Parallel Composition and Novel Programming Models

Laxmikant (Sanjay) Kale

http://charm.cs.uiuc.edu

Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana Champaign
Outline of this talk

• Some problems we face at petascale and beyond
  – Overview of our solution strategies via adaptive RTS
  – This is useful for setting up collaborations, since it is a broad review of work in my group

• Efficient parallel composition
  – What it is, obstacles
  – What is needed to support
  – Some initial languages and ideas
Enabling CS technology of parallel objects and intelligent runtime systems has led to several CSE collaborative applications.
Our Guiding Principles

• Design abstractions based solidly on use-cases
  – Application-oriented yet computer-science centered approach

• Seek an optimal division of labor between the system and the programmer
  – Corollary: No magic
    • Parallelizing compilers have achieved close to technical perfection, but are not enough
    • Sequential programs obscure too much information
Object based over-decomposition

• Let
  – the programmer decompose computation into objects
    • Work units, data-units, composites
  – Let an intelligent runtime system assign objects to processors
  – RTS can change this assignment (mapping) during execution

• Locality of data references is a critical attribute for performance

• A parallel object can access only its own data
  – Asynchronous method invocation for accessing other’s data
Object-based over-decomposition: Charm++

- Multiple “indexed collections” of C++ objects
- Indices can be multi-dimensional and/or sparse
- Programmer expresses communication between objects – with no reference to processors

System implementation

User View
Object-based over-decomposition: AMPI

- Each MPI process is implemented as a user-level thread
- Threads are light-weight and migratable!
  - <1 microsecond context switch time, potentially >100k threads per core
- Each thread is embedded in a charm++ object (chare)

Thread migration:
On many machines, we use our implementation of a technique called isomalloc first developed in PM2, by Raymond Namist, Jean Francois Mehaut, and others
Principle of persistence

Computational loads and communication patterns tend to persist, even in dynamic computations.

So, recent past is a good predictor of near future.

This enables measurement-based load balancing.
Charm++ provides a suite of load balancing strategies, including GreedyLB, RefineLB, OrbRefineLB, …
ChaNGa: Parallel Gravity

- Collaborative project (NSF)
  - with Tom Quinn, Univ. of Washington
- Gravity, gas dynamics
- Barnes-Hut tree codes
  - Oct tree is natural decomp
  - Geometry has better aspect ratios, so you “open” up fewer nodes
  - But is not used because it leads to bad load balance
  - Assumption: one-to-one map between sub-trees and PEs
  - Binary trees are considered better load balanced

With Charm++: Use Oct-Tree, and let Charm++ map subtrees to processors

Evolution of Universe and Galaxy Formation
Load balancing with OrbRefineLB

Need sophisticated balancers and ability to choose the right ones automatically
ChaNGa: Parallel Gravity Code
Developed in Collaboration with Tom Quinn (Univ. Washington) using Charm++

ChaNGa Preliminary Performance

Time per Iteration (s)

Number of Cores

Blue Gene/L
Cray XT3

12/02/2009
JointLab
Load balancing for large machines: I

• **Centralized balancers achieve best balance**
  – Collect object-communication graph on one processor
  – But won’t scale beyond tens of thousands of nodes

• **Fully distributed load balancers**
  – Avoid bottleneck but… Achieve poor load balance
  – Not adequately agile

• **Hierarchical load balancers**
  – Careful control of what information goes up and down the hierarchy can lead to fast, high-quality balancers

• Need for a universal balancer that works for all applications

Load balancing for large machines: II

• Interconnection topology starts to matter again
  – Was hidden due to wormhole routing etc.
  – Latency variation is still small
  – But bandwidth occupancy is a problem

• Topology aware load balancers
  – Some general heuristic have shown good performance
    • But may require too much compute power
  – Also, special-purpose heuristic work fine when applicable
  – Still, many open challenges
OpenAtom
Car-Parinello ab initio MD
NSF ITR 2001-2007, IBM

Molecular Clusters:
G. Martyna (IBM)
M. Tuckerman (NYU)
L. Kale (UIUC)

Nanowires:

Semiconductor Surfaces:

3D-Solids/Liquids:
New OpenAtom Collaboration (DOE)

- Principle Investigators
  - G. Martyna (IBM TJ Watson)
  - M. Tuckerman (NYU)
  - L.V. Kale (UIUC)
  - K. Schulten (UIUC)
  - J. Dongarra (UTK/ORNL)

- Current effort focus
  - QMM
  - M
  - v
  - i
  - a integration with NAMD2
  - ORNL
  - Cray

