Crash Course in Supercomputing

Rebecca Hartman-Baker, PhD
User Engagement Group Lead
Charles Lively III, PhD
Science Engagement Engineer
Helen He, PhD
User Engagement Group

June 28, 2024

Computing Sciences Summer Student Program & NERSC/ALCF/OLCF Supercomputing User Training 2024

Today's Pipeline Continued…

Afternoon Session Overview (after Lunch)

- Introduction to OpenMP - 01:00 p.m. PDT
- Understanding OpenMP + Hybrid OpenMP Concepts
- BREAK - 02:45 p.m. - 03:00 p.m. PDT
- Interactive Hands-On Exercises &&|| Q&A

Please refer to Event Web Page for Specific Times

Introduction to OpenMP

Outline

I. About OpenMP
II. OpenMP Directives
III. Data Scope
IV. Runtime Library Routines and Environment Variables
V. Using OpenMP
VI. Hybrid Programming

I. ABOUT OPENMP

About OpenMP

- Industry-standard shared memory programming model
- Developed in 1997
- OpenMP Architecture Review Board (ARB) determines additions and updates to standard
- Current standard: 5.2 (November 2021)
- Standard includes GPU offloading (since 4.0), not discussed today

Advantages to OpenMP

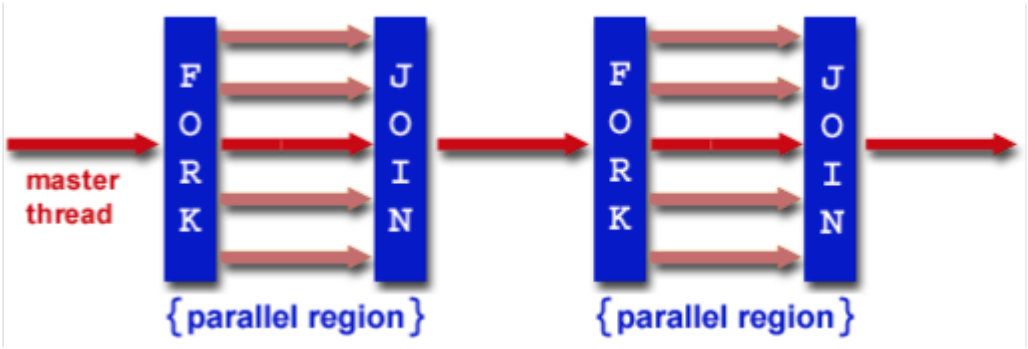
- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Can express simple or complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, so code is easy to read
- Single source code for OpenMP and non-OpenMP – non-OpenMP compilers simply ignore OMP directives

OpenMP Programming Model

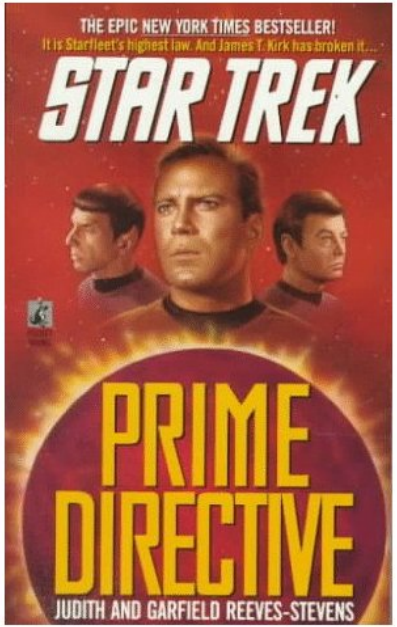
- Application Programmer Interface (API) is combination of
  - Directives
  - Runtime library routines
  - Environment variables
- API falls into three categories
  - Expression of parallelism (flow control)
  - Data sharing among threads (communication)
  - Synchronization (coordination or interaction)

Parallelism

- Shared memory, thread-based parallelism
- Explicit parallelism (parallel regions)
- Fork/join model



Source: https://hpc-tutorials.llnl.gov/openmp/



II. OPENMP DIRECTIVES

Star Trek: Prime Directive by Judith and Garfield Reeves-Stevens, ISBN 0671744666

II. OpenMP Directives

- Syntax overview
- Parallel
- Worksharing Loop
- Schedule
- Synchronization
- Reduction
- Loop

Syntax Overview: C/C++

- Basic format
  - **`#pragma omp directive-name`** *`[clause]`* **`newline`**
- All directives followed by newline
- Uses pragma construct (pragma = Greek for “thing done”)
- Case sensitive
- Directives follow standard rules for C/C++ compiler directives
- Use curly braces (not on pragma line) to denote scope of directive
- Long directive lines can be continued by escaping newline character with \

Syntax Overview: Fortran

- Basic format:
  - ***sentinel directive-name*** *[clause]*
- Three accepted sentinels: **`!$omp *$omp c$omp`**
- Some directives paired with end clause
- Fixed-form code:
  - Any of three sentinels beginning at column 1
  - Initial directive line has space/zero in column 6
  - Continuation directive line has non-space/zero in column 6
  - Standard rules for fixed-form line length, spaces, etc. apply
- Free-form code:
  - **`!$omp`** only accepted sentinel
  - Sentinel can be in any column, but must be preceded by only white space and followed by a space
  - Line to be continued must end in **`&`** and following line begins with sentinel
  - Standard rules for free-form line length, spaces, etc. apply

OpenMP Directives: Parallel

- A block of code executed by multiple threads
- Syntax:

```
#pragma omp parallel private(list) shared(list)
{
   /* parallel section */
}

!$omp parallel private(list) &
!$omp shared(list)
! Parallel section
!$omp end parallel
```

Simple Example (C/C++)

```
#include <stdio.h>
#include <omp.h>
int main (int argc, char *argv[]) {
       int tid;
       printf("Hello world from threads:\n");
       #pragma omp parallel private(tid)
       {
              tid = omp_get_thread_num();
              printf("<%d>\n", tid);
       }
       printf("I am sequential now\n");
       return 0;
}
```

Simple Example (Fortran)

```
program hello
 integer tid, omp_get_thread_num
 write(*,*) 'Hello world from threads:'
 !$omp parallel private(tid)
 tid = omp_get_thread_num()
 write(*,*) '<', tid, '>'
 !$omp end parallel
 write(*,*) 'I am sequential now'
end
```

Simple Example: Output

Output 1

```
Hello world from threads:
<0>
<1>
<2>
<3>
<4>
I am sequential now
```

Output 2

```
Hello world from threads:
<1>
<2>
<0>
<4>
<3>
I am sequential now
```

Simple Example: Output

Output 1

```
Hello world from threads:
<0>
<1>
<2>
<3>
<4>
I am sequential now
```

Output 2

```
Hello world from threads:
<1>
<2>
<0>
<3>
I am sequential now
```

Order of execution is scheduled by OS!!!

