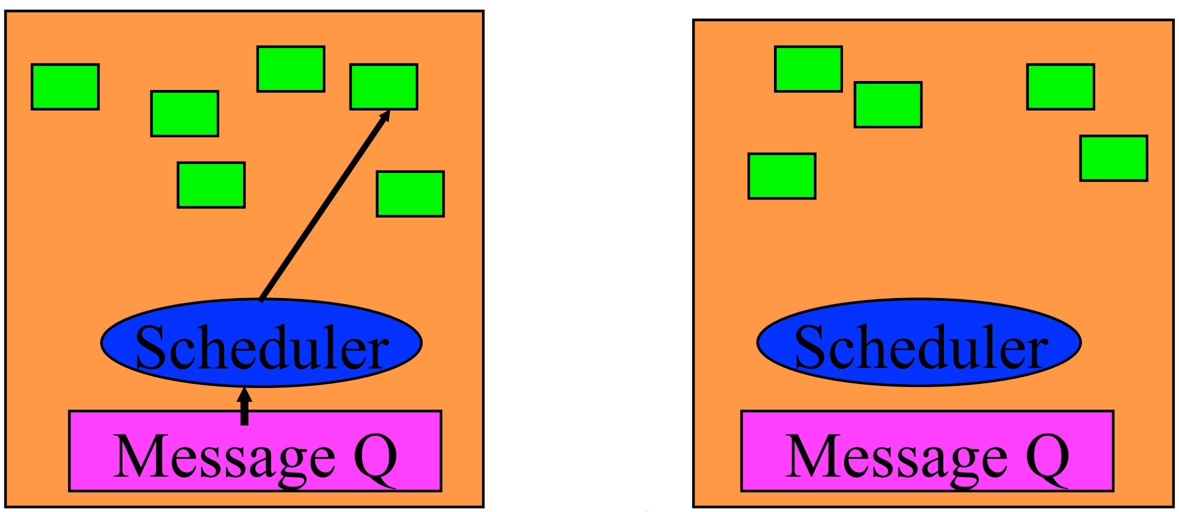
Observations: Exascale applications

- Development of new models must be driven by the needs of exascale applications
  - Multi-resolution
  - Multi-module (multi-physics)
  - Dynamic/adaptive: to handle application variation
  - Adapt to a volatile computational environment
  - Exploit heterogeneous architecture
  - Deal with thermal and energy considerations
- So? Consequences:
  - Must support automated resource management
  - Must support interoperability and parallel composition