- A unique parallel decomposition of the Car-Parinello method.
- Using Charm++ virtualization, we can efficiently scale small (32 molecule) systems to thousands of processors
Decomposition and Computation Flow
Topology aware mapping of objects

3D Torus of the machine

RealSpace

States

Planes

RealSpace Prisms perpendicular to Gspace Prisms

Density

States

Planes

PairCalculator

Rectangular Gspace Prisms

GSpace
Improvements wrought by network topological aware mapping of computational work to processors

The simulation of the right panel, maps computational work to processors taking the network connectivity into account while the left panel simulation does not. The “black” or idle time processors spent waiting for computational work to arrive on processors is significantly reduced at left. (256waters, 70R, on BG/L 4096 cores)
OpenAtom: Topology Aware Mapping

BlueGene/L

Cray XT3

No. of processors

Number of cores

Time (secs per step)

Time (secs per step)
Fault Tolerance

• Automatic Checkpointing
  – Migrate objects to disk
  – In-memory checkpointing as an option
  – Automatic fault detection and restart

• “Impending Fault” Response
  – Migrate objects to other processors
  – Adjust processor-level parallel data structures

• Scalable fault tolerance
  – When a processor out of 100,000 fails, all 99,999 shouldn’t have to run back to their checkpoints!
  – Sender-side message logging
  – Latency tolerance helps mitigate costs
  – Restart can be speeded up by spreading out objects from failed processor

More info http://charm.cs.uiuc.edu/research/ft
Scalable Performance Analysis

- Trace data can be very large
  - Problems in storing
  - Problems in visualizing/understanding
- Use data mining
- Use end-of-run on-line analysis on parallel computer
- Streaming analysis: example on next slide

See: Chee Wai Lee et al, HIPS 2008, HiPC 2009, (and Chee Wai’s PhD thesis)
Example: Online (Live) Visualization of Streamed Performance Data

10-second snapshots of live NAMD detailed performance profiles from start-up (left) to the first major load-balancing phase (right) on 1024 Cray XT5 processors

Low overhead: 0.43%
Scalable Debugging Tools

1. Debugging on large machines
   - Autoinspection
   - Using fewer resources
2. Virtualized emulation
   - Separation of entities
3. Processor slicing
4. Conditional message delivery

See: Papers by Filippo Gioachin et al
http://charm.cs.uiuc.edu/papers
Including PADTAD 2008, IPDPS 2009, Teragrid 2009

12/02/2009
How to tune performance for a future machine?

• For example, Blue Waters will arrive in 2011
  – “Sustained PetaFLOPs”, at least 200,000 Power-7 cores
  – But need to prepare applications for it stating now

• Even for existing machines:
  – Full size machine may not be available as often as needed for tuning runs

• A simulation-based approach is needed
Our Approach: BigSim

• Emulation:
  – Run an existing, full-scale application using (say) 10k cores
  – Using an emulation layer that pretends to be (say) 100k cores
    • Leverages Charm++’s object-based virtualization
  – Generates message-level traces

• Trace Driven Parallel Simulation (using 10-100 procs)
  – Multiple resolution modes for sequential performance and network performance
  – Generates Timing traces just as a real app on full scale machine

• Phase 3: Analyze performance
  – Identify bottlenecks, even w/o predicting exact performance
  – Carry out various “what-if” analyses
Possible collaborations

• We have identified several collaborations:
  • BigDFT (Jean Francois Mehaut et al)
    – Has resonance with our Quantum Chemistry codes
    – Although it’s a different method: Car-Parinello
    – Could use some of the adaptive RTS in Charm++/AMPI
    – Idea: start parallelizing with AMPI
  • *PU (Raymond Namyst et al)
    – Possible integration in Charm++, so Charm++ programmers can use this
    – Also, compare with our own heterogeneity work (Kunzman, Weselowski)
  • User level threads: potential standardization
PART TWO
EFFICIENT PARALLEL COMPOSITION
Parallel Decomposition and Processors

• **MPI-style encourages**
  – Decomposition into $P$ pieces, where $P$ is the number of physical processors available
  – If your natural decomposition is a cube, then the number of processors must be a cube
  – …

• **Charm++/AMPI style “virtual processors”**
  – Decompose into natural objects of the application
  – Decomposition is independent of number of cores
What I mean by Parallel Composition

• Imagine 3 parallel modules, A, B, C
  – Independently developed
  – No data/control dependence between B and C
  – A1; (B || C); A2;
  – We should be able to do this without modifying B and C

• What I mean by efficient parallel Composition
  – Don’t achieve modularity at the cost of performance
    • Because you can’t: programmers will break modularity to achieve performance
Decomposition independent of numCores

- Rocket simulation example under traditional MPI vs. Charm++/AMPI framework

![Diagram showing decomposition of solids and fluids](image-url)
Why Support Parallel Composition?

