# Two Roads to Parallelism: Compilers and Libraries

Lawrence Rauchwerger

Parasol
Smarter computing.
Texas A&M University

# Parallel Computing

Parasol

- It’s back (again) and ubiquitous
- We have the hardware (multicore …. petascale)
- Parallel software + Productivity: not yet…
- And now ML needs it …

Our Road towards a productive parallel software development environment

# For Existing Serial Programs

## Previous Approaches

- Use Instruction Level Parallelism (ILP): HW + SW
  - compiler (automatic) BUT not scalable
- Thread (Loop) Level (Data) Parallelism: HW+SW
  - compiler (automatic) BUT insufficient coverage
  - manual annotations more scalable but labor intensive

## Our Approach

- Hybrid Analysis: A seamless bridge of static and dynamic program analysis for loop level parallelization
  - USR - a powerful IR for irregular application
  - Speculation as needed for dynamic analysis

# For New Programs

## Previous Approaches

- Write parallel programs from scratch
- Use parallel language, library, annotations
- Hard Work !

## Our Approach

- STAPL: Parallel Programming Environment
  - Library of parallel algorithms, distributed containers, patterns and run-time system
  - Used in PDT, an important app for DOE & Nuclear Engineers, influenced Intel's TBB
  - …and perhaps similar to Tensorflow

# Parallelizing Compilers

## Auto-Parallelization of Sequential Programs

- Around for 30+ years: UIUC, Rice, Stanford, KAI, etc.
- Requires complex static analysis + other technology
- Not widely adopted

## Our Approach

- Initially:  speculative parallelization
- Better: Hybrid Analysis is best of both: static + dynamic
- Aspects of these techniques used in mainstream compilers and STM based systems.
- Excellent results – Major Effort – Don’t try at home

# Static Data Dependence Analysis: An Essential Tool for Parallelization

Parasol

The Question: Are there cross iteration dependences?

- Equivalent to determining if system of equations has integer solutions
- In general, undecidable – until symbols become numbers (at runtime)

## Linear Reference Patterns

- Solutions restricted to linear addressing and control (mostly small kernels)
  - Geometric view: Polytope model
    - Some convex body contains no integral points
  - Existential solutions: GCD Test, Banerjee Test, etc
    - Potentially overly conservative
  - General solution: Presburger formula decidability
    - Omega Test: Precise, potentially slow

```
DO j = 1, 10
 a(j) = a(j+40)
ENDDO
```

$$\begin{cases} 1 \le j_w \le 10 \\ 1 \le j_r \le 10 \\ j_w \ne j_r \\ j_w = j_r + 40 \end{cases}$$

## Nonlinear Reference Patterns

- Common cases: indirect access, recurrence without closed form
- Approaches: Linear Approximation, Symbolic Analysis, Interactive

```
DO j = 1, 10
 IF (x(j)>0) THEN
  A(f(j)) = …
 ENDIF
ENDDO
```

# Run-time Dependence Analysis: Speculative Parallelization

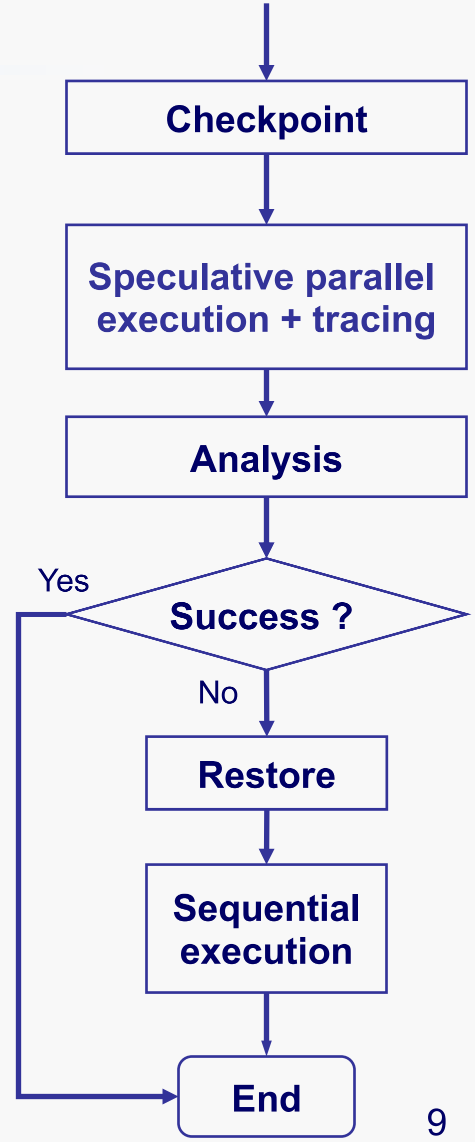
Parasol

**Problem:**

```
FOR i = …
  A[W[i]] = A[R[i]] + C[i]
```

**Main Idea:**

- Speculatively execute the loop in parallel and record reference in private shadow data structures
- Afterwards, check shadow data structures for data dependences
  - if  no dependences loop was parallel
  - else re-execute safely (loop not parallel)




**Cost:**

- Worst case: proportional to data size

# Hybrid Analysis

| | Compile-time Analysis | **Hybrid Analysis** | Run-time Analysis |
|---|---|---|---|
| STATIC (compiler) | Symbolic analysis | Symbolic analysis<br>Extract conditions | |
| DYNAMIC (run-time) | | Evaluate conditions | Full reference-by-reference analysis |

**PROs**
- No run-time overhead

**CONs**
- Conservative when
  - Input/computed values
    - Indirection, Control
  - Weak symbolic analysis
    - Complex recurrences
- Impractical Combinatorial explosion

