

---

---

# Maintaining Efficiency and Scalability in a Wildly Extensible Physics Simulation Code

---

---

**Patrick Miller**

***Center for Applied Scientific Computing***

**The Conference on High Speed Computing, Salishan Lodge**

**Glenden Beach, OR, April, 2002**

**April 23, 2002**



# UCRL-PRES-xxxxxx

---

---

Work performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48

# Abstract

---

---

In a research context, the context of parallel studies and the key to its acceptance is the efficiency and scalability of the algorithms involved. This forward looking nature of research is to prepare for truly massively parallel machines of tens of thousands of processors. Production applications, on the other hand are more focused on the "now" or the near future. The concern of the "apps" developer is to improve usability through both performance enhancements and capability enhancements. Performance enhancements are challenging in this environment because an apps code is alive and constantly changing. Typically apps codes are developed by larger teams making it still more difficult to apply performance enhancements. The teams typically are cross-disciplinary, so responsibility and the knowledge necessary for code changes may be diffused across individuals or disciplines!

Here, we will examine a modern code system designed to be modular, extensible, and long-lived. The system is dynamically driven by a top-level interpreter, so even the component composition isn't necessarily known by runtime. Despite these drawbacks, efficiency and scalability are still crucial to massively parallel performance. We will show the systems engineering approach that led to the structure of the code, its simplifying assumptions, and the costs and the limits these choices imply. We will close with some ideas for possible future directions.

# Overview

---

---

- What is a “*Wildly Extensible Code*”
  - Characteristics
- How is that different from a “*Research Kernel*” or “*Parallel Component.*”
- Why scaling and efficiency aren’t problematic
  - Scary things that didn’t come to pass
- Why scaling and efficiency are problematic
  - Scary things that are likely to come to pass
- What the future holds

# Characterization of our “*Wildly Extensible Code*”

---

---

- Focus on capability and time to solution
- Use of “modern” languages
- High levels of abstraction
- Heavy reliance on 3<sup>rd</sup> party libraries
- Modular or component-wise construction
- Constant integration of new modules/packages

# The W.E.C. team

---

---

- **Human cost drives many decisions**
  - **Maintenance cost**
  - **Training cost**
  - **Recruiting cost**
- **Big, inter-disciplinary team**
- **“Lives” together**
- **Widely varying levels of programming expertise**
  - **Multiple focus, multiple jobs**
  - **Reliance on in-house training and “new” grads**
  - **1<sup>st</sup> or 2<sup>nd</sup> job for most team members**

# My apps-centric view of a parallel research kernel

---

---

- **Small development team**
  - **Parallelism & speedup is the goal**
  - **Deep knowledge of parallel tools**
  - **High expertise**
  - **Years of experience**
  - **Multiple applications**
- **Shared deep knowledge of internal data and flow of control**
- **Single, clean abstraction**
- **Tight control of the problem, execution parameters, and environment**

# How do “*Research Kernels*” and “*Parallel Components*” differ from a W.E.C

---

---

- Freedom to experiment with implementation
  - Rebuild from scratch IS an option
- Consequences of radical change are less dramatic
- Shorter time frames
- Often, research sets direction and focus
- Typically “lives” outside the code team
- Components are less interwoven

# A Concrete Example: The KULL project

---

---

- **Joint A/X Division ICF code**
- **Mostly C++ with Python, F77, F90, a bit of C**
- **Framework is a Python interpretive shell**
- **32+ full- and part-time developers**
- **About 700 classes & 7000 function points**
- **1600 files**
- **450,000 lines of code**

# Small Parallel footprint

---

---

- **32 core files contain MPI calls**
- **Only 28 unique MPI calls and 28 unique parameters**
- **12 C++ OMP pragmas, 24 in FORTRAN**
  - **OMP restrictions disallow the use of iterators**

