Charm++
Motivations and Basic Ideas

Laxmikant (Sanjay) Kale
http://charm.cs.illinois.edu
Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana Champaign
Challenges in Parallel Programming

• Applications are getting more sophisticated
  – Adaptive refinements
  – Multi-scale, multi-module, multi-physics
  – E.g. Load imbalance emerges as a huge problem for some apps

• Exacerbated by strong scaling needs from apps

• Future challenge: hardware variability
  – Static/dynamic
  – Heterogeneity: processor types, process variation, ..
  – Power/Temperature/Energy
  – Component failure

• To deal with these, we must seek
  – Not full automation
  – Not full burden on app-developers
  – But: a good division of labor between the system and app developers
What is Charm++?

• Charm++ is a generalized approach to writing parallel programs
  – An alternative to the likes of MPI, UPC, GA etc.
  – But not to sequential languages such as C, C++, and Fortran

• Represents:
  – The style of writing parallel programs
  – The runtime system
  – And the entire ecosystem that surrounds it
So, What is Charm++?

• Object Oriented Parallel Programming

• Identify the entities being simulated (say atoms, routers, humans)

• Define the computational tasks being performed (think force computation)

• Create C++ classes to encapsulate them

• Use member functions to interact

• What about processors? Do you really want to worry about them?
Overdecomposition

- Decompose the work units & data units into many more pieces than execution units
  - Cores/Nodes/..

- Not so hard: we do decomposition anyway
Migratability

• Allow these work and data units to be migratable at runtime
  – i.e. the programmer or runtime, can move them

• Consequences for the app-developer
  – Communication must now be addressed to logical units with global names, not to physical processors
  – But this is a good thing

• Consequences for RTS
  – Must keep track of where each unit is
  – Naming and location management
Asynchrony:
Message-Driven Execution

• Now:
  – You have multiple units on each processor
  – They address each other via logical names

• Need for scheduling:
  – What sequence should the work units execute in?
  – One answer: let the programmer sequence them
    • Seen in current codes, e.g. some AMR frameworks

  – Message-driven execution:
    • Let the work-unit that happens to have data ("message") available for it execute next
    • Let the RTS select among ready work units
    • Programmer should not specify what executes next, but can influence it via priorities
Realization of this model in Charm++

- Overdecomposed entities: chares
  - Chares are C++ objects
  - With methods designated as “entry” methods
    • Which can be invoked asynchronously by remote chares
  - Chares are organized into indexed collections
    • Each collection may have its own indexing scheme
      - 1D, ..7D,
      - Sparse
      - Bitvector or string as an index
  - Chares communicate via asynchronous method invocations
    • A[i].foo(....); A is the name of a collection, i is the index of the particular chare.
Message-driven Execution

Processor 0

Scheduler

Message Queue

A[..].foo(...)

Processor 1

Scheduler

Message Queue
Empowering the RTS

The Adaptive RTS can:

- Dynamically balance loads
- Optimize communication:
  - Spread over time, async collectives
- Automatic latency tolerance
- Prefetch data with almost perfect predictability
Adaptive Runtime Systems

• Decomposing program into a large number of Objects empowers the RTS, which can:
  – Migrate Objects at will
  – Schedule tasks (Dependent Execution Blocks) at will
  – Instrument computation and communication at the level of these logical units
    • Object A communicates y bytes to B every iteration
    • Sequential Block S has a high cache miss ratio
  – Maintain historical data to track changes in application behavior
    • Historical => previous iterations
    • E.g., to trigger load balancing
Benefits in Charm++

- Over-decomposition
- 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
Benefits in Charm++

- Over-decomposition
- Message-driven execution
- Migratability
- Introspective and adaptive runtime system

- Scalable Tools
  - Automatic overlap of Communication and Computation
  - Perfect prefetch
  - Compositionality
  - Fault Tolerance
  - Dynamic load balancing (topology-aware, scalable)
  - Temperature/Power/Energy Optimizations
Utility for Multi-cores, Many-cores, Accelerators:

- 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 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
Impact on communication

• With overdecomposition
  – Communication is spread over an iteration
  – Also, adaptive overlap of communication and computation

Overdecomposition enables overlap
Decomposition Challenges

• Current method is to decompose to processors
  – But 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

• Adaptive scheduling of the objects on available resources by the RTS
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 || 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
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
So, What is Charm++?

• Charm++ is a way of parallel programming based on
  – Objects
  – Overdecomposition
  – Message
  – Asynchrony
  – Migratability
  – Runtime system
Overdecomposed Objects

Parallel Address Space
Message-driven

Parallel Address Space

• Certain member functions of certain classes are globally visible
• Invocation of a member function may lead to communication
• **Charm++ Basics:**
• **Structured Dagger Notation**
• **Designing Charm++ programs, with application case studies**