Lecture 15 - Shared-Memory Parallel Computing

- **Now let's start to discuss how we can use shared-memory systems to perform parallel computing**

- **Remember that in shared-memory systems, all of the processors see all of the data**

Motivation

- **Popularity of shared memory systems is increasing:**
  - Distributed Shared-Memory (DSM) computers (SGI Altix, SUN Ultra 10000,HP Exemplar, Intel Xeon Phi) have replaced vector systems in many supercomputing centers
- **Compiler directed parallelism is attracting attention:**
  - Single version of sequential and parallel code
  - Industrial standard is now available: OpenMP

Motivation

- **New generation of parallel machines with multiple CPUs per node**
  - IBM SP3 (older technology):
    - each node is a 16-way shared memory machine
    - there is a single bus to the high performance network per node
  - IBM BlueGene/L (later technology):
    - Each node is a 2-way shared memory machine
    - 1024 nodes per rack, 16-128 I/O nodes per rack, 3D Torus interconnect 350 Mb/s bandwidth, 1.5 μs latency
  - Quad-, Octo-, and now higher Core Chip-Sets (eg. Mic Xeon Phi, latest technology)
- **Mixed programming model:**
  - OpenMP in each node
  - MPI across nodes

Programming Models

- **Shared memory options:**
  - Automatic parallelization (some compilers)
  - Pthreads (POSIX threads)
  - Compiler directives: OpenMP

- **Message passing options:**
  - MPI: message passing interface
  - PVM: parallel virtual machine
  - HPF: high performance Fortran

OpenMP

- **OpenMP is an API for writing multi-threaded applications:**
  - A set of compiler directives and library routines for parallel application programmers
  - Makes it easy to create multi-threaded programs in Fortran, C and C++

OpenMP Supporters

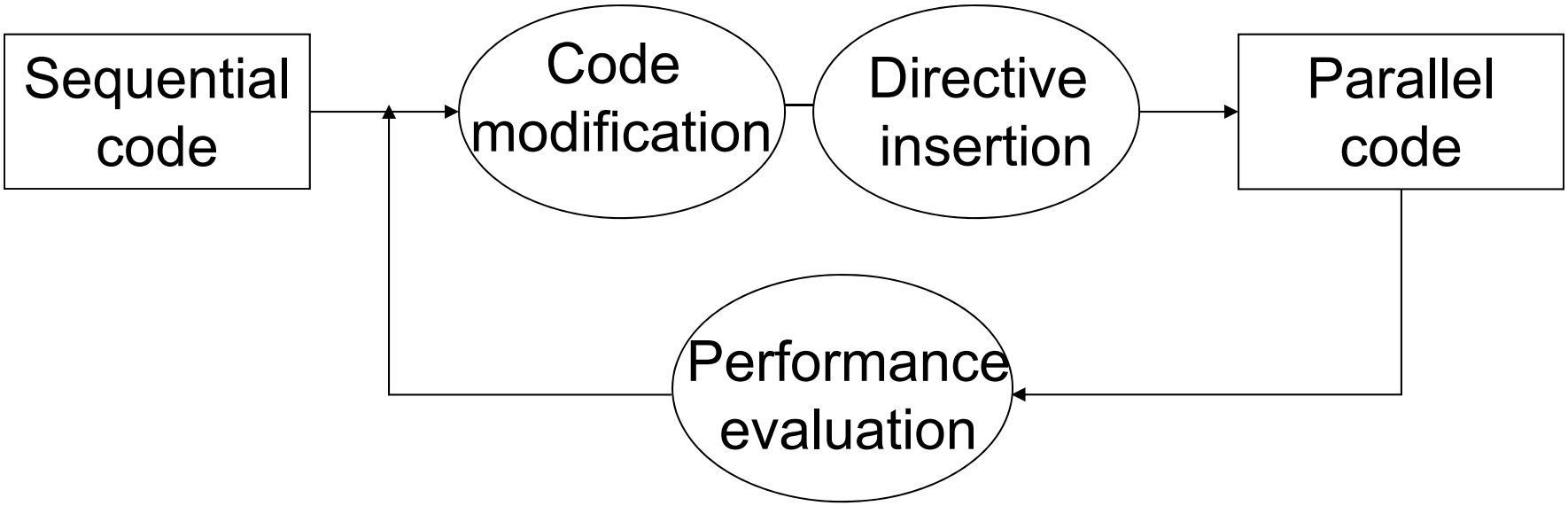
- **Hardware vendors:**
  - Compaq/HP, IBM, Intel, SGI, SUN,…
- **Software vendors**
  - KAI (now Intel), PGI, Absoft, PSR,…