**PROs**
- Always finds answers
- Minimizes runtime overhead

**CONs**
- More Complex static analysis

**PROs**
- Always finds answers

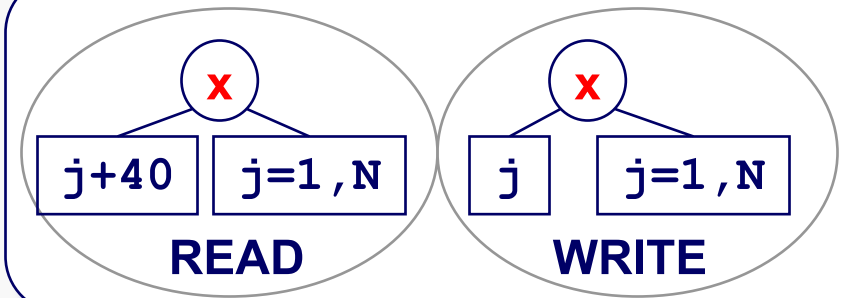
**CONs**
- Run-time overhead
- Ignores compile-time analysis

# Hybrid Analysis

## Compile-time Phase

```
DO j=1,N
 a(j)=a(j+40)
ENDDO
```

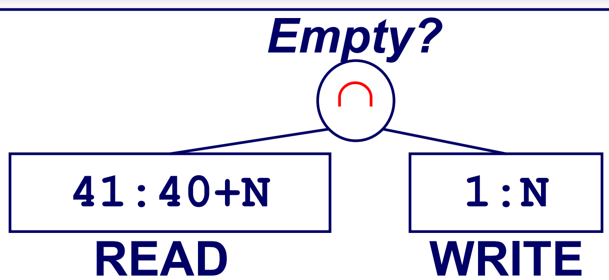
*Under what conditions can the loop be executed in parallel?*




1. Collect and classify memory references.

| READ | WRITE |
|---|---|
| `41:40+N` | `1:N` |

2. Aggregate them symbolically




3. Formulate independence test.

4.a) If we can prove $1 \le \mathbf{N} \le \mathbf{40}$
Declare loop parallel.

4.b) If **N** is unknown,
Extract run-time test. $\mathbf{N} \le 40$

# Hybrid Analysis

## Run-time Phase

```
DO j=1,N
 a(j)=a(j+40)
ENDDO
```

*Execute the loop in parallel if possible.*

4.a) If we can prove $1 \leq$ **N** $\leq$ **40**, Declare loop parallel.

4.b) If **N** is unknown, Extract run-time test. $N \leq 40$

*Compile Time*

*Run Time*

**Parallel Loop**

```
DO PARALLEL j=1,N
 a(j)=a(j+40)
ENDDO
```

No run-time tests performed if not necessary!

**Run-time Test**

**Parallel Loop**

**Sequential Loop**

```
IF (N ≤ 40) THEN
 DO PARALLEL j=1,N
  a(j)=a(j+40)
 ENDDO
ELSE
 DO j=1,N
  a(j)=a(j+40)
 ENDDO
ENDIF
```

# Hybrid Analysis: a slightly deeper dive

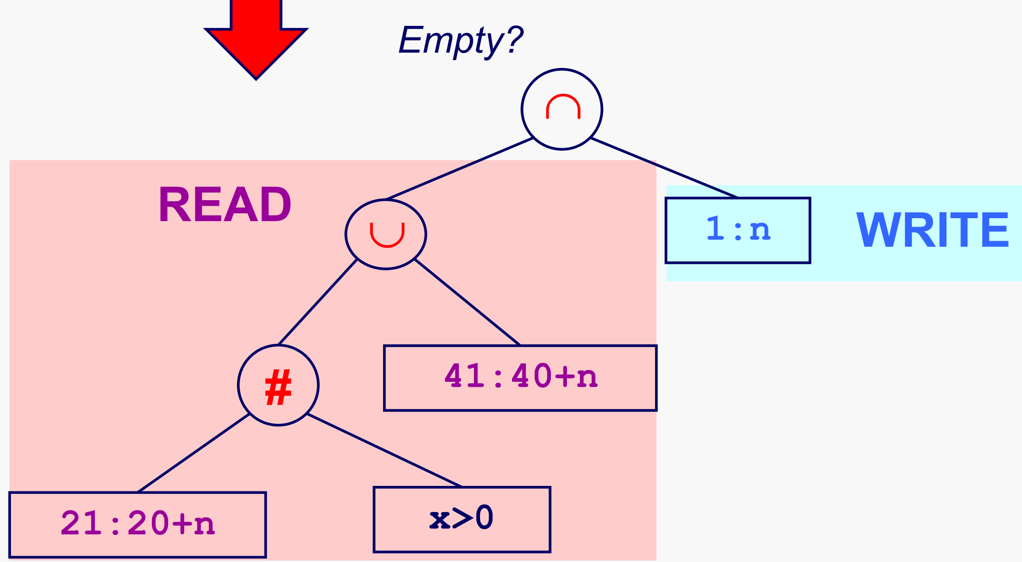
WRITE

```
DO j = 1, n
  A(j) = A(j+40)
  IF (x>0) THEN
    A(j) = A(j) + A(j+20)
  ENDIF
ENDDO
```

