

Center for Exascale Radiation Transport

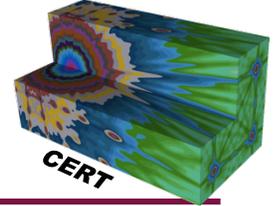
Toward Exascale Computing with STAPL

Lawrence Rauchwerger

Parasol Lab, Computer Science and Engineering

PSAAP II Kick-off Meeting, Dec. 9 – Dec. 10, 2013

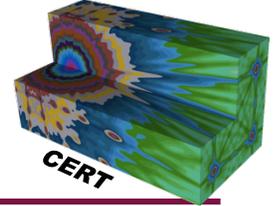
Goal: Exa-code for Parallel Deterministic Transport



1. State of the Parallel Det. Transport code (PDT): Peta
2. Intro to STAPL – a C++ high level parallel library
3. Roadmap from Peta to Exa
4. Taking STAPL from Peta to Exa
5. Building an Exa PDT on top of STAPL using TAXI
6. Beyond PDT: Our contributions to Comp Sci at Exascale

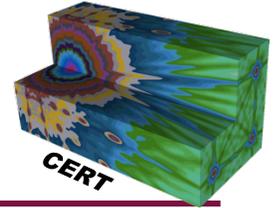


Exa – Peta ... where is PDT now ?



- Previously developed PDT using PTTL library
 - *Scale to 128k cores*
- Currently developing PDT using STAPL library
 - *STAPL is general purpose parallel library*
 - *75 K LOC in PDT only*
 - *Scales upto 393k processors on BG/Q (Sequoia)*
- Exploit space (geometry) level parallelism
 - *Sequence of parallel sweeps across the *rectangular* grids with Pipelined directions*
 - *Asynchronous (step wise) communication*
 - *No fault tolerance*
 - *Homogeneous computer system (BG/Q) – no accelerators*
- Our software environment : STAPL ...

STAPL: Standard Template Adaptive Parallel Library



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



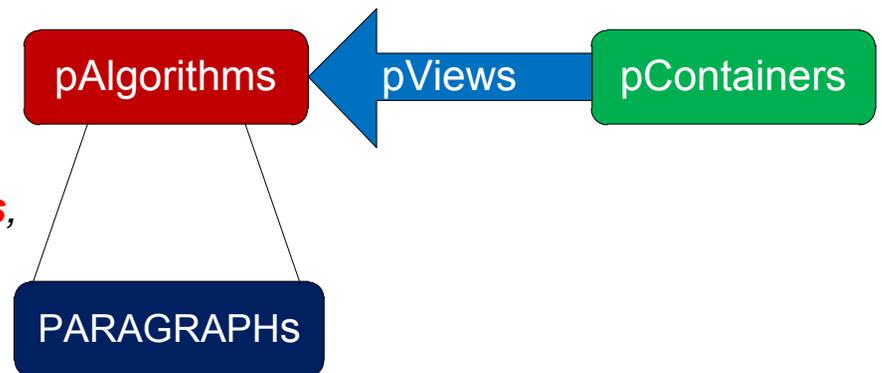
➤ STL

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

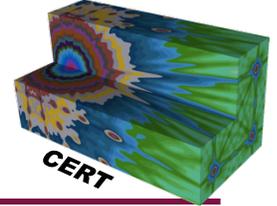


➤ STAPL

- **pViews** provide abstracted access to distributed data stored in **pContainers**.
- **pAlgorithms** specified 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.



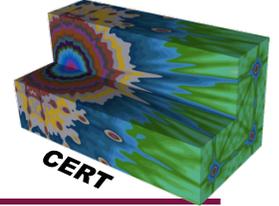
Programming Model with STAPL



➤ STAPL Programming Model.

