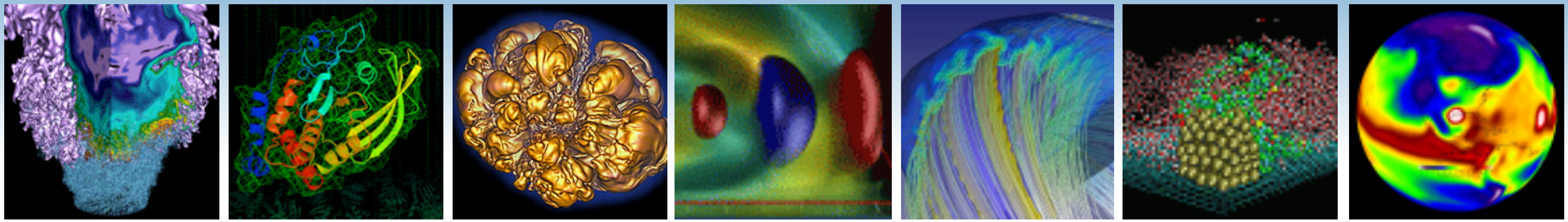
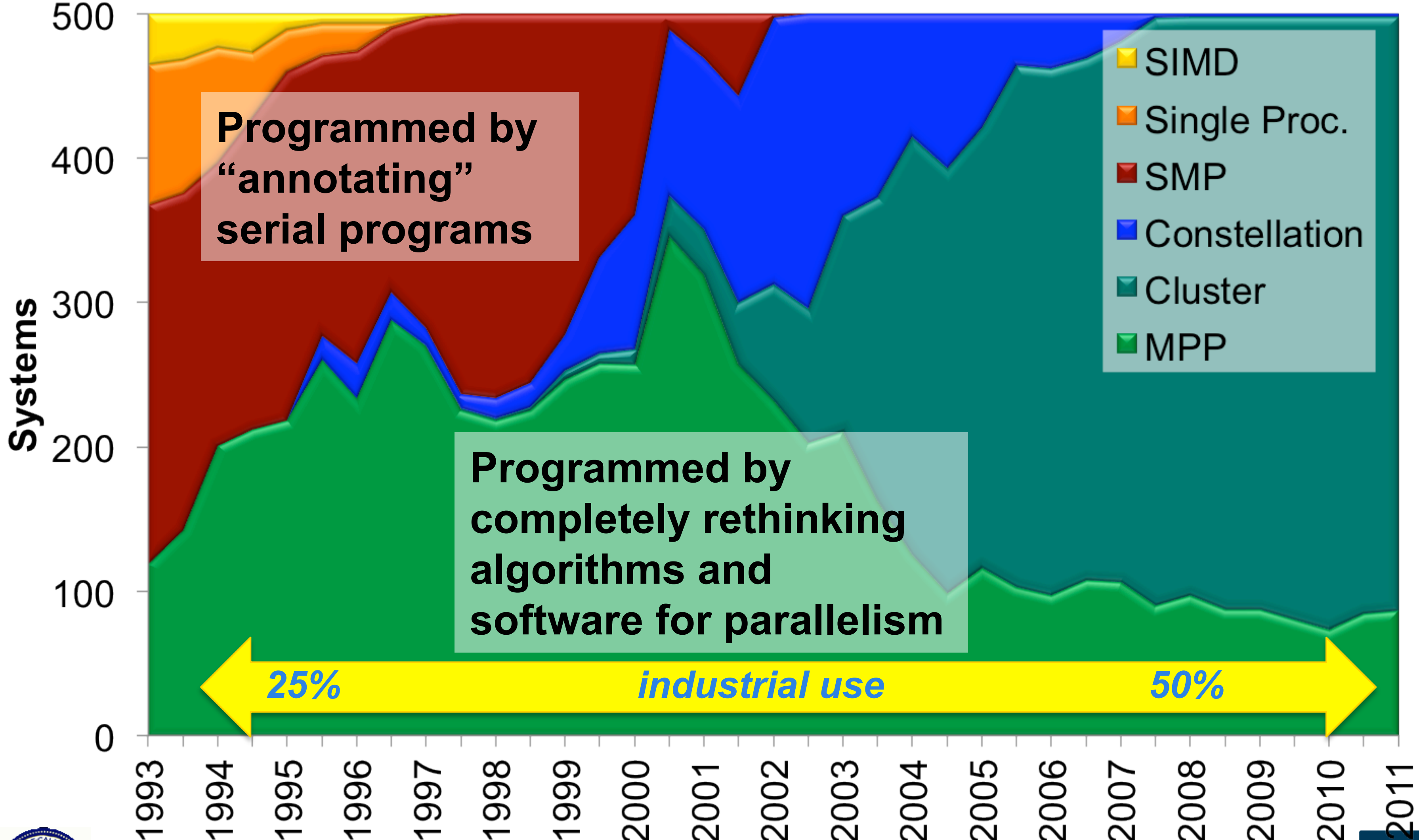