READ

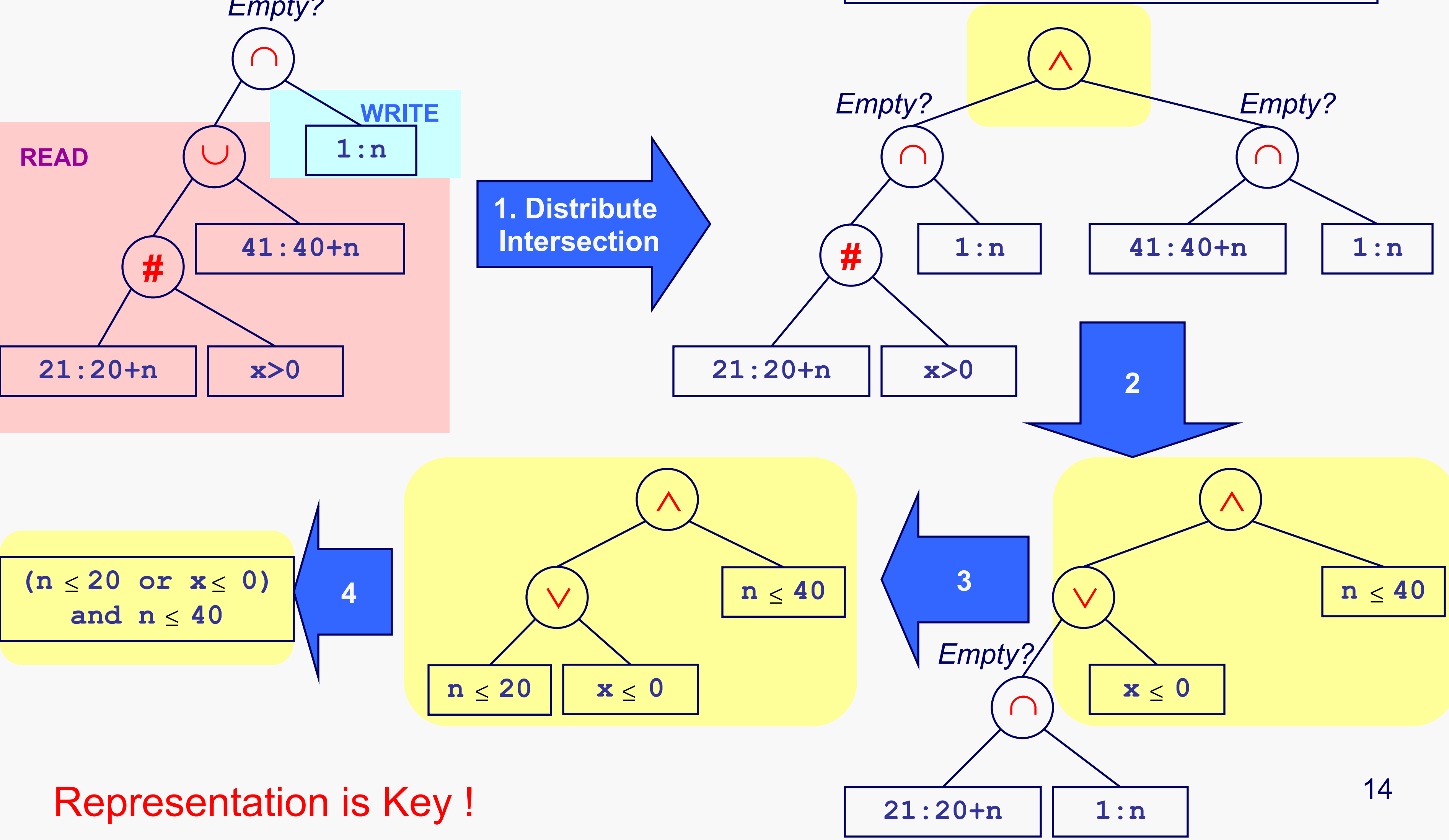
***READ* ∩ *WRITE* = Empty?**

Program Level Representation of References (**USR**)



# Set expression to Logic expression

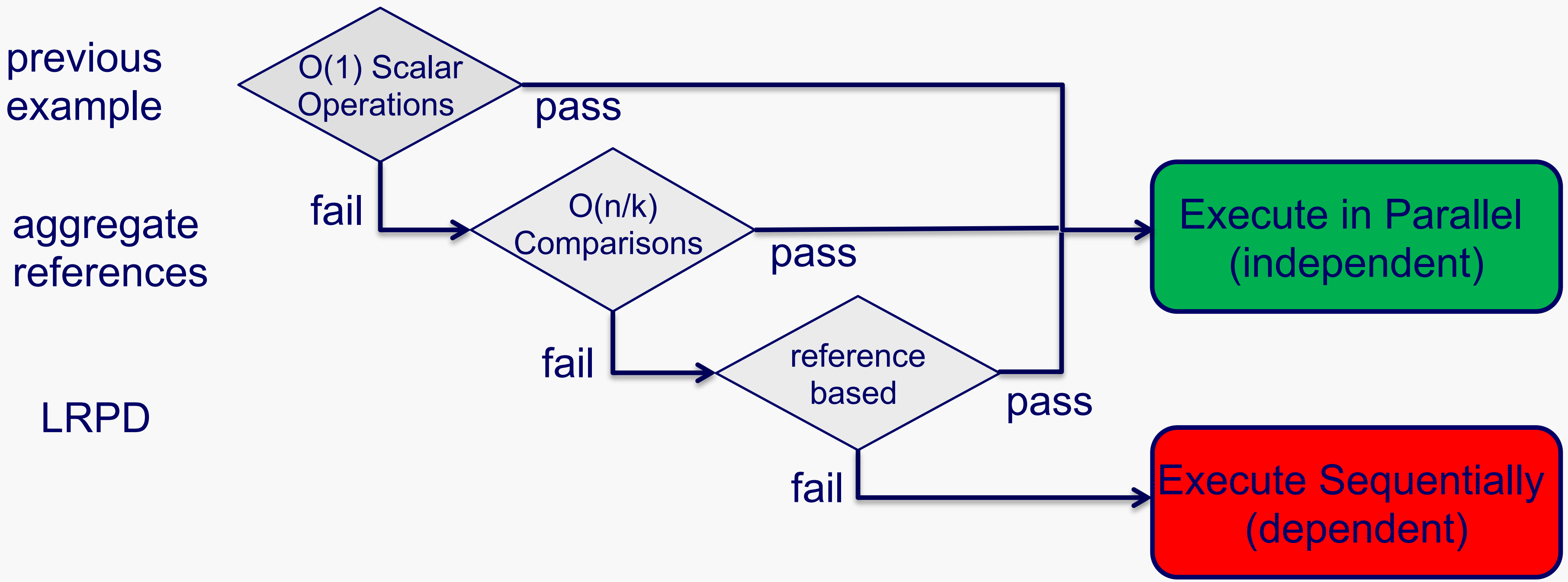
```
DO j = 1, n
  A(j) = A(j+40)
  IF (x>0) THEN
    A(j) = A(j) + A(j+20)
  ENDIF
ENDDO
```



