Don’t Parallelize!
Think Parallel & Write Clean Code

Lawrence Rauchwerger
http://parasol.tamu.edu
Texas A&M University
Parallel Computing: It’s back!

But...

Applications must be parallel to benefit from them.

If they are not then multicores are doomed.

.... Intel will not like the idea.
Why is parallel programming so hard?

Because we are used to think serially
(Cormen, Leiserson, Rivest & Stein) is the bible for all CS students
only 1 chapter on parallel algos in latest edition

Because the programmer needs to manage the cooperation of the many cores, i.e., manage concurrency

Because it is dangerous
Prone to a new kind of errors (communications/synchronizations)

Because we cannot estimate (model) parallel program performance (no good parallel computation model)
So what is a poor CS going to do?

- **Ignore the issue and pray to compiler god**
- Hide concurrency management (the parallel instructions) with low level language (e.g., OpenMP)
  - Still labor intensive
- **Use a higher level language (> C)**
  - Raise the level of abstraction
  - Focus on parallel algorithms, applications
Automatic Parallelization (aka Compiler Heaven)

- The magic solution for last 30+ years …
- Worked partially for scientific codes (C-tran)
- High hopes on speculation….

Our (latest) contribution:
- We combined static and dynamic analysis into a unified framework (*Hybrid Analysis*)
- We used logic inference to extract lightweight dynamic parallelization tests.
Hybrid Analysis (HA) on 1 slide

- Language to represent memory ref. summaries (USR) – closed under composition to prog. level.
- Translate to a language of predicates
- Set Summary (USR) \(\Rightarrow\) Predicate Language \(F(S)\)
- \(F(S)\) factorized into a cascade of predicates to be tested at runtime in the order of their complexity.
- Evaluation on 27 benchmarks
- Details in PLDI 2012 paper
HA Phase 1: Build Access Summary (USR)

Loop is Independent if RW Access Summary for A is empty

DO i = 1, N
  A(i+100) = ...
  IF (x > 0) THEN
    ... = A(i)
  ENDIF
ENDDO
Build RO, RW, WF Summaries (USR)

Summaries (RO, RW, WF) are
• constructed via a bottom-up parse of the CALL and CD graphs,
• data-flow equations dictate how to compose consecutive regions, aggregate/translate across loops/callsites, ...

Simplified solvh_do20 from dyfesm (PERFECT CLUB)

```fortran
DO i = 1, N
    CALL geteu (XE(IA(i)), NP, SYM)
    CALL matmul(XE(IA(i)), NS)
ENDDO

SUBROUTINE matmul(XE, NS)
    INTEGER NS, XE(*)
    DO j = 1, NS
        ... = XE(j) ...
        XE(j) = ...
    ENDDO
END

SUBROUTINE geteu(XE, NP, SYM)
    INTEGER NP, SYM, XE(16, *)
    IF (SYM .NE. 1) THEN
        DO i = 1, NP
            DO j = 1, 16
                XE(j, i) = ...
            ENDDO
        ENDDO
    ENDDO
END
```
Summarizing Subroutine \texttt{geteu}

\textbf{Loop Aggregation} uses (intuitively) interval arithmetic:

\begin{itemize}
  \item Loop \texttt{i}: \{16 * i + j – 1 \mid j = \{1..16\}\} \Rightarrow 16 * i + [0, 15]
  \item Loop \texttt{j}: \{16 * i + [0, 15] \mid i = \{1..NP\}\} \Rightarrow [0, 16 * NP – 1]
  \item Branches introduce predicated nodes, e.g.,\textbf{,} \begin{align*}
    WF_{S_{if}}^{XE} &= WF_{S_{geteu}}^{XE} \\
    WF_{S_{Li}}^{XE} &= [0, 16 * NP – 1] \\
    WF_{S_{Lj}}^{XE} &= 16 * i + [0, 15] \\
    WF_{S_{WF}}^{XE} &= \{16 * i + j – 1\}
  \end{align*} \quad (SYM \neq 1)
\end{itemize}

\begin{verbatim}
SUBROUTINE geteu(XE, NP, SYM)
    INTEGER NP, SYM, XE(16, *)
    IF (SYM .NE. 1) THEN
        DO i = 1, NP
            DO j = 1, 16
                XE(j, i) = ...
            END DO
        END DO
    ENDIF
END
\end{verbatim}
### Summary for Subroutine `matmul`

- **Composing read-only** $RO_{S_1}$ **and write-first** $WF_{S_2}$ **regions:**
  - $RO = RO_{S_1} - WF_{S_2}$, $WF = WF_{S_2} - RO_{S_1}$, $RW = RO_{S_1} \cap WF_{S_2}$
  - In our case $RO = 0$, $WF = 0$, $RW = \{j - 1\}$