OpenMP Directives: Worksharing Loop

- Iterations of the loop following the directive are executed in parallel
- Syntax (C):

```
#pragma omp for schedule(type [,chunk]) private(list)\
shared(list) nowait
 {
   /* for loop */
 }
```

OpenMP Directives: Worksharing Loop

- Syntax (Fortran):

```
!$omp do schedule (type [,chunk]) &
!omp private(list) shared(list)
C do loop goes here
!$omp end do nowait
```

- *type* = {static, dynamic, guided, runtime}
- If `nowait` specified, threads do not synchronize at end of loop

OpenMP Directives: Scheduling

- Default scheduling determined by implementation
- Static
  - ID of thread performing particular iteration is function of iteration number and number of threads
  - Statically assigned at beginning of loop
  - Best for known, predictable amount of work per iteration
  - Low overhead
- Dynamic
  - Assignment of threads determined at runtime (round robin)
  - Each thread gets more work after completing current work
  - Load balance is possible for variable work per iteration
  - Introduces extra overhead

OpenMP Directives: Scheduling

Type	Chunks ?	Chunk Size	# Chunks	Overhead	Description
`static`	N	N/P	$\boldsymbol{P}$	Lowest	Simple Static
`static`	Y	$\boldsymbol{C}$	N/C	Low	Interleaved
`dynamic`	N	N/P	$\boldsymbol{P}$	Medium	Simple dynamic
`dynamic`	Y	$\boldsymbol{C}$	N/C	High	Dynamic
`guided`	N/A	$\leq N/P$	$\leq N/C$	Highest	Dynamic optimized
`runtime`	Varies	Varies	Varies	Varies	Set by environment variable

Note: N = size of loop, P = number of threads, C = chunk size

Which Loops are Parallelizable?

Parallelizable

- Number of iterations known upon entry, and does not change
- Each iteration independent of all others
- No data dependence

Not Parallelizable

- Conditional loops (many while loops)
- Iterator loops (e.g., iterating over `std:: list<...>` in C++)
- Iterations dependent upon each other
- Data dependence

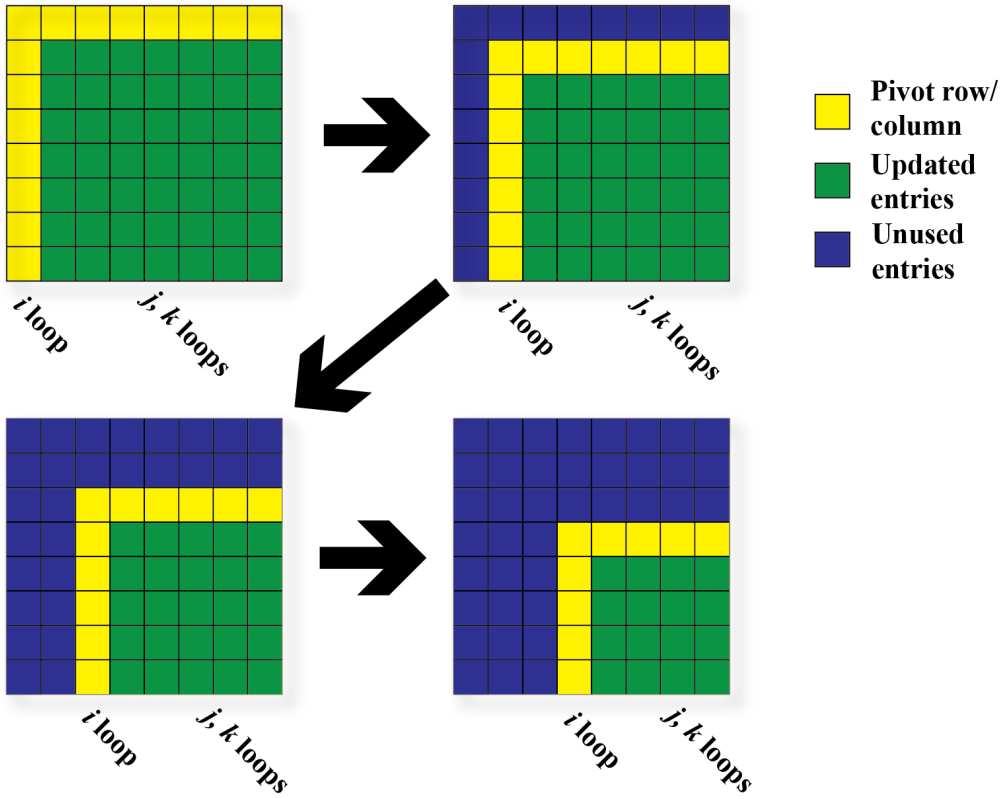
Trick: If a loop can be run backwards and get the same results, then it is almost always parallelizable!

Example: Parallelizable?

```
/* Gaussian Elimination (no pivoting):   x = A\b   */

for (int i = 0; i < N-1; i++) {
  for (int j = i; j < N; j++) {
    double ratio = A[j][i]/A[i][i];
    for (int k = i; k < N; k++) {
      A[j][k] -= (ratio*A[i][k]);
      b[j] -= (ratio*b[i]);
    }
  }
}
```

Example: Parallelizable?



Example: Parallelizable?