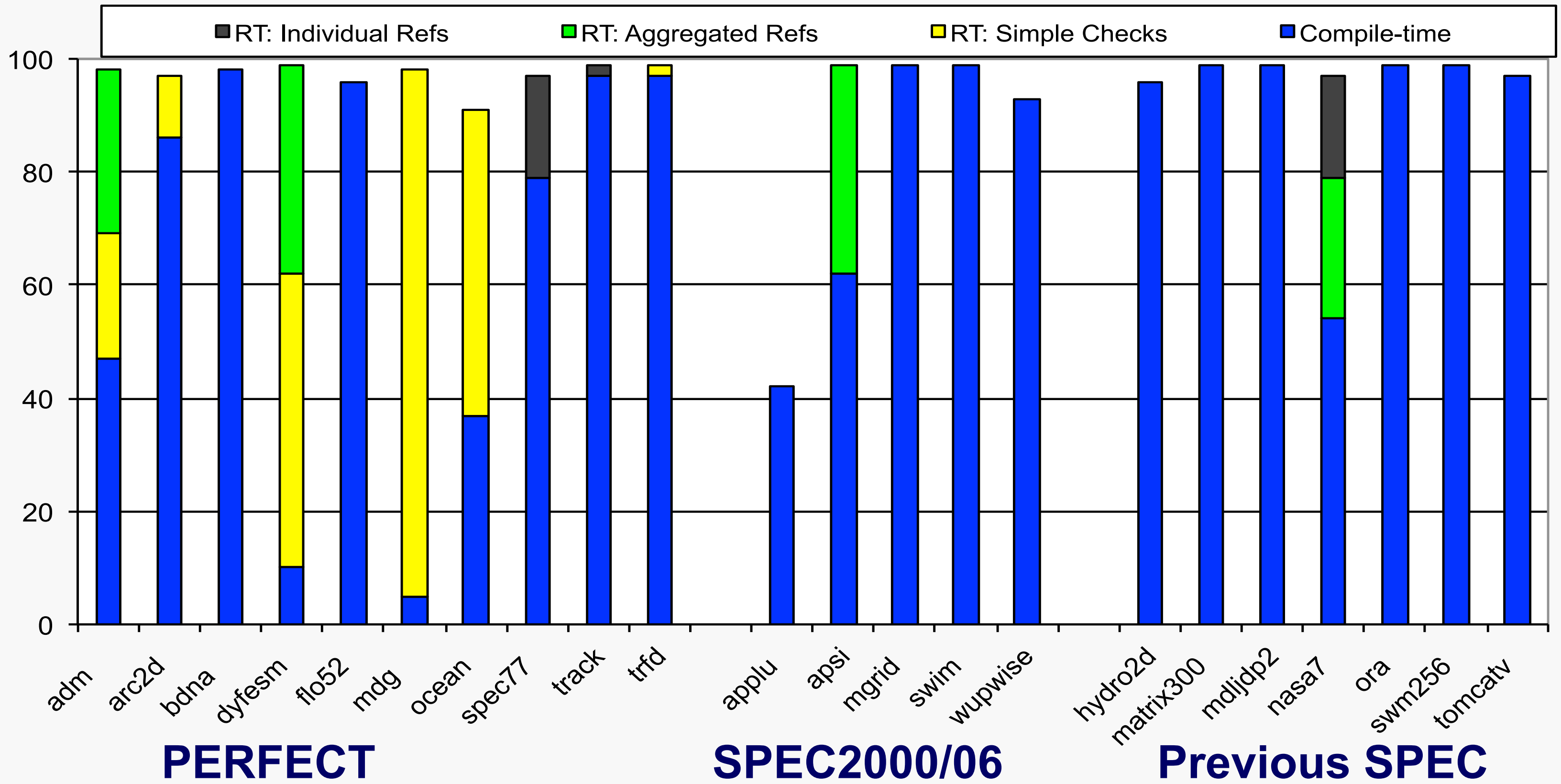

Representation is Key !

# Hybrid Analysis Strategy

Independence conditions factored into a series of sufficient conditions tested at runtime in the order of their complexity

