

THE CENTER FOR EXASCALE SIMULATION OF PLASMA-COUPLED COMBUSTION(XPACC)

Tools to Enable Access to Exascale

William D. Gropp

Center for Exascale Simulation of
Plasma-Coupled Combustion

Parallel Computing Institute

Computer Science

National Center for Supercomputing

Applications



Our View of Exascale Challenges

- ▶ **Plateau in clock rates** requiring *increased concurrency*
- ▶ **Simpler and more specialized processing elements** requiring support for *heterogeneous computing*
- ▶ **Increasing likelihood of faults** requiring *multiple strategies for resilience*
- ▶ **Scalability and performance irregularity** requiring *new algorithms and adaptive methods*
- ▶ **High latency at all levels** requiring *latency hiding* software and algorithms
- ▶ Most apply to computing in general; for exascale, solutions to these challenges must work well *together*.
- ▶ Time frame another challenge — must have usable tools within life of project



Consequences and Approaches

- ▶ **Concurrency** Algorithms and software to expose multiple levels of parallelism. Systems built with a hierarchy of compute elements — not a flat architecture with 10^9 cores, but (say) 10^5 nodes each with 10^3 concurrent streams each with 10-way functional units.
 - Do not need a single programming approach — can compose methods
- ▶ **Heterogeneity** Use GPUs as a *stand-in* for likely integrated yet heterogeneous single chip architectures.
 - Software tools to support multiple architectures, load balance.
- ▶ **Fault detection and resilience** Purely automatic methods are too costly in time and energy.
- ▶ **Complexity of Code** Overall complexity impacts ability to perform V&V; performance tuning; enhancements
 - Use tools to address code complexity; maintain straightforward “golden copy” and auto-generate and tune code as needed



Overarching Approach

- ▶ **Scalable numerical algorithms** Exposing the amounts and kind of parallelism is key to enabling CS techniques
- ▶ **Overdecomposition** Divide data structures into more than one chunk per core to provide load balancing (redistribute), performance (match memory hierarchy), latency hiding (schedule), fault tolerance (unit of repair), heterogeneity (schedule)
- ▶ **Adaptive Computation** Have the code at runtime adapt runtime execution to respond to compile *and runtime* information about progress, efficiency
- ▶ **Source-to-source transformations** Allow the computational scientist to focus on expressing the algorithm and use tools to provide customized versions of code for different situations and platforms.

Always preserving the golden copy.



Composition of Solutions

Rather than one do-everything solution (e.g., a new parallel language), build interoperable tools that address one *or more* issues.

- ▶ Enables exploitation of tools developed for commodity and smaller-scale systems (can plug into framework)
- ▶ Mitigate risk by reducing single points of failure
- ▶ Increases potential for adoption by enabling incremental adoption
- ▶ Addresses complexity by factoring methods and services (complexity becomes primarily additive rather than multiplicative)

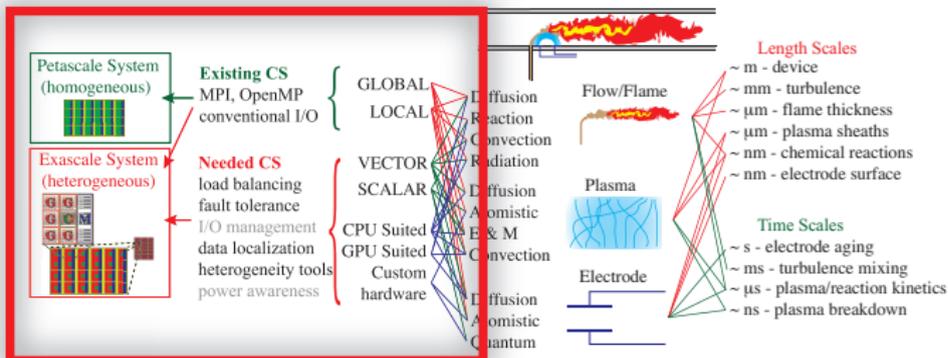


Overall Approach

- ▶ *PlasComCM* provides the overall framework; operator (mathematics) based structure, union of logically regular (including overlapping) meshes. Well-defined and mathematically accurate method for information transfer between modules.
- ▶ “Golden copy” of source code
- ▶ Use mathematically-oriented abstractions to connect with source-to-source transformations (to map code onto different hardware) and with adaptive runtime to efficiently manage execution order and mask latency.

