Parallel Programmability
and the **Cowichan Problems**

Andrew Borzenko
Cameron Gorrie

The Parallel Problem

- Parallel hardware is becoming ubiquitous faster than developers are learning how to use it effectively.
  - as a result, hundreds of **parallel programming systems** for various computer systems and programming languages have cropped up—with various degrees of polish, usefulness, and adoption
- We are conducting a survey of these systems' **programmability.**

Different Paradigms

- These different **parallel programming systems** have been created to abstract away the backend and create precise, succinct, and useful ways to define parallel programs
  - many commonalities between systems
- We group the systems with certain commonalities into different categories of **parallel paradigms**.
- e.g. **Message Passing**, **Shared Variables**, **Data-Parallel**, **Tuple Space**, **Channel-Based Parallelism**, etc.

Programmability

- We define **programmability** as a qualitative metric encompassing effort on the part of the application developer—how difficult it is to get ideas “onto paper” with the parallel system (intuitiveness).
- Our survey hopes to convey the sense of usefulness of parallel systems with a “bang-for-your-buck” approach

Programmability, continued

- What exactly are we looking for?
  - **ease of integration** into existing applications
  - **support**, **user base**, and **documentation**
  - **code size**
  - code **readability**
  - ease of installation/code **portability**
  - **licensing** issues
- We look for anything that might make the parallel system a pain to use!

The Cowichan Problem Set

- Wilson [94] developed a set of “toy” problems to implement using various parallel systems to assess their degrees of programmability, dubbed the “**Cowichan problems**”.
- We have implemented an updated version of the Cowichan problem set using two state-of-the-art parallel systems in order to assess their ease of implementation and overall usefulness.
- By using this problem set, we standardize our metric and get performance evaluation for free.

The Cowichan Problem Set

- **Examples** of some of the Cowichan problems.
- The ideal parallel programming system would be able to solve each problem in an amount of code that is proportional to the amount of explaining the problem needs in words.
  - examples of intuitive devices (parallel_for, ...)
  - Intuitive concepts that build on each other
- We ask questions like, “why does this concept take so much code to implement using this system?” and “what freedom do I give up by using this system?”

Message Passing Interface

- Message Passing Interface (MPI) is a specification for an API that allows many computers to communicate with one another.
- Boost.MPI is a library for message passing in high-performance parallel applications.
- Boost MPI is not a completely new parallel programming library. Rather, it is a C++-friendly interface to the standard Message Passing Interface, the most popular library interface for high-performance, distributed computing.
- MPICH is an MPI implementation that efficiently supports various computation and communication platforms.
- There are other implementations of MPI, but we used MPICH since it is platform independent (in particular, it runs on windows)

Sources: boost.org, www.mcs.anl.gov

Threading Building Blocks

- TBB is a high-level parallelism library developed by Intel for C++ developers, incorporating many basic threading ideas not covered by system call interfaces
  - It also contains primitives and classes that implement a form of the data-parallel paradigm; refer to this as TBB/DP
- The **data-parallel** paradigm (DP) expresses work-sharing by doing exactly the same thing to (potentially) large collections of data
  - Worker sub-processes *cannot communicate*
  - Programmer defines the data domain (*where*), and the operation to do with that data (*what*)
  - **Key Insight:** TBB decides the *how* (i.e. splitting up the range onto different workers)

Parallel Implementation

- Assuming we have a good serial implementation of a solution to a problem, we want to create a parallel solution.
- What is the effort required (time) to transform it into parallel code?
  - can be hard to estimate until you actually write the code
- What is the benefit (speedup) from parallelizing the code?
  - this very much depends on the problem being solved

Parallel Implementation

- These factors influence the effort required on the part of the programmer:
  - How much code needs to be changed?
  - How complex are the changes?

Example One

Matrix-Vector Product

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} x_1 y_1 + x_2 y_2 + x_3 y_3 \\ x_4 y_1 + x_5 y_2 + x_6 y_3 \\ x_7 y_1 + x_8 y_2 + x_9 y_3 \end{bmatrix}$$

Example One

- We looked at the serial, TBB, and MPI implementations of a matrix-vector product.
  - Both parallel implementations **scale incredibly well** given enough data (>1.95 when using 2 processors).
- Most effort goes into:
  - **TBB**: typing up the class.
  - **MPI**: writing the function to split work.

Example Two
Mandelbrot Set Generation

Key issue to deal with when parallelizing this code:

Mandelbrot set matrix is **very irregular**, i.e. for example the top half of the matrix might take twice as long to compute.

Example Two

- For the **TBB** implementation:
  - the layout of the class is similar to Product
  - the programmer does NOT deal with workload balancing
- Thread Building Blocks **parallel_for** implementation is powered by high level abstraction - task. Task stealing is used to do load balancing behind the scenes.

Example Two

- **MPI** scales slightly better (on 2 cores, 1.6 compared to 1.4 for TBB), but requires an **additional process** for balancing/distributing the load.
- Most effort goes into:
  - **TBB**: reformatting original implementation to be object oriented; this may not be necessary if the original implementation is already OO.
  - **MPI**: constructing a task farm manually.

Example Three
Convex Hull Computation

Quickhull is a **divide-and-conquer algorithm** which
recursively divides the set of points into two sets
based on the point of maximum distance from a line.
The convex hull is then computed from the most radical points.

Example Three

- For the **TBB** implementation:
  - NOTE: the initial computation of left-most and right-most points is not parallelized here; to do so, you can define an additional class (MinX_MaxX_Point) that is similar to MaximumPoint.

Example Three

- Threading Building Blocks performs slightly better for Hull (1.75 for TBB, 1.55 for MPI given 2 cores).
- Most effort goes into:
  - **TBB**: Writing a separate class for each type of reduction. Once again OO program structure is forced.
  - **MPI**: Defining extra datatypes for reductions, defining reduction functions, adding serialization routines.
- Neither is hard to get right, but the number of lines of code changed is quite large.

Example Four

Vector Normalization

$$\begin{bmatrix} 10 \\ 22 \\ 17 \end{bmatrix} \Rightarrow \frac{1}{\sqrt{873}} \begin{bmatrix} 10 \\ 22 \\ 17 \end{bmatrix}$$

Example Four

- TBB performs much better than MPI for Norm.
  - The norm operation is so fast that it takes longer to transfer the resulting vector to all processes than to perform the actual computation serially.
- Speedup: 1.95 for **TBB**, 0.65 for **MPI** (using 2 cores).
- Of course, results may vary depending on the system. These particular tests were done on an Intel Q9450 clocked at 3.6Ghz.
- Norm is most likely limited (using Boost.MPI) by the memory copying from one process to another.

The Survey So Far

TBB PROS

- Object-Oriented Design produces nice, clean code
- Easy setup (as simple as including a library)
- Decent performance for most Cowichan problems
- Scales well (tested for small number of processes)

TBB CONS

- Very Object-Oriented Design; possibly requires a lot of refactoring.
- **No distributed memory**/cluster support (however, multi-processor shared-memory computers are rapidly developing now)

BOOST.MPI PROS

- Works/tested with many different MPI back-ends (and thus legacy systems)
- Cleaned-up, lean, and object-oriented interface as compared to base MPI
- Built-in object serialization helps with communicating custom objects.
- "skeletons" to improve communication performance.

BOOST.MPI CONS

- Not straightforward setup; must run program with wrapper stub
- **Expensive** communications limit performance.
- Doesn't take advantage of shared memory in multi-processor setups.
- **Whole program** run from multiple processes