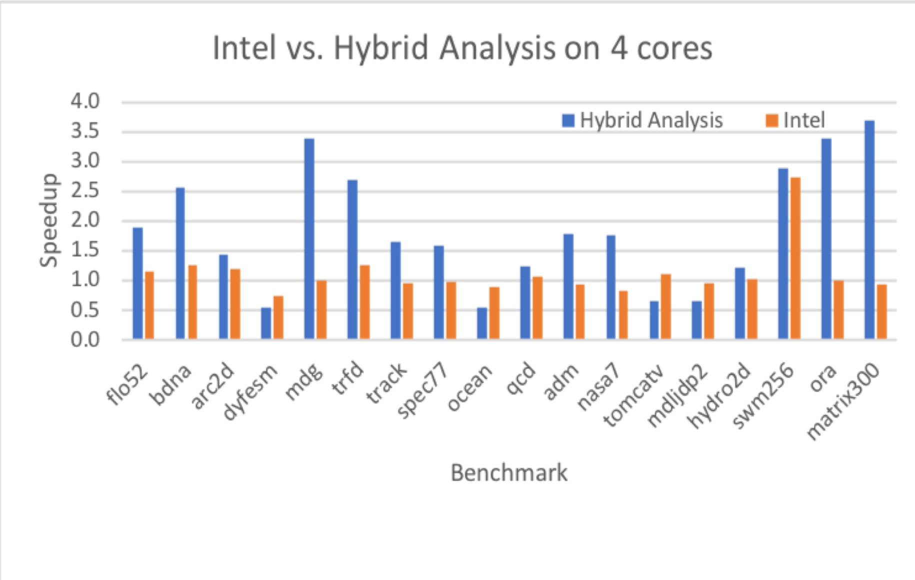


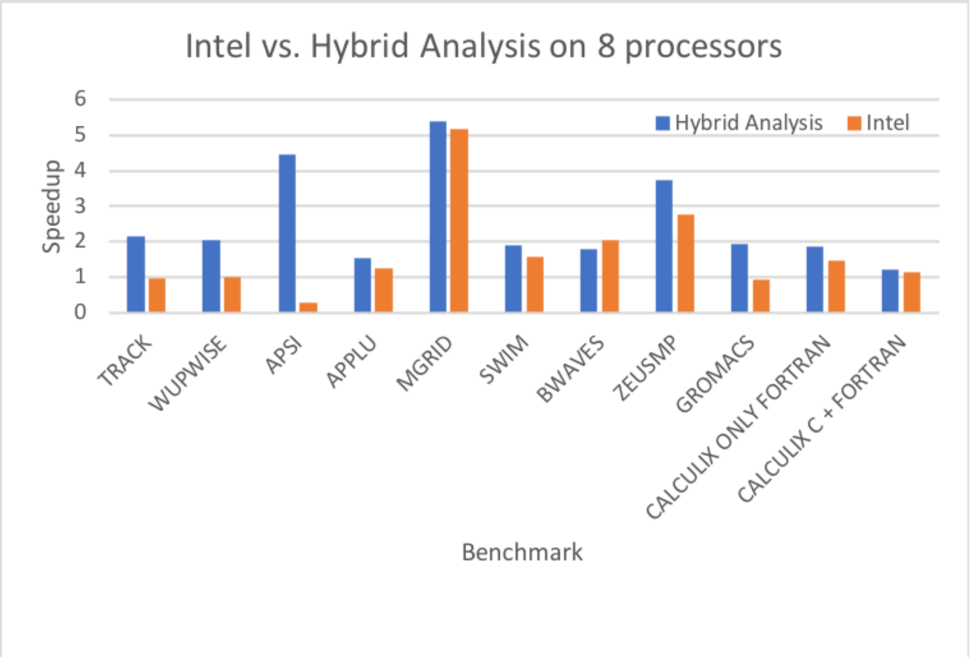
# Hybrid Analysis Parallelization Coverage




- Parallelized 380 loops of 2100 analyzed loops: **92% seq. coverage**

# Speedups: Hybrid Analysis vs. Intel ifort




- Older Benchmarks with smaller datasets on 4 cores only
- Better performance on 14/18 benchmarks on 4 cores
- Better performance on 10/11 benchmarks on 8 cores

# So….

- What did we accomplish?
  - Full Parallelization of C-tran codes (28 benchmarks at >90% coverage)
  - A IR representation & a technique

- We cannot declare victory because:
  - Required Heroic Efforts
  - Commercial compilers adopt slowly
  - **Compilers cannot create parallelism -- only programmers can!**

# How else?

First

- Think Parallel!

Then

- Develop parallel algorithms
- Raise the level of abstraction
- Use algorithm level (not only) abstraction

→

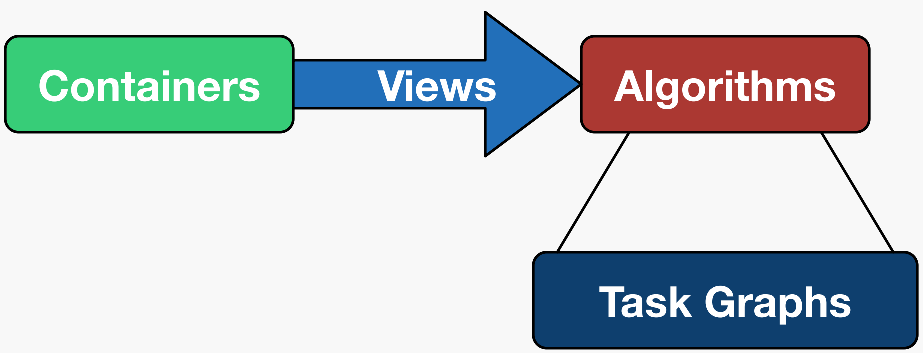
- Expressivity + Productivity
- Optimization can be compiler generated

# **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**
  - ***Views*** provide abstracted access to distributed data stored in ***Distributed Containers***.
  - ***Parallel Algorithms*** specified by **Skeletons**
    - Run-time representation is Task Graph

# STAPL Components

High Level of Abstraction ~ similar to C++ STL

Task & Data parallelism: Asynchronous

- Parallelism (SPMD) implicit – Serialization explicit
- imperative + functional: Data flow+Containers

SPMD Programs defined by

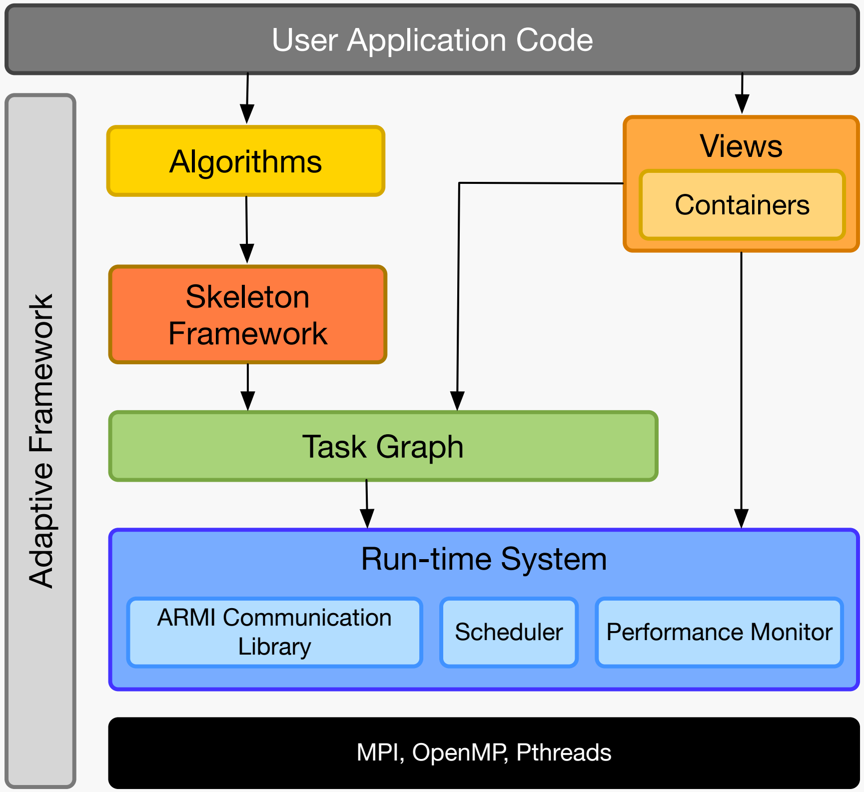
- Data Dependence Patterns ➔ Skeletons
  - Composition: parallel, serial, nested, …
- Tasks: Work function & Data
  - **Fine grain** tasks (coarsened)
  - Data in distributed containers

Execution Defined by:

Data Flow Graphs (Task Graphs)

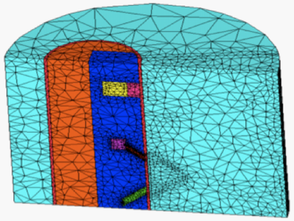
Execution policies: scheduling, asynchrony..