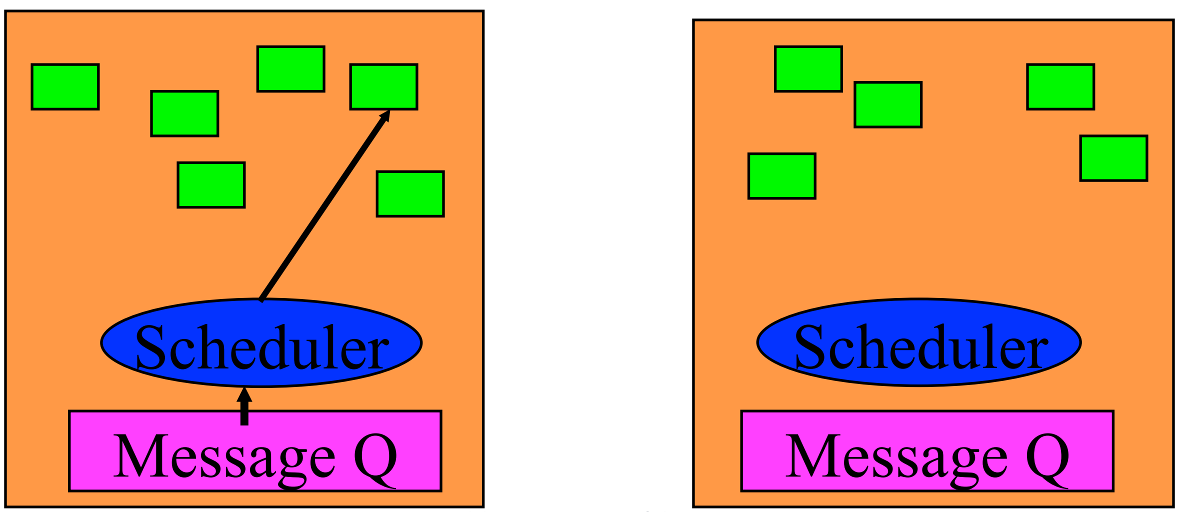
The Execution Model

- Several objects live on a single "processor"
  - We will come back what we mean by a processor.
    - For now, think of it as a core
  - As a result,
    - the method invocations directed at objects on that processor will have to be stored in a pool,
    - And a user-level scheduler will select one invocation from the Quque and and runs it completion



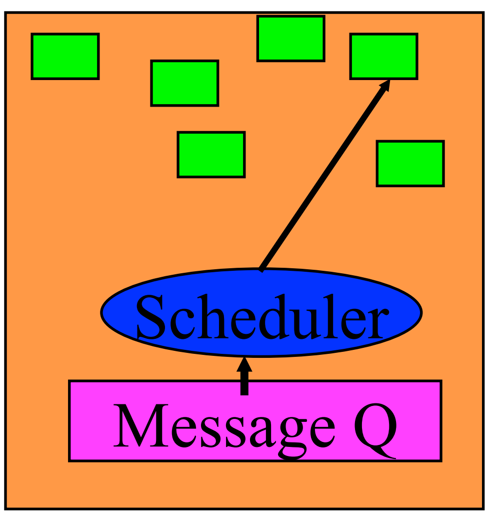
Message-driven Execution

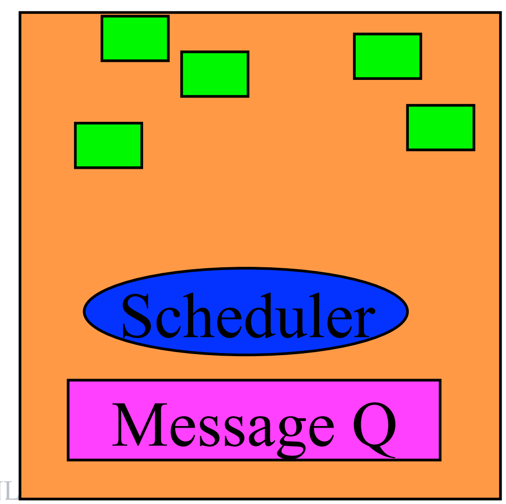
- Execution is trigggered by availability of a "message" (a method invocation)
- When an entry method executes,
  - it may generate messages for other objects
  - The RTS deposits them in the message Q on the target processor



Utility for Multi-cores, Many-cores, Accelerators:

- Objects connote and promote locality
- Message-driven execution is
  - A strong principle of prediction for data and code use
  - Much stronger than principle of locality
    - Can use to scale memory wall:
    - Prefetching of needed data:
      - into scratch pad memories, for example