Computer Science Contributions

- ▶ Load Balance and Overdecomposition
- ▶ Exploiting Specialized Processing Elements
- ▶ Locality and Performance
- ▶ Autotuning
- ▶ Fault Tolerance and Resilience



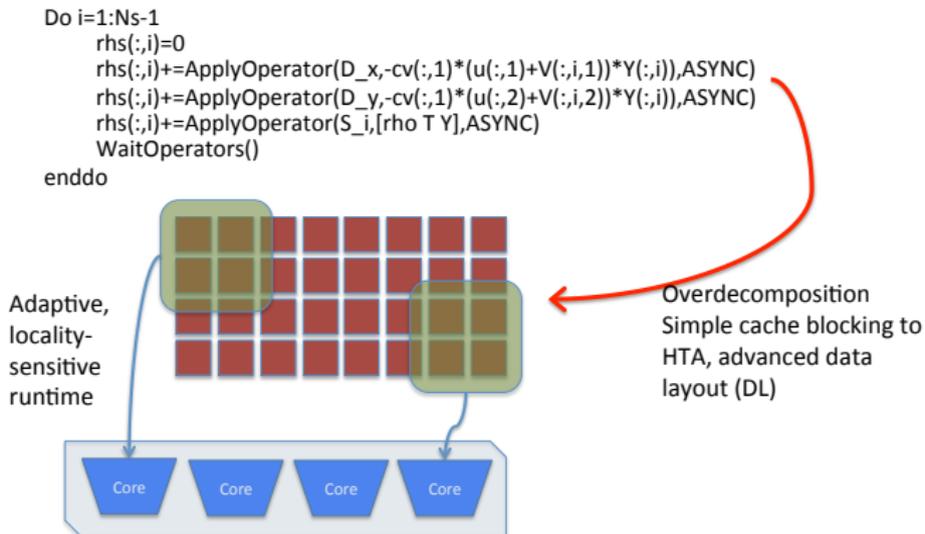
Load Balance and Overdecomposition

- ▶ Many sources of dynamic load imbalance:
 - Problem itself (geometry, chemistry, multiphysics)
 - System runtime and OS steals resources (CPU, memory bandwidth, interconnect for I/O)
 - Faults, including recovery costs
- ▶ Load balance requires dynamic reassignment of work, retaining low overhead
 - Very fine grain (within an multithreaded MPI process; within a loop). Build on current work with LLNL collaborators Gamblin, de Supinski.
 - Intermediate grain (e.g., between MPI processes; mesh blocks). Build on extensive experience with Charm++ as runtime, including scalable (20k nodes on BW, e.g.), low-overhead load balancing.
 - Intermediate grain *also* addresses fault tolerance (recover on basis of block) and resource heterogeneity (schedule on available resource).



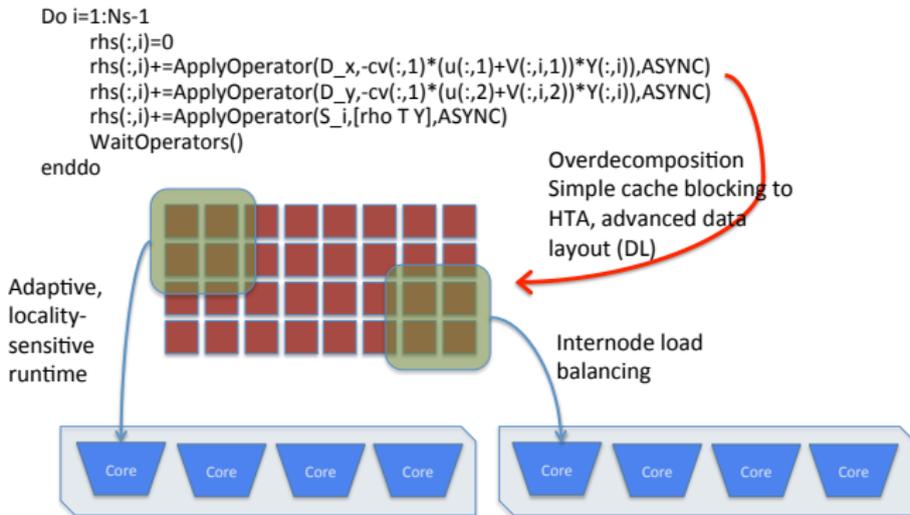
Load Balancing and Overdecomposition

- ▶ Research includes defining and building methods that seamlessly handle very fine grain through very coarse.



Common Approach for Load Balancing

- ▶ Internode load balancing important for multi-physics, refinement, ...



