C Tutorial

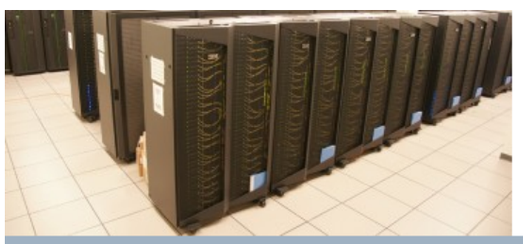
SciNet Parallel Scientific Computing Course

Aug 31 - Sept 4, 2009

SciNet High Performance Computing

Why C?

Because we can't in good conscience espouse Fortran.



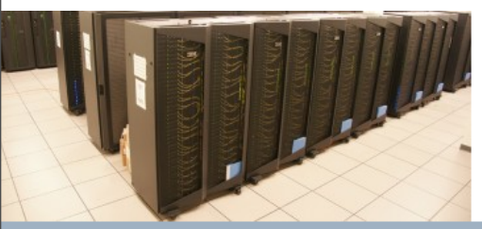
C Hello World

Code:

```c
#include <stdio.h>
/*C itself is a very spare language.  Much standard functionality
  is in libraries.  The libraries usually come with header files that
  describe how a program should interface with the library calls.
  We include the header files (the C pre-processor will copy the files
  into our program at compile-time).

  stdio contains standard I/O functions, including printing to the screen.*/
//  Comments can be in blocks marked off by /*...*/, or single-line, using //
int main(int argc, char *argv[])
     /*Main is a special routine that gets called when we run a program.
       It has two input argumens:  an integer argc that tells us how
       many command-line arguments we had, and argv, an array of strings
       of those arguments.*/
{
  //Blocks of code in C are contained in curly braces{}
  printf("Hello world.\n");
  //printf dumps a string to standard out (the terminal).  The character \n is
  //a newline.  The string must be in double quotes.  Note that we have
  //to end each line in C with a semicolon.

}
```

Output:

```
[sievers@tpb4 c-tutorial]$ gcc -o hello_world hello_world.c
[sievers@tpb4 c-tutorial]$ ./hello_world
Hello world.
[sievers@tpb4 c-tutorial]$
```



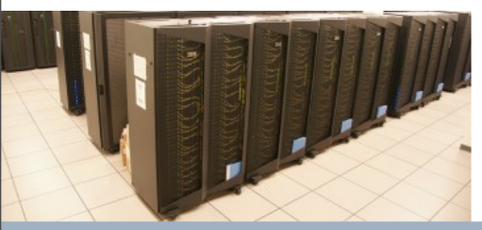
C For Loop

Code:

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;  //declare a variable to be an integer.  Other useful types are char, float, double...

  /*C for-loop syntax is " for (A;B;C) ".  Command A gets set at the start of the for loop.
    It will continue as long as statement B remains true.  Finally, at each iteration,
    command C gets executed.
    In C, i++ is shorthand for i=i+1;
    Also have i+=j (i=i+j).  --, *=, /= are also commonly used*/
  for (i=0;i<argc;i++)
    printf("Argument %3d is %s\n",i,argv[i]);
  /*formatted output in C.  prinf will take an arbitrarily long string of arguments.  The first
    is always a string, the rest are variables that will be put inside the string.
    %d means an integer, %f means a float, %e is a float using scientific notation, %g is
    the prettier of %f and %g, %c is a charater, and %s is a string.  We can put numbers between
    the % and the type, which tells printf how many characters to use.  So, every integer
    will be given three characters of space.*/
}
```

Output:

```
[sievers@tpb4 c-tutorial]$ gcc -o for_loop for_loop.c
[sievers@tpb4 c-tutorial]$ ./for_loop This is a test of C for loops.  Would like 10 arguments or more.
Argument   0 is ./for_loop
Argument   1 is This
Argument   2 is is
Argument   3 is a
Argument   4 is test
Argument   5 is of
Argument   6 is C
Argument   7 is for
Argument   8 is loops.
Argument   9 is Would
Argument  10 is like
Argument  11 is 10
Argument  12 is arguments
Argument  13 is or
Argument  14 is more.
[sievers@tpb4 c-tutorial]$
```



C Functions

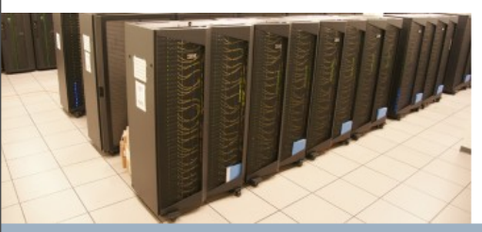
Code:

```c
#include <stdio.h>
#include <stdlib.h>

double get_square(double x)
     /*we give each function a unique name.  We tell the compiler what sort of variable it
       will return, at most one (can be zero if declared to be of type void).  We also tell it
       how many variables and of what kind the input is.  The keyword return tells the function
       what value to return.  The function stops once it hits return.*/
{
  return x*x;
}

/*=========================================================================*/
int main(int arc, char *argv[])
{
  double x=atof(argv[1]);  //atof is a standard library function to convert a string to a float/double
  printf("%g squared is %g\n",x,get_square(x));
}
```

Output:

```
[sievers@tpb4 c-tutorial]$ gcc -o c_functions c_functions.c
[sievers@tpb4 c-tutorial]$ ./c_functions 4
4 squared is 16
[sievers@tpb4 c-tutorial]$
```

Unlike Fortran, there is no distinction in C between functions and subroutines.



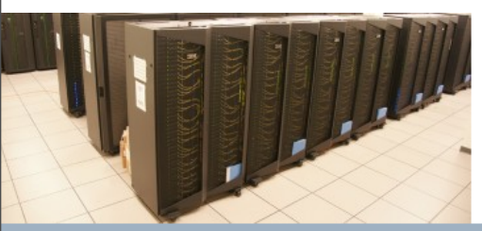
C Pointers

Code:

```c
#include <stdio.h>
#include <stdlib.h>
/*Pointers are a source of much confusion.  We have a class of variables that contain not
  values but locations in memory.  The syntax in C to go from a variable to its location, or from a
  pointer to the value of the memory to which it points is very compact.  As such, it can be
  difficult until one gets used to it.
  Short summary:  If x is a variable, then &x is a pointer to x.  If xp is a pointer, then *xp is
                  the value of the memory pointed to by xp.  When we declare pointers, we put a *
                  before them to mark them as pointers.  so
                  int *xp;  make a variable names xp (not *xp) that points to an integer.
  C always passes by value (makes a copy of whatever arguments you send to a function, then
  passes those copies).  So if we want to set lots of numbers in a function, we would pass in pointers
  to variables, and then the function would modify the memory pointed to by the pointers.
*/

void square(double *x)  //this function takes a pointer to a double, not an actual double.
{
  *x = (*x)*(*x);  //Set the value pointed to by x to be equal to the value pointed to by x squared
}

/*============================================================================*/
int main(int argc, char *argv[])
{
  double z = atof(argv[1]);
  printf("Input z is %g\n",z);
  square(&z);  //take the address of z and pass it to the function
  printf("Z is now %g\n",z);
}
```

Output:

```
[sievers@tpb4 c-tutorial]$ ./c_pointers 6
Input z is 6
Z is now 36
[sievers@tpb4 c-tutorial]$
```



C Arrays

Code:

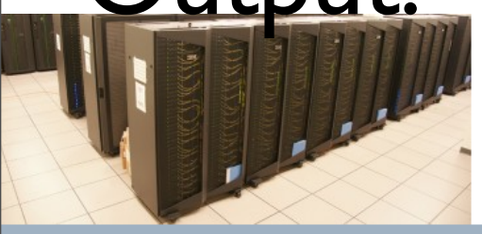
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>  //some macros for quick & dirty sanity checks.

/*In C, arrays are just pointers to blocks of memory.  C is zero-offset, the first element
  of an array x is x[0].  We can access the i'th element of x by using x[i].  The compiler knows
  how big different types are, which is why we need to specify what sort of data a pointer
  points to.  */

void print_array(int n, double *x)
{
  for (int i=0;i<n;i++)
    printf("x[%2d]=%g\n",i,x[i]);
}

/*============================================================================*/
int main(int argc, char *argv[])
{
  double *x;
  int n;
  assert(argc>1);  //Bail if we don't have an input.  Assert will tell you if it failed, and stop.
  n=atoi(argv[1]);
  assert(n>0);
  x=(double *)malloc(n*sizeof(double));
  /*OK - malloc creates a chunk of useable memory, and returns a pointer to its beginning.
    The argument is the number of bytes we want.  The function sizeof() tells you how many
    bytes a datatype takes up.  By default, malloc returns a void *.  The (double *) tells the
    compiler we know a void pointer came back, treat it like a double pointer anyways.*/
  for (int i=0;i<n;i++) //Now fill up the array.
    x[i]=i*i;
  print_array(n,x);
  free(x);  //If we malloced space, we get it back with free.
}
```

assert in action

```
[sievers@tpb4 c-tutorial]$ ./c_arrays -3
c_arrays: c_arrays.c:23: main: Assertion `n>0' failed.
Abort
[sievers@tpb4 c-tutorial]$ ./c_arrays
c_arrays: c_arrays.c:21: main: Assertion `argc>1' failed.
Abort
[sievers@tpb4 c-tutorial]$
```

Output:

```
[sievers@tpb4 c-tutorial]$ ./c_arrays 5
x[ 0]=0
x[ 1]=1
x[ 2]=4
x[ 3]=9
x[ 4]=16
```



C 2-D Arrays

Code:

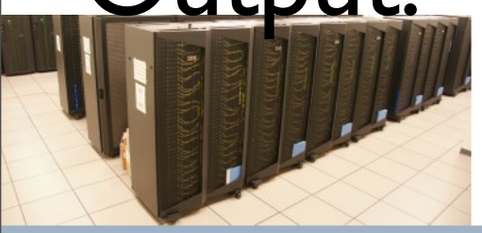
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
void print_array(double **x, int n, int m)
{
  for (int i=0;i<n;i++) {
    for (int j=0;j<m;j++)
      printf(" %3g ",x[i][j]);
    printf("\n");
  }
}
/*-------------------------------------------------------------------------*/
double **allocate_matrix(long n, long m) //long is a 64-bit integer
{
  double *vec=(double *)malloc(n*m*sizeof(double)); //allocate enough space for the matrix
  assert(vec!=NULL);
  double **mat=(double **)malloc(n*sizeof(double *)); //Now allocate a row of pointers.
  assert(mat!=NULL);
  for (int i=0;i<n;i++)
    mat[i]=&vec[m*i];  //assign each pointer to the location of a new row
  return mat;
}
/*=========================================================================*/
int main(int argc, char *argv[])
{
  int n=atoi(argv[1]);
  int m=atoi(argv[2]);
  double **mat=allocate_matrix(n,m);  //C knows how to convert our integers to longs
  for (int i=0;i<n;i++)
    for (int j=0;j<m;j++)
      mat[i][j]=10*i+j;
  print_array(mat,n,m);
}
```

assert in action

Output:

```
[sievers@tpb4 c-tutorial]$ gcc -std=c99 -o c_arrays_2d c_arrays_2d.c
[sievers@tpb4 c-tutorial]$ ./c_arrays_2d 4 6
   0    1    2    3    4    5
  10   11   12   13   14   15
  20   21   22   23   24   25
  30   31   32   33   34   35
[sievers@tpb4 c-tutorial]$
```

```
[sievers@tpb4 c-tutorial]$ ./c_arrays_2d 400000 600000
c_arrays_2d: c_arrays_2d.c:16: allocate_matrix: Assertion `vec!=((void *)0)' failed.
Abort
[sievers@tpb4 c-tutorial]$
```

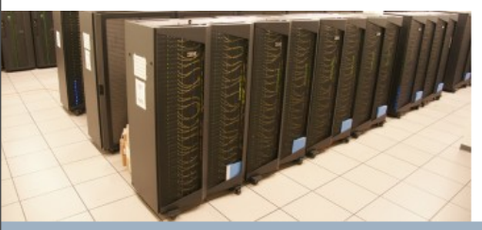


C Structures

Note that C does not initialize values. Forgetting this is an extremely common bug. Often compilers will warn - take them seriously.

Output:

```
[sievers@tpb4 c-tutorial]$ ./c_structures
Charles Babbage is 4195968 years old at first.
Charles Babbage is now 217 years old.
[sievers@tpb4 c-tutorial]$
```



Code:

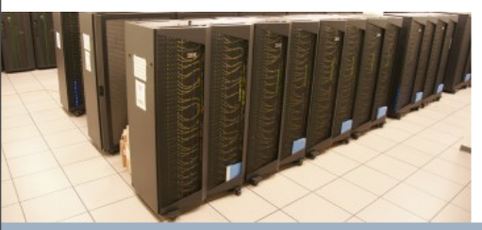
```c
#include <stdio.h>
#define MAXLEN 256  //#define tells the preprocessor to replace MAXLEN by 256
/*This is how we say what a structure will look like*/
struct record_t{
  char name[MAXLEN];
  int age;
  int dob;
};
typedef struct record_t Record; //this step is optional but makes structures more
                                // convenient to declare throughout the code.
/*--------------------------------------------------------------------------*/
void assign_age_wrong(Record dat, int year)
{
  dat.age=year-dat.dob;  //I have passed in the value of the structure.  Nothing
                         //changed here will be seen by the calling routine.
}
/*--------------------------------------------------------------------------*/
void assign_age_right(Record *dat, int year)
{
  dat->age=year-dat->dob; //the -> is C shorthand to get at members of a structure
                          //that comes in as a pointer.
}
/*--------------------------------------------------------------------------*/
int main(int argc, char *argv[])
{
  Record mydata;  //declare the structure using the template defined above
  struct record_t data_copy;  //Without the typedef, we have to do this every time
  sprintf(mydata.name,"Charles Babbage");  //sprintf works like printf, but into a string
  mydata.dob=1791;  //we can assign structure members like this.
  assign_age_wrong(mydata,2008);  //structures get passed around
  printf("%s is %d years old at first.\n",mydata.name, mydata.age);

  /*almost always, you want to pass a pointer to a structure.  This prevents having to
  make copies of potentially large things.  It also is an extremely convenient way of
  getting lots of data into or out of a routine.  If we want an extra argument, we simply
  add that field to the structure.  The function prototype doesn't change.  Large programs
  would sipmly explode and be unmaintainable without this.*/
  assign_age_right(&mydata,2008);
  printf("%s is now %d years old.\n",mydata.name, mydata.age);
}
```

Printing in C

