Lecture 11 – Compilers

- **We want to make the compiler's job easy:**

1. **Overhead:**

   Subroutine calls, indirect memory references, test within loops, wordy tests, variables preserved unnecessarily

2. **Things that restrict compiler analysis:**

   Pointers, subroutine calls, indirect memory references

A good algorithm may give you far more speed than tuning!!

Numerical Libraries

• Use the libraries provided by the vendors.

Several smart people spent weeks (months) tuning the implementation and rewriting critical sections in assembler.

• There are now some self-tuning libraries (FFTW, ATLAS,...) that have performances comparable to hand-tuned code.

For more information, see:
http://en.wikipedia.org/wiki/Automatically_Tuned_Linear_Algebra_Software

Efficient Programming

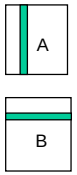
- **Design cache-friendly data structures**
  - Make each cache-line transfer pay
  - Nested loops - Fortran leftmost subscript varies fastest
  - Nested loops - C rightmost subscript varies fastest
- **Avoid aliasing – arguments pointing to same memory**
- **Ensure loops can be analyzed**
  - Avoid branches within innermost loop
  - choose "counted" loops (which include DO or IF loops, but not DO WHILE loops)

Blocking

Transpose: $A=B^T$

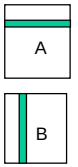
```
DO I=1,N
   DO J=1,N
      A(J,I)=B(I,J)
   END DO
END DO
```

A accessed with unit stride,
B with non-unit stride N

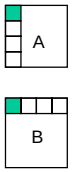



```
DO J=1,N
   DO I=1,N
      A(J,I)=B(I,J)
   END DO
END DO
```

B accessed with unit stride,
A with non-unit stride N



We want to block references, processing small blocks of A and B, to conserve cache entries.



Possible Parallel Programming Models

- **Shared memory:**
  - Automatic parallelization
  - Pthreads (Coarse, Medium Grain)
  - Compiler directives (Coarse, Medium Grain): OpenMP for cores including new Intel Mic (Many Integrated Core processor) and even GPUs, OpenACC for GPUs
  - Graphical processors (Fine Grain) : CUDA (C or Fortran), OpenCL (C)
- **Message passing:**
  - MPI: message passing interface
  - PVM: parallel virtual machine
  - HPF: high performance fortran

Shared-Memory Pthreads

- **Posix threads:**
  - Standard definition but non-standard implementation
  - Hard to code
  - Set of low-level primitives that give user more control at the price of complication
  - Predominantly used by system programmers rather than application programmers

A website where you can learn more about Posix threads is:
http://www.lynuxworks.com/products/posix/threads.php3

Shared-Memory OpenMP

- **New "de-facto" standard available on all major platforms**
- **Easy to implement (easier than distributed memory parallelization)**
  - Higher-level primitives in the form of compiler directives
  - Easier to use for application programmers
- **Single source for parallel and serial code**
- **OpenMP is included in most Fortran90/95 compilers**
  - Is included in pgf90 on wopr
  - Latest versions can also be used for GPUs

A website where you can learn more about OpenMP is:
http://www.openmp.org

Shared Memory on GPUs (CUDA, OpenCL, OpenACC)

- **Graphical processors (GPUs) have their own shared memory**
- **Processing on the GPUs can be performed with standards that include**
  - CUDA (in C or Fortran) for NVIDIA GPUs
  - OpenCL for any GPU (NVIDIA or other vendors)
  - Both languages use the concept of Threads similar to OpenMP which now can be used directly for GPUs as well as the OpenACC directive package
- **Algorithm Kernels are called from a "kernel" routine that calls another "thread" routine**

Shared Memory on GPUs (CUDA, OpenCL, OpenACC)

- **Thread routines consist of instructions that are performed for every micro-processor**

  Websites where you can learn more about CUDA or OpenCL are

  http://developer.nvidia.com/opencl

  http://developer.nvidia.com/category/zone/cuda-zone

Distributed-Memory MPI

- **Standard parallel interface:**
  - MPI 1: two-sided communication
  - MPI 2: extend MPI 1 with single side communication (GAS)
    - See http://www.cs.berkeley.edu/~bonachea/upc/mpi2.html for a good explanation
- **Need to rewrite the code to perform message passing**
- **Code portable on all architectures (open-source code available)**
- **There are different implementations of MPI depending on the computer: OpenMPI, MPICH, Portland Group MPI (with pgi)**