Modern Parallel Languages

Kathy Yelick

http://www.eecs.berkeley.edu/~yelick/cs294-f13

HPC: From Vector Supercomputers to Massively Parallel Systems

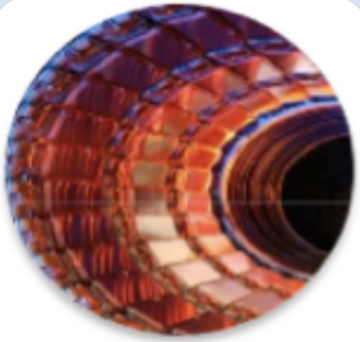


A Brief History of Languages

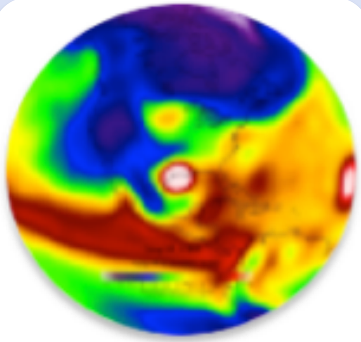
- When vector machines were king
  - Parallel "languages" were loop annotations (IVDEP)
  - Performance was fragile, but there was good user support
- When SIMD machines were king
  - Data parallel languages popular and successful (CMF, *Lisp, C*, …)
  - Quite powerful: can handle irregular data (sparse mat-vec multiply)
  - Irregular computation is less clear (multi-physics, adaptive meshes, backtracking search, sparse matrix factorization)
- When shared memory multiprocessors (SMPs) were king
  - Shared memory models, e.g., OpenMP, Posix Threads, are popular
- When clusters took over
  - Message Passing (MPI) became dominant

We are at the mercy of hardware, but we get blamed.

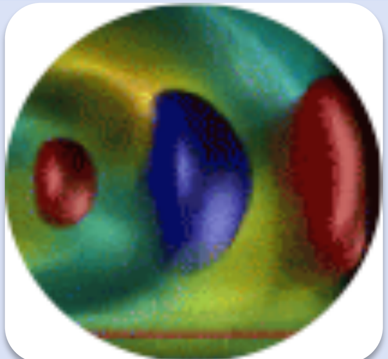
Science Across the “Irregularity” Spectrum



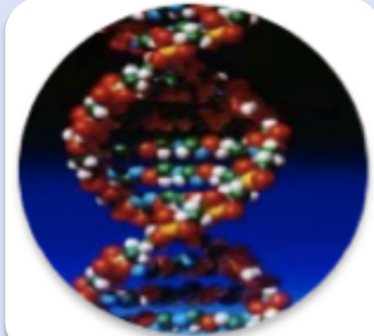
Massive Independent Jobs for Analysis and Simulations