- **Over loop** $DO$ $j$:
  - $RO_{loop} = 0$, $WF_{loop} = 0$, $RW_{loop} = [0, NS - 1]$
Summary Access for Target Loop

**RW summary for loop**

```plaintext
DO i: RW^i = ?
```

```plaintext
INTEGER NS, NP, IA(*), XE(*)
```

**S_loop**

```plaintext
DO i = 1, N
```

**S_WF**

```plaintext
CALL geteu (XE(IA(i)), NP, SYM)
```

**S_RW**

```plaintext
CALL matmul (XE(IA(i)), NS)
```

ENDDO

```
S_WF \mean\ S_RW = \{\emptyset, WF^i = WF_{geteu}, RW^i\}
```

- Composing write-first \(WF^i_{S_1}\) and read-write \(RW^i_{S_2}\) regions:

\[
RO^i = \emptyset, WF^i = WF^i_{S_1}, RW^i = RW^i_{S_2} - WF^i_{S_1}
\]

- In our case \(RO^i = \emptyset, WF^i = WF^i_{geteu}\) not shown, \(RW^i\) as above
Summary (USR) Based Independence Equations

Flow and Anti Independence Equation for loop of index i:

\[ S_{\text{find}} = \left\{ \left( \bigcup_{i=1}^{N} WF_i \right) \cap \left( \bigcup_{i=1}^{N} RO_i \right) \right\} \cup \]
\[ \left\{ \left( \bigcup_{i=1}^{N} WF_i \right) \cap \left( \bigcup_{i=1}^{N} RW_i \right) \right\} \cup \]
\[ \left\{ \left( \bigcup_{i=1}^{N} RO_i \right) \cap \left( \bigcup_{i=1}^{N} RW_i \right) \right\} \cup \]
\[ \left\{ \bigcup_{i=1}^{N} (RW_i \cap (\bigcup_{k=1}^{i-1} RW_k)) \right\} = \emptyset \]  (1)

Output Independence Equation for loop of index i:

\[ S_{\text{oind}} = \left\{ \bigcup_{i=1}^{N} (WF_i \cap (\bigcup_{k=1}^{i-1} WF_k)) \right\} = \emptyset \]  (2)

\( S_{\text{find}} \) and \( S_{\text{oind}} \) exact computation – NOT necessary, too expensive

Loop Independence: when are \( S_{\text{find}} \) and \( S_{\text{oind}} \) empty?
HA Phase 2: Summary to Predicate Translation

Translation Scheme $\mathcal{F}$ from summary to predicate language:

$\mathcal{F}$ : Summary $\Rightarrow$ Predicates $\quad S = \emptyset \iff F(S)$

Predicates constructed via a top-down pass of the indep. summary.

Translating the Flow-Independence Summary for XE in solh_do20

- Assume $S_{\text{ind}} = 0$ indep. equation:
  \[ \bigcup_{i=1}^{N} (RW_i \cap (\bigcup_{k=1}^{i-1} RW_k)) = \emptyset \]

- $\bigcup_{i=1}^{N} S_i = \emptyset \iff (\bigwedge_{i=1}^{N} F(S_i)) \vee \ldots$

- $F (RW_i \cap (\bigcup_{k=1}^{i-1} RW_k)) = ?$

- $S_1 \cap S_2 = \emptyset \iff F(S_1) \vee \ldots$

- $F(RW_i) = ?$
A sufficient XE-independence condition is:

\[
F \left( \bigcup_{i=1}^{N} (RW_i \cap \bigcup_{k=1}^{i-1} RW_k) \right) = \bigwedge_{i=1}^{N} (SYM \neq 1 \land NS \leq 16 \times NP)
\]
Predicate Simplification/Extraction Techniques

Predicate Simplifications:

- **Invariant Hoisting:**
  \[ \bigwedge_{i=1}^{N} \left( \bigvee \left( A_{1}^{inv}, \ldots, A_{r}^{inv}, B_{1}^{var}, \ldots, B_{p}^{var} \right) \right) \rightarrow \left( \bigvee \left( A_{1}^{inv}, \ldots, A_{r}^{inv} \right) \right) \vee \left( \bigwedge_{i=1}^{N} \left( \bigvee \left( B_{1}^{var}, \ldots, B_{p}^{var} \right) \right) \right) \]

- **With our example:**
  \[ \bigwedge_{i=1}^{N} \left( \text{SYM} \neq 1 \land \text{NS} \leq 16 \ast \text{NP} \right) \rightarrow \text{SYM} \neq 1 \land \text{NS} \leq 16 \ast \text{NP} \]

  which is O(1) runtime.