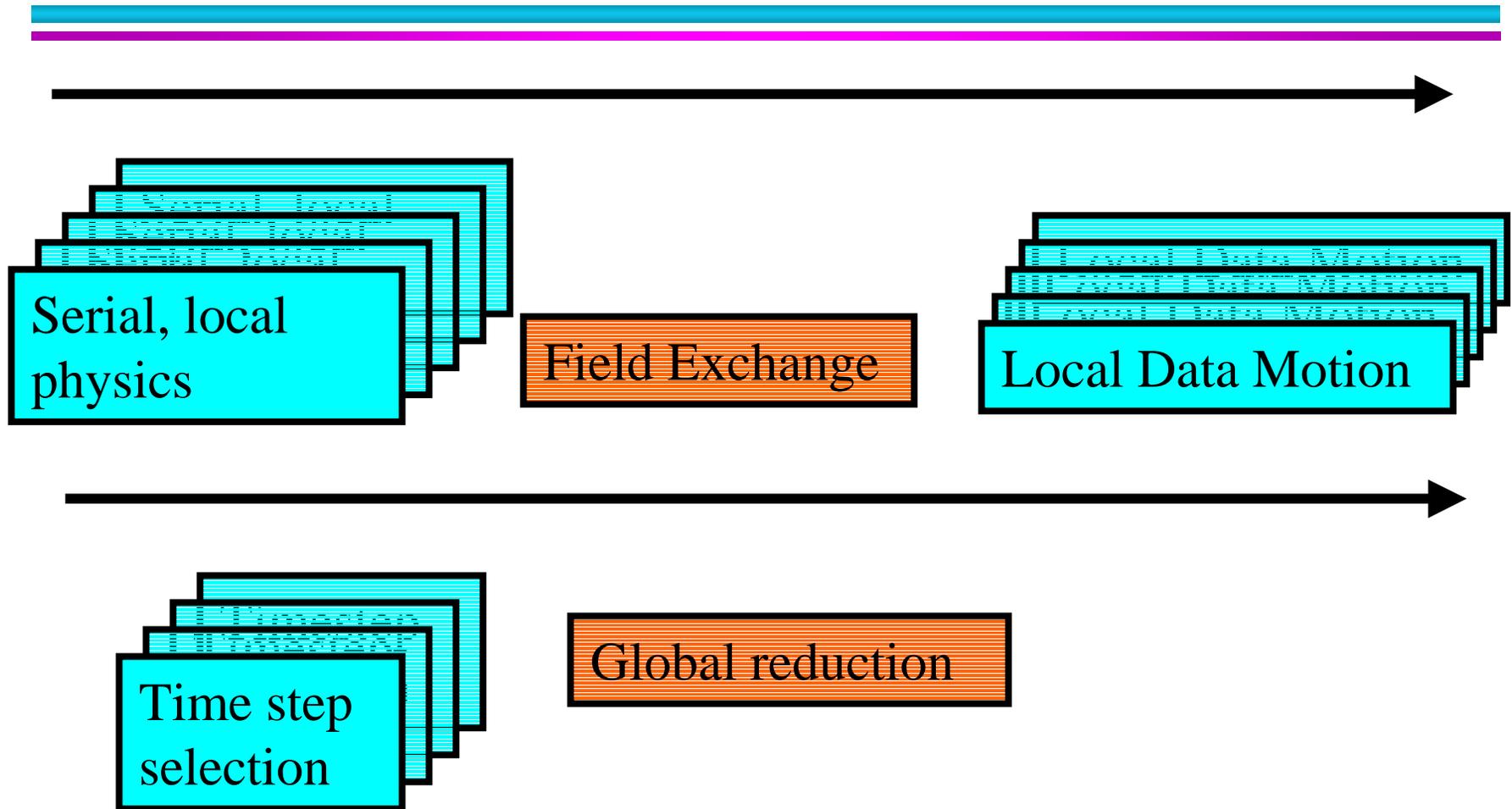
# MPI symbols in KUIII

---

---

- **MPI\_2DOUBLE\_PRECISION, MPI\_ALLGATHER, MPI\_ALLREDUCE, MPI\_ALLTOALL, MPI\_Allreduce, MPI\_Barrier, MPI\_Bcast, MPI\_CHAR, MPI\_COMM\_WORLD, MPI\_Comm, MPI\_Comm\_dup, MPI\_Comm\_rank, MPI\_Comm\_size, MPI\_DOUBLE, MPI\_Datatype, MPI\_ERRORS\_RETURN, MPI\_Errhandler\_set, MPI\_Error\_string, MPI\_Finalize, MPI\_Get\_count, MPI\_INT, MPI\_INTEGER, MPI\_Init, MPI\_Irecv, MPI\_Isend, MPI\_MAX, MPI\_MAXLOC, MPI\_MAX\_ERROR\_STRING, MPI\_MIN, MPI\_MINLOC, MPI\_Op, MPI\_PACKED, MPI\_PROD, MPI\_Pack, MPI\_Pack\_size, MPI\_REAL8, MPI\_RECV, MPI\_REQUEST\_NULL, MPI\_Recv, MPI\_Request, MPI\_Request\_free, MPI\_SEND, MPI\_STATUS\_SIZE, MPI\_SUCCESS, MPI\_SUM, MPI\_Scan, MPI\_Send, MPI\_Status, MPI\_Type\_commit, MPI\_Type\_contiguous, MPI\_Type\_free, MPI\_Unpack, MPI\_Wait, MPI\_Waitall, MPI\_Wtick, MPI\_Wtime**

# The IDEAL big picture



# This looks great!

---

---

- **Local physics and operator splitting are our friends!**
- **Logical extension of serial implementation**
- **Most communication operations are point-to-point**
- **Computation costs dwarf global reduction cost**
- **Load imbalance is a bigger issue than the global reduction**
- **Everything scales, serial run efficiency is the only real bottleneck, life is beautiful, buy me a bigger machine that looks exactly like the one I have now except faster!**

# Oops

---

---

- **What about other intra-package parallelism**
  - Improvements in solver technology have masked scaling issue in Diffusion solve
  - Particle methods use a vastly different idea of exchange
  - 3rd party libraries use internal parallelism as well
  - Internally generated edits & debug information are often parallel aware
- **Serial-first design led to.....**
  - Very little overlap in communication and computation
  - Over use of synchronization

# Oops(2)

---

---

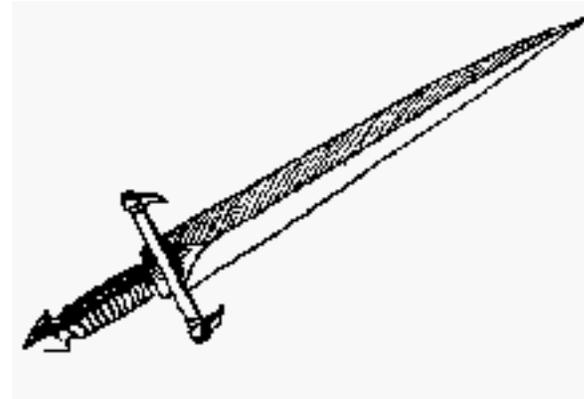