```c
#include <stdio.h>  //This has most of the printing functions in C

int main(int argc, char *argv[])
{
  double xbig=3.456e27;
  double xsmall=4.67890;
  //%e prints a floating point number as an exponent "%%" prints one % sign
  printf("%%e:  xbig is %e, xsmall is %e.\n",xbig,xsmall);
  //%14.6 says use 14 digits w/ 6 decimal points.
  printf("%%14.6e:  xbig is %14.6e, xsmall is %14.6e.\n",xbig,xsmall);
  //if you use %f, it's forced to use fixed-place notation:
  printf("%%14.6f:  xbig is %14.6f, xsmall is %14.6f.\n",xbig,xsmall);
  //%g lets the computer decide between %e and %f
  printf("%%g:   xbig is %g, xsmall is %g.\n",xbig,xsmall);

  int ii=32;
  //%d is the print code for integers
  printf("ii is *%d*, with %%5d is  *%5d*, and %%05d is *%05d*.\n",ii,ii,ii);

  char c='Q', *s="Qwerty";
  //note that chars are single quotes, strings double quotes.
  printf("%c is a char, %s is a string.\n",c,s);

  char line[256];
  //we can also print directly to strings. and to files.
  sprintf(line,"I would like to have %d cores on my laptop.",ii);
  printf("line is:  %s\n",line);
  FILE *outfile=fopen("test_output.txt","w");
  fprintf(outfile,"%s\n",line);
  fclose(outfile);

  return 0;
}
-uuu:---F1  c_printing.c   All L30    (C/l Abbrev)----------------------
```

formatted printing in C is handled by the standard i/o library, not the language itself. Still, a few tips on how to print things in a formatted way are useful. If you need something fancier, possibly there, look on-line.

```
Macintosh-270:c-tutorial sievers$ gcc -Wall -o c_printing c_printing.c
Macintosh-270:c-tutorial sievers$ ./c_printing
%e:  xbig is 3.456000e+27, xsmall is 4.678900e+00.
%14.6e:  xbig is   3.456000e+27, xsmall is   4.678900e+00.
%14.6f:  xbig is 3456000000000000010737418240.000000, xsmall is       4.678900.
%g:   xbig is 3.456e+27, xsmall is 4.6789.
ii is *32*, with %5d is  *   32*, and %05d is *00032*.
Q is a char, Qwerty is a string.
line is:  I would like to have 32 cores on my laptop.
Macintosh-270:c-tutorial sievers$ cat test_output.txt
I would like to have 32 cores on my laptop.
Macintosh-270:c-tutorial sievers$
```



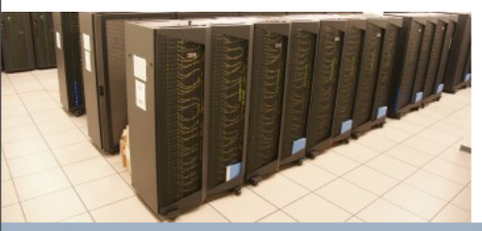
Math in C

Finally, here are some examples of how to do math in C. Because C is a compact language, math functions have been offloaded into a library. The library is very standard, so any C compiler will support its functions.

Some compilers may have math routines built in, and hence not need the "-lm" flag to the right, but this is not standard, and you will likely be punished for your sins the first time you run on a new machine.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/*Math functions are not built into C.  Instead they are library
calls, and a header file must be included.  This applies to
pretty much anything more complicated than +,-,*,/. */

/*Because the math functions are in a library, you must
tell the compiler to include that library.  The compile
flag is "-lm": "-l" tells the compiler it's going to get
a library, and "m" tells it the library is "libm.[a,so...]"*/
#include <assert.h>
int main(int argc, char *argv[])
{
  assert(argc>2);
  double a=atof(argv[1]);  //get a float number from a string.
  double b=atof(argv[2]);
  printf("%g ^ %g = %g\n",a,b,pow(a,b));
  printf("cos(%g)=%0.6g\n",a,cos(a));
  printf("sqrt(%g)=%0.6g\n",a,sqrt(a));

  return 0;
}
```

```
Macintosh-270:c-tutorial sievers$ gcc -Wall -o math_example math_example.c -lm
Macintosh-270:c-tutorial sievers$ ./math_example 2.25 2.5
2.25 ^ 2.5 = 7.59375
cos(2.25)=-0.628174
sqrt(2.25)=1.5
Macintosh-270:c-tutorial sievers$
```



And Now for Parallel

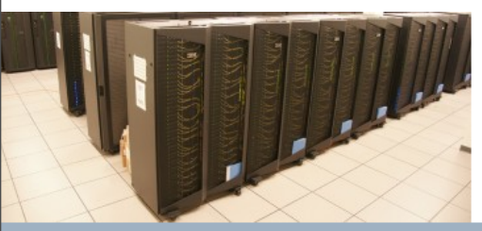
SciNet Parallel Scientific Computing Course

Aug 31 - Sept 4, 2009

SciNet High Performance Computing

Intro to OpenMP

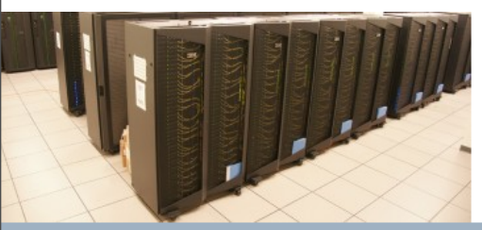
- Or, "Gee, I wish that loop were faster."
- (although newer OpenMP more flexible)



OpenMP Philosophy

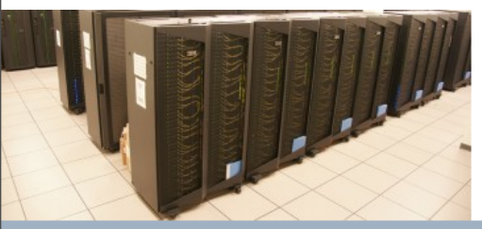
- Goal: Add parallelism to a functioning serial code.
- Add *compiler directives* to parallelize parts of code.

- Requires *shared-memory* machine.
- Pros: Often very easy to add to existing codes.
- Major con: Large shared-memory machines $$$$



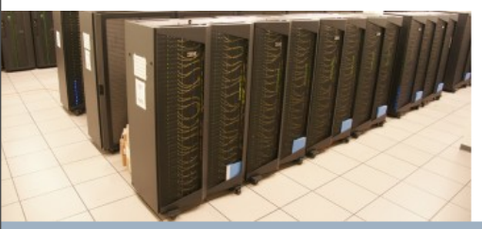
OpenMP Philosophy II

- We tell OpenMP compiler to parallelize a block of code. In practice, mostly fixed-length loops.
- Mark off parallel block: C use #pragma omp ... and {}, FORTRAN use !$OMP and !$OMP END.
- Compiler will spawn threads and split up work for us. Thank you Mr. Compiler!
- We must tell compiler how to use variables. Is a variable *shared* between threads, or does each thread have a *private* copy?



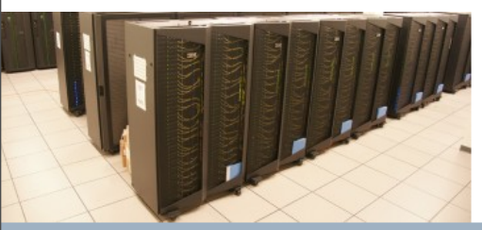
OpenMP Philosophy III

- Not all compilers OpenMP-compatible. OpenMP designed to be ignored by non-OpenMP compilers.
- Most OpenMP implemented with compiler directives. Non-OpenMP compilers will think they're comments.
- OpenMP also provides some library calls. For compatibility, #ifdef guard these calls. OpenMP always defines _OPENMP for this reason.
- Backwards-compatibility rapidly becoming unimportant. Even cheap machines have multiple cores!