http://openmp.org/wp

OpenMP

- **Fine grained parallelism (at loop level)**
- **Coarse grained parallelism**
- **Compiler directives, library and environment variables *extend* base language**
- **NOT automatic parallelization**
- **Since the constructs are directives, an OpenMP program can be compiled by compilers that do not support OpenMP**

Incremental Parallelization



Work can be done incrementally!!!!!!!

Serial Code

```
      program compute_pi
      integer n, i
      double precision w, x, sum, pi, f, a
! function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)
      print *, 'Enter number of intervals: '
      read *,n
! calculate the interval size
      w = 1.0d0/n
      sum = 0.0d0
      do i = 1, n
            x = w * (i - 0.5d0)
            sum = sum + f(x)
      end do
      pi = w * sum
      print *, 'computed pi = ', pi
      stop
      end
```

MPI Code

```
      program compute_pi
      include 'mpif.h'
      double precision mypi, pi, w, sum, x, f, a
      integer n, myid, numprocs, i, rc
! function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)
      call MPI_INIT( ierr )
      call MPI_COMM_RANK(MPI_COMM_WORLD,
     &                   myid, ierr )
      call MPI_COMM_SIZE(MPI_COMM_WORLD,
     &                   numprocs, ierr )
      if ( myid .eq. 0 ) then
          print *, 'Enter number of intervals: '
          read *, n
      endif
      call MPI_BCAST(n,1,MPI_INTEGER,0,
     &               MPI_COMM_WORLD,ierr)
! calculate the interval size
      w = 1.0d0/n
      sum = 0.0d0
      do i = myid+1, n, numprocs
          x = w * (i - 0.5d0)
          sum = sum + f(x)
      enddo
      mypi = w * sum
! collect all the partial sums
      call MPI_REDUCE(mypi,pi,1,
     &                MPI_DOUBLE_PRECISION,
     &                MPI_SUM,0,
     &                MPI_COMM_WORLD,ierr)
! node 0 prints the answer
      if (myid .eq. 0) then
          print *, 'computed pi = ', pi
      endif
      call MPI_FINALIZE(rc)
      stop
      end
```