Distributed Memory Model (PGAS)

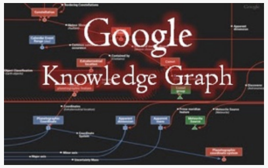


# The STAPL Graph Library (SGL)

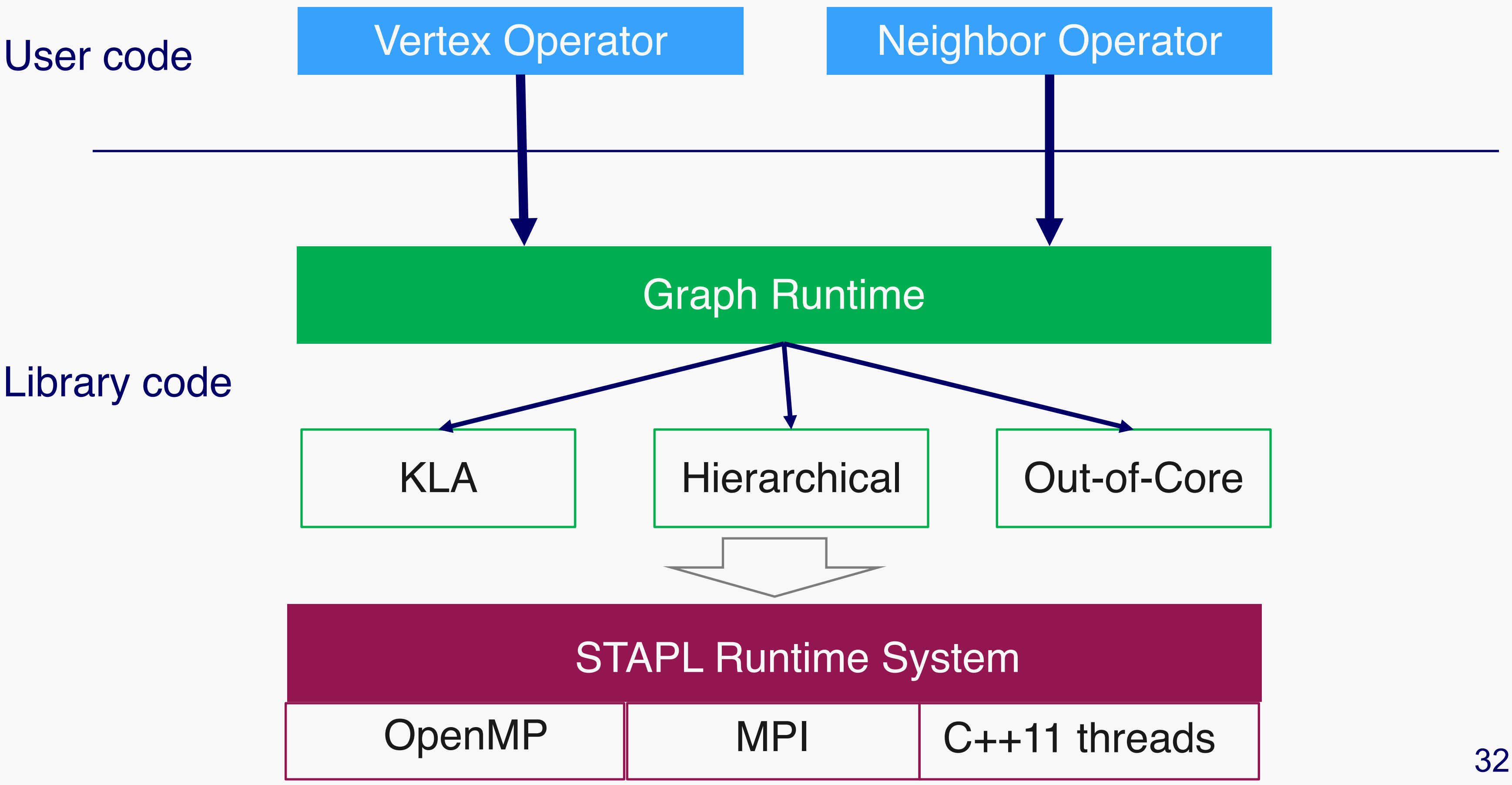
- Many problems are modeled using graphs:
  - Web search, data-mining (Google, Youtube)
  - Social networks (Facebook, Google+, Twitter)
  - Geospatial graphs (Maps)
  - Scientific applications



- Many important graph algorithms:
  - Breadth-first search, single-source shortest path, strongly connected components, k-core decomposition, centralities

