Charm++ Benefits

- Overdecomposition
- Message-driven execution
- Migratability
- Introspective and adaptive runtime system
- Scalable tools
- Automatic overlap of communication and computation
- Perfect prefetch
- Compositionality
- Emulation for performance prediction
- Fault tolerance
- Dynamic load balancing (topology-aware, scalable)
- Temperature/power/energy optimizations

Charm Tutorial
Locality and Prefetch

- Objects connote and promote locality
- Message-driven execution
  - 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 scratchpad memories, for example
Impact on Communication

• Current use of communication network:
  – Compute-communicate cycles in typical MPI apps
  – The network is used for a fraction of time
    • And is on the critical path

• Current communication networks are over-engineered by necessity
Impact on Communication

• With overdecomposition:
  – Communication is spread over an iteration
  – Adaptive overlap of communication and computation
Decomposition Challenges

• Current method is to decompose to processors
  – This has many problems
  – Deciding which processor does what work in detail is difficult at large scale

• Decomposition should be independent of number of processors – enabled by object based decomposition

• Let runtime system (RTS) assign objects to available resources adaptively
Decomposition Independent of numCores

• Rocket simulation example under traditional MPI

• With migratable-objects:

  - Benefit: load balance, communication optimizations, modularity
Compositionality

• It is important to support parallel composition
  – For multi-module, multi-physics, multi-paradigm applications…

• What I mean by parallel composition
  – $B \parallel C$ where $B$, $C$ are independently developed modules
  – $B$ is parallel module by itself, and so is $C$
  – Programmers who wrote $B$ were unaware of $C$
  – No dependency between $B$ and $C$

• This is not supported well by MPI
  – Developers support it by breaking abstraction boundaries
    • E.g., wildcard recvs in module $A$ to process messages for module $B$
  – Nor by OpenMP implementations
Without message-driven execution (and virtualization), you get either:

**Space-division**
OR: Sequentialization
Recall: different modules, written in different languages/paradigms, can overlap in time and on processors, without programmer having to worry about this explicitly.

Parallel Composition: \( A1; (B \parallel C); A2 \)