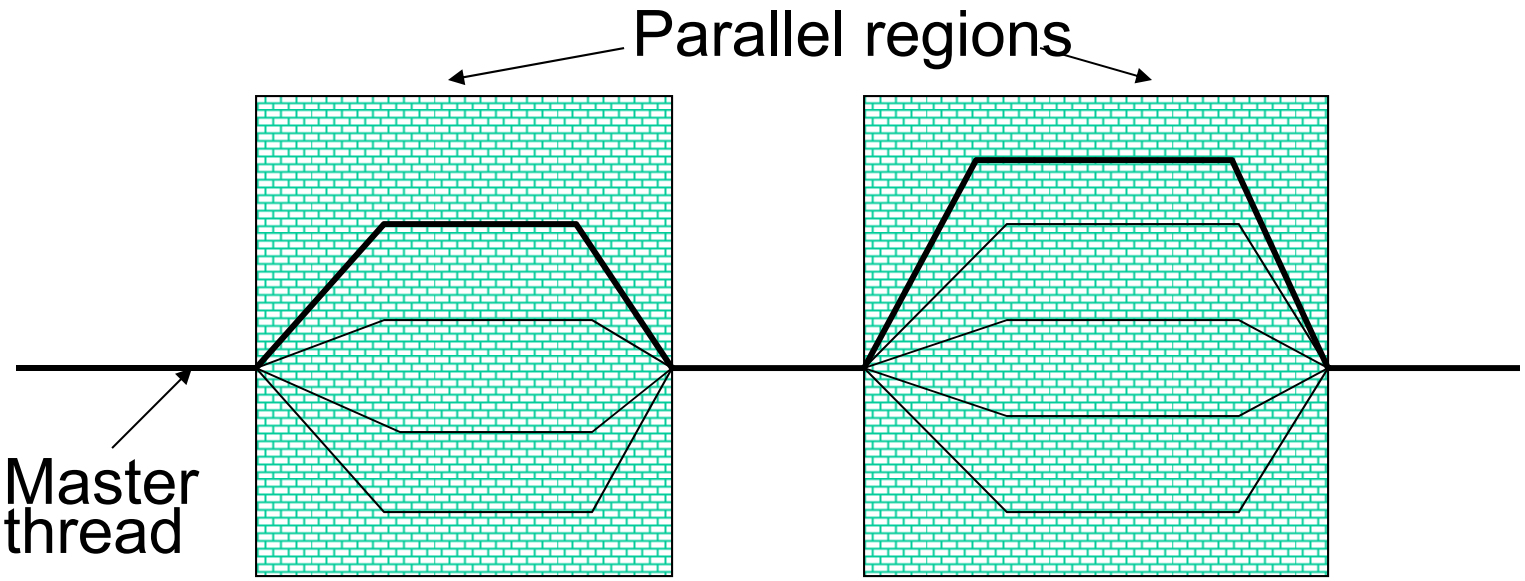
OpenMP Code

```
      program compute_pi
      integer n, i
      double precision w, x, sum, pi, f, a
! function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)
      print *, 'Enter number of intervals: '
      read *,n
! calculate the interval size
      w = 1.0d0/n
      sum = 0.0d0
!$OMP PARALLEL DO PRIVATE(x), SHARED(w,n)
!$OMP& REDUCTION(+: sum)
      do i = 1, n
         x = w * (i - 0.5d0)
         sum = sum + f(x)
      end do
!$OMP END  PARALLEL DO
      pi = w * sum
      print *, 'computed pi = ', pi
      stop
      end
```

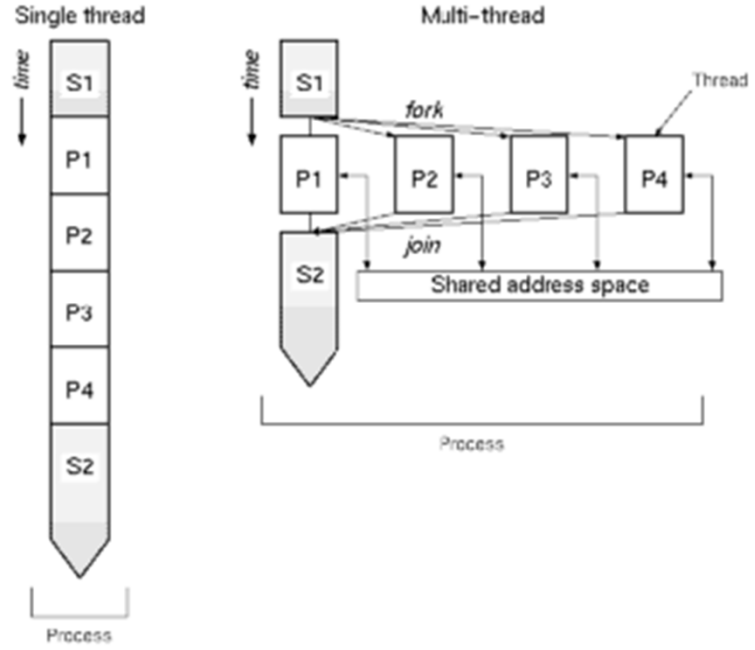
Execution Model

- **Master thread spawns a team of threads as needed**
  - Concept of threads also used in GPU parallel computing

- **Parallelism is added incrementally**



Execution Model



- OpenMP programs start as single thread
- In parallel regions, additional threads are created
- Master thread is part of the team
- Outside parallel regions, additional threads go away (or sleep)

Directives or pragmas

- **Fortran (fixed format)**

  !$omp …

  c$omp …

  *$omp …

  If the character in column 6 is different from a space or a 0 it is a continuation line

- **Fortran (free form)**
  - A line that begins with !$omp is an OpenMP directive
  - A directive that needs to be continued on the next line:

    !$OMP ……. &

- **C:**

  #pragma omp

Conditional Compilation

- **The selective disabling of OpenMP construct applies only to directives.**
- **Application may contain statements that are specific to OpenMP but not intended for OpenMP**