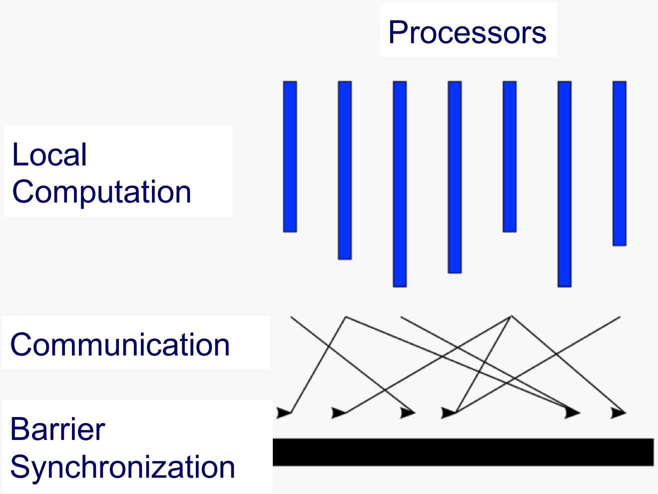


# SGL Programming Model

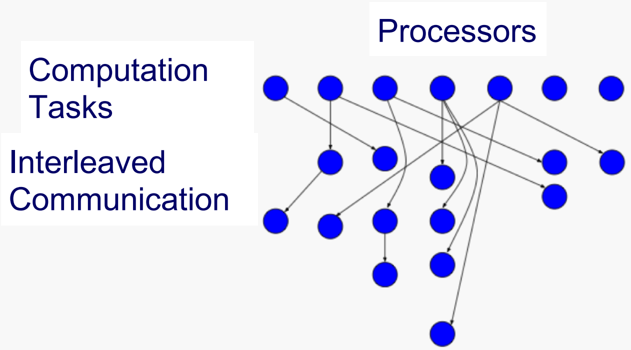


# Parallel Graph Algorithms May Use

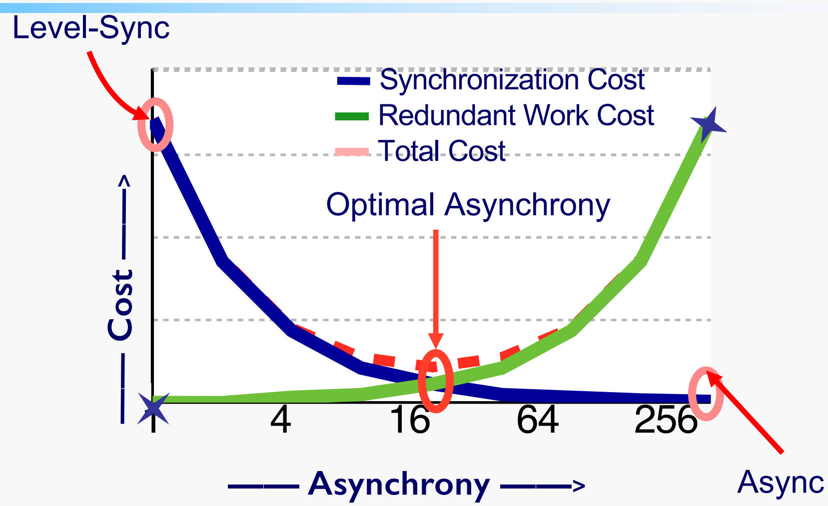
- Level-Synchronous Model
  - BSP-style iterative computation
  - Global synchronization after each level, no redundant work




- Asynchronous Model
  - Asynchronous task execution
  - Point-to-point synchronizations, possible redundant work



# Having Your Cake and Eating it Too

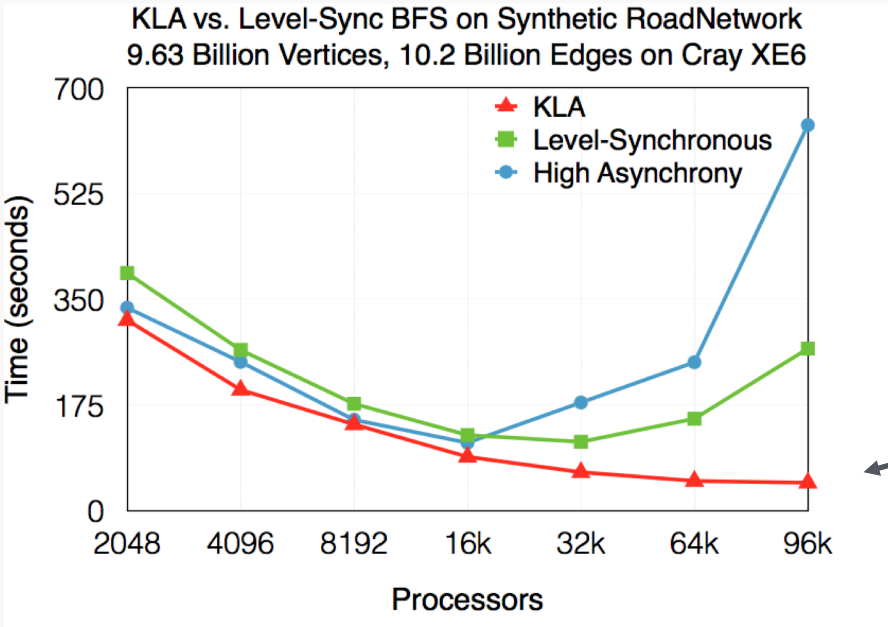



Asynchrony = $\frac{\text{levels}}{\text{supersteps}}$

k-Level Asynchronous Model

- k defines depth of superstep (KLA-SS)
- Unifies existing models
  - k=1: Level-synchronous
  - k=d: Asynchronous

# k-Level Asynchronous (KLA) BFS

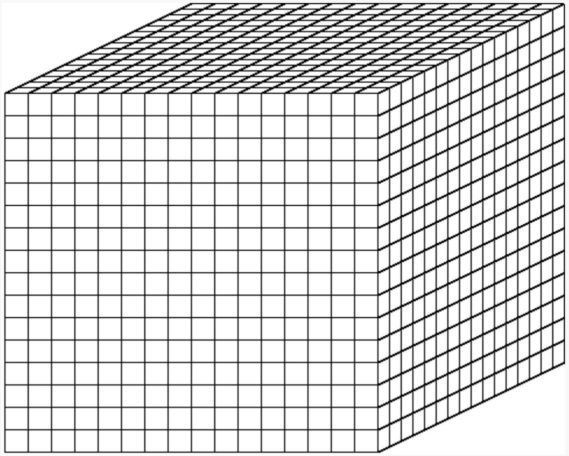



Diameter = 3218
k = 9
KLA-SS = 358

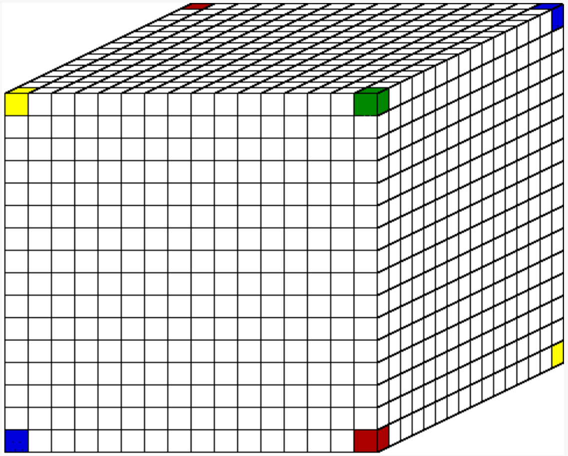
- Other strategies stop scaling after 32,768 cores
- KLA strategy faster, scales better
- Adaptively change asynchrony to balance global-synchronization costs and asynchronous penalty