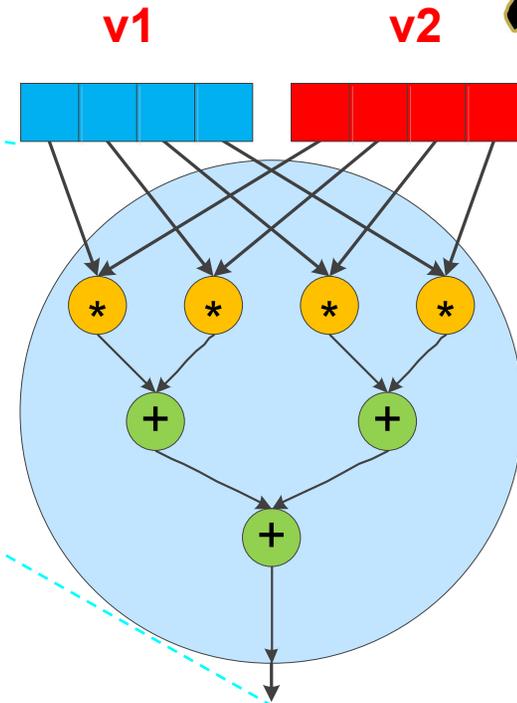
- *High Level of Abstraction ~ similar to C++ STL*
- **Fine grain** expression of parallelism – can be coarsened
- *Implicit parallelism – Serialization is explicit*
- *Distributed Memory Model (PGAS)*
- *Algorithms defined by*
 - Data Dependence Patterns (Library)
 - Distributed containers
 - Execution policies (scheduling, data distributions, etc)
- *Algorithm run-time representation: Task Graphs (PARAGRAPHS)*

pAlgorithms are PARAGRAPHS

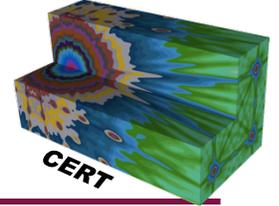


```
inner_product(View1 v1,  
View2 v2) {  
  return map_reduce(  
    multiplies(), plus(),  
    v1, v2  
  );  
}
```

- `inner_product()` specified by **PARAGRAPH**.
- Employs `map_reduce` parallel pattern.
- Defines a new pattern we can use to **compose** a **nested PARAGRAPH**.



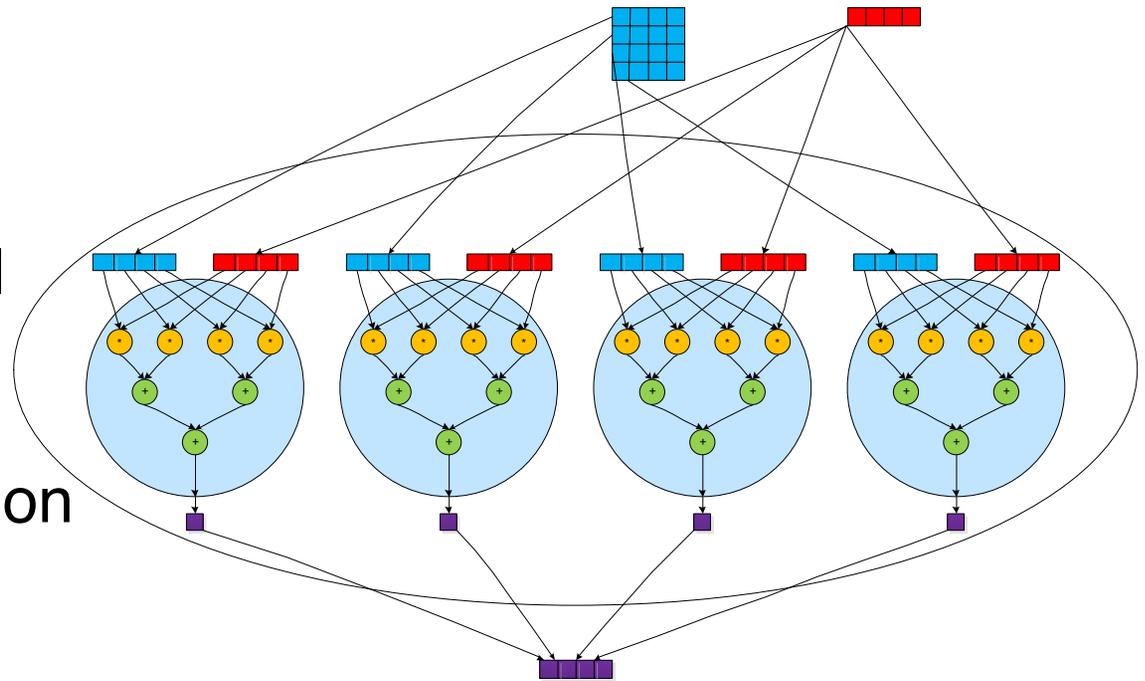
PARAGRAPH Composition



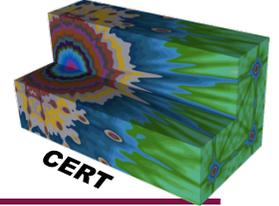
Matrix Vector Multiplication

```
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.



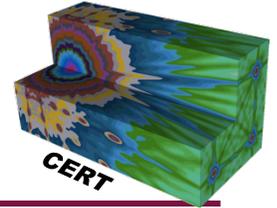
Example: NAS CG in STAPL



```
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.
- NO Barriers – Only point-to-point communication/synchro
- For simplicity / space, we next consider just the first two statements.