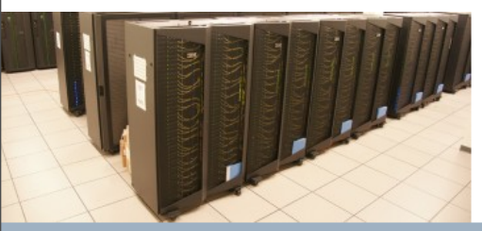
My First OpenMP Program

- ***Goal of first program: figure out in serial region total number of parallel threads.***
- Let's see how many threads we have. (We set this at run time using environment variable OMP_NUM_THREADS).
- omp_get_num_threads() returns *total* number of threads.
- omp_get_thread_num() returns which thread ***I*** am.
- omp_get_num_threads() will return current number of working threads. This will be one if we call it from a serial region.



My First OpenMP Program II

- To find out number of threads, we must ask in a parallel region. To start a parallel region, use command #pragma omp parallel (or !$OMP parallel).
- First, let's get greetings from each thread.
- You will need to include <omp.h>, which has defines and function prototypes for OpenMP



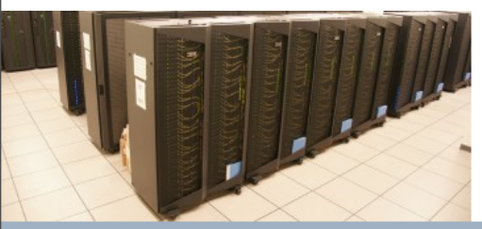
My First OpenMP Program Output

Code:

```
//omp_simplest_program.c
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{

#pragma omp parallel
  {
#ifdef _OPENMP
    printf("I am thread number %d\n",omp_get_thread_num());
#endif
  } //End parallel region.
}
```

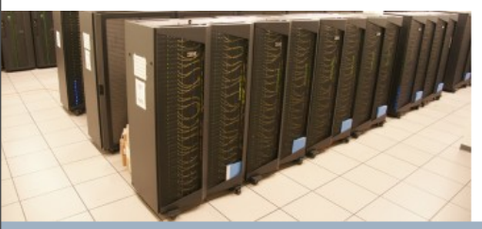
Output:

```
[sievers@tpb4 omp-intro]$
[sievers@tpb4 omp-intro]$ export OMP_NUM_THREADS=8
[sievers@tpb4 omp-intro]$ ./omp_simplest_program
I am thread number 0
I am thread number 1
I am thread number 2
I am thread number 3
I am thread number 4
I am thread number 5
I am thread number 7
I am thread number 6
[sievers@tpb4 omp-intro]
```



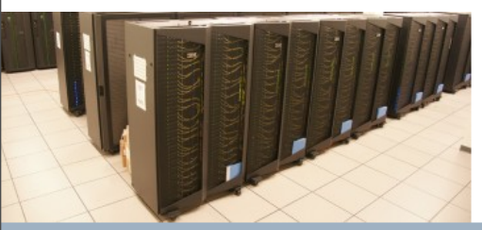
MFOMPP: What Happened?

- We started a parallel region, and each thread printed out its thread ID number.
- What didn't happen? The threads printed out in random order. Threads execute independently, and in general, order will be random.
- What else didn't happen? No variables. Now lets introduce some so we can see how they behave.



MFOMPP: Add a Variable

- Let's assign the number of threads to a shared variable, *nthread*.
- Only one thread needs to do this. So, let's save each thread's number to *mythread*, and only have thread 0 write to *nthread*.
- By default, variables are shared. But each thread needs its own copy of *mythread*. We will declare that to be private.
- (Also going to drop #ifdef's to reduce clutter)



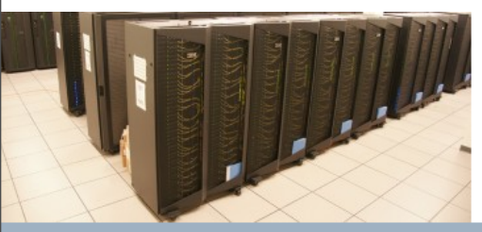
MFOMPP: Getting nthread

Code:

```c
//omp_simplest_program2.c
#include <stdio.h>
#include <omp.h>


int main(int argc, char *argv[])
{

  int mythread,nthread;
#pragma omp parallel private(mythread)
  {
    mythread=omp_get_thread_num();
    if (mythread==0)
      nthread=omp_get_num_threads();
  } //End parallel region.
  printf("I have a total of %d threads in the parallel region.\n",nthread);
}
```

Output:

```
[sievers@tpb4 omp-intro]$ ./omp_simplest_program2
I have a total of 8 threads in the parallel region.
[sievers@tpb4 omp-intro]$
```

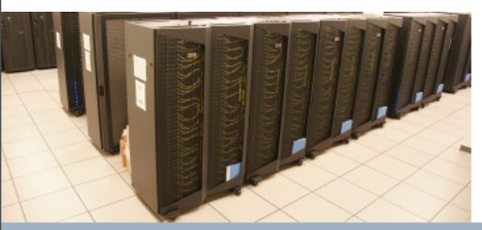


MFOMPP: What Happened Now?

- The shared variable *nthread* was only written to by thread 0, and because it was shared, it maintained its value outside of the parallel region.

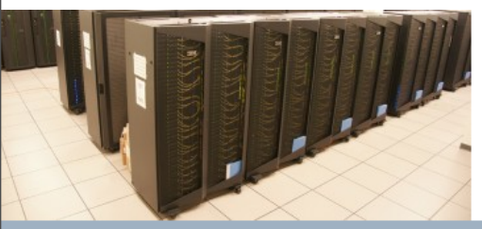
- The compiler created a private copy of *mythread* for each thread. If it had been shared, each thread would have tried to write its own value to *mythread*. There's no telling what *mythread* would have been by the *if* statement. Program behavior would have been indeterminate.

- Another choice (and a very good one): declare *mythread* **locally** inside parallel region.



Quick Note on Initialization

- By default, the initial values of private variables are undefined.
- By default, the values of private variables are lost outside of the parallel region.
- In the #pragma directive, we can override this behavior by declaring variables to be *firstprivate* (import the value from before the parallel region) or *lastprivate* (put the value from the final loop iteration into the serial variable).



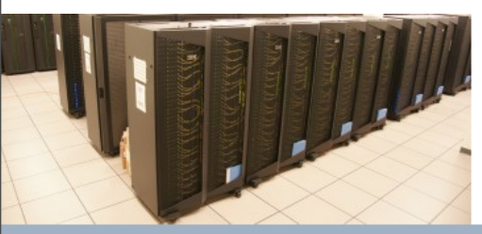
MFOMPP: Getting nthread II

Code:

```c
//omp_simplest_program3.c
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{

  int nthread;
#pragma omp parallel
  {
    int mythread=omp_get_thread_num();
    if (mythread==0)
      nthread=omp_get_num_threads();
  } //End parallel region.
  printf("I have a total of %d threads in the parallel region.\n",nthread);
}
```

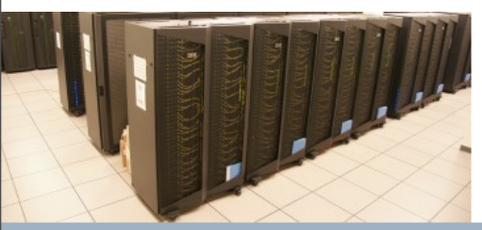
Output:

```
[sievers@tpb4 omp-intro]$ ./omp_simplest_program3
I have a total of 8 threads in the parallel region.
[sievers@tpb4 omp-intro]$
```



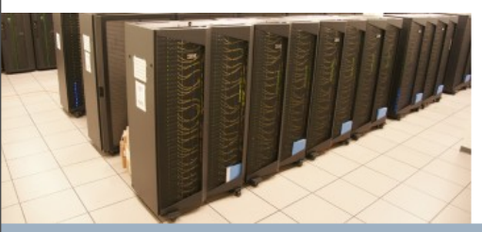
MFOMPP: What Was So Different?