Nearest Neighbor Simulations



All-to-All Simulations



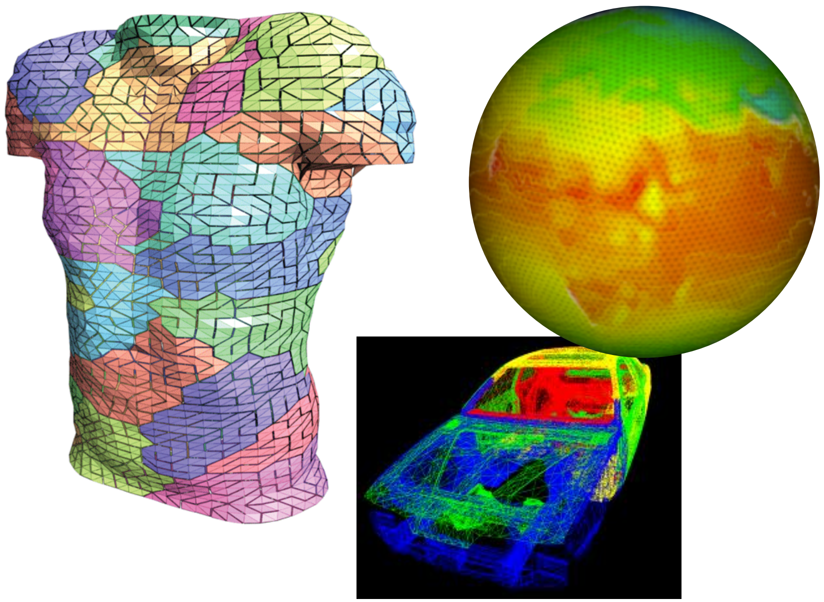
Random access, large data Analysis

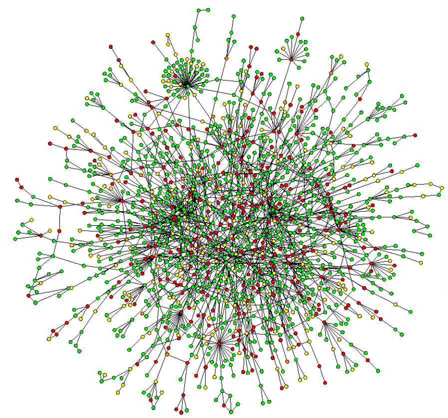
Data analysis and simulation

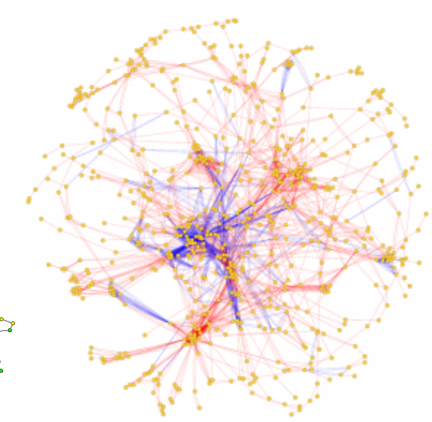
Analytics vs. Simulation Kernels:

7 Giants of Data	7 Dwarfs of Simulation
Basic statistics	Monte Carlo methods
Generalized N-Body	Particle methods
Graph-theory	Unstructured meshes
Linear algebra	Dense Linear Algebra
Optimizations	Sparse Linear Algebra
Integrations	Spectral methods
Alignment	Structured Meshes

Programming Challenges and Solutions







<u>Message Passing Programming</u>

Divide up domain in pieces

Each compute one piece

Exchange (send/receive) data

PVM, MPI, and many libraries

<u>Global Address Space Programming</u>

Each start computing

Grab whatever you need whenever

Global Address Space Languages and Libraries

Why Consider New Languages at all?

- Most of work is in runtime and libraries
- Do we need a language? And a compiler?
  - If higher level syntax is needed for productivity
    - We need a language
  - If static analysis is needed to help with correctness
    - We need a compiler (front-end)
  - If static optimizations are needed to get performance
    - We need a compiler (back-end)
- All of these decisions will be driven by application need

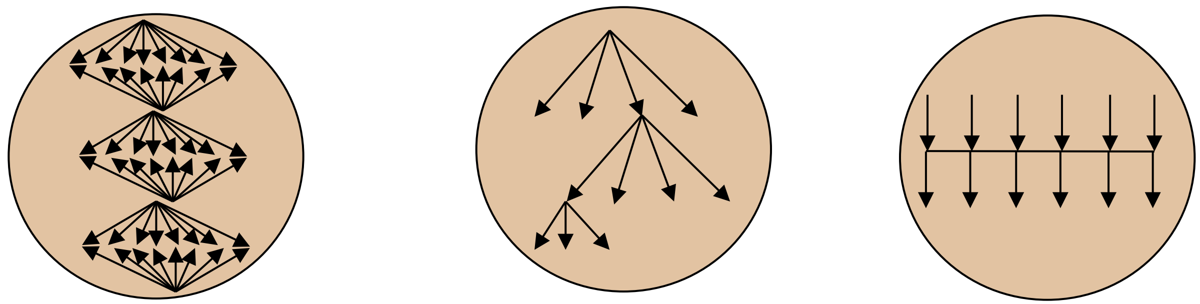
Libraries vs. Languages

- Use libraries when: clear interface between operations
  - Dense and sparse matrix operations: can be done in the library
  - FFTs and other spectral transforms
- Use compilers when: cannot be captured in a traditional library
  - Stencils on structured grids (LBMD and Heat)
  - Graph traversal algorithms
- But aren't these just higher order functions?
  - Yes, but optimization requires they are instantiated