Predicate Extraction:

- To extract O(1) predicates, apply aggressive invariant rules:
  \[ \bigwedge_{i=1}^{N} \left( \bigwedge \left( A_{1}^{inv} \lor B_{1}^{var}, \ldots, A_{p}^{inv} \lor B_{p}^{var} \right) \right) \rightarrow \bigwedge \left( A_{1}^{inv}, \ldots, A_{p}^{inv} \right) \]

- To extract O(N) predicates: replace inner-loop nodes with \texttt{false} and simplify the result.
Enabling Transformations for Predicate Extraction

Summary Transformations:

- “Strength Reduction”: union preferred to repeated subtraction,
- Preserving the shape of a Union of Mutually Exclusive Gates.

*a) Reshaping Repeated Subtraction*

Assume mutually exclusive gates \( C_1, C_2, C_3 \) and \( C = C_1 \lor C_2 \lor C_3 \)

\[
X = \bigcup \begin{array}{c}
C_1 \downarrow \quad S_1 \\
C_2 \downarrow \quad S_2 \\
C_3 \downarrow \quad S_3 
\end{array}
\]

\[
Y = \begin{array}{c}
\bigcup \quad S_6 \\
C_1 \downarrow \quad S_4 \\
C_2 \downarrow \quad S_5 
\end{array}
\]

\[
C = \begin{array}{c}
\bigcup \quad S_6 \\
C_1 \downarrow \quad S_6 - S_3 \\
C_2 \downarrow \quad S_6 - S_3 \\
C_3 \downarrow 
\end{array}
\]

*b) Preserving the shape of an Union Of Mutually Exclusive Gates*
HA Phase 3: Run Time

- **O(1) Scalar Comparison**
  - Pass
  - Fail

- **O(n/k) Comparisons**
  - Pass
  - Fail

- **Reference Based**
  - Pass
  - Fail

- **Independent**
- **Dependent**
Experimental Setup

- TAMU variant Polaris compiler
- F77 Source to (source + OpenMP)
- Two architectures:
  - Smaller data-set Perfect-Club and SPEC89/92 vs. with INTEL’s ifort 11.1 (-O2 –ipo –parallel) on a commodity INTEL quad-core Q9550@2.83GHz machine with 8GB memory,
  - Larger data-set SPEC2000/2006 vs. xlf_r.13 (-O4 –qsmp=auto) on a 8 dual-core POWER5+@1.9GHz, 32GB-memory machine.
Normalized Parallel Timing on a Quad-Core. Sequential Time is 1, compiler options -O2 -ipo

Normalized Parallel Timing on 4 Processors. Sequential Time is 1. Compiler Option -O2 -ipo.


Perfect Club Suite
- flo52
- arc2d
- dyfesm
- mdg
- trfd
- spec77
- ocean

SPEC89/92 Suites
- tomcatv
- mdljdp2
- hydro2d
- ora
- matrix300

Spec2000 and 2006 Suites
- WUPWISE
- APSI
- MGRID
- BWAVES
- GROMACS
- CALCULIX
Speedup Scalability Up To 16 Procs for SPEC2000 and 2006 Suites

SPEC2000 and SPEC2006 Benchmarks

- APSI
- WUPWISE
- MGRID
- APPLU
- BWAVES
- SWIM
- ZEUSMP
- GROMACS
- CALCULIX

Speedup:

- 1 Proc
- 2 Proc
- 4 Proc
- 8 Proc
- 16 Proc

Values:

- APSI: 5.83 (1 Proc), 12.64 (2 Proc), 13.07 (8 Proc), 8.06 (16 Proc)
- WUPWISE: 1.57 (2 Proc), 8.95 (4 Proc), 9.29 (8 Proc), 9.45 (16 Proc)
- MGRID: 11.21 (4 Proc), 12.64 (8 Proc)
- APPLU: 1.57 (2 Proc), 11.21 (8 Proc)
- BWAVES: 9.29 (8 Proc), 9.45 (16 Proc)
- SWIM: 8.06 (16 Proc)
- ZEUSMP: 9.29 (8 Proc), 9.45 (16 Proc)
- GROMACS: 8.06 (16 Proc)
- CALCULIX: 9.29 (8 Proc), 9.45 (16 Proc)
Is this good enough?

- 26 Perfect-Club and SPEC benchmarks: 2100 loops,
- Measured 380 loops representing 92% seq. coverage,
- On 4 procs: Max. speedup 4.5 ; Avg. 2.4
- On 8 procs: Max. speedup 8.4 ; Avg. 5.4

- Predicate evaluation: negligible runtime overhead.