  Websites where you can find out more about OpenMPI and MPICH (open-source standards) is:

  http://www.open-mpi.org/

  http://www-unix.mcs.anl.gov/mpi/mpich

Distributed-Memory PVM

- **Parallel Virtual Machine**
- **Another popular message passing interface**
- **Useful in environment with multiple vendors and for MPMD approach**
- **Again, PVM is open-source software that can be downloaded**

  The website where you can find out more about PVM is:

  http://www.csm.ornl.gov/pvm/pvm_home.html

Simple Code to Compute π

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

```
      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/real(n)
      sum = 0.0d0
      do i = 1, n
            x = w * (real(i) - 0.5d0)
            sum = sum + f(x)
      end do
      pi = w * sum
      print *, 'computed pi = ', pi
      stop
      end
```

Pthreads C-code 1/3

```
#include <stdio.h>
#include <unistd.h>
#include <sys/times.h>
#include <pthread.h>
#include <stdlib.h>

float  sumall, width;
int i, iend;
__private int ibegin;
__private int cut;
__private float x;
__private float  xsum;
void *do_work();
main(argc,argv)
int argc;
char *argv[];
{
/* Pi - Program loops over slices in interval, summing area of each slice
*/
     struct tms time_buf;
     float ticks, t1, t2;
     int intrvls,   numthreads, istart;
     pthread_t pt[32];

/* get intervals from command line */

     intrvls = atoi(argv[1]);
     numthreads =  atoi(getenv( "PL_NUM_THREADS"));
     printf(" intervals = %d PL_NUM_THREADS = %d \n",intrvls, numthreads);
```


Pthreads C-code 2/3

```
/* get number of clock ticks per second and initialize timer */

     ticks = sysconf(_SC_CLK_TCK);
     t2 =times(&time_buf);

/*  - - Compute width of cuts */

     width = 1. / intrvls;
     sumall = 0.0;

/* - - Loop over interval, summing areas */
     istart = 1;
     iend = intrvls/numthreads;
     for (i = 0; i < numthreads - 1 ; i++)
     {
       pthread_create(&pt[i], pthread_attr_default, do_work,(void *) istart);
       istart += iend;
     }
     do_work( istart);
       istart += iend;
       for (i = 0; i < numthreads - 1  ; i++)
         {
           pthread_join(pt[i], NULL);
         }
/*   - - fininish any remaining slices */
       iend = intrvls - (intrvls/numthreads)* numthreads;
     if( iend) do_work( istart);
/* - - Finish overall timing and write results */
      t1 = times(&time_buf);
     printf("Time in main = %20.14e   sum = %20.14f \n",(t1 -t2)/ticks,sumall);
     printf("Error = %20.15e \n",sumall - 3.14159265358979323846);
}
```


Pthreads C-code 3/3

```
void *do_work(istart)
int istart;
{
     ibegin = istart;
     xsum = 0.0;
     for (cut = ibegin ; cut < ibegin+iend; cut++)
     {
       x = (( (float ) cut) - .5) * width ;
       xsum += width * 4. /(1. + x * x);
     }
     sumall += xsum;
}
```

- As you can see, Posix (pthreads) gives a lot of control at the expense of complexity. This is probably not the technique to use in engineering simulations.
- OpenMP is much simpler!!

OpenMP code 1/1

```
      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/real(n)
       sum = 0.0d0
!$OMP PARALLEL DO PRIVATE(x), SHARED(w)
!$OMP& REDUCTION(+: sum)
       do i = 1, n
               x = w * (real(i) - 0.5d0)
               sum = sum + f(x)
       end do
!$OMP END  PARALLEL DO
       pi = w * sum
       print *, 'computed pi = ', pi
       stop
       end
```

- Directives are much simpler !!