Use an approach that matches the problem

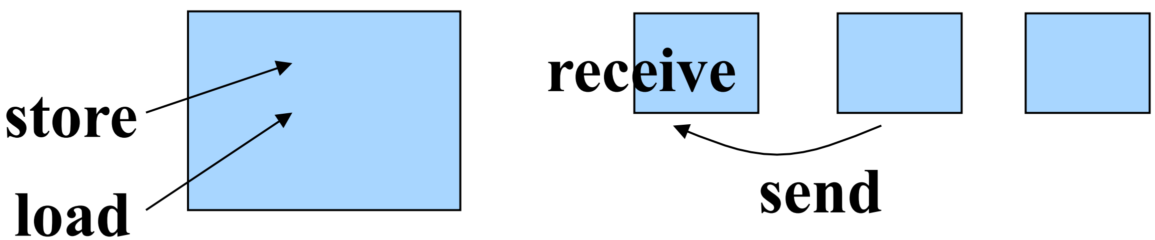
Two Parallel Language Questions

- What is the parallel control model?



data parallel (singe thread of control) | dynamic threads | single program multiple data (SPMD)

- What is the model for sharing/communication?



shared memory | message passing

implied synchronization for message passing, not shared memory

Task Cost Spectrum

Schedule a set of tasks under one of the following assumptions:

Easy: The tasks all have equal (unit) cost. — branch-free loops

Harder: The tasks have different, but known, times. — sparse matrix-vector multiply

Hardest: The task costs unknown until after execution. — GCM, circuits, search

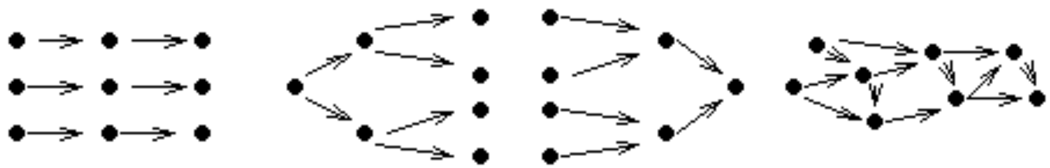
Task Dependency Spectrum

Schedule a graph of tasks under one of the following assumptions:

Easy: The tasks can execute in any order.

dependence free loops

Harder: The tasks have a predictable structure.



wave–front out–tree in–tree general dag

balanced or unbalanced

matrix computations (dense, and some sparse, Cholesky)

linear programming

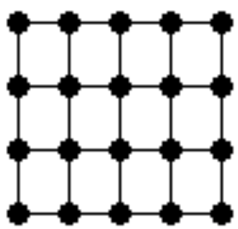
Hardest: The structure changes dynamically (slowly or quickly) search, sparse LU

Task Locality Spectrum

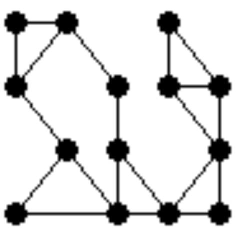
Schedule a set of tasks under one of the following assumptions:

Easy: The tasks, once created, do not communicate. — embarrassingly parallel

Harder: The tasks communicate in a predictable pattern. — unstructured and structured grids



regular



irregular

Hardest: The communication pattern is unpredictable. — discrete event simulation

Liskov's Goals of Language Design (circa 1981)

0. Well-define semantics (a requirement, not just a goal)
1. Simplicity: easy to learn; minimality of concepts
2. Generality: computationally complete
3. Expressibility
4. Writability
5. Readability
6. Efficiency of compiler and programs (implementable)
7. Uniformity, economy of concepts,
8. Familiarity (consistent with common notation)
9. Orthogonality
10. Extensible and (maybe) subsetable
11. Secure (safe)
12. Machine independent (portable)

MacLennan's Principles of Language Design

- **Abstraction:** Information about implementations should be hidden from use; recurring patterns should be reusable.
- **Orthogonality:** Independent functions should be controlled by independent mechanisms.
- **Portability:** Avoid features or facilities that are dependent on a particular machine or a small class of machines.
- **Automation:** Automate mechanical, tedious, or error-prone activities.
- **Redundancy:** Have a series of defenses so that if an error isn't caught by one, it will probably be caught by another.
- **Transparency:** Expensive things should look expensive.
- **Localized Cost:** Users should only pay for what they use; avoid distributed costs.
- **Consistency:** Regular rules, without exceptions, are easier to learn, use, describe, and implement. Similar things should look similar; different things different.
- **Security:** No program that violates the definition of the language, or its own intended structure, should escape detection.
- **Simplicity:** A language should be as simple as possible. There should be a minimum number of concepts, with simple rules for their combination.