- Outermost Loop (`i`):
  - `N-1` iterations
  - Iterations depend upon each other (values computed at step `i-1` used in step `i`)
- Inner loop (`j`):
  - `N-i` iterations (constant for given `i`)
  - Iterations can be performed in any order
- Innermost loop (`k`):
  - `N-i` iterations (constant for given `i`)
  - Iterations can be performed in any order

Example: Parallelizable?

```
/* Gaussian Elimination (no pivoting):   x = A\b   */

for (int i = 0; i < N-1; i++) {
#pragma omp parallel for
  for (int j = i; j < N; j++) {
    double ratio = A[j][i]/A[i][i];
    for (int k = i; k < N; k++) {
      A[j][k] -= (ratio*A[i][k]);
      b[j] -= (ratio*b[i]);
    }
  }
}
```

Note: can combine **`parallel`** and **`for`** into single **`pragma`**

OpenMP Directives: Synchronization

- Sometimes, need to make sure threads execute regions of code in proper order
  - Maybe one part depends on another part being completed
  - Maybe only one thread need execute a section of code
- Synchronization directives
  - Critical
  - Barrier
  - Single

OpenMP Directives: Synchronization

- Critical
  - Specifies section of code that must be executed by only one thread at a time
  - Syntax: C/C++

    ```
    #pragma omp critical (name)
    ```

  - Fortran

    ```
    !$omp critical (name)
    !$omp end critical
    ```

  - Names are global identifiers – critical regions with same name are treated as same region

OpenMP Directives: Synchronization

- Single
  - Enclosed code is to be executed by only one thread
  - Useful for thread-unsafe sections of code (e.g., I/O)
  - Syntax: C/C++ Fortran

```
#pragma omp single
```

```
!$omp single
```

```
!$omp end single
```

OpenMP Directives: Synchronization

- Barrier
  - Synchronizes all threads: thread reaches barrier and waits until all other threads have reached barrier, then resumes executing code following barrier
  - Syntax: C/C++ Fortran

```
#pragma omp barrier          !$OMP barrier
```

  - Sequence of work-sharing and barrier regions encountered must be the same for every thread

OpenMP Directives: Reduction

- Reduces list of variables into one, using operator (e.g., max, sum, product, etc.)
- Syntax

```
#pragma omp reduction(op : list)
!$omp reduction(op : list)
```

  - where list is list of variables and op is one of following:
    - C/C++: `+, -, *, &, ^, |, &&, ||, max, min`
    - Fortran: `+, -, *, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor`

OpenMP Directives: Loop

- Iterations of the loop following the directive are executed in parallel
- omp loop gives an OpenMP implementation the freedom to choose the best parallelization scheme
- Syntax (C):

```
#pragma omp loop private(list)\  shared(list) nowait
 {
   /* for loop */
 }
```

OpenMP Directives: Loop

- Syntax (Fortran):

```
!$omp loop &
!omp private(list) shared(list)
! do loop goes here
!$omp end loop nowait
```

- omp loop gives an OpenMP implementation the freedom to choose the best parallelization scheme
- If `nowait` specified, threads do not synchronize at end of loop

III. VARIABLE SCOPE

"M119A2 Scope" by Georgia National Guard, source: http://www.flickr.com/photos/ganatlguard/5934238668/sizes/l/in/photostream/

III. Variable Scope

- About variable scope
- Scoping clauses
- Common mistakes

About Variable Scope

- Variables can be shared or private within a parallel region
- Shared: one copy, shared between all threads
  - Single common memory location, accessible by all threads
- Private: each thread makes its own copy
  - Private variables exist only in parallel region

About Variable Scope

- By default, all variables shared *except*
  - Index values of parallel region loop – **private by default**
  - Local variables and value parameters within subroutines called within parallel region – **private**
  - Variables declared within lexical extent of parallel region – **private**
- Variable scope is the most common source of errors in OpenMP codes
  - Correctly determining variable scope is key to correctness and performance of your code

Variable Scoping Clauses: Shared

- Shared variables: **`shared (list)`**
  - By default, all variables shared unless otherwise specified
  - All threads access this variable in same location in memory
  - Race conditions can occur if access is not carefully controlled

Variable Scoping Clauses: Private

- Private: **`private (list)`**
  - Variable exists only within parallel region
  - Value undefined at start and after end of parallel region
- Private starting with defined values: **`firstprivate (list)`**
  - Private variables initialized to be the value held immediately before entry into parallel region
- Private ending with defined value: **`lastprivate(list)`**
  - At end of loop, set variable to value set by final iteration of loop

Common Mistakes

- A variable that should be private is public
  - Something unexpectedly gets overwritten
  - Solution: explicitly declare all variable scope
- Nondeterministic execution
  - Different results from different executions
- Race condition
  - Sometimes you get the wrong answer
  - Solutions:
    - Look for overwriting of shared variable
    - Use a tool such as Cray Reveal or Codee to rescope your loop

Find the Mistake(s)!

```
/* Gaussian Elimination (no pivoting):   x = A\b   */
int i, j, k;
double ratio;
for (i = 0; i < N-1; i++) {
#pragma omp parallel for
  for (j = i; j < N; j++) {
    ratio = A[j][i]/A[i][i];
    for (k = i; k < N; k++) {
      A[j][k] -= (ratio*A[i][k]);
      b[j] -= (ratio*b[i]);
    }
  }
}
```

Find the Mistake(s)!

```
/* Gaussian Elimination (no pivoting):   x = A\b   */
int i, j, k;
double ratio;
for (i = 0; i < N-1; i++) {
#pragma omp parallel for
  for (j = i; j < N; j++) {
    ratio = A[j][i]/A[i][i];
    for (k = i; k < N; k++) {
      A[j][k] -= (ratio*A[i][k]);
      b[j] -= (ratio*b[i]);
    }
  }
}
```

k & **ratio** are shared variables by default. Depending on compiler, **k** may be optimized out & therefore not impact correctness, but **ratio** will always lead to errors! Depending how loop is scheduled, you will see different answers.