Fortran :

– A line beginning with a *sentinel (!$,c$,*$* in fixed format, !$ in free format) is ignored if not in OpenMP

```
!$      iam=omp_get_num_thread()
```

C:

– Preprocessor macro _OPENMP

```
#pragma omp
```

Programming Model

- **OpenMP is a shared memory model**
  - Threads communicate by sharing variables

- **Unintended sharing of data can lead to race conditions**
  - Race conditions: the program's outcome changes as the threads are scheduled differently

Example in C

Sequential code

```
void main()
{
  double a[100000];
  for (int i=0,i<100000,i++){
     do_something(a[i]);
                    }
 }
```

Parallel code

```
void main()
{
  double a[100000];
#pragma omp parallel for
  for (int i=0,i<100000,i++){
     do_something(a[i]);
                    }
 }
```

Example in Fortran

Sequential code

```
real*8 a(100000)
do i = 0,100000
    do_something(a)
enddo
```

Parallel code

```
real*8 a(100000)
!#omp parallel do
do i = 0,100000
     do_something(a)
 enddo
```

OpenMP Constructs

- **OpenMP's constructs (directives) fall into 5 categories:**
  - Parallel regions
  - Worksharing
  - Synchronization
  - Data environment
  - Runtime functions/environment variables

OpenMP Constructs

- **OpenMP's constructs (directives) fall into 5 categories:**
  - Parallel regions
  - Worksharing
  - Synchronization
  - Data environment
  - Runtime functions/environment variables

Parallel regions

- **Threads are created in OpenMP with the omp parallel directive**

FORTRAN

```
!$OMP PARALLEL
        Code to be executed by each thread
!$OMP END PARALLEL
```

C

```
#pragma omp parallel
{Code to be executed by each thread}
```

Parallel Regions

- **Dynamic mode (the default):**
  - The number of threads used can vary from one parallel region to another
  - Setting the number of threads only sets the maximum number of threads
- **Static mode:**
  - The number of threads is fixed and controlled by the programmer:
    - Use library routine:

      call omp_set_num_threads(16)
    - Enviromental variable:

      setenv OMP_NUM_THREADS 16

OpenMP Constructs

- **OpenMP's constructs (directives) fall into 5 categories:**
  - Parallel regions
  - Worksharing
  - Synchronization
  - Data environment
  - Runtime functions/environment variables

C Example: Work Sharing: for

Serial

```
for (i=0;i<N,i++){ a[i]=b[i]+c[i];}
```

OpenMP
Parallel region

```
#pragma omp parallel
{ int myid,i,nthreads,mystart,myend;
  myid=omp_get_thread_num();
  nthreads=omp_get_num_threads();
  mystart=myid*N/nthreads;
  myend=(Myid+1)*N/nthreads;
  for (i=mystart;i<myend,i++){ a[i]=b[i]+c[i];}
}
```

OpenMP
Parallel region
and work-sharing

```
#pragma omp parallel
#pragma omp for schedule(static)
  for (i=0;i<N,i++){ a[i]=b[i]+c[i];}
```

Fortran Example: Work Sharing: for

Serial

```
do i = 0,N
   a(i) = b(i)+c(i)
enddo
```

OpenMP
Parallel region

```
!$omp parallel
myid= omp_get_thread_num()
nthreads= omp_get_num_threads()
mystart = myid*N/nthreads
myend  = (myid+1)*N/nthreads
do I = mystart, myend
   a(i) = b(i)+c(i)
enddo
!$omp end parallel
```

Allows the programmer to break up loop into nthreads

Notice that operation within loop is index-independent!!