Adapted from MacLennan's Programming Language Design Principles

Rules for Language Adoption

- Community with need
- Significant advantage (performance or productivity)
- Incremental adoption path (interoperability)
  - Not the whole shebang!
- Portability
- Familiarity
  - Consider C, C++, Java history
- Access to powerful libraries!

Top Goals of Parallel Language Design

- Performance
  - Control
  - Locality
  - Parallelism and synchronization features
- Portability (main goal of productivity)
- Productivity (expressiveness, simplicity,…)
- Familiarity

- Others?

To Virtualize or Not

- The fundamental question facing in parallel programming models is:

  What should be virtualized?

- Hardware has finite resources
  - Processor count is finite
  - Registers count is finite
  - Fast local memory (cache and DRAM) size is finite
  - Links in network topology are generally < n^2
- Does the programming model (language+libraries) expose this or hide it?
  - E.g., one thread per core, or many?
    - Many threads may have advantages for load balancing, fault tolerance and latency-hiding
    - But one thread is better for deep memory hierarchies
- How to get the most out of your machine?

Programming Model Research: Early 90s

- Data-parallel languages
  - Fine-grained parallelism, similar to vectorization, with hard compiler problem to map to coarse-grained machines
  - Examples: HPF, pC++, NESL
- Task parallel
  - Especially for divide-and-conquer problems with little inherent locality
  - Small compilers with sophisticated runtime systems
  - Examples: CILK, Charm++
- Object-oriented parallel languages
  - Computation follows data
  - Examples: CC++, CA
- Global address space languages
  - Small compilers and lightweight runtimes
  - Examples: Split-C, PCP, AC, F--

What Happened to HPF?

- High Performance Fortran (data parallel)
  - Language effort by users, language researchers, vendors
  - Ambitious goals: tried to address many domains
    - Dense linear algebra: block-cyclic data layouts (index overhead)
    - Sparse, unstructured problems: irregular layouts
  - Compiler performs difficult mapping for coarse-grained machines
    - Kennedy: “10-15 years for compiler technology to mature”
  - Abandoned in the U.S. after O(5 years)
  - Seeing some success in Japan/Europe
  - Surprising Gordon Bell prize on the Earth Simulator

Bringing Users Along: UPC Experience

1991
Active Msgs are fast

1992
First AC (accelerators + split memory)

1992 First Split-C (compiler class)

1993
Split-C funding (DOE)

1997
First UPC Meeting

"best of" AC, Split-C, PCP

2001
First UPC Funding

2002 GASNet Spec

Other GASNet-based languages

2001
gcc-upc at Intrepid

2003 Berkeley Compiler release

2006
UPC in NERSC procurement

2010
Hybrid MPI/UPC

- Ecosystem:
  - Users with a need (fine-grained random access)
  - Machines with RDMA (not full hardware GAS)
  - Common runtime; Commercial and free software
  - Sustained funding and Center procurements
- Success models:
  - Adoption by users: vectors → MPI, Python and Perl, UPC/CAF
  - Influence traditional models: MPI 1-sided; OpenMP locality control
  - Enable future models: Chapel, X10,…

PyGAS: Combine two popular ideas

- Python
  - No. 6 Popular on http://langpop.com and extensive libraries, e.g., Numpy, Scipy, Matplotlib, NetworkX
  - 10% of NERSC projects use Python
- PGAS
  - Convenient data and object sharing
- PyGAS : Objects can be shared via *Proxies* with operations intercepted and dispatched over the network:

```
num = 1+2*j
    = share(num, from=0)
```

```
print pxy.real # shared read
pxy.imag = 3   # shared write
print pxy.conjugate() # invoke
```

- Leveraging duck typing:
  - *Proxies* behave like original objects.
  - Many libraries will automatically work.

Can Architectures Help Make Programming Easier?

