

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

- NAS Parallel Benchmarks: EP

\[ N(m, p) = \left\lfloor \frac{2^{m-16} \cdot (u + 2^{16})}{p} \right\rfloor \]

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

\[ E_P = \frac{2^m}{p \left\lfloor \frac{2^m}{p} \right\rfloor} \]

\[ E_P \approx 1 \text{ if } p \leq 2^m \]
\[ E_P \approx \frac{2^m}{p} \text{ if } p > 2^m \]

---

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

problem size = 1024

---

parallel efficiency

number of processes

0 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0 128 256 512 1024 1500 2000 2500 3000

---

63
Case studies

CG – conjugate gradient

IS – integer sort

\[ W \approx k_1 \left[ \frac{m}{p} \right] + k_2 \sqrt{\frac{m}{p}} + k_3 \log_2 \left( \sqrt{p} \right) \]

\[ D = T_\infty = \infty \left( 3(b+t) + 2 \left[ \frac{m}{p} \right] + p + u_1 + u_2 \right) \]

\[ E_p = \frac{D = T_\infty = \infty}{k_4} \]

\[ p \left( k_1 \left[ \frac{m}{p} \right] + k_2 \sqrt{\frac{m}{p}} + k_3 \log_2 \left( \sqrt{p} \right) \right) \]
Counting Arbitrary Affine Loop Nests

- Why affine loops?
  - Closed form representation of the loop

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

\[
n(x_0, c, g) = \arg \min_d (c^T \cdot x(d, x_0) \geq g)
\]
## Counting Arbitrary Affine Loop Nests

### Why affine loops?
- **Closed form representation of the loop**

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

- **Example**

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

n(x_0,c,g) = \arg \min_d (c^T \cdot x(d,x_0) \geq g)
```

```plaintext
x(i,x_0) = \begin{bmatrix} 1 & 0 \\ i & 1 \end{bmatrix} x_0 + \begin{bmatrix} 0 \\ 0 \end{bmatrix}

n(x_0) = \begin{bmatrix} m-k_0 \\ j_0 \end{bmatrix}
```

where

```plaintext
x_0 = \begin{bmatrix} j_0 \\ k_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \end{bmatrix}
```
Loops

- Multipath affine loops

```plaintext
x = 1;
while (x < n/p + 1) {
    y = x;
    while (y < m) {
        S1; y = 2*y;
    }
    z = x;
    while (z < m) {
        S2; z = z + x;
    }
    x = 2*x;
}
```