- We declared *mythread* inside the parallel region. Variables declared inside regions are always private.
- What's the big deal? Well, we didn't have to list *mythread* in the #pragma line. Plus, we naturally treated *mythread* as a new, private variable and initialized it accordingly. While trivial, this will save ***a lot*** of debugging time.
- I strongly encourage this for even serial codes. If you get into this habit, you will *never* accidentally loop with the same variable twice!



MFOMPP: Final Version

- We don't really care which thread assigns *nthread*, only that it happens once. OpenMP supports this with the "*#pragma omp single*" command inside a parallel region.
- Another point: we can switch the default behavior of variables. C supports (shared, none), Fortran also supports private.
- Your instructors *strongly* suggest you always use *default(none)*. This will protect you from many, many bugs. Combined with structures (which you should use) and local declarations, overhead of *default(none)* is small.



MFOMPP: Final Version

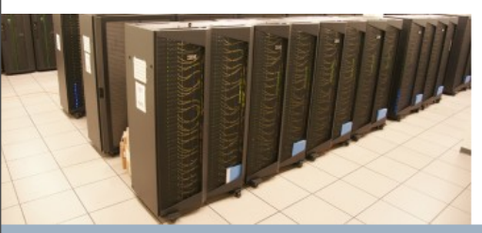
Code:

```c
//omp_simplest_program4.c
#include <stdio.h>
#include <omp.h>


int main(int argc, char *argv[])
{

  int nthread;

#pragma omp parallel shared(nthread) default(none)
#pragma omp single
  nthread=omp_get_num_threads();

  printf("I have a total of %d threads in the parallel region.\n",nthread);
}
```

Output:

```
[sievers@tpb4 omp-intro]$ ./omp_simplest_program4
I have a total of 8 threads in the parallel region.
[sievers@tpb4 omp-intro]$
```

Use of #pragma omp single has made code cleaner and more readable. Use of default(none) has made it safer.



MFOMPP: Final Version in Fortran

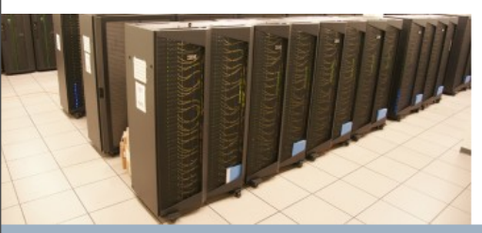
Code:

```
      program omp_simplest_program4_f
      include "omp_lib.h"
      integer nthread

!$OMP parallel shared(nthread)
!$OMP single
      nthread=omp_get_num_threads()
!$OMP end single
!$OMP end parallel

      print *,'In Fortran, we have a total of ',nthread,' threads.'

      end
```

Output:

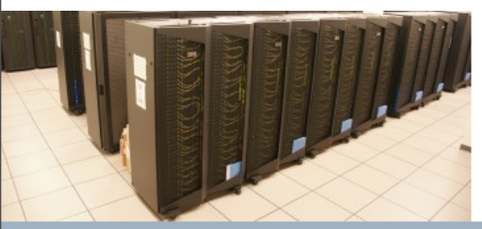
```
[sievers@tpb4 omp-intro]$ ./omp_simplest_program4_f
 In Fortran, we have a total of           8  threads.
[sievers@tpb4 omp-intro]$
```

Code looks similar in Fortran. We needed to include “omp_lib.h” instead of <omp.h>



My First OpenMP Loop

- Now let's look at a simple loop. OpenMP will split up the loop for us, so we don't have to think about it.
- OpenMP shorthand for a single loop: *#pragma omp parallel for* (*omp parallel do* in Fortran). We use the same shared, private clauses as before.
- For each element in the loop, we will print out which thread owns it.



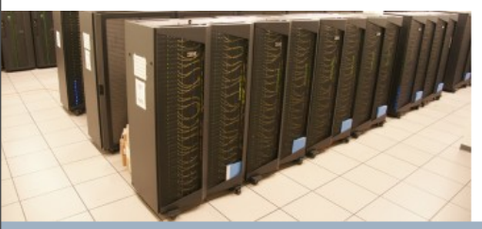
MFOMP Loop

Code:

```
//omp_simple_loop.c
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{

#pragma omp parallel for default(none)
  for (int i=0;i<15;i++)
    printf("Process %d gets i=%d\n",omp_get_thread_num(),i);
}
```

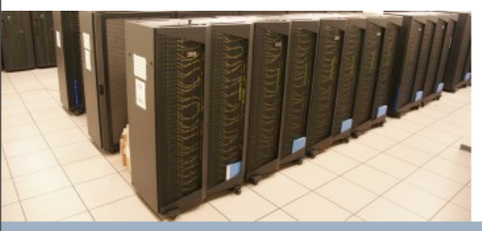
Output:

```
[sievers@tpb4 omp-intro]$ setenv OMP_NUM_THREADS 4
[sievers@tpb4 omp-intro]$ ./omp_simple_loop
Process 3 gets i=12
Process 3 gets i=13
Process 3 gets i=14
Process 0 gets i=0
Process 0 gets i=1
Process 0 gets i=2
Process 0 gets i=3
Process 2 gets i=8
Process 2 gets i=9
Process 2 gets i=10
Process 2 gets i=11
Process 1 gets i=4
Process 1 gets i=5
Process 1 gets i=6
Process 1 gets i=7
[sievers@tpb4 omp-intro]$
```



MFOMPL Debrief

- The *parallel for* directive told OpenMP to split up the work.
- Each node got a chunk of the loop and spat it out.
- *parallel for* is a shorthand for a *parallel* region with a split-up *for* loop.
- We could avoid the repeated calls to omp_get_thread_num() by separating the *parallel* and the *for*.



MFOMP Loop II

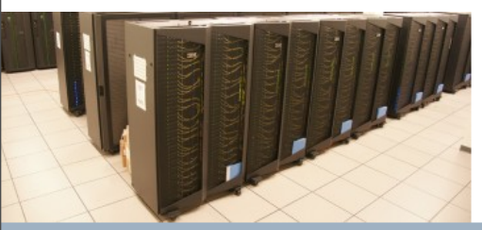
Code:

```
//omp_simple_loop2.c
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{

#pragma omp parallel default(none)
  {
    int mythread=omp_get_thread_num();
#pragma omp for
  for (int i=0;i<15;i++)
    printf("Process %d gets i=%d\n",mythread,i);

  }
}
```

Behaves same as previous version, but we have now saved the repeated calls to omp_get_thread_num().