MPI code 1/2

```
      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/real(n)
      sum = 0.0d0
      do i = myid+1, n, numprocs
            x = w * (real(i) - 0.5d0)
            sum = sum + f(x)
      enddo
      mypi = w * sum
```


MPI code 2/2

```
!     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
```

- We will use MPI across the CPU/Cores in this course since it is relatively simple, readily available for many computers with both Fortran 95/03 and C, and has become the standard.
- However, we could instead use OpenMP within a node (blocks).
- OR, we could use a combination of MPI across the nodes and OpenMP within the node (blocks).

PVM master code 1/3

```
      program compute_pi_master
      include '~/pvm3/include/fpvm3.h'
      parameter (NTASKS = 5)
      parameter (INTERVALS = 1000)
      integer mytid
      integer tids(NTASKS)
      real sum, area
      real width
      integer i, numt, msgtype, bufid, bytes, who, info

      sum = 0.0
!       Enroll in PVM
      call pvmfmytid(mytid)
!       spawn off NTASKS workers
      call pvmfspawn('comppi.worker', PVMDEFAULT, ' ', NTASKS, tids, numt)
      width = 0.0
      i = 0
!       Multi-cast initial dummy message to workers
      msgtype = 0
      call pvmfinitsend(0, info)
      call pvmfpack(INTEGER4, i, 1, 1, info)
      call pvmfpack(REAL4, width, 1, 1, info)
      call pvmfmcast(NTASKS, tids, msgtype, info)
!       compute interval width
      width = 1.0 / INTERVALS
```


PVM master code 2/3

```
      do i = 1, NTASKS
!       for each interval, 1) receive area from worker 2) add area to sum 3) send worker new interval
                number and width
        call pvmfrecv(-1, -1, bufid)
        call pvmfbufinfo(bufid, bytes, msgtype, who, info)
        call pvmfunpack(REAL4, area, 1, 1, info)
        sum = sum + area
        call pvmfinitsend(PvmDataDefault, info)
        call pvmfpack(INTEGER4, i, 1, 1, info)
        call pvmfpack(REAL4, width, 1, 1, info)
        call pvmfsend(who, msgtype, info)
      enddo

!     Signal to workers that tasks are done
      i = -1
      call pvmfinitsend(0, info)
      call pvmfpack(INTEGER4, i, 1, 1, info)
      call pvmfpack(REAL4, width, 1, 1, info)
```

PVM master code 3/3

```
!       Collect the last NTASK areas and send the completion signal
      do i = 1, NTASKS
        call  pvmfrecv(-1,-1, bufid)
        call pvmfbufinfo(bufid, bytes, msgtype, who, info)
        call pvmfunpack(REAL4, area, 1, 1, info)

        sum = sum + area

        call pvmfsend(who, msgtype, info)
      enddo

      print 10,sum
 10   format(x,'Computed value of Pi is ',F8.6)

      call pvmfexit(info)
      end
```

Note that the PVM code is generally more complicated than it's MPI counter-part due to buffer manipulation

PVM worker code 1/2

```
      integer mytid, master
      real area
      real width, int_val, height
      integer int_num
!     Enroll in PVM
      call pvmfmytid(mytid)
!     who is sending me work?
      call pvmfparent(master)

!     receive first job from the master
      call pvmfrecv(-1, -1, info)
      call pvmfunpack(INTEGER4, int_num, 1, 1, info)
      call pvmfunpack(REAL4, width, 1, 1, info)

!  While I've not been sent the signal to quit, I'll keep processing
 40   if (int_num .eq.  -1) goto 50
!       compute interval value from interval number
        int_val = int_num * width
!       compute height of given rectangle
        height = F(int_val)

!       compute area
        area = height * width
```


PVM worker code 2/2

```
!       send area back to master
        call pvmfinitsend(PvmDataDefault, info)
        call pvmfpack(REAL4, area, 1, 1, info)
        call pvmfsend(master, 9, info)

!       Wait for next job from master
        call pvmfrecv(-1, -1, info)
        call pvmfunpack(INTEGER4, int_num, 1, 1, info)
        call pvmfunpack(REAL4, width, 1, 1, info)
      goto 40
!     all done
 50   call pvmfexit(info)
      end

      real function f(x)
      real x
      f = 4.0/(1.0+x**2)
      return
      end
```

This is one reason why MPI has now become the standard for distributed-memory parallel