# PDT:
# Application Development with STAPL

- Compute flow of subatomic particles across a spatial domain
- Discretized spatial domain represented using pGraph
- Iterative algorithm (e.g., GMRES) iterates until particle flow in space, direction, and energy level stabilizes.
  - Matrix-vector multiplication is 90% of execution time and is implemented as sweep of spatial domain in all directions.
  - Each sweep is a task graph

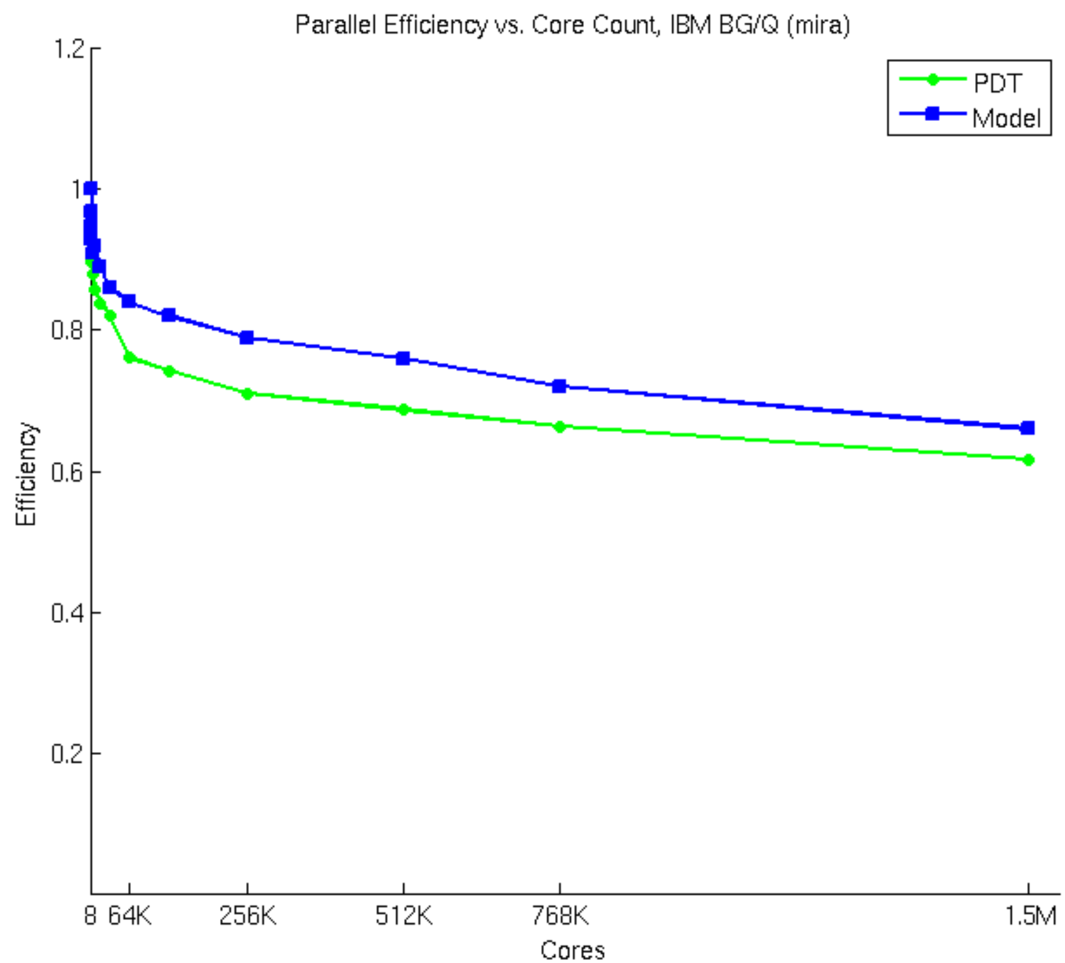


One sweep



Eight simultaneous sweeps

# Particle Transport in STAPL




Experiment keeps number of unknowns per processor constant.

PARAGRAPH size and communication increases with processor count.

Model assumes immediate processing of messages

# Conclusions: What did we accomplish? What did we learn?

- Auto - parallelization: Major Effort
  - 28 benchmarks parallelized with good coverage
  - Possible but very hard
  - Autopar: Extracts but does not create parallelism
  - Technology can be (re)used in other areas (TF compilation)

- STAPL for new parallel programs (e.g., TF)
  - New(ish) asynchronous algorithms (Data Flow, ..)
  - Distributed environment (containers, Data Flow Graph)
  - Adaptive environment & polymorphism

Avenues are complementary

- Legacy Code: Parallelization may be a good idea
- Always: Think Parallel & Write Clean Code

STAPL on https://gitlab.com/parasol-lab/stapl
and several National Labs repos

# Why is this relevant ?

- From obsolescence to point technology – just wait 10 years

- Static & Dynamic Array reference analysis – Basis for ML optimizing transformations – Tensors ~ n-dim arrays

- STAPL design facilitates: Compose and Conquer
  - Programs = Skeleton Composition
  - Global properties = Component Property Composition
    - Correctness, performance models, approximation, fault tolerance, energy
- Compile the composition (fuse TF components)

Parasol

# Questions?