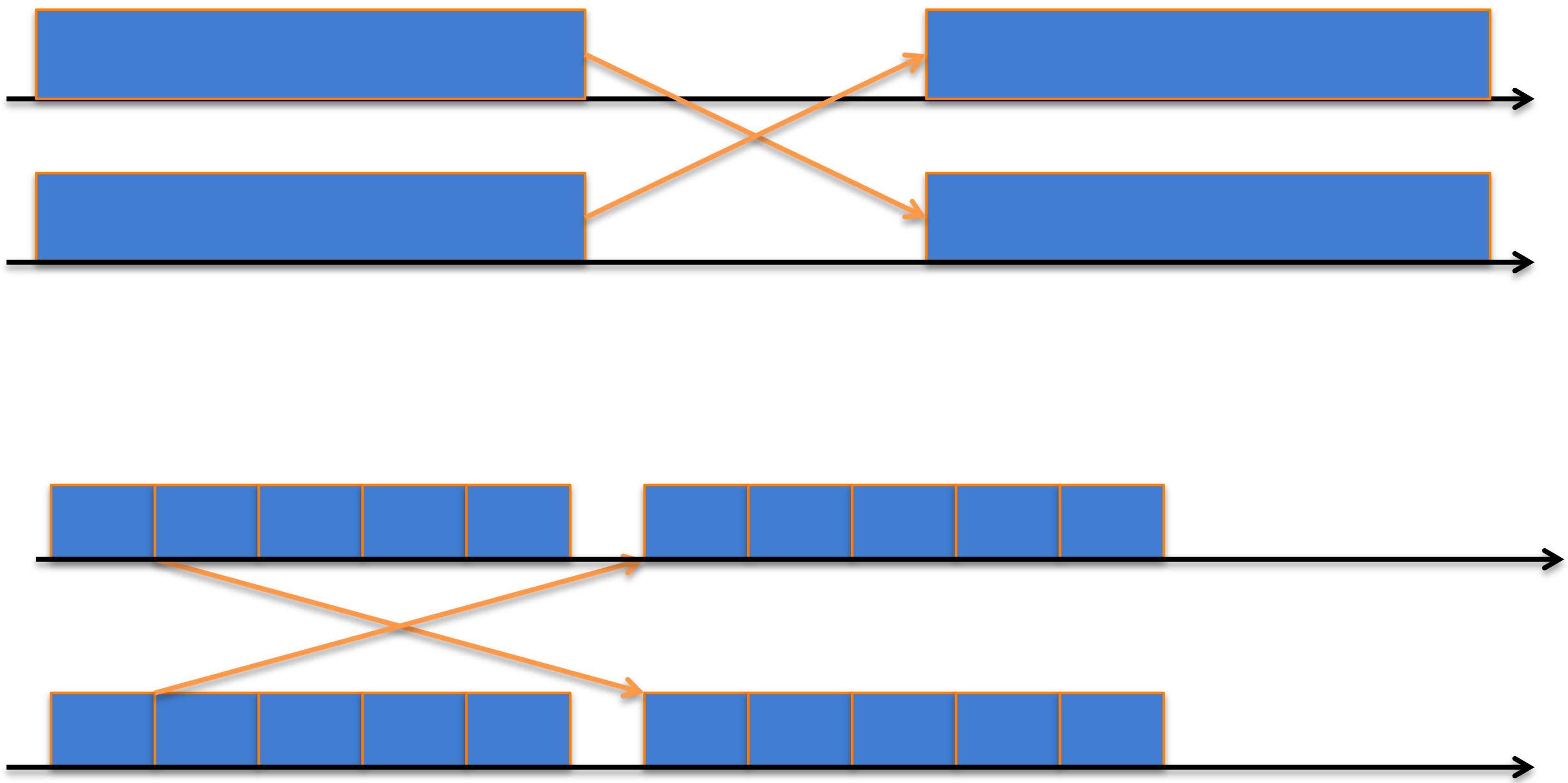


Impact on communication

- Current use of communication network:
  - Compute-communicate cycles in typical MPI apps
  - So, the network is used for a fraction of time,
  - and is on the critical path
- So, current *communication networks are over-engineered for by necessity*
- With overdecomposition
  - Communication is spread over an iteration
  - Also, adaptive overlap of communication and computation