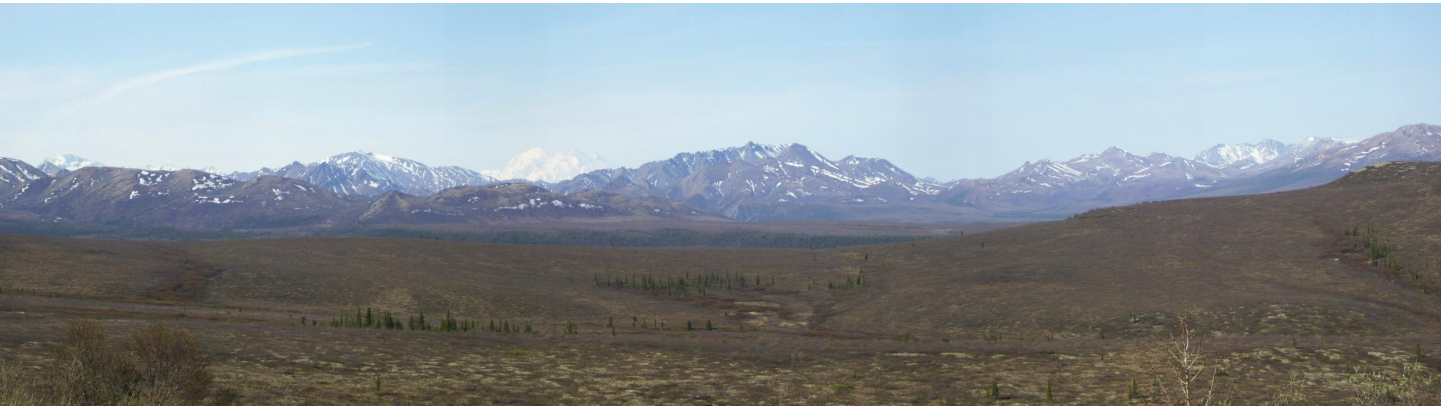
Fix the Mistake(s)!

```
/* Gaussian Elimination (no pivoting):   x = A\b   */
int i, j, k;
double ratio;
for (i = 0; i < N-1; i++) {
#pragma omp parallel for private (j,k,ratio) \
shared (i,A,b,N) default (none)
  for (j = i; j < N; j++) {
    ratio = A[j][i]/A[i][i];
    for (k = i; k < N; k++) {
      A[j][k] -= (ratio*A[i][k]);
      b[j] -= (ratio*b[i]);
    }
  }
}
```

Fix the Mistake(s)!

```
/* Gaussian Elimination (no pivoting):   x = A\b   */
int i, j, k;
double ratio;
for (i = 0; i < N-1; i++) {
#pragma omp parallel for private (j,k,ratio) \
shared (i,A,b,N) default (none)
  for (j = i; j < N; j++) {
    ratio = A[j][i]/A[i][i];
    for (k = i; k < N; k++) {
      A[j][k] -= (ratio*A[i][k]);
      b[j] -= (ratio*b[i]);
    }
  }
}
```

By setting **`default (none)`**, compiler will catch any variables not explicitly scoped



IV. RUNTIME LIBRARY ROUTINES & ENVIRONMENT VARIABLES

Panorama with snow-capped Mt. McKinley in Denali National Park, Alaska, USA, May 2011, by Rebecca Hartman-Baker.

OpenMP Runtime Library Routines

- **`void omp_set_num_threads(int num_threads)`**
  - Sets number of threads used in next parallel region
  - Must be called from serial portion of code
- **`int omp_get_num_threads()`**
  - Returns number of threads currently in team executing parallel region from which it is called
- **`int omp_get_thread_num()`**
  - Returns rank of thread
  - **`0 ≤ omp_get_thread_num() < omp_get_num_threads()`**

OpenMP Environment Variables

- Set environment variables to control execution of parallel code
- **`OMP_SCHEDULE`**
  - Determines how iterations of loops are scheduled
  - E.g., **`export OMP_SCHEDULE="dynamic, 4"`**
- **`OMP_NUM_THREADS`**
  - Sets maximum number of threads
  - E.g., **`export OMP_NUM_THREADS=4`**

Various Methods to Set Number of Threads

1) Use num_threads clause

```
#pragma omp parallel num_threads (4)
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

2) Call omp_set_num_threads API

```
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID,A);
}
```

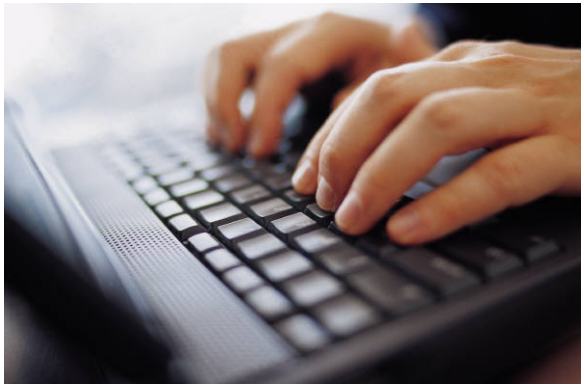
3) Set runtime environment variable

```
export OMP_NUM_THREADS=4
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

4) Do none of the three above

Code will use an implementation dependent default number of threads defined by the compiler.

- Precedence: 1) > 2) > 3) > 4)
- You may get fewer threads than you requested, check with omp_get_num_threads()



V. USING OPENMP

Conditional Compilation

- Can write single source code for use with or without OpenMP
  - Pragmas are ignored if OpenMP disabled
- What about OpenMP runtime library routines?
  - `_OPENMP` macro is defined if OpenMP available: can use this to conditionally include `omp.h` header file, else redefine runtime library routines

Conditional Compilation

```
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
#endif
…
int me = omp_get_thread_num();
…
```

Enabling OpenMP

- Most standard compilers support OpenMP directives
- Enable using compiler flags