(No dependencies within loop)

OpenMP
Parallel region
and work-sharing

```
!$omp parallel
!$omp do schedule(static)
do i =0,N
   a(i) = b(i)+c(i)
enddo
!$omp end schedule
!$omp end parallel
```

These directives could be combined to:
!$omp parallel do schedule(static)

The loop is automatically broken up into nthreads as evenly as possible

Work Sharing: schedule

- **Schedule keyword specifies how loop is split:**
  - schedule(static,[chunk]):
    - Each thread gets a block of size chunk
  - schedule(dynamic,[chunk]):
    - Each thread gets a block of size chunk from a queue until all iterations are scheduled
  - schedule(guided,[chunk]):
    - Dynamic schedule that shrinks down to size “chunk” as the calculation proceeds.
  - schedule(runtime):
    - Schedule and chunk size taken from OMP_SCHEDULE environment variable

Work Sharing: sections

- **The "sections" construct is used to give different workload to different threads:**
  - Assuming that subroutines (tasks) sub_x, sub_y, sub_z, etc. are independent of each other

```
!$OMP PARALLEL
!$OMP SECTIONS
   Call sub_x(…)
!$OMP SECTION
   Call sub_y(…)
!$OMP SECTION
   Call sub_z(…)
!$OMP END SECTIONS
!$OMP END PARALLEL
```

OpenMP Constructs

- **OpenMP's constructs (directives) fall into 5 categories:**
  - Parallel regions
  - Worksharing
  - Synchronization
  - Data environment
  - Runtime functions/environment variables

Synchronization

- **There are different OpenMP constructs that allow the processors to synchronize:**
  - Mutual exclusion
  - Atomic update
  - Barrier synchronization
  - Master section

Mutual Exclusion

- **CRITICAL section:**
  - Can be named, but the name should not conflict with subroutine or module
  - Used to ensure that *only one thread* (first to arrive) is within a "critical" section (i.e.serial execution)
  - Can not jump into or out of critical sections

```
!$OMP PARALLEL
          .........
!$OMP CRITICAL (name)
           a(i)=a(i)+alocal
!$OMP END  CRITICAL (name)
        .........
!$OMP END PARALLEL
```

Atomic Update

- **Optimization of *critical* directive for a certain case**
  - Operation should not be broken into sub-parts (single-thread operation)
  - Update to a single memory location should be performed as a single-threaded operation

```
!$OMP PARALLEL
          .........
!$OMP ATOMIC
           a(i)=a(i)+alocal
!$OMP END ATOMIC
          .........
!$OMP END PARALLEL
```

Barrier Synchronization

- **BARRIER directive**
  - Threads wait until all the threads reach this point:
  - An implicit barrier exists at the end of each parallel region

```
!$OMP PARALLEL
        .........
!$OMP BARRIER
        .........
!$OMP END PARALLEL
```

Master section

- **Only the master thread executes the section**
  - Rest of the team skip execution
  - No barrier at the end of the master section
  - A specialization of the “single” directive

```
!$OMP PARALLEL
        ……..
!$OMP MASTER
        print *,”I am the master”
!$OMP END MASTER
        ……..
!$OMP END PARALLEL
```

OpenMP Constructs

- **OpenMP's constructs fall in 5 categories:**
  - Parallel regions
  - Synchronization
  - Worksharing
  - Data environment
  - Runtime functions/environment variables

Default Storage Attributes

- **Global variables are SHARED among threads**
  - FORTRAN:
    - SAVE variables
    - MODULE variables
  - C: File scope variables, static

- **Stack variables in sub-programs called from parallel regions are PRIVATE**

Default Storage Attributes

```
program sort
use input
integer index(10)
call input
!$OMP PARALLEL
        call work
!$OMP END PARALLEL
print *,index(1)
….
```

```
subroutine work
use input
real dummy(10)
integer count
save count
…..
```

```
module input
  dimension a(10)
end module input
```

a(10), index(10), count are shared

dummy(10) is private

Storage Attributes

- **Storage attributes can be changed using the following *clauses of an OMP directive*:**
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
  - THREADPRIVATE

- **The value of private data inside a parallel loop can be transmitted to a global value outside the loop with:**
  - LASTPRIVATE

- **The default status can be modified with:**
  - DEFAULT (PRIVATE | SHARED | NONE)

PRIVATE Clause

- **PRIVATE(var) creates a local copy of var for each thread**
  - The value is initialized
  - Private copy is not storage associated with the original

```
      program wrong
      is=0
!$OMP PARALLEL DO PRIVATE(IS)
      do j=1,1000
      is=is+j       !is is not initialized since it is declared private
      end do
!$OMP END PARALLEL DO
      print *,is    !regardless of initialization, "is" is undefined
      stop
      end
```