Example: Sequence Composition - CG



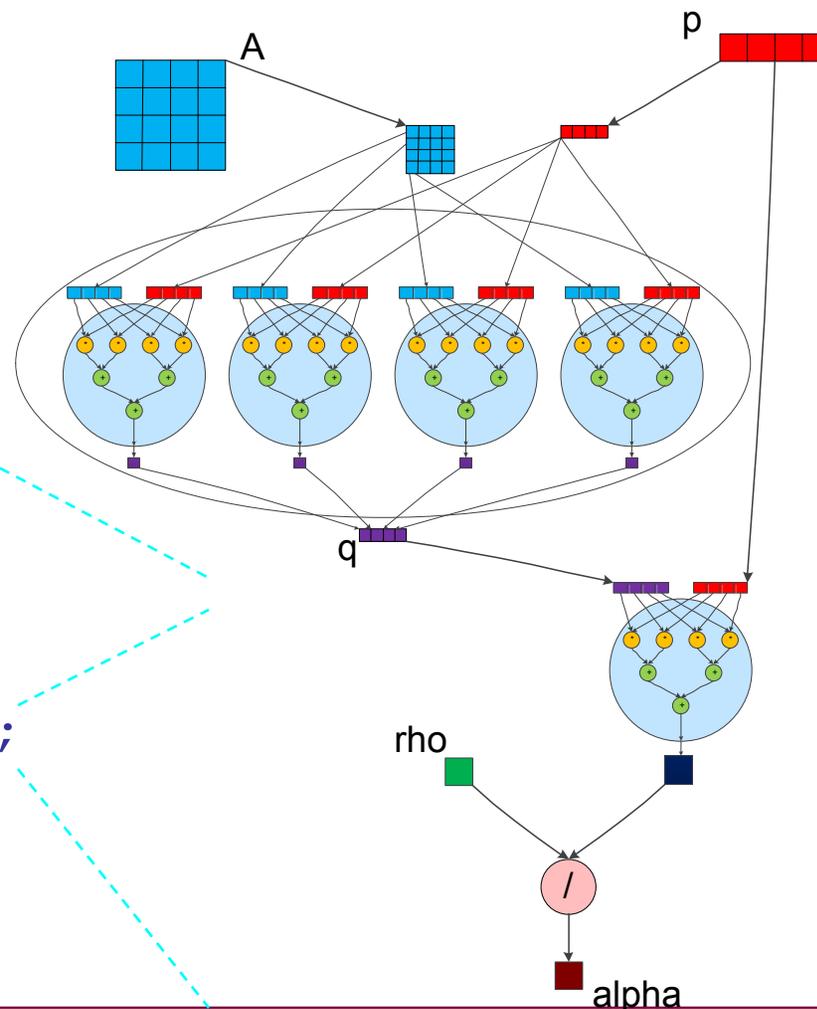
*Matvec() pAlgorithm on
2D_view of pMatrix and
1D_view of pArray.*

$$q = A * p;$$

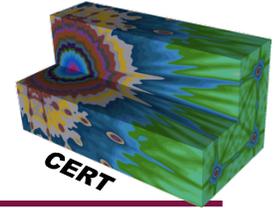
*Inner product of two 1D_view
views whose scalar result is
divisor of dividend rho.*