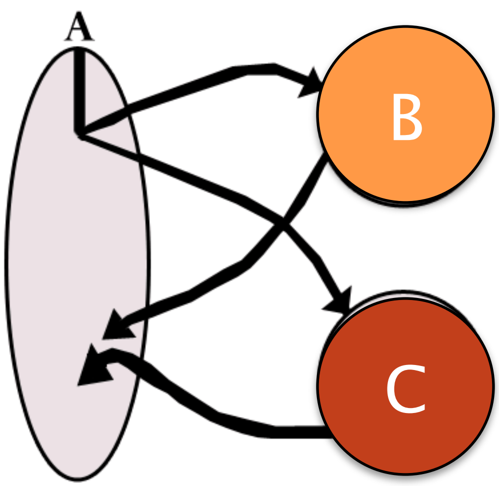
Example: Stencil Computation

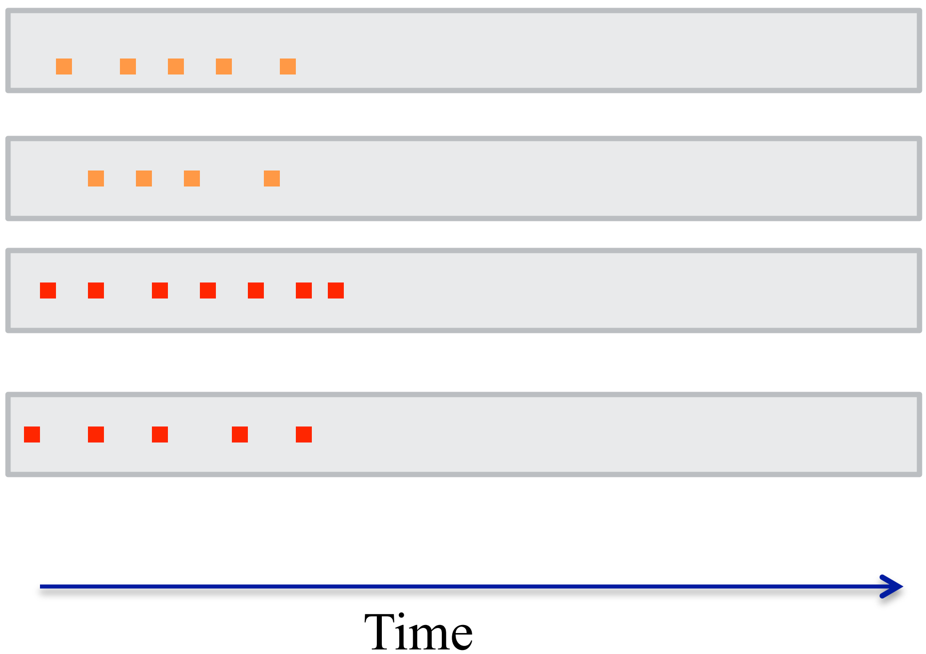
- Consider a simple stencil computation
  - With traditional design based on traditional methods (e.g. MPI-based)
    - Each processor has a chunk, which alternates between computing and communicating
  - With Charm++
    - Multiple chinks on each processor
    - Wait time for each chunk overlapped with useful computation for the other
    - Communication spreads over time
  - A Simple schematic timeline for both approaches?