FIRSTPRIVATE Clause

- **FIRSTPRIVATE(var) creates a local copy of var for each thread with the corresponding value from the master thread**

```
      program wrong2
      is=0
!$OMP PARALLEL DO FIRSTPRIVATE(IS)
      do j=1,1000
      is=is+j       !is is now initialized to 0
      end do
!$OMP END PARALLEL DO
      print *,is    !regardless of initialization, “is” is undefined because it
                    ! was not lastprivate
      stop
      end
```

LASTPRIVATE Clause

- **LASTPRIVATE(var) passes the value of a private from the last iteration to a global variable**

```
          program wrong3
          is=0
!$OMP PARALLEL DO FIRSTPRIVATE(IS) LASTPRIVATE(IS)
          do j=1,1000
             is=is+j       !is is now initialized to 0
          end do
!$OMP END PARALLEL DO
          print *,is        !is is wrong value, however
           stop
           end
```

Example

```
      program wrong3
      ls=0
!$OMP PARALLEL DO FIRSTPRIVATE(IS)
!$OMP LASTPRIVATE(IS)
      do j=1,1000
         is=is+j        !is starts at 0
      end do
!$OMP END PARALLEL DO
      print *,is
      stop
      end
```

Serial output

```
~> a.out
 500500
```

Parallel output:

```
~> setenv OMP_NUM_THREADS 2
~> a.out
 375250

~> setenv OMP_NUM_THREADS 4
~> a.out
 218875
```

Why do we get different answers?

Answer: Because ‘is’ is the final value for each thread, not the global sum. A reduction is needed!

Note: The loop will be broken up into NUM_THREADS even if it is greater than the number of available CPUs (in which case, a warning will be issued).

REDUCTION Clause

- **REDUCTION(op: list) :local copies are reduced into a single global copy at the end of the construct**
  - The variables in “list” must be shared.
  - Inside a parallel or worksharing construct a local variable is made and properly initialized.
  - “op” includes +, *, -, .AND., .OR., .EQV., .NEQV., or the intrinsics MAX, MIN, IAND, IOR, or IEOR

Example

```
      program correct3
      is=0
      call omp_set_num_threads(2)
!$OMP PARALLEL DO REDUCTION(+:IS)
      do j=1,1000
        is=is+j       !is is now initialized to 0
      end do
!$OMP END PARALLEL DO
      print *,is
      stop
      end
```

Parallel Output:

```
./correct3
      500500
FORTRAN STOP
```

"Variables that appear in a REDUCTION clause must be SHARED in the enclosing context. A private copy of each variable in *list* is created for each thread as if the PRIVATE clause had been used. The private copy is initialized according to the operator.

At the end of the REDUCTION, the shared variable is updated to reflect the result of combining the original value of the (shared) reduction variable with the final value of each of the private copies using the operator specified."

THREADPRIVATE Clause

- **Make global data private to a thread:**
  - FORTRAN: MODULE blocks
  - C: file scope and static variables

- **Each thread has its own copy of the module block**
  - Persistent across multiple parallel regions provided that the number of threads remains the same

OpenMP's Constructs

- **OpenMP's constructs fall in 5 categories:**
  - Parallel regions
  - Synchronization
  - Worksharing
  - Data environment
  - Runtime functions/environment variables
    - We'll discuss this during the next lecture

Project 5

- **Now that you have "completed" Projects 1-3 and nearly finished with Project - 4(hopefully), you are ready to parallelize a multi-block code using MPI**

- **Take your multi-block code that was developed under Project 3 and parallelize it using MPI across the node/CPU partitions created in Project 4**

Project 5

- **Due Thursday, December 3 (Start on this NOW!)**
  - Overall description of heat conduction problem including governing equations, discretization scheme (recursion formulas), domain decomposition, processor mapping, parallel processing strategy, etc.
  - Demonstration that the parallel heat conduction code works on hpc1 with:
    - For 501x501 grid using 10x10 decomposition with 4 processors, figures of
      - Grid
      - Converged solution
      - Convergence history (log(residual) vs iteration) comparison with single-block and multi-block serial codes. Plot the 3 histories on the same plot.
    - Plots of computational speed-up and efficiency vs number of processors for up to 8 processors for 501x501 grid using 10x10 decomposition
      - Show comparison with ideal (linear) and optimal (based on your load balance numbers in Project 4)
  - Listing of your parallel heat conduction code