$$\text{alpha} = \text{rho} / \text{inner_product}(q, p);$$

*Expressive syntax quickly yields
nested/hierarchical PARAGRAPHS.*



NAS EP Peta – Scalability Sanity Check



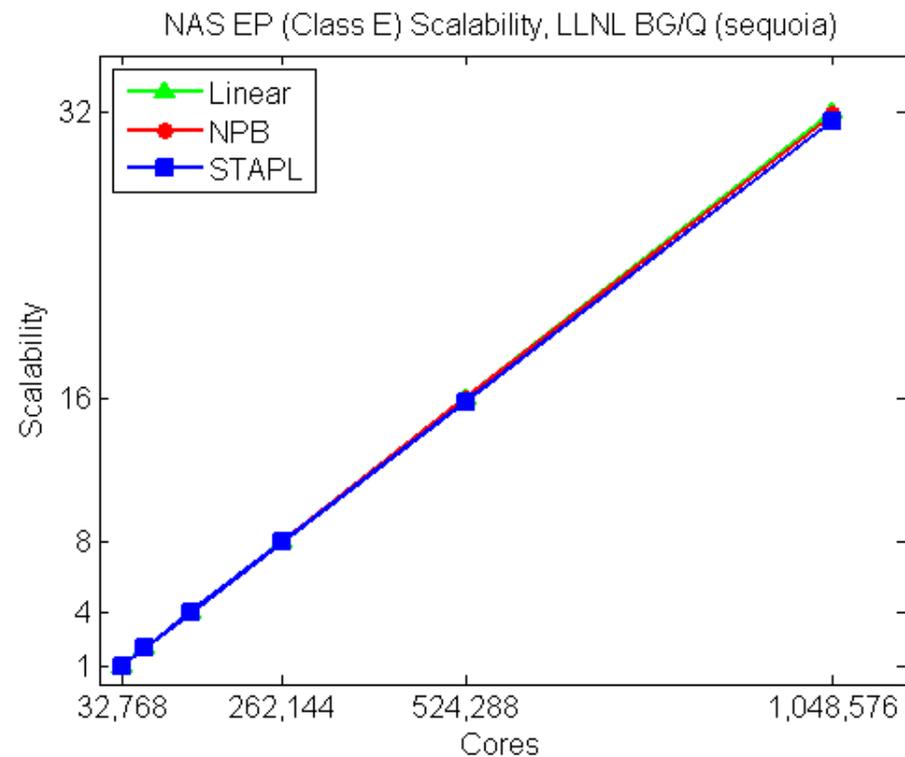
➤ LLNL BlueGene/Q System

- 16-core PowerPC A2 processor per node
- 16GB RAM per node
- Nodes connected in 5-D torus

