On the Impact of Execution Models: A Case Study in Computational Chemistry

DANIEL CHAVARRÍA-MIRANDA, MAHANTESH HALAPPANAVAR, SRIRAM KRISHNAMOORTHY, JOSEPH MANZANO, ABHINAV VISHNU, ADOLFY HOISIE

2015 Large-Scale Parallel Processing workshop (LSPP)
29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)
May 18-21, Hyderabad, India
Future extreme scale computing systems will be significantly power constrained

Current petascale era software & hardware ecosystem:
- Does it need to evolve?
- Or be completely replaced?

Many of these questions relate to the underlying execution model

Execution model:
- Conceptual framework describing orchestration of computation on parallel hardware & software resources
- Connects application & algorithms to the underlying architecture & systems software

How do we keep thousands of compute nodes busy?
- Load balancing problem
- Under different execution models
Execution Models

- **Communicating Sequential Processes (CSP)** as defined in *Hoare’78* ¹
  - Core execution model at the heart of MPI-1 *two-sided* communication

- **Bulk Synchronous Processing (BSP)** as defined in *Valiant’90* ²
  - Core execution model for many PGAS environments
  - MPI-RMA, OpenSHMEM, ComEx, GASNet, Global Arrays, etc.
  - MapReduce-like systems

- Load balancing can be implemented under both CSP & BSP

- **Shared memory execution models:**
  - Considered under a single umbrella in this work (Shared Address Space, SAS)
  - *Key feature*: direct access to a common data store

- Study the **performance impact** of different combinations of **execution models** together with load balancing techniques
  - *Target*: modern multicore clusters

Case study using the Self-Consistent Field (SCF) method
  - Electronic structure calculation in computational chemistry

Several important challenges for execution models:
  - Irregular work distribution
  - Dependent on structural properties of the input
  - Block-sparse data accesses
  - Tradeoffs between locality & load balance

Lessons from SCF are broadly applicable

Explored elements:
  - Execution models
    - CSP, BSP, Hierarchical CSP & BSP with SAS
  - Load balancing
    - Novel semi-matching formulation
    - Hypergraph partitioning
    - Work stealing
Outline

- Motivation
  - Self-Consistent Field method
- Work Partitioners
- Execution Model Variants
  - CSP only: MPI
  - BSP only: Global Arrays
  - CSP + SAS, BSP + SAS
  - Work stealing
  - Work stealing + SAS
- Experimental Results
- Conclusions
Fundamental quantum chemistry calculation
- Used to build the Hartree-Fock matrix
- Building block for higher-order methods: Coupled Cluster, Density Functional Theory

Dominant computational kernel in SCF:
- Two-electron contribution to the Fock matrix

Principal data structures: Schwarz, density & Fock matrices
- 2D block distribution amongst processes on a cluster

Computationally sparse $n^4$ calculation over $n^2$ data space
- $n$ is number of basis sets in input
- Set of $n^4$ tasks over the data space
- Each task: reads tiles from Schwarz & density matrices, accumulate results onto tiles of the Fock matrix
- Most tasks do not contribute significantly to the result
- < 1% of tasks do contribute, for large inputs
Pure locality-based schedule:

- Maximizing locality with respect to data tile access produces severe load imbalance.
- Computational cost of each task varies (roughly proportional to number of non-zeroes in data tiles).
Given these challenges from the SCF application
- Explore best options for mapping it efficiently onto a cluster

Deal with load imbalance first

Task weights:
- Each task accesses two distinct data tiles from Schwarz matrix
- Examine all elements in the Cartesian product of the tiles
- Weight corresponds to number of non-zeroes in the product

Map tasks to processes on the cluster:
- Such that the sum of the task weights per process is approximately equal

Two static approaches:
- Hypergraph partitioning
- Weighted semi-matching over bipartite graph
PaToH hypergraph formulation:

- Each task accesses six different tiles (read 4 tiles of Schwarz & density, write 2 tiles of Fock)
  - However, only four unique sets of coordinates

Multi-constraint formulation:
- Equalize weight and number of tasks per process
- Larger number of lighter tasks is not equivalent to a few heavy tasks
Weighted semi-matching formulation:

Bi-partite graph with tasks & processes

Single-constraint formulation:
- Equalize sum of task weights per process
Outline

- Motivation
- Self-Consistent Field method
- Work Partitioners
  - Execution Model Variants
    - CSP only: MPI
    - BSP only: Global Arrays
    - CSP + SAS, BSP + SAS
    - Work stealing
    - Work stealing + SAS
- Experimental Results
- Conclusions
MPI is the de facto programming model for clusters
- Core execution model is CSP

Basic concept: need a two-sided communication schedule
- What does each rank need to send and receive?
- Use static work partitioners and 2D data decomposition to create schedule

Computation & communication macro-steps
Main difference with CSP:

- One sided communication enables “position-independent” representation of tasks
- Can execute on any process since data accesses are specified in absolute/global terms
- No need to build communication schedule

Key concept to enable work stealing across processes
CSP + SAS, BSP + SAS

- Couple CSP & BSP implementations with SAS execution model
  - As realized in the OpenMP programming model
- Two-level load balancing:
  - Inter-node using CSP or BSP
  - Intra-node load balancing across threads
- CSP + SAS
  - Implemented on top of the hypergraph partitioning approach
  - Master thread performs communication
  - All threads access communicated data from shared buffers
  - Synchronization to prepare write-back (Fock) buffer
- BSP + SAS
  - Communication is done inline by all threads (synchronized by locks)
  - Not much overhead due to large computational load
Work Stealing

- Distributed work stealing across processes on a cluster
- Dynamic adaptivity in the presence of load imbalance
- Two variants:
  - Per-core work stealing
  - Work stealing + SAS
- Both use one-sided communication to access task queues on remote compute nodes
- Persistence-based approach:
  - Initial seeding of task queues based on pure locality approach
  - Keep track of which actual tasks were executed by the process to seed queues for following iterations
- Work stealing + SAS
  - Steal tasks at a coarser granularity
  - Execute them using OpenMP work sharing constructs
Outline

- Motivation
- Self-Consistent Field method
- Work Partitioners
- Execution Model Variants
  - CSP only: MPI
  - BSP only: Global Arrays
  - CSP + SAS, BSP + SAS
  - Work stealing
  - Work stealing + SAS

☞ Experimental Results

▶ Conclusions
Experimental Results

- Ran on up to 2048 cores of PNNL’s Olympus cluster
  - Dual-socket AMD Interlagos processors (16 cores per sockets), 64GB RAM per node
  - Use every other core for runs due to shared floating point units (Bulldozer)
  - QDR Infiniband interconnect
- 16 processes per node or 16 threads w/single process for OpenMP runs
- Two input decks:
  - 256 atoms of Beryllium (Be), 356 atoms of Be
- Work stealing granularities:
  - Process-based: 1 task
  - SAS-based: 1024 tasks
    - To reduce steal overhead and keep threads busy

<table>
<thead>
<tr>
<th>Intra-node/Inter-node</th>
<th>BSP</th>
<th>CSP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WS</td>
<td>SM</td>
</tr>
<tr>
<td>Process-centric</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OpenMP guided (MT)</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

*Table 1:* SCF two-electron kernel versions where WS is Work Stealing, SM is Semi-Matching, and HG is Hypergraph.
Experimental Results (cont.)

Process Execution Breakdown: 256 Atoms

MT Execution Breakdown: 256 Atoms

Process Execution Breakdown: 352 Atoms

MT Execution Breakdown: 352 Atoms
Studied a large number of execution model variants for SCF benchmark
- Different communication primitives, task scheduling, concurrency

Statically scheduled versions can match and sometimes exceed work stealing-based version

Semi-matching executes fast for a static partitioning
- Can produce lower quality partitionings

Hypergraph is better but very slow

System wide dynamic adaptation
- Requires the right kind of communication & concurrency primitives

Execution model design choices & assumptions can limit critical optimizations
- Such as global, dynamic load balancing

Future work: consider other execution & programming models, improve accuracy of static partitioning formulations