Compiler	Intel	GNU	PGI/Nvidia	Cray
Flag	`-qopenmp`	`-fopenmp`	`-mp`	`-fopenmp`

Running Programs with OpenMP Directives

- Set OpenMP environment variables in batch scripts (e.g., include definition of **`OMP_NUM_THREADS`** in script)
- Example: to run a code with 8 MPI processes and 4 threads/MPI process on Perlmutter CPU:
  - `export OMP_NUM_THREADS=4`
  - `export OMP_PLACES=threads`
  - `export OMP_PROC_BIND=spread`
  - `srun -n 8 -c 64 --cpu_bind=cores ./myprog`
- Use the NERSC jobscript generator for best results: https://my.nersc.gov/script_generator.php

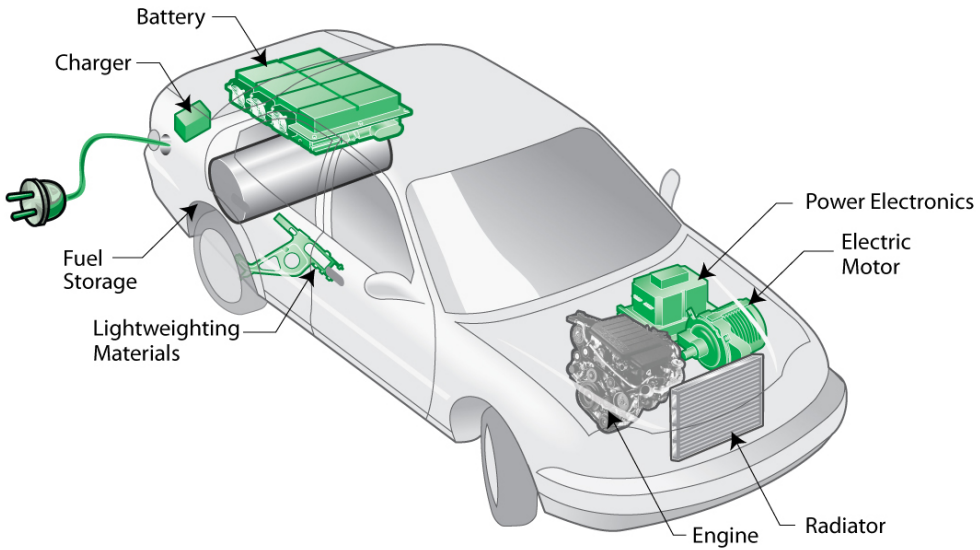
INTERLUDE 3: COMPUTING PI WITH OPENMP

"Happy Pi Day (to the 69th digit)!" by Mykl Roventine from http://www.flickr.com/photos/myklroventine/3355106480/sizes/l/in/photostream/

Interlude 3: Computing π with OpenMP

- Think about the original darts program you downloaded (**`darts.c/lcgenerator.h`** or **`darts.f90`**)
- How could we exploit shared-memory parallelism to compute π with the method of darts?
- What possible pitfalls could we encounter?
- Your assignment: parallelize the original darts program using OpenMP
- Rename it **`darts-omp.c`** or **`darts-omp.f90`**

OpenMP + Hybrid Parallel Programming



VI. HYBRID PROGRAMMING

VI. Hybrid Programming

- Motivation
- Considerations
- MPI threading support
- Designing hybrid algorithms
- Examples

Motivation

- Multicore architectures are here to stay
  - Macro scale: distributed memory architecture, suitable for MPI
  - Micro scale: each node contains multiple cores and shared memory, suitable for OpenMP
- Obvious solution: use MPI between nodes, and OpenMP within nodes
- Hybrid programming model

Considerations

- Sounds great, Rebecca, but is hybrid programming always better?
  - No, not always
  - Especially if poorly programmed 😅
  - Depends also on suitability of architecture
- Think of accelerator model
  - in omp parallel region, use power of multicores; in serial region, use only 1 processor
  - If your code can exploit threaded parallelism “a lot”, then try hybrid programming

Considerations

- Hybrid parallel programming model
  - Are communication and computation discrete phases of algorithm?
  - Can/do communication and computation overlap?
- Communication between threads
  - Communicate only outside of parallel regions
  - Assign a manager thread responsible for inter-process communication
  - Let some threads perform inter-process communication
  - Let all threads communicate with other processes

MPI Threading Support

- MPI-2 standard defines four threading support levels
  - (0) MPI_THREAD_SINGLE only one thread allowed
  - (1) MPI_THREAD_FUNNELED master thread is only thread permitted to make MPI calls
  - (2) MPI_THREAD_SERIALIZED all threads can make MPI calls, but only one at a time
  - (3) MPI_THREAD_MULTIPLE no restrictions
  - (0.5) MPI calls not permitted inside parallel regions (returns MPI_THREAD_SINGLE) – this is MPI-1

What Threading Model Does My Machine Support?

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
int provided;

MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

printf("Supports level %d of %d %d %d %d\n", provided,
  MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED,
  MPI_THREAD_SERIALIZED, MPI_THREAD_MULTIPLE);

MPI_Finalize();
return 0;
}
```

What Threading Model Does My Machine Support?

```
rjhb@perlmutter> cc -o threadmodel threadmodel.c
rjhb@perlmutter> salloc -C cpu -q interactive
salloc: Granted job allocation 10504403
salloc: Waiting for resource configuration
salloc: Nodes nid005664 are ready for job
rjhb@nid005664:~/test> srun -n 1 ./threadmodel

Supports level 3 of 0 1 2 3
```

MPI_Init_thread

- **`MPI_Init_thread(int required, int *supported)`**
  - Use this instead of **`MPI_Init(…)`**
  - **`required`**: the level of thread support you want
  - **`supported`**: the level of thread support provided by implementation (ideally **`= required`**, but if not available, returns lowest level **`> required`**; failing that, largest level **`< required`**)
  - Using **`MPI_Init(…)`** is equivalent to **`required = MPI_THREAD_SINGLE`**
- **`MPI_Finalize()`** should be called by same thread that called **`MPI_Init_thread(…)`**

Other Useful MPI Functions

- **`MPI_Is_thread_main(int *flag)`**
  - Thread calls this to determine whether it is main thread
- **`MPI_Query_thread(int *provided)`**
  - Thread calls to query level of thread support

Supported Threading Models: Single

- Use single pragma