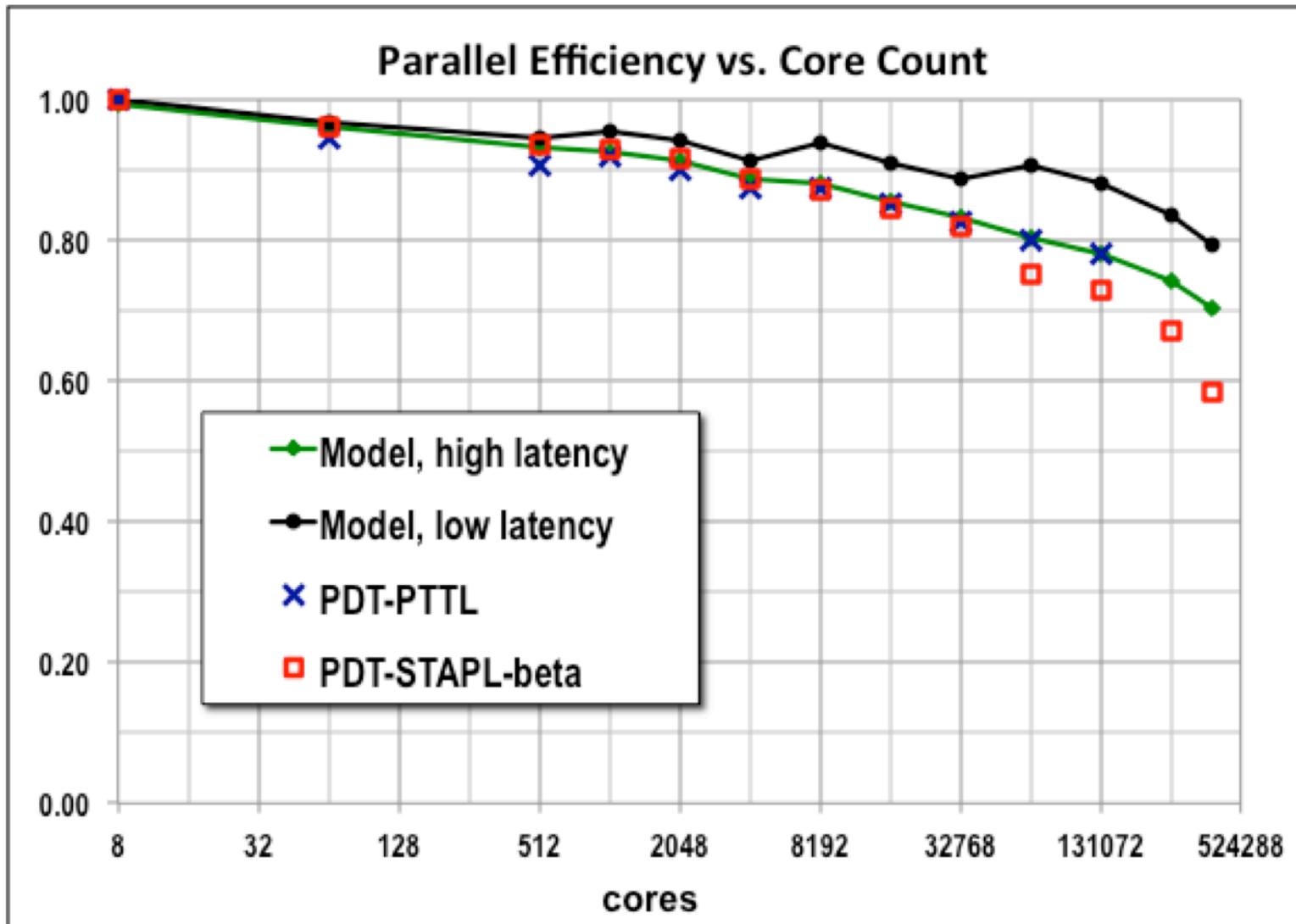
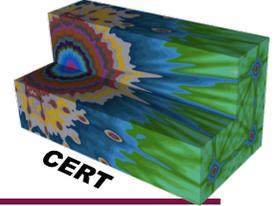
➤ NAS EP

- Transforms stream of uniformly distributed random numbers into normally distributed stream.
- Combines statistics of each processor's output stream to validate.

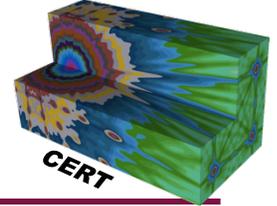
➤ STAPL implementation scales as well as native Fortran+MPI to one million cores.



Where is PDT Now ? In PETA



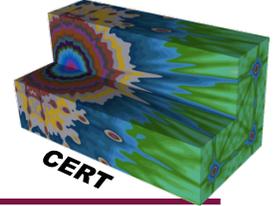
Our Roadmap from Peta to Exa



Immediate Plans:

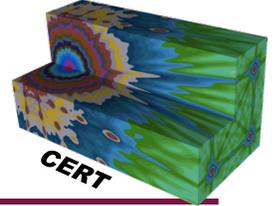
- Exa scalable STAPL
- Fault tolerant STAPL → fault tolerant DSL & PDT
- TAXI: A Domain Specific Library (DSL) for Rad. Transport (built on top of STAPL)
- Longer Term:
 - Adaptive STAPL → Adaptive PDT (all levels !)
 - *Tunable granularity: Fine ↔ Coarse Grain Algorithms*
 - *Communication aggregation*
 - *Load balancing*
 - *Use a fraction of processors for monitor/control performance*
 - Study Approximate methods for TAXI (and STAPL) to improve scalability, in context of UQ