- **Needed a change in philosophy**
  - I can run bigger in the same time, but not necessarily the same problem faster
- **Assumes a simplifying, uniform view of**
  - Memory
  - Communications
  - I/O
  - Operating system
- **Trusts that the component composition is simply the “sum of the parts.”**

# The two-edged sword of abstraction

---

---

- **Abstraction costs...**
  - Optimization
  - Low-level control of memory allocation
  - Data privacy interferes with layout
  - Iterators are NOT pointers
  
- **Abstraction gains...**
  - Control hooks
  - Validity check hook
  - Type safety
  - Intelligent pre-compilers



# Many types, many function points

---

---

- **700 classes (C++ and Python)**
- **7000 function points**
- **Over 85% are 32 lines or less**

# Why scaling and efficiency aren't problematic

---

---

- **Current runtime to communication ratios do or can provide reasonable balance**
- **Parallel architecture is rather stable across a wide range of platforms**
- **Pure MPI decomposition has proven workable**
  - **For much of the code, threads are used because of system MPI limitations**
- **Simple parallel regions**
  - **Simplifies sequential implementation**
  - **Speeds package integration**
  - **Cuts developer training**

# Why scaling and efficiency are problematic

---

---

- Upcoming changes to machine architectures
  - E.g. BGL
- NUMA (memory, messaging, I/O)
- Need to program to the MPI/Threads “sweet spot”
  - Achieving good surface-to-volume ratios
- Better handle on partitioning and dynamic load balance

# What the future holds

---

---

- **Moving edits into Python and out of core code**
  - **Role of pyMPI**
- **Integrating better tools into the framework**
  - **Parallelism integrated into basic mesh and field structures**
  - **A start on a shared particle framework**
- **M-N way parallelism to cut diameter of global messaging where possible**

# EOF