```
#pragma omp parallel
{
  #pragma omp barrier
  #pragma omp single
  {
    MPI_Xyz(…);
  }
  #pragma omp barrier
}
```

Supported Threading Models: Funneled

- Cray & Intel MPI implementations support funneling
- Use master pragma

```
#pragma omp parallel
{
  #pragma omp barrier
  #pragma omp master
  {
    MPI_Xyz(…);
  }
  #pragma omp barrier
}
```

Supported Threading Models: Serialized

- Cray & Intel MPI implementations support serialized
- Use single pragma

```
#pragma omp parallel
{
  #pragma omp barrier
  #pragma omp single
  {
    MPI_Xyz(…);
  }
  //Don't need omp barrier
}
```

Supported Threading Models: Multiple

- Intel MPI implementation supports multiple!
  - (Cray MPI can turn on multiple support with env variables, but performance is sub-optimal)
- No need for pragmas to protect MPI calls
- Constraints:
  - Ordering of MPI calls maintained within each thread but not across MPI process -- user is responsible for preventing race conditions
  - Blocking MPI calls block only the calling thread
- Multiple is rarely required; most algorithms can be written without it

Which Threading Model Should I Use?

Depends on the application!

Model	Advantages	Disadvantages
Single	Portable: every MPI implementation supports this	Limited flexibility
Funneled	Simpler to program	Manager thread could get overloaded
Serial	Freedom to communicate	Risk of too much cross-communication
Multiple	Completely thread safe	Limited availability; sub-optimal performance

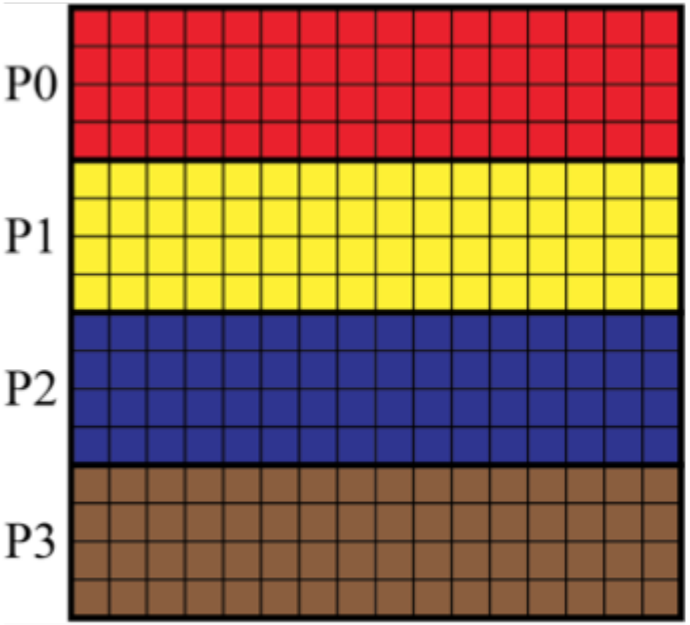
Designing Hybrid Algorithms

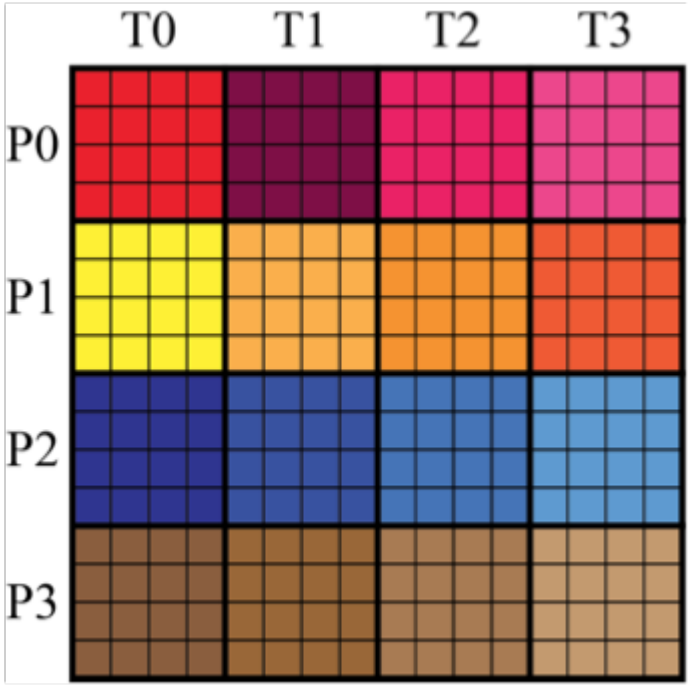
- Just because you *can* communicate thread-to-thread, doesn't mean you *should*
- Tradeoff between lumping messages together and sending individual messages
  - Lumping messages together: one big message, one overhead
  - Sending individual messages: less wait time (?)
- Programmability: performance will be great, when you finally get it working!

Example: Mesh Partitioning

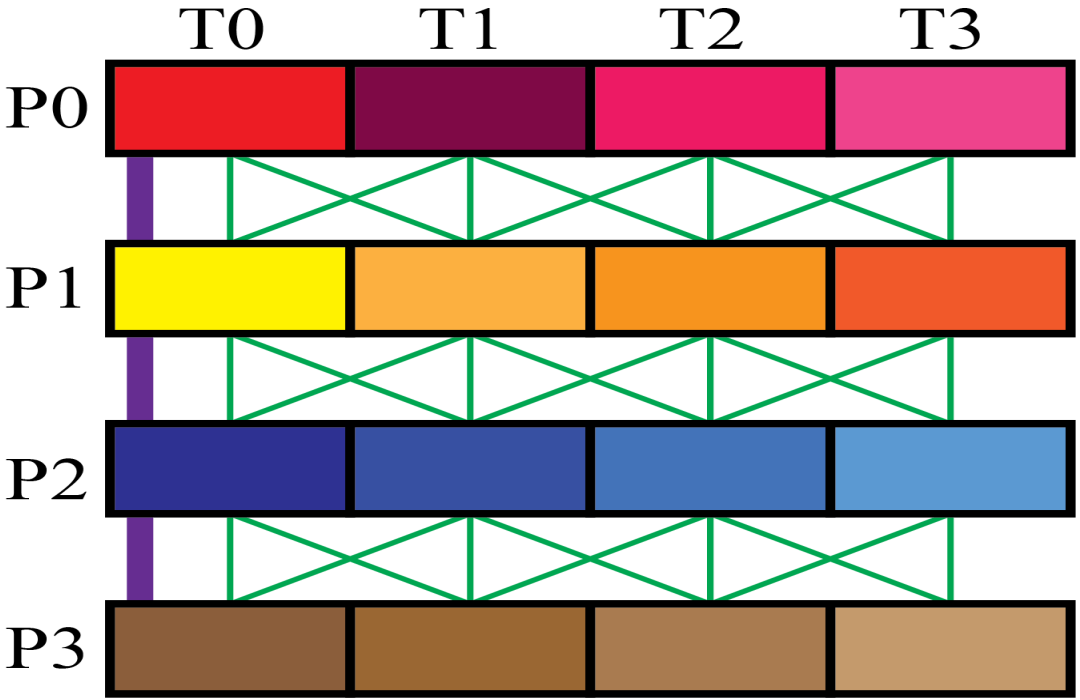
- Regular mesh of finite elements
- When we partition mesh, need to communicate information about (domain) adjacent cells to (computationally) remote neighbors

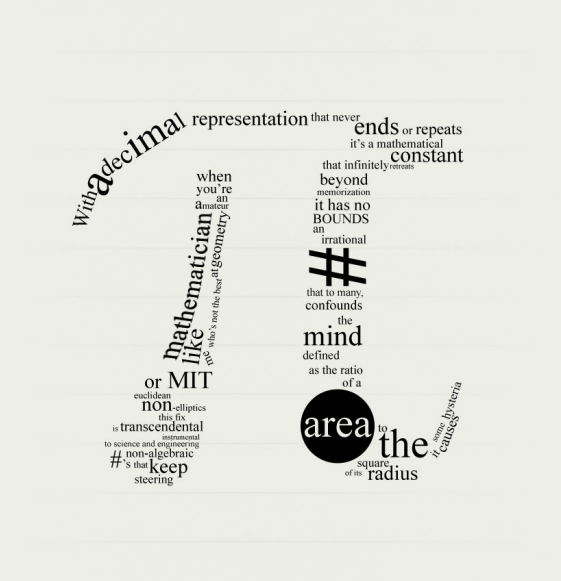
Example: Mesh Partitioning





Example: Mesh Partitioning





INTERLUDE 4: COMPUTING PI WITH HYBRID PROGRAMMING

"pi" by Travis Morgan from http://www.flickr.com/photos/morgantj/5575500301/sizes/l/in/photostream/

- Putting it all together:
  - How can we combine inter-node and intra-node parallelism to create a hybrid program that computes π using the method of darts?