Some features for Exa-scalable STAPL



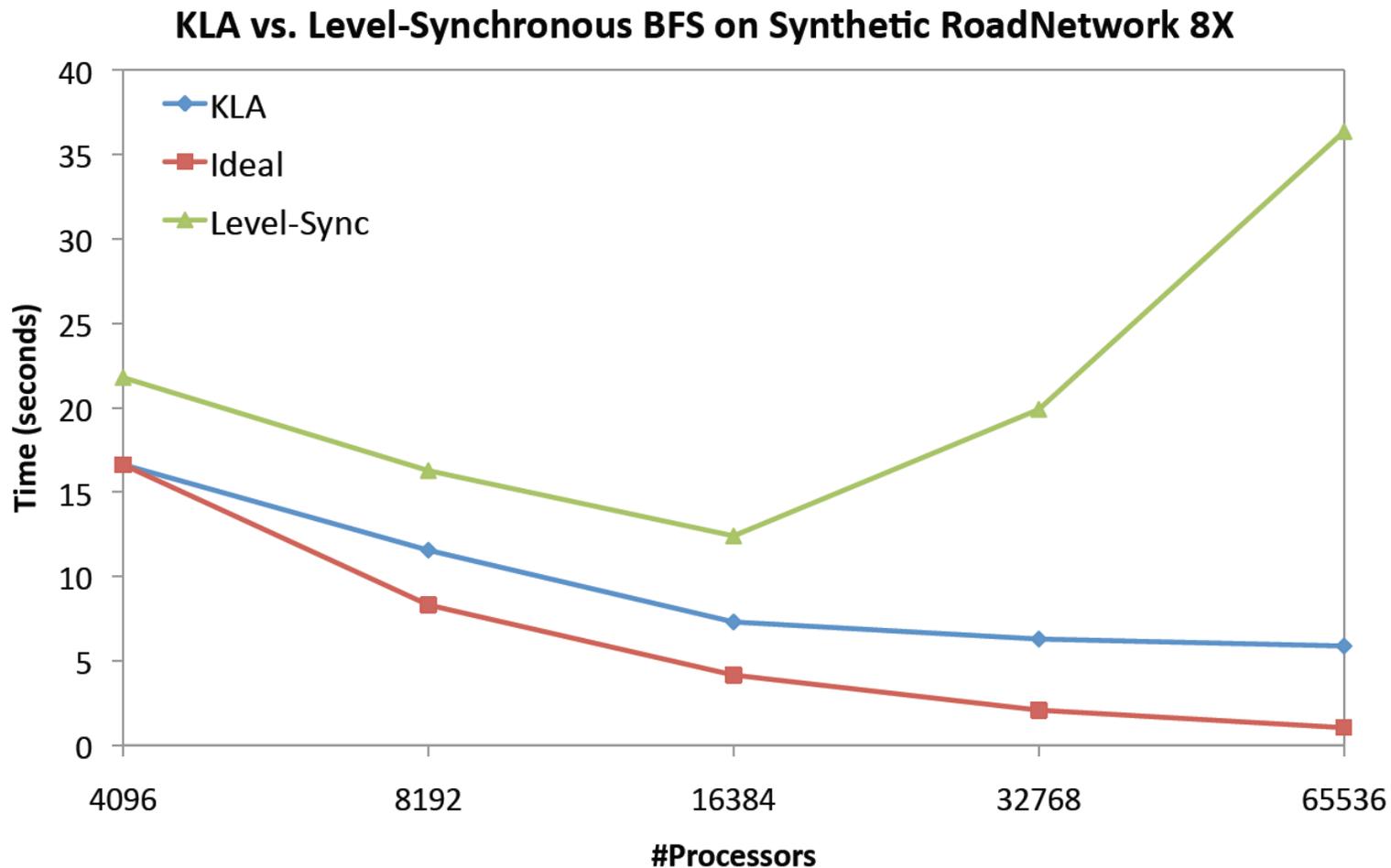
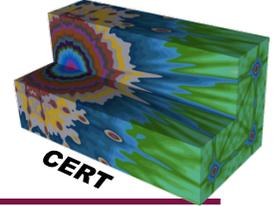
- Asynchronous Algorithms (a bit later)
- Nested/Hierarchical parallelism (parallel algorithms)
- Extension to heterogeneous architectures - GPUs
- Special support for
 - *AMR (space/angle)*
 - *Arbitrary grids, sparse data structures*
- Adaptive behavior
 - *Granularity control of tasks (data + work)*
Fine ↔ Coarse Grain Parallel Algorithms Morphing
 - *Communication/Synch aggregation AND Customization(pt2pt)*
- Algorithmic Composition for Productivity & Performance (skeleton library + composition operators)

Asynchronous Algorithms & Communication



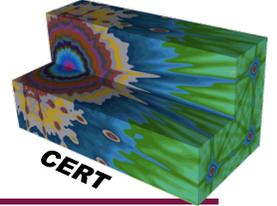
- Asynchrony → Latency Hiding
- Asynch communication: STAPL: ARMI comm. Library
 - *Asynch active messages – never waits for a return value.*
 - *Futures – place holders for return values not yet computed but needed for current evaluation (increases asynchrony).*
 - *Recursively nested communication subgroups (and subcontainer registration) → locality, load balancing + affinity, work reduction (efficiency)*
- Asynchronous Algorithms – not an easy task ...