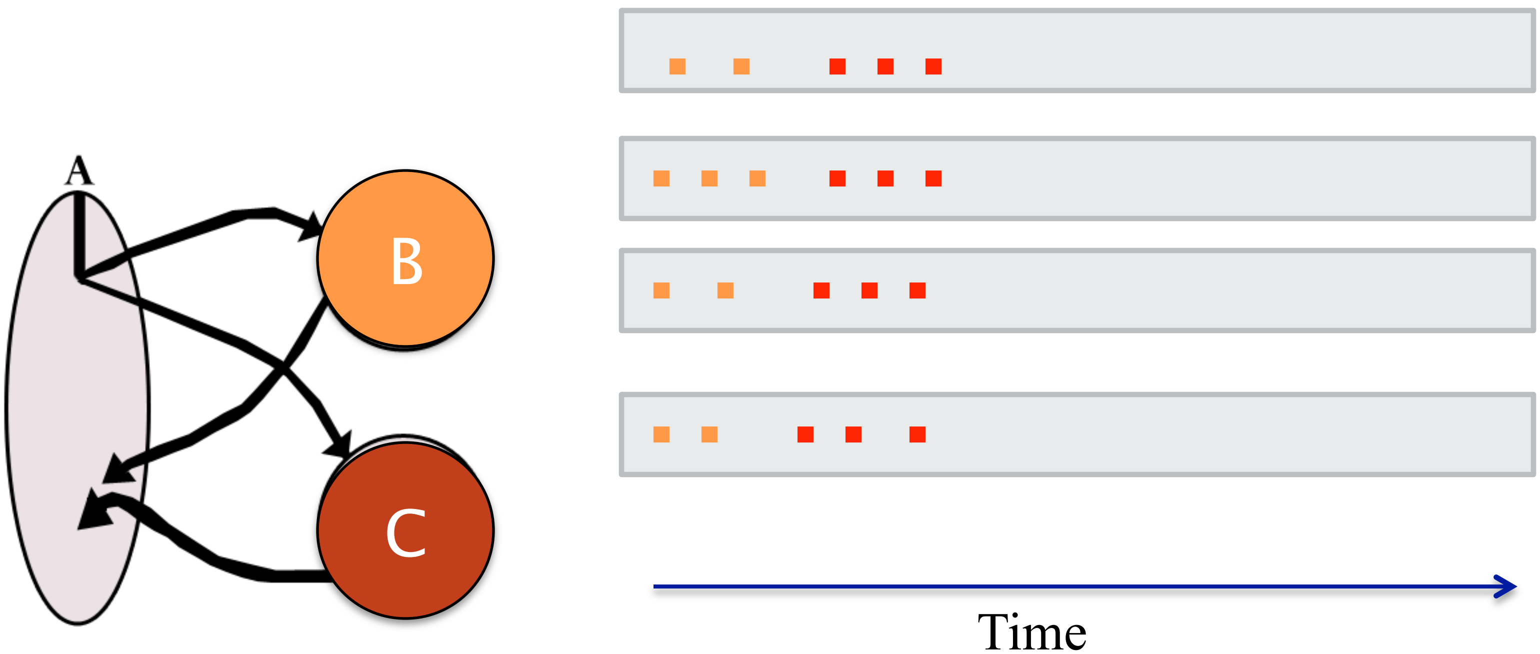
Without message-driven execution (and virtualization), you get either:

Space-division

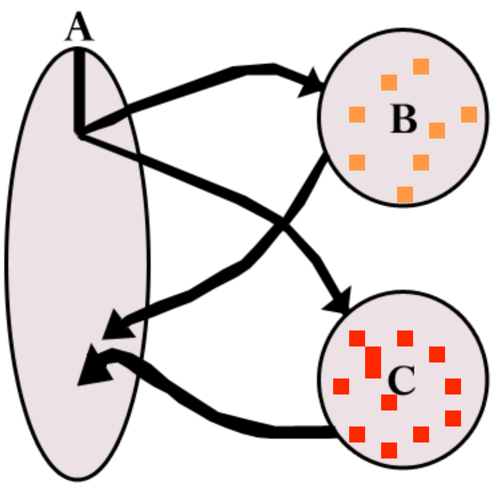




OR: Sequentialization



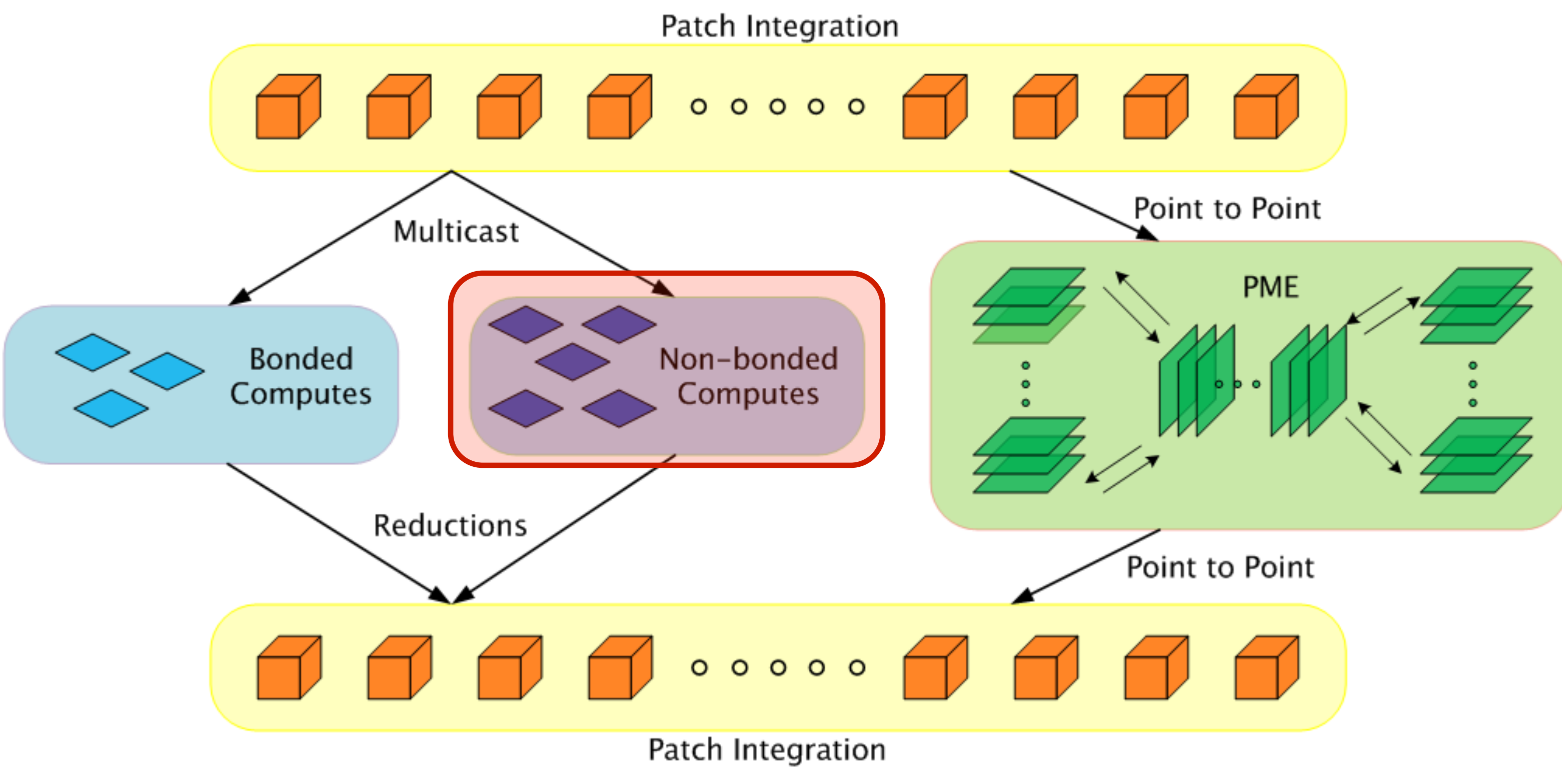
Parallel Composition: A1; (B || C); A2



Recall: Different modules, written in different languages/paradigms, can overlap in time and on processors, without programmer having to worry about this explicitly