  - What potential pitfalls do you see?
- Your assignment: create a code, **`darts-hybrid.c`** or **`darts-hybrid.f90`**, developed from **`darts-collective.c`**/**`darts-collective.f90`** and **`darts-omp.c`**/**`darts-omp.f90`**, that uses OpenMP to exploit parallelism within the node, and MPI for parallelism between nodes

Bibliography/Resources: OpenMP

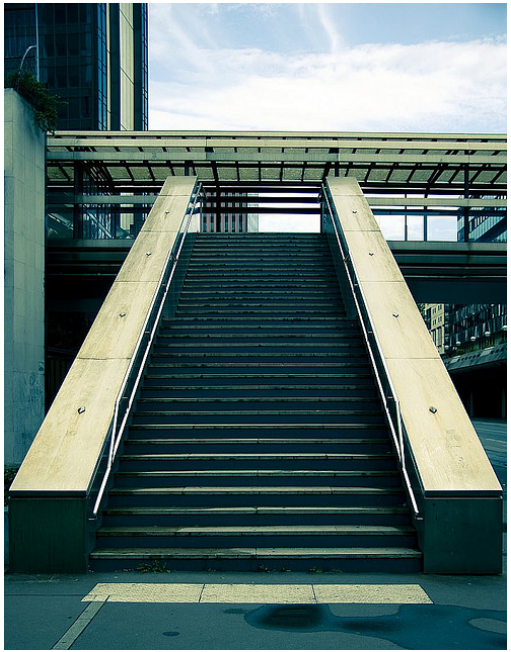
- Mattson, Timothy, Yun (Helen) He, Alice Koniges (2019) *The OpenMP Common Core*, Cambridge, MA: MIT Press
- Chapman, Barbara, Gabrielle Jost, and Ruud van der Pas. (2008) *Using OpenMP*, Cambridge, MA: MIT Press.
- LLNL OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/
- Mattson, Tim, and Larry Meadows (2008) *SC08 OpenMP "Hands-On" Tutorial*, https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
- Bull, Mark (2018) *OpenMP Tips, Tricks and Gotchas*, http://www.archer.ac.uk/training/course-material/2018/09/openmp-imp/Slides/L10-TipsTricksGotchas.pdf

Bibliography/Resources: OpenMP

- Logan, Tom, *The OpenMP Crash Course (How to Parallelize your Code with Ease and Inefficiency)*, http://ffden-2.phys.uaf.edu/608_lectures/OmpCrash.pdf
- OpenMP.org: https://www.openmp.org/
- OpenMP Standard: https://www.openmp.org/specifications/
  - 5.2 Specification: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf
  - 5.2 code examples: https://www.openmp.org/wp-content/uploads/openmp-examples-5-2.pdf

Bibliography/Resources: Hybrid Programming

- Cuma, Martin (2015) *Hybrid MPI/OpenMP Programming*, https://www.chpc.utah.edu/presentations/images-and-pdfs/MPI-OMP15.pdf
- INTERTWinE (2017) *Best Practice Guide to Hybrid MPI + OpenMP Programming*, http://www.intertwine-project.eu/sites/default/files/images/INTERTWinE_Best_Practice_Guide_MPI%2BOpenMP_1.1.pdf
- Rabenseifner, Rolf, Georg Hager, Gabriele Jost (2013) SC13 Hybrid MPI and OpenMP Parallel Programming Tutorial, https://openmp.org/wp-content/uploads/HybridPP_Slides.pdf



APPENDIX 1: COMPUTING PI

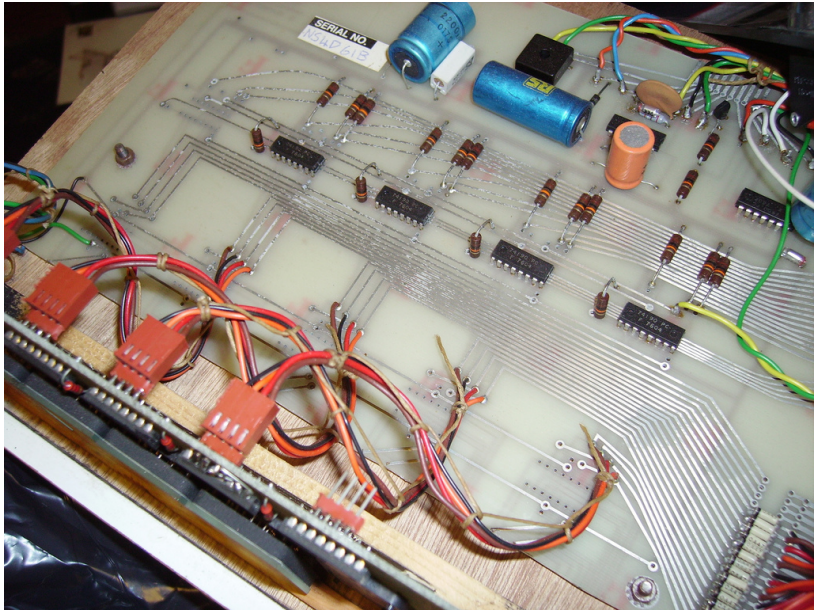
"Pi" by Gregory Bastien, from http://www.flickr.com/photos/gregory_bastien/2741729411/sizes/z/in/photostream/

Computing π

- Method of Darts is a TERRIBLE way to compute π
  - Accuracy proportional to square root of number of darts
  - For one decimal point increase in accuracy, need 100 times more darts!
- Instead,
  - Look it up on the internet, e.g., http://www.geom.uiuc.edu/~huberty/math5337/groupe/digits.html
  - Compute using BBP (Bailey-Borwein-Plouffe) formula:

$$\pi = \sum_{n=0}^{\infty} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^n$$

  - For less accurate computations, try your programming language's constant, or quadrature or power series expansions



APPENDIX 2: ABOUT RANDOM NUMBER GENERATION

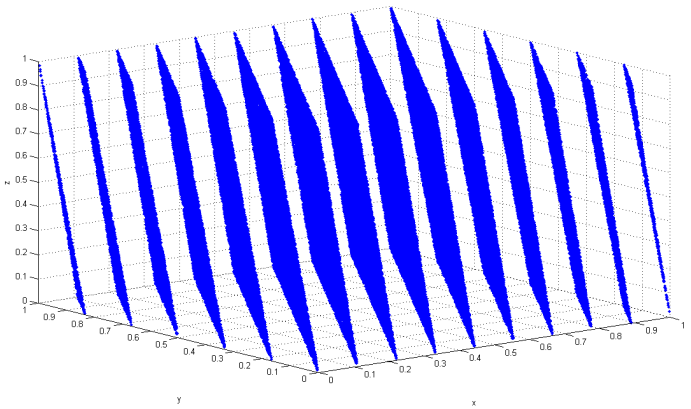
"Random Number Generator insides" by mercuryvapour, from http://www.flickr.com/photos/mercuryvapour/2743393057/sizes/l/in/photostream/

About Random Number Generation

- No such thing as random number generation – proper term is pseudorandom number generator (PRNG)
- Generate long sequence of numbers that seems “random”
- Properties of good PRNG:
  - Very long period
  - Uniformly distributed
  - Reproducible
  - Quick and easy to compute

Pseudorandom Number Generator

- Generator from **`lcgenerator.h`** is a Linear Congruential Generator (LCG)
  - Short period (= **`PMOD`**, 714025)
  - Not uniformly distributed – known to have correlations
  - Reproducible
  - Quick and easy to compute
  - Poor quality (don't do this at home)



Correlation of RANDU LCG (source: http://upload.wikimedia.org/wikipedia/commons/3/38/Randu.png)

Good PRNGs

- For serial codes
  - Mersenne twister
  - GSL (GNU Scientific Library), many generators available (including Mersenne twister) http://www.gnu.org/software/gsl/
  - Also available in Intel MKL
- For parallel codes
  - SPRNG, regarded as leading parallel pseudorandom number generator http://sprng.cs.fsu.edu/

- Putting it all together:
  - How can we combine inter-node and intra-node parallelism to create a hybrid program that computes π using the method of darts?
  - What potential pitfalls do you see?
- Your assignment: create a code, **`darts-hybrid.c`** or **`darts-hybrid.f90`**, developed from **`darts-collective.c`**/**`darts-collective.f90`** and **`darts-omp.c`**/**`darts-omp.f90`**, that uses OpenMP to exploit parallelism within the node, and MPI for parallelism between nodes

Bibliography/Resources: OpenMP

- Mattson, Timothy, Yun (Helen) He, Alice Koniges (2019) *The OpenMP Common Core*, Cambridge, MA: MIT Press
- Chapman, Barbara, Gabrielle Jost, and Ruud van der Pas. (2008) *Using OpenMP*, Cambridge, MA: MIT Press.
- LLNL OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/
- Mattson, Tim, and Larry Meadows (2008) *SC08 OpenMP "Hands-On" Tutorial*, https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
- Bull, Mark (2018) *OpenMP Tips, Tricks and Gotchas*, http://www.archer.ac.uk/training/course-material/2018/09/openmp-imp/Slides/L10-TipsTricksGotchas.pdf