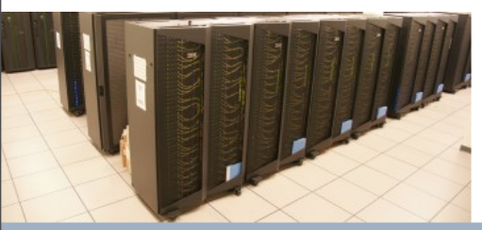
Output:

```
[sievers@tpb4 omp-intro]$ ./omp_simple_loop2
Process 3 gets i=12
Process 3 gets i=13
Process 3 gets i=14
Process 0 gets i=0
Process 0 gets i=1
Process 0 gets i=2
Process 0 gets i=3
Process 1 gets i=4
Process 1 gets i=5
Process 1 gets i=6
Process 1 gets i=7
Process 2 gets i=8
Process 2 gets i=9
Process 2 gets i=10
Process 2 gets i=11
[sievers@tpb4 omp-intro]$
```



Now Let's do Something Useful

- So far, we haven't gotten our threads to do anything.
- Second problem: sum x[i]*y[i] vectors x and y
- We will use OpenMP work-sharing constructs to split up the loop amongst different threads.
- First let's look at a serial version. This will show some of the utilities we will be using in the course of the workshop.



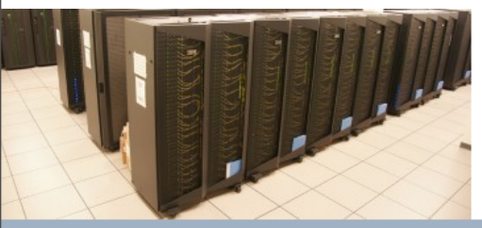
Serial ndot

Code:

```
//serial_ndot.c
#include <stdio.h>
#include "../util/pca_utils.h"

NType ndot(int n, NType *x, NType *y)
{
  NType tot=0;
  for (int i=0;i<n;i++)
    tot+=x[i]*y[i];
  return tot;
}

/*============================================================================*/
int main(int argc, char *argv[])
{
  int n=1e7;
  NType *x=vector(n);
  NType *y=vector(n);
  for (int i=0;i<n;i++) {
    x[i]=i;
    y[i]=i;
  }
  NType nn=n-1;
  NType ans=nn*(nn+1)*(2*nn+1)/6.0;
  pca_time tt;
  tick(&tt);
  NType dot=ndot(n,x,y);
  printf("Dot product is %14.4e (vs %14.4e) for n=%d.  Took %12.4e seconds.\n",dot,ans,n,tocksilent(&tt));
}
```

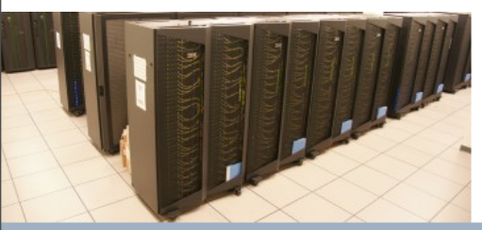
Output:

```
[sievers@tpb4 omp-intro]$ ./serial_ndot
Dot product is     3.3333e+20 (vs     3.3333e+20) for n=10000000.  Took   8.2650e-02 seconds.
[sievers@tpb4 omp-intro]$
```



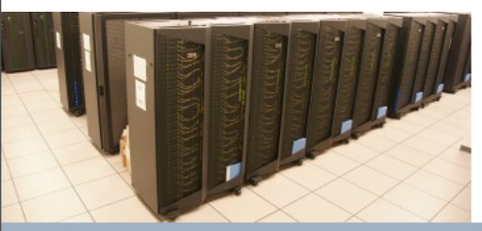
Things to Note:

- We have put some definitions and utility routines in pca_utils.[ch]. We will use them in the example codes.
- We have created a *typedef* called *NType*. By default it will be double, but can also be recast as a float (or even an int). (N originally stood for N-body, since I wasn't sure if that should be done single or double)
- *vector* allocates a vector of NTypes and returns a pointer to the beginning.
- *pca_time* is a datatype to store microsecond-precision time. The *tick()* function resets a timer, and *tock()* tells you how much time has passed since *tick()* was called.



A Parallel Dot Product

- We could clearly parallelize the loop.
- We need the sum from everybody. We could make *tot* shared, then all threads can add to it.
- *Don't!!!* Multiple threads may try to update *tot* at the same time. If they do, then we'll get wrong answers.
- This is known as a *race* condition. Threads race each other to change shared objects. A classic parallel bug.
- Let's have a look:



Parallel ndot - Data Race

Code:

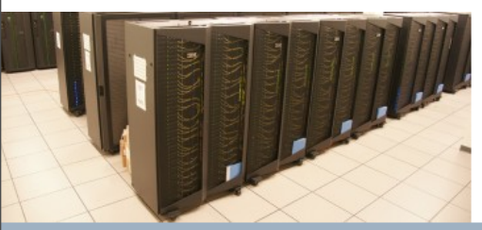
```
//omp_ndot_race.c
#include <stdio.h>
#include "../util/pca_utils.h"

NType ndot(int n, NType *x, NType *y)
     // This version of ndot will produce wrong answers because multiple threads
     // will try to update tot at the same time.
{
  NType tot=0;
#pragma omp parallel for shared(x,y,n,tot)
  for (int i=0;i<n;i++)
    tot+=x[i]*y[i];
  return tot;
}
```

(Main part of code unchanged, only showing the dot product routine.)

Output:

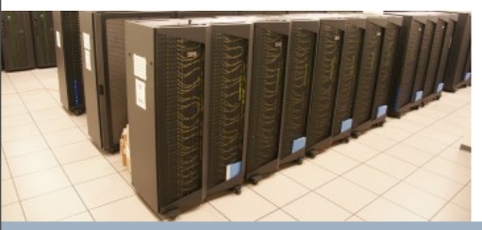
```
[sievers@tpb4 omp-intro]$ setenv OMP_NUM_THREADS 1
[sievers@tpb4 omp-intro]$ ./omp_ndot_race
Dot product is     3.3333e+20 (vs     3.3333e+20) for n=10000000.  Took   1.2495e-01 seconds.
[sievers@tpb4 omp-intro]$ setenv OMP_NUM_THREADS 8
[sievers@tpb4 omp-intro]$ ./omp_ndot_race
Dot product is     7.4971e+19 (vs     3.3333e+20) for n=10000000.  Took   2.0928e-01 seconds.
[sievers@tpb4 omp-intro]$
```

Not only is the answer wrong, it was slower to compute!



Data Races

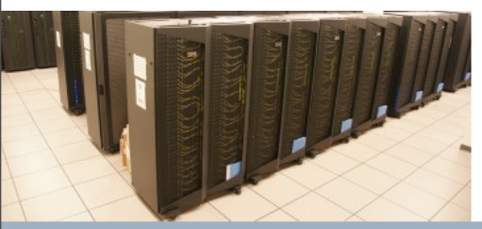
- So we got a wrong answer. What happened inside the hardware?
- Shared variables live in main memory. Cores process data in their cache.
- When a thread wants to update tot, it will pull it to its cache, modify it, and return to main memory.
- If threads try to change at the same time, both pull the same value, update, and return to main memory. Whoever finishes second wins.
- Can be very subtle in practice. Errors are not repeatable, and may not show up until problems get surprisingly large.



Critical Directive

- If threads waited for other threads to update, then we would get the correct answer.
- OpenMP supports this. The *#pragma omp critical* directive tells the compiler to only let one thread in at a time.

- The overhead for critical regions can be large. In this case, the OpenMP run-time system needs to keep track of all threads for every iteration.
- However, answer should be correct.