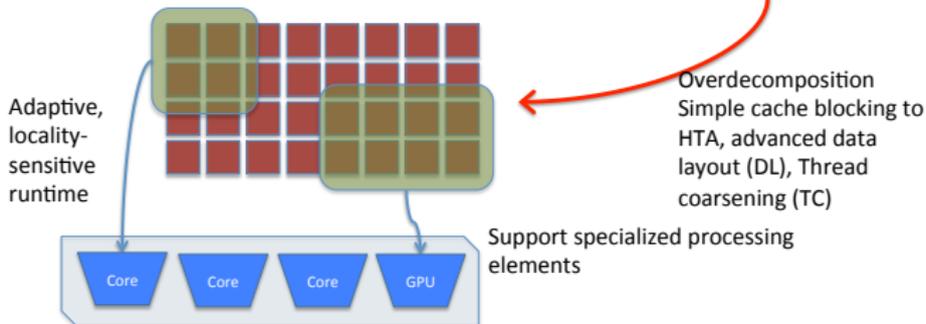
Exploiting Specialized Processing Elements

- ▶ For power, density, etc., processors likely to specialize (GPUs are a current example; we expect integrated, specialized cores rather than the current attached-processor)
- ▶ Address by creating tools that optimize code for specific needs
- ▶ Example: Thread coarsening. Allows programmer/algorithm developer to think in terms of maximum parallelism (good for Exascale!) and let tool perform hardware-appropriate aggregation.
- ▶ Example: Data layout. Even for dense matrices, the “natural” layout is not the most efficient for computation. (Near) optimal layout depends on details of hardware, so best if a tool can manage all code and data structure transformations.
- ▶ Scheduling/load balancing handled through overdecomposition



Exploiting Specialized Processing Elements

```
Do i=1:Ns-1
  rhs(:,i)=0
  rhs(:,i)=ApplyOperator(D_x,-cv(:,1))*(u(:,1)+V(:,i,1))*Y(:,i),ASYNC)
  rhs(:,i)=ApplyOperator(D_y,-cv(:,1))*(u(:,2)+V(:,i,2))*Y(:,i),ASYNC)
  rhs(:,i)=ApplyOperator(S_i,[rho T Y],ASYNC)
  WaitOperators()
enddo
```



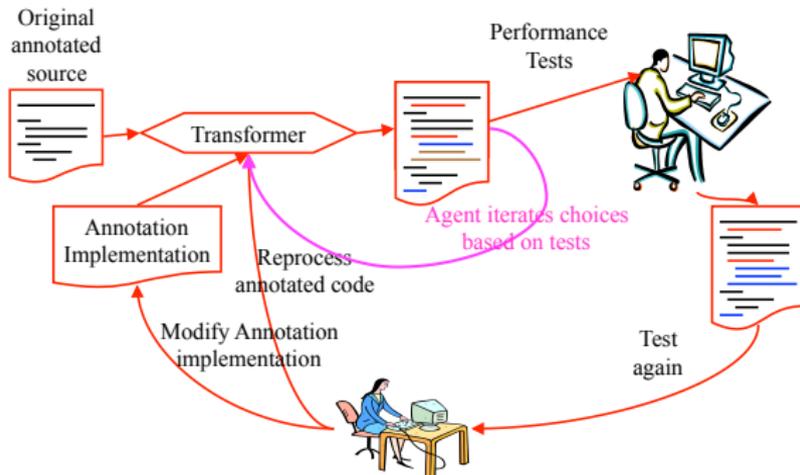
Autotuning for Performance

- ▶ Generating efficient code is necessary but error prone, labor intensive
- ▶ Building an optimizing compiler is not realistic (best results exploit application's data structure properties)
- ▶ Alternative is to open up the compilation process and use the (probably vendor) compiler for code generation and fine grain optimization at the instruction level and use other tools to apply larger grain optimizations
- ▶ Problem with this approach is (a) the compilers' performance model is unavailable and (b) performance model too inaccurate for picking optimization strategies
- ▶ One approach is *autotuning*, where a family of possibilities is defined and experiments run to pick solution
- ▶ We will build on experience with Spiral



Autotuning for Performance

- ▶ Our approach: higher-level abstractions, easily integrated into code, preserving a “golden copy” from which computational scientist can work



Programming Approach

- ▶ At all levels: abstractions hide implementation details, provide flexibility
- ▶ MPI-3 for internode
 - Concurrency $O(10^5)$ – $O(10^6)$
 - Includes MPI-3 one-sided and shared memory; provides advantages of PGAS languages but without risk of compiler development; based on MPI communicators, gives better support for multi-physics
 - Alternatives include other systems supporting one-sided operations (lighter weight), but methods unlikely to require few word put/get/accumulate
 - Productivity issues mostly associated with distributed data structure support, performance optimization; handled with other tools
 - Internode scalability requires adaptive load balance
 - Leverage extensive Charm++ experience



Programming Approach

- ▶ Adaptive runtime for intranode
- ▶ Tools for performance optimization at node, core level (source-to-source transformations)
 - Shares with so-called domain specific languages (really abstract data-structure specific languages)
- ▶ Established language compilers for core(s)
 - C/C++/Fortran 90
 - OpenACC, OpenMP, CUDA
 - Aided by autotuners, code generators