MD Parallelization Using Charm++

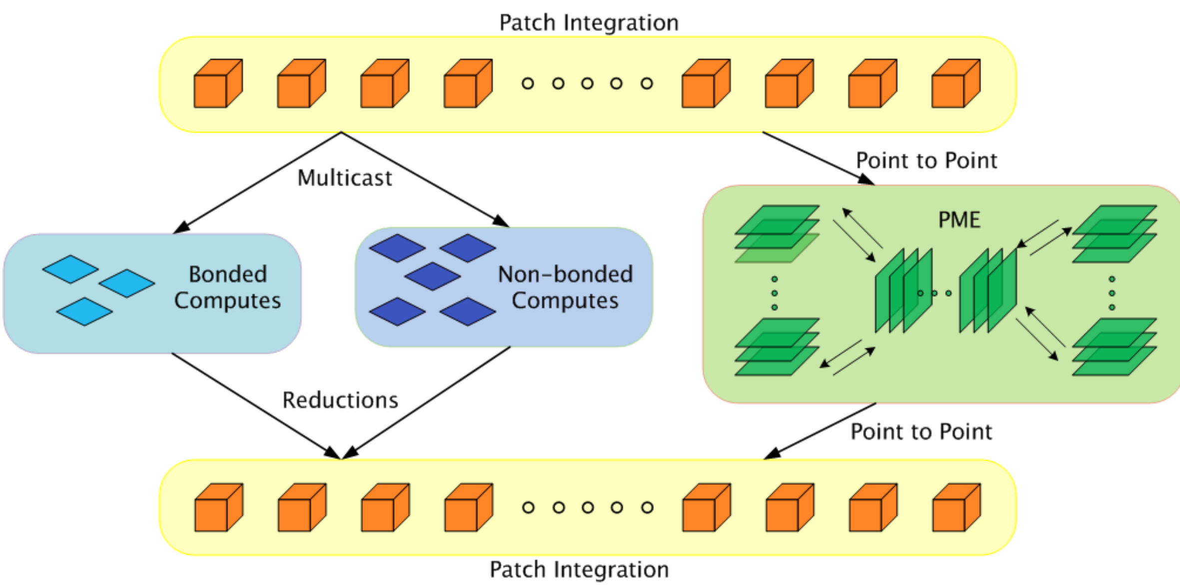
The computation is decomposed into "natural" objects of the application, which are assigned to processors by Charm++ RTS