• It is important to support compositionality
  – For multi-module, multi-physics, multi-paradigm applications..

• This is not well-supported 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.
NAMD: A Production MD program

NAMD
- Fully featured program
- NIH-funded development
- Distributed free of charge (~20,000 registered users)
- Binaries and source code
- Installed at NSF centers
- User training and support
- Large published simulations
Parallelization using Charm++

Shallow valleys, high peaks, nicely overlapped PME

Apo-A1, on BlueGene/L, 1024 procs
Charm++’s “Projections” Analysis too

Time intervals on x axis, activity added across processors on Y axis

94% efficiency

green: communication
Red: integration
Blue/Purple: electrostatics
Orange: PME
turquoise: angle/dihedral

Graph type:
- Line Graph
- Bar Graph
- Area Graph
- Stacked

X-axis scale:
- X-Axis Scale: 1.0

Y-axis scale:
- Y-Axis Scale: 1.0
Performance of NAMD

STMV: ~1 million atoms
ApoA1: ~92K atoms

Blue Gene results based on work on DCMF many-to-many pattern by Sameer Kumar, IBM Research

No. of cores

12/02/2009 JointLab
Raising the Level of Abstraction

• What we have seen so far, will help:
  – by automating resource management, FT, modularity ..

• But: parallel interactions are still low-level
  – Programmer has to spend considerable efforts in spelling out interactions, and data decompositions, in particular

• Clearly, we need new programming models
Simplifying Parallel Programming

• By giving up completeness!
  – A paradigm may be simple, but
    • not suitable for expressing all parallel patterns
    • yet, if it can cover a significant classes of patterns (applications, modules), it is useful

• Our own examples: both outlaw non-determinism
  – Charisma: Static data flow among collections of objects
    • LCR’04, HPDC ’07
  – Multiphase Shared Arrays (MSA): restricted shared memory
    • LCPC ’04
Observation: many CSE applications or modules involve static data flow in a fixed network of entities.

The amount of data may vary from iteration to iteration, but who talks to whom remains unchanged.

Observation 2:
It is sometimes better software engineering to separate parallel and sequential code.

- **Program consists of**
  - Orchestration (.or) code
    - Chare arrays declaration
    - Orchestration with parallel constructs
    - Global flow of control
  - “Physics” code
    - Regular C++
    - User variables
    - Sequential methods
while (e > threshold)
    forall i in J
    <+e, lb[i], rb[i]> := J[i].compute(rb[i-1], lb[i+1]);
Simplifying Parallel Programming

• By giving up completeness!
  – A paradigm may be simple, but
    • not suitable for expressing all parallel patterns
    • yet, if it can cover a significant classes of patterns (applications, modules), it is useful
  – A collection of incomplete models, backed by a few complete ones, will do the trick

• Our own examples: both outlaw non-determinism
  – Multiphase Shared Arrays (MSA): restricted shared memory
    • LCPC ‘04
  – Charisma: Static data flow among collections of objects
    • LCR’04, HPDC ‘07
MSA: Multiphase Shared Arrays

Observations:
General shared address space abstraction is complex
Certain special cases are simple, and cover most uses

- In the simple model:
- A program consists of
  - A collection of Charm threads, and
  - Multiple collections of data-arrays
    - Partitioned into pages (user-specified)
- Each array is in one mode at a time
  - But its mode may change from phase to phase
- Modes
  - Write-once
  - Read-only
  - Accumulate
  - Owner-computes
A View of an Interoperable Future

Interoperability, Composibility, Resource Management

Virtualization based on Migratable Objects supported by an Adaptive Runtime System
Summary

• Over-decomposition with adaptive Runtime System
  • Decomposition and Compositionality
  • Dynamic Load balancing
  • Fault Tolerance
    – Fault tolerance
    – Scalable Performance analysis, and debugging
    – Use simulation to do early performance tuning (BigSim)

• Raise the level of abstraction
  – Via “Incomplete yet simple” paradigms, and
  – Domain-specific frameworks
  – Supported by an interoperable adaptive runtime system

• Exciting times ahead

More Info: http://charm.cs.uiuc.edu /