The vicious cycle:

All high end programs
written in MPI

All machines support
only MPI well

Who is affected:

- the scientists who are not paid enough to program in MPI
- the customers who bought hardware and threw away the performance

Course Goals

- Training in well-defined language design and sufficient documentation: how to recognize good/bad specs
- Design of a future language, e.g., PyGAS
- Design of future autotuners, aka DSLs
- Language and compiler support for communication-avoiding algorithm support
- Priorities based on interests of participants

Course Mechanics

- Web page:
  http://www.cs.berkeley.edu/~yelick/cs294-f13
- Course lectures in 3 parts:
  - GP Language overviews: X10, Chapel, Titanium or UPC,…
  - Cross-cuts: Memory Consistency, Collectives, Liveness, Array abstractions…
  - Presentations by students and auditors on topics of interest: DSLs (SEJITS), communication avoiding compilers (HBL), Data languages (Hadoop), PyGAS
- Prerequisites:
  - Understanding of parallel (scientific) computing: CS267 or equivalent
  - Familiarity with multiple languages and interest in learning more!
- Grading: This is an advanced graduate class
  - Programming assignments in first half of semester
  - Final projects
- Class computer accounts at NERSC (using mp309, CS267 repo)
  - Search for "NERSC new user" which should take you to the following URL
  - https://nim.nersc.gov/nersc_account_request.php

How to fill out the NERSC Account form

NERSC New Account Request Form

Please fill out and submit this form to request a new NERSC account to be associated with an existing NERSC repository, project, or share.

If you are already a NERSC user, please contact your project's Principal Investigator to be added to an existing NERSC repository.

If you are not a NERSC user and wish to submit a proposal to create a new NERSC project, please go to "ERCAP Access to new PIs and PI Proxies. If you are an existing NERSC user and wish to submit a new project proposal, please do so through NIM.

Account Type: Standard

If you are completely new to NERSC and wish to make an allocation request as a new Principal Investigator (PI), please go to the "ERCAP Access Request Form for new PIs and PI Proxies". You can find the information on submitting an ERCAP request here: NERSC Allocations Overview and Eligibility

First Name: Jane

Middle Initial (optional):

Last Name: Doe

Preferred Username: jdoe *(If you do not already have a NERSC username, enter a preferred username. Maximum of 8 characters.)*

Citizenship:

Email Address: *Email address is required.*

Telephone: *Work phone is required.* *(format: 123-456-7890, non-U.S. users please include country code)*

Organization: USA: University of California Berkeley (if your site is not listed, click here to add it)

Principle Investigator and repository name: Kathy Yelick : mp309 – Class Account for UCB CS267 / CS194 "Applications of Parallel Computing" – Proxies:

Select the project you wish to be added to. You can search by PI name, repository, share name, or experiment, the Project's description or by a PI Proxy's name. Be sure to select from the list that is presented.

Please provide a description of the work you will be doing and the name of the person you will be working with:

(2000 character maximum) 44 characters entered. : 1956 characters remaining.

Exploring new parallel programming languages

NERSC Electronic Computer User Agreement form

The following is a list of general computer use policies and security rules that apply to individual users of NERSC. Further information on NERSC security policies and practices can be found on the NERSC Computer Security page. Principal Investigators are responsible for implementing these policies and procedures in their organization and ensuring that users fulfill their responsibilities.

User Accountability

Users are accountable for their actions. Violations of policy may result in applicable administrative or legal sanctions.

Resource Use

Resources provided by NERSC are to be used only for activities authorized by the Department of Energy (DOE) or the NERSC Director. The use of NERSC resources for personal or private benefit is prohibited. NERSC resources are provided to users without warranty. NERSC will not be held liable in the event of any system failure or loss of data. NERSC resources cannot be used for any military or defense end use or application, or to facilitate any transaction that would otherwise violate U.S. export control regulations.

Data Parallelism

- For next week, read about data parallelism:
- NESL: Nested Data parallelism
  - http://www.cs.cmu.edu/~scandal/nesl.html
  - http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-95-170.html
- HTA: Hierarchically Tiled Arrays
  - http://polaris.cs.uiuc.edu/hta/
  - http://dl.acm.org/citation.cfm?id=1122981
- Chapel (more than just data parallel)
  - http://chapel.cray.com
  - http://chapel.cray.com/spec/spec-0.93.pdf

In each case the first link is to a web page for the project, and the send is a specific paper you should read.