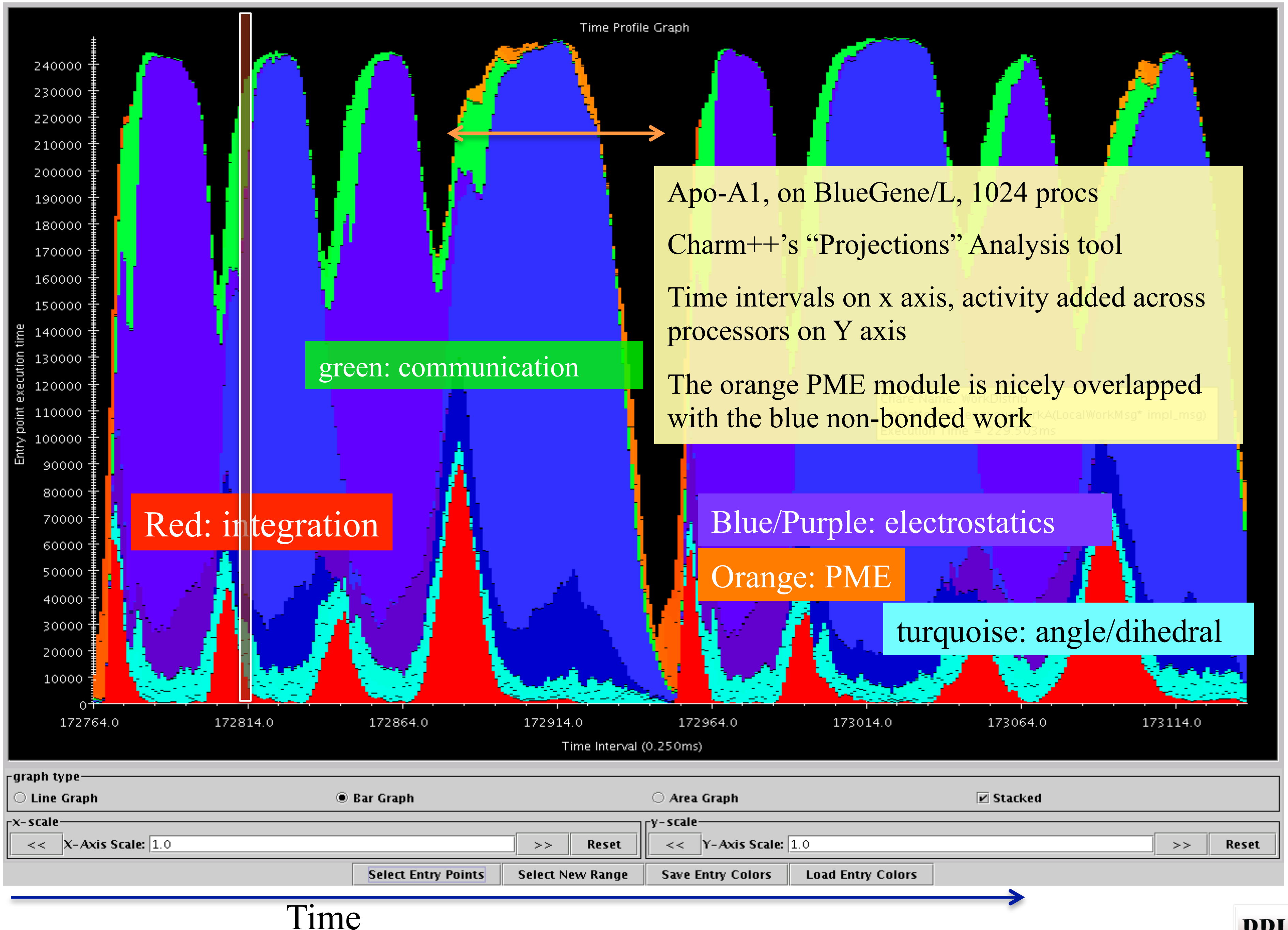


MD Parallelization Using Charm++

The PME module is communication-dominated, with a long critical path, and small computation'

The non-bonded force computation module has a lot of floating point work, but relatively small computaiton

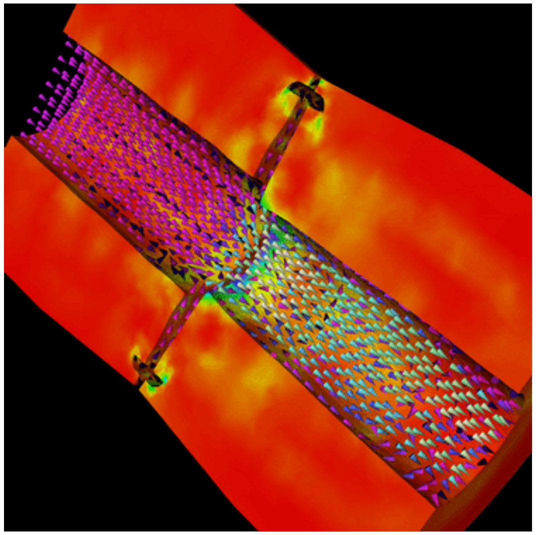



Time Profile Graph
240000
230000
220000
210000
200000
190000
180000
170000
160000
150000
140000
130000
120000
110000
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
Entry point execution time
172764.0
172814.0
172864.0
172914.0
172964.0
173014.0
173064.0
173114.0
Time Interval (0.250ms)
Apo-A1, on BlueGene/L, 1024 procs
Charm++'s "Projections" Analysis tool
Time intervals on x axis, activity added across processors on Y axis
The orange PME module is nicely overlapped with the blue non-bonded work
green: communication
Red: integration
Blue/Purple: electrostatics
Orange: PME
turquoise: angle/dihedral
graph type
Line Graph
Bar Graph
Area Graph
Stacked
x-scale
<<
X-Axis Scale: 1.0
>>
Reset
y-scale
<<
Y-Axis Scale: 1.0
>>
Reset
Select Entry Points
Select New Range
Save Entry Colors
Load Entry Colors
Time

Decomposition Independent of numCores

- Rocket simulation example under traditional MPI

1	2		P
Solid	Solid	. . .	Solid
Fluid	Fluid		Fluid



- With migratable-objects:

$Solid_1$ $Solid_2$ $Solid_3$ . . . $Solid_n$

$Fluid_1$ $Fluid_2$ . . . $Fluid_m$

  - Benefit: load balance, communication optimizations, modularity

Migratability

- Once the programmer has written the code without reference to processors
  – With all the communication expressed as that between objects
- The system is free to migrate the objects across processors as and when it pleases
  – It must ensure it can deliver method invocations to the objects, whereever they go
  – This migratability turns out to be a key attribute for empowering an adaptive runtime system