Parallel ndot - Pure Critical

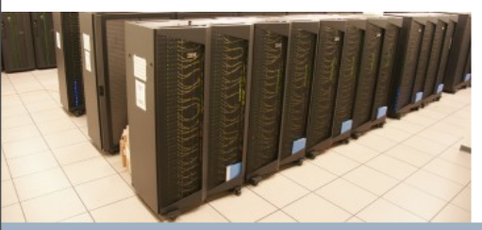
Code:

```
NType ndot_critical(int n, NType *x, NType *y)
     // This version of ndot will produce correct answers but
     // be very slow because of the critical overhead.
{
  NType tot=0;
#pragma omp parallel for shared(x,y,n,tot)
  for (int i=0;i<n;i++)
#pragma omp critical
    tot+=x[i]*y[i];
  return tot;
}
```

Output:

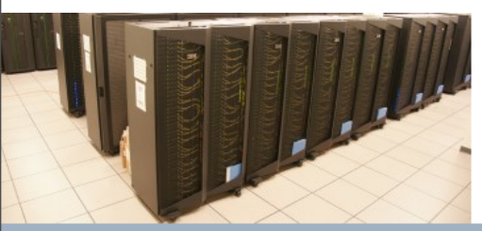
```
[sievers@tpb4 omp-intro]$ ./omp_ndot_race
Dot product is     3.3333e+20 (vs     3.3333e+20) for n=10000000.  Took   3.8808e+00 seconds.
[sievers@tpb4 omp-intro]$
```

Answer is now correct, but we are 30 times slower than the serial version!!!



Atomic Directive

- *#pragma omp critical* will work for arbitrary block of code. There usually exists specialized hardware for reading, modifying, and writing to a single memory location.
- OpenMP supports this. The *#pragma omp atomic* directive lets the compiler take advantage of this hardware support. Supports limited commands: =, +=, *=, a few others.
- Due to lower overhead, atomic should be faster. Still won't be that good, however.



Parallel ndot - Pure Atomic

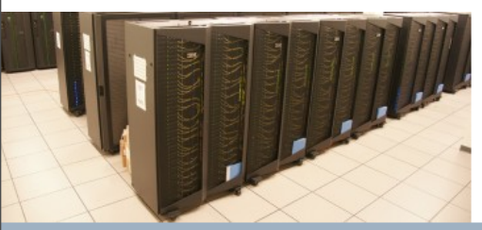
Code:

```
NType ndot_atomic(int n, NType *x, NType *y)
     // This version of ndot will produce correct answers.  It
     // will be faster than critical, but still be slow.
{
  NType tot=0;
#pragma omp parallel for shared(x,y,n,tot)
  for (int i=0;i<n;i++)
#pragma omp atomic
    tot+=x[i]*y[i];
  return tot;
}
```

Output:

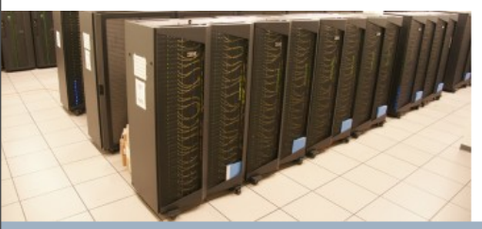
```
[sievers@tpb4 omp-intro]$ ./omp_ndot_race
Dot product is     3.3333e+20 (vs     3.3333e+20) for n=10000000.  Took   2.7980e+00 seconds.
[sievers@tpb4 omp-intro]$
```

For this case, *atomic* about 30% faster than *critical*. Still 20 times slower than serial!



Better Reduction

- The big problem is that many threads are trying to update the same location.
- Dot product doesn't depend on order of summation. So, let each thread sum its bit into its own private variable, then combine.
- We will have a shared variable *tot*, updated at the end, and private variables *mytot* for each thread.
- At end of loop, sum *mytot* into *tot* using an *atomic* directive.



Parallel ndot - Atomic Reduction

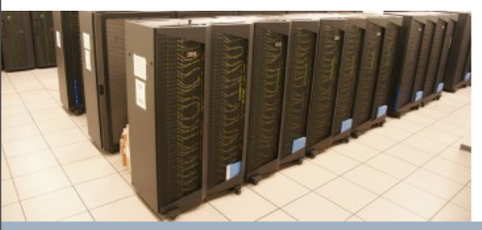
Code:

```
NType ndot_atomic_reduce(int n, NType *x, NType *y)
     // This version of ndot should be OK.
{
  NType tot=0;
#pragma omp parallel shared(x,y,n,tot)
  {
    NType mytot=0;
#pragma omp for
    for (int i=0;i<n;i++)
      mytot+=x[i]*y[i];

#pragma omp atomic
    tot+=mytot;
  }
  return tot;
}
```

Output:

```
[sievers@tpb4 omp-intro]$ setenv OMP_NUM_THREADS 1
[sievers@tpb4 omp-intro]$ ./omp_ndot_race
Dot product is     3.3333e+20 (vs     3.3333e+20) for n=10000000.  Took   9.3732e-02 seconds.
[sievers@tpb4 omp-intro]$ setenv OMP_NUM_THREADS 8
[sievers@tpb4 omp-intro]$ ./omp_ndot_race
Dot product is     3.3333e+20 (vs     3.3333e+20) for n=10000000.  Took   3.6198e-02 seconds.
[sievers@tpb4 omp-intro]$
```

Now we're in business! Correct answer, ~3x faster than serial.



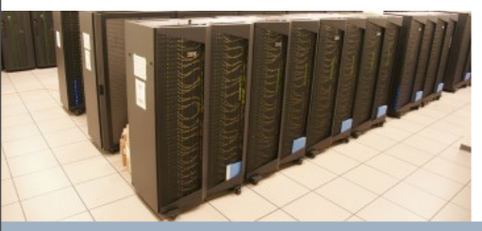
What Did We Do? What Next?

- Started a parallel region. Declared a private variable. Summed our piece of a parallel loop. Finally, at end, summed our piece into the total.

- This operation, where we sum private copies into a shared variable, is called a *reduction*. Reductions are extremely common in scientific parallel programming.

- OpenMP has reductions built into the standard. Instead of declaring a variable to be *private* or *shared*, we can declare it to be *reduction*, and OpenMP will take care of it for us.

- C supports +, -, and * reductions (plus some bit mask ones). Fortran also supports *min* and *max*.



Parallel ndot - OpenMP Reduction

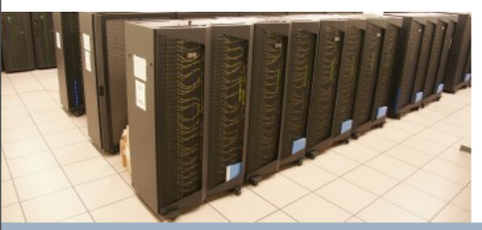
Code:

```
NType ndot_reduce(int n, NType *x, NType *y)
     // This version of ndot will be OK.  The use of the
     // reduction clause makes it much more compact.
{
  NType tot=0;
#pragma omp parallel for shared(x,y,n)   reduction(+:tot) default(none)
  for (int i=0;i<n;i++)
    tot+=x[i]*y[i];

  return tot;
}
```

Output:

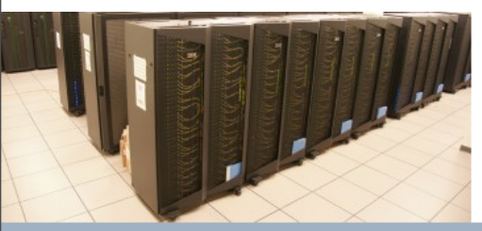
```
[sievers@tpb4 omp-intro]$ ./omp_ndot_race
Dot product is     3.3333e+20 (vs     3.3333e+20) for n=10000000.  Took   3.8276e-02 seconds.
[sievers@tpb4 omp-intro]$
```

Same answer, time as our manual reduction. But much simpler to code!