Computer Science Impact on Our Simulation

- ▶ *PlasComCM* can scale efficiently across all current architectures and is expected to scale on anticipated architectures
- ▶ *PlasComCM* can effectively harness the high- and evolving-complexity of current and anticipated computers
- ▶ *PlasComCM* can both efficiently utilize full machines for maximum-scale-of-the-day simulations and still be useful for extensive small-simulation sampling for UQ or parametric investigation
- ▶ Within a reasonable set of guidelines, *PlasComCM*'s development (algorithms and physical models) proceed unhindered by evolving hardware norms and constraints
- ▶ The tools that enable *PlasComCM* in this way are portable to other applications



Our CS Contributions

- ▶ *Integrated* use of overdecomposition for load balance, resilience, performance, latency tolerance and heterogeneity and *tools* to implement
- ▶ Comprehensive attention to *locality* for performance and *tools* to implement
- ▶ *Adaptive computing* to address scaling, load-balance, performance irregularity, resilience
- ▶ A *cooperative, component-oriented* approach to permit incremental adoption to a significantly more advanced code
- ▶ Success: *PlasComCM* unconstrained by the challenges of exascale

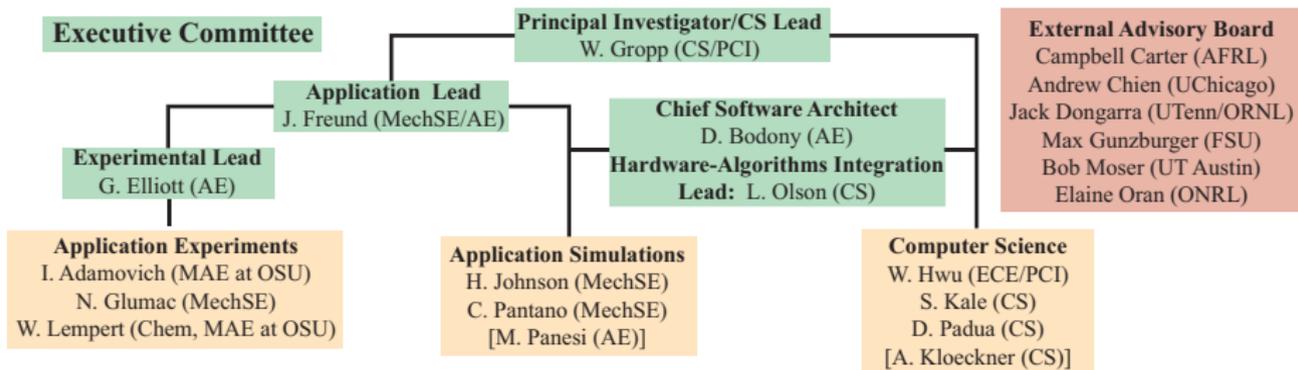


Advancing Exascale Computation

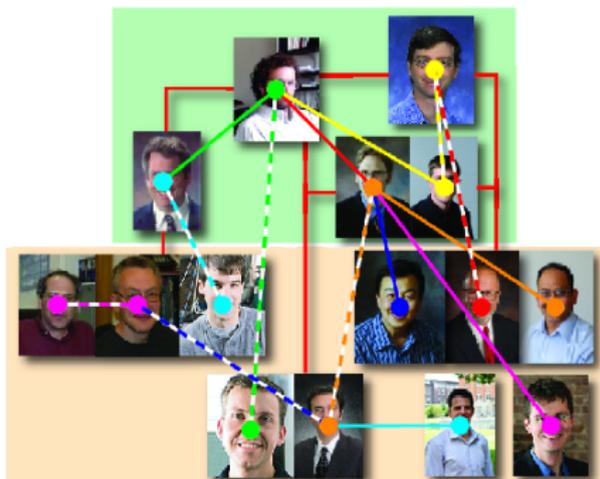
- ▶ Exascale requires extreme care in *everything*. A large part of the contribution is an approach that permits and encourages *combining* the best possible tools and techniques
- ▶ Specific issues: load balance and noise (over-decomposition); locality (Hierarchically Tiled Arrays, vector/thread/task decompositions tested in GPU context)
- ▶ Resilience (Checkpoint-restart; support for algorithmic approaches)
- ▶ Demonstrate practicality of *composition of programming systems*



The Team



Initial PhD Student Projects



- Discrete Adjoint Across Meshes
- PIC in *PlasComCM* and at Exascale
- Exper. Design for UQ
- Practical Plasma-Combustion Kinetics
- Multi-hardware Programming
- Multi-timestepping
- Runtime Over-decomposition
- - - Autotuning/HTA for *PlasComCM* (2)
- - - Finite elements/Multigrid in *PlasComCM* (2)
- - - Multiscale MD of Electrode Aging
- - - Diagnostics/Full-scale Experiment (2)
- - - Kinetics Modeling for Diffusion Flames
- - - Advanced Diagnostics for Calibration
- - - Sub-grid-scale Plasma Comb. Models

- Planning a highly integrated, collaborative effort