Asynch Algos: K-level Asynchronous BFS



➤ Removing synchs more important at higher proc. counts

Nested, Hierarchical Algos and RT



- Nested parallelism:

While{forall (reduce (sweep (.....)))}

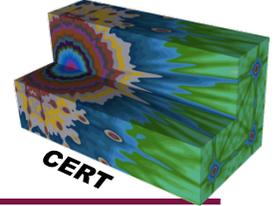
- Hierarchical parallelism (algos): nested and mapped onto the machine memory hierarchy

forall (view_i, forall (view_j, wf{})) where view_i = U{view_j}

mapped hierarchically on machine hierarchy (Locales)

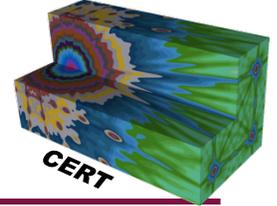
- Support for various Runtime Systems (MPI/OpenMP/Pthreads...recursive constructs)
- **Nested/Hierarchical → Latency reduction (locality) + Expressivity (and productivity)**

Support for Heterogeneous Architectures



- Storage: STAPL is distributed and GPU means another address space (Locale)
- Algos+Code: GPUs use different code, algorithms than CPU (needs engineering)
- STAPL: Allows global memory tracking– all data structures have GIDs.
- STAPL will enable simpler programming but not make compiler/user level decisions.

Peta to Exa: Fault tolerance via STAPL

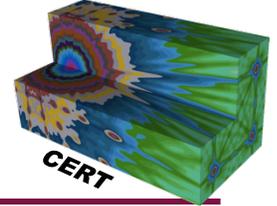


- STAPL – distributed **virtualized** system makes it easier
- Fault tolerant STAPL components → Fault tolerant composed program
- Fault Detection: extend ARMI + other techniques
- Fault Recovery: Distributed Checkpoints + Task graph replication
 - *Groups of re-work processors/memory(plenty of them)*
 - *Paragraphs with built-in replication/redundancy*
 - *See Manteuffel's coarse grain replication of data (CU)*

Open Question:

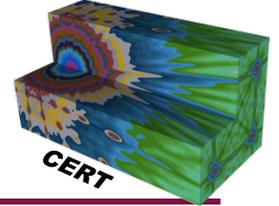
- Fault resilient algorithms: error ↔ fault tolerance

Open Question: Approximate Computation



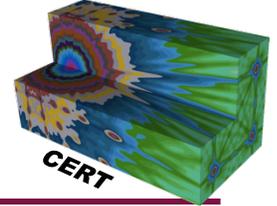
- Increased Asynchrony requires
 - *tolerance of stale info*
 - *otherwise approximating it*
 - *Example: use of old data in sweeps on re-entrant graphs*
- Relaxation of dependences to keep computation local
- Non-determinism
- Tradeoff: Algorithm induced error \leftrightarrow performance (parallelism)
- UQ in the presence of approximate computation

From Peta to Exa PDT by TAXI



- TAXI library will contain data structures and algorithms for radiation transport
 - *Extend STAPL data structures (Graph -> Grid)*
 - *Composition of Algorithms (skeletons) into transport specific algorithms (simultaneous sweeps)*
 - *BiCG, etc*
 - *Composition of building blocks will allow Transport exploration*

Beyond PDT and TAXI: Contributions to Exascale Issues in CompSci



- Exa-scaled parallel *generic* library STAPL
- Answers to many general questions:
 - *AMR/Arbitrary Grids*
 - *Fault tolerant STAPL Library and trade-offs with speed*
 - *Hierarchical/Heterogeneous parallelism mapped on H/H Machines*
 - *Dynamic Load Balancing*
 - *Transformation between Fine-Coarse grain of algorithms*
 - *Asynchrony, (weaker) memory consistency and programming productivity tradeoffs.*
- How to build a useful DSL
- Make peta scale good for general use.
- ➔ Almost nothing presented is Transport exclusive !