Performance

- We threw in 8 cores, got a factor of 3 speedup. Why?
- Often we are limited not by CPU power but by how quickly we can feed CPU's.
- For this problem, we had 10^7 long vectors, with 2 numbers 8 bytes long flowing through in 0.036 seconds.
- Combined bandwidth from main memory was 4.3 GB/s. Not far off of what we could hope for on this architecture.
- One of the keys to good OpenMP performance is using data when we have it in cache. Complicated functions: easy. Low work-per-element (dot product, FFT): hard.



Parallel ndot - Lots of Work Per *i*

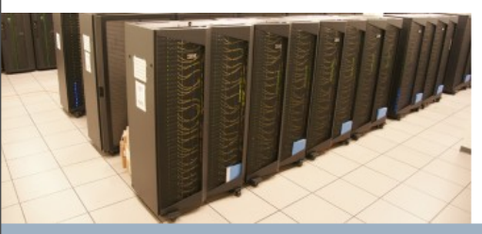
Code:

```
NType ndot_log_reduce(int n, NType *x, NType *y)
     // This version of ndot will be OK.  The use of the
     // reduction clause makes it much more compact.
{
  NType tot=0;
#pragma omp parallel for shared(x,y,n)   reduction(+:tot) default(none)
  for (int i=0;i<n;i++)
    tot+=log(1+x[i])*pow(1.5+x[i],2.5)+log(1+y[i])*pow(1.5+y[i],3.98);

  return tot;
}
```

Output:

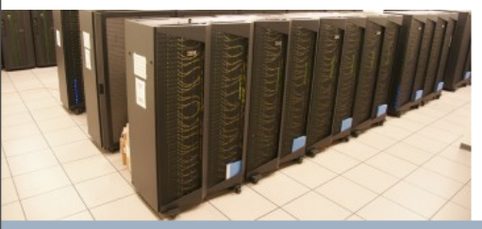
```
[sievers@tpb5 omp-intro]$ setenv OMP_NUM_THREADS 1
[sievers@tpb5 omp-intro]$ ./omp_ndot_race
Dot product is     2.3155e+35 (vs     3.3333e+20) for n=10000000.  Took   6.1813e+00 seconds.
[sievers@tpb5 omp-intro]$ setenv OMP_NUM_THREADS 8
[sievers@tpb5 omp-intro]$ ./omp_ndot_race
Dot product is     2.3155e+35 (vs     3.3333e+20) for n=10000000.  Took   7.9433e-01 seconds.
[sievers@tpb5 omp-intro]$
```

8 threads gives me 7.8 times faster job. That's more like it!



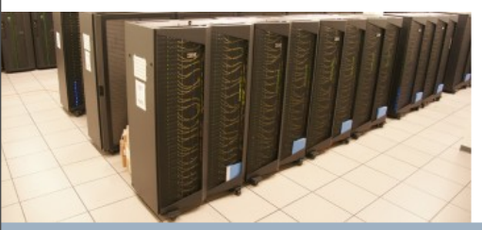
OpenMP Versions

- So far, OpenMP is good mainly for loops. This was generally true for a long time.
- OpenMP 3.0 is more flexible - will will meet some of it tomorrow.
- OpenMP3 very new - best documentation is standard itself. gcc 4.3.2 & later support(?) it.
- See: http://openmp.org/wp/openmp-specifications/ for more info. Strongly encouraged - many good sample programs.



Hand-on 0

- Make a directory in your pca/src directory called ‘hw2’.
- Copy /scratch/sievers/pca/src/hw2/Makefile into hw2, do your work there.
- Digital version of these slides available on the scinet wiki: https://support.scinet.utoronto.ca/wiki/index.php

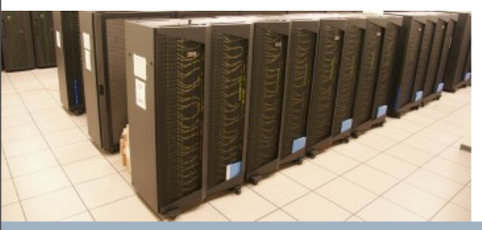


Hands On 1:

- Write and compile a C program from scratch to allocate a 2-D array using pointers. The user should be able to specify the both dimensions of the matrix on the command line. The allocation should be in a function, not main()

- When the matrix successfully allocates, write a function to fill it such that matrix[i][j]=sin(sqrt((1.0+i)/(1.0+j))). Print the matrix to the screen in a separate function. Finally, write a fourth function to sum the elements of the matrix and print it out. Call this program mat_2d.c

- If it has worked correctly, matrix is antisymmetric, elements (2,1) and (3,1) are -0.339677 -0.522096

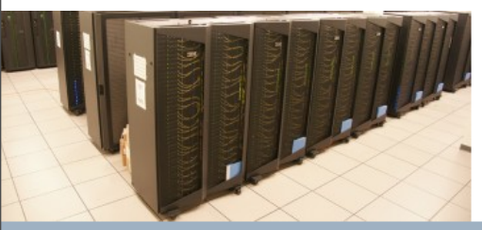


Hands On 2

- Now let's add check timings and add parallelism. Copy *mat_2d.c* to *mat_2d_omp1.c*. How long does it take to fill a 3,000 by 3,000 matrix? A 3 by 3,000,000? A 3,000,000 by 3 matrix? You may wish to turn off printing the matrix.

- Parallelize the fill using OpenMP. Repeat the same three timing tests. How much did we improve? Did we get a factor of 2 speedup in all cases? Any cases?

- For a 5x5 matrix, print which process did which assignments.

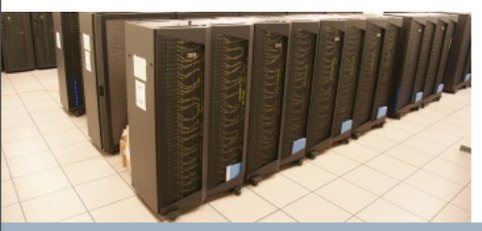


Hands On 3

- Now introduce a counter so that every time a thread assigns a value to the matrix, it increments its counter by one. For our same cases (3e6 by 3, 3 by 3e6, 3k by 3k) how much work are the different threads doing? Why? (If the work reported by threads and the timings seem to disagree, don't worry, we will see what's happening in the next lecture.)

- Can you change the parallelization so that the broken case is fixed? What happened to the other cases?

- OpenMP 3.0 introduces a "collapse" clause to tell the compiler to combine loops. Make sure the parallel is attached to the outer loop and add collapse(2) to the *#pragma omp for* directive, and re-run the same 3 cases. How much work is each thread doing now? Call this program *mat_2d_omp2.c*



Hands On 4

- Finally, write routines to sum the elements of the matrix in parallel in a program called *mat_2d_omp3.c*.
- One routine should have each thread explicitly keep track of its private sum and then combined using *critical* or *atomic* directives.
- One routine should use the *reduction* clause. This should require only one extra line over the serial case.
- You have seen each of these code snippets in the lectures, but it is good practice to write them yourselves.