- Commercial compiler technology slow to adopt
So .....  

How did we get here?
• Heroic Efforts (see paper in PLDI 2012, Beijing)

What did we accomplish?
• Only C-tran code !!!
• Commercial compilers move slowly – too bad!
  (See D. Padua’s talk)
• Compilers work only with an algo *implementation*
• Will not scale because …they reverse engineer
  **Compilers cannot create parallelism!**
  Only programmers can
How else?

**Use a higher level language (> C)**
- Raise the level of abstraction
- Focus on parallel algorithms, applications
- Create parallelism *ab initio*. 
Our Approach: STAPL

High-level Programming Model.

- High Level of Abstraction ~ similar to C++ STL
- **Fine grain** expression of parallelism – can be coarsened
- Implicit parallelism – Serialization is explicit
- Distributed Memory Model (PGAS)
- Algos are Data Flow Graphs specified with Data Dependence Patterns
  - Asynchronous execution with granularity control
  - Library of Dependence Patterns
STAPL: Standard Template Adaptive Parallel Library

A library of parallel components that adopts the generic programming philosophy of the C++ Standard Template Library (STL).

**STL**
- **Iterators** provide abstract access to data stored in *Containers*.
- **Algorithms** are sequences of instructions that transform the data.

**STAPL**
- **pViews** provide abstracted access to distributed data stored in *pContainers*.
- **pAlgorithms** spec-ed by **PARAGRAPHS**, parallel task graphs that transform the input data.
  - Can use existing PARAGRAPHS, defined in collection of common **parallel patterns**.
  - **Extensible** - users can define new patterns.
Example - STAPL Inner Product

array<int> arr1(100);
array<int> arr2(100);
view_1D view1(arr1);
view_1D view2(arr1);

x = inner_product(view1, view2);

In specification, user can be unaware of data distribution and locality.
pAlgorithms are PARAGRAPHS

inner_product(View1 v1, View2 v2) {
    return map_reduce(
        multiplies(), plus(),
        v1, v2
    );
}

• inner_product() specified by PARAGRAPHS.
• Employs map_reduce parallel pattern.
• Defines a new pattern we can use to compose a nested PARAGRAPHS.
PARAGRAPh Composition

Matrix Vector Multiplication

matvec(View2D A, View1D x) {
  using functional::inner_product;
  return map_func(inner_product(), full_overlap(x), A.rows());
}

View transformations and PARAGRAPh reuse in composition enable an exact, succinct specification of matvec task graph.
Example: NAS CG in STAPL

cg_iteration(View2D A, View1D p, Ref rho, ...) {
    q       = A * p;
    alpha   = rho / inner_product(q, p);
    new_z   = z + alpha * p;
    new_r   = r - alpha * q;
    new_rho = inner_product(new_r, new_r);
    beta    = new_rho / rho;
    new_p   = new_r + beta * p;
    ...
}

- Operator overloads call pAlgorithms: A * p \rightarrow \text{matvec}(A, p)
- Sequence composition is non blocking:
  Specification proceeds concurrently with execution.
- For simplicity / space, we next consider just the first two statements.
Example: Sequence Composition - CG

Matvec() pAlgorithm on 2D_view of pMatrix and 1D_view of pArray.

\[ q = A \cdot p; \]

Inner product of two 1D_view views whose scalar result is divisor of dividend rho.

\[ \alpha = \frac{\rho}{\text{inner_product}(q, p)}; \]

Expressive syntax quickly yields large, hierarchical PARAGRAPHS.
NAS CG Results - Scalability

STAPL implementation versus NAS MPI implementation.

CRAY6

Hopper @ NERSC.
Cray XE6, AMD ManyCours.
6,384 compute nodes.
24 cores, 32GB per node.
‘Gemini’ 3D torus interconnect.
PDT Scalability

LLNL Opteron cluster
- 4 quad-core AMD 2.3GHz processors per node
- InfiniBand DDR (Mellanox)
- 32GB RAM per node

Model
- Simple estimate of execution time
  Time = Computation + Communication
- Uses seq. exec. time for computation
- MPI bandwidth/latency measured at execution startup
- Assumes messages processed immediately

Input
- Models a tall box 4:1 height to width
- Number of unknowns/core constant
- Code matches 94% efficiency predicted by model at 4,096 cores
Parallel languages and libraries are available. But how far can they get us? Not very far unless …
We will not get very far unless we ...

Think parallel
No Language/Library is substitute for parallel algos
Autoparallelizing Compilers can do even less

Write Clean Parallel Code: Use high level language
- important for productivity, market penetration
- Programmer needs parallelism awareness

Develop Performance models for parallel systems
• For both programmers, algo writers and compilers