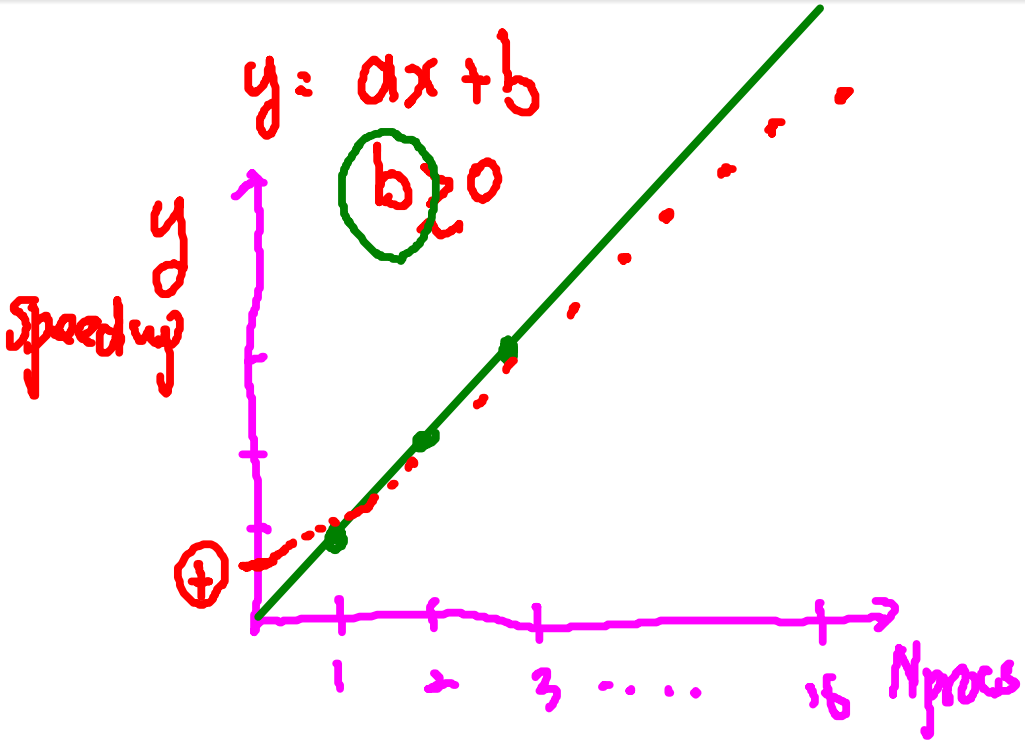
CEE 618 Scientific Parallel Computing (Lecture 7): OpenMP (con'td) and Matrix Multiplication

Albert S. Kim

Department of Civil and Environmental Engineering
University of Hawai'i at Manoa
2540 Dole Street, Holmes 383, Honolulu, Hawaii 96822

Table of Contents

Outline



Latency

- The time it takes to send an empty message over the communication medium, from the time the send routine is called to the time the empty message is received by the recipient. Programs that generate large numbers of small messages are sensitive to the latency and are called "**latency bound**" programs.
  1. Network latency = the time to make a message transfer through the network
  2. Communication latency = the total time to send th message, including the software overhead and interface delays.
  3. Message latency, or startup latency = the tie to send a zero message, which is essentially the software and hardware overhead in sending a message (finding the route, pakcing, unpacking, ets.) onto which must be added the actual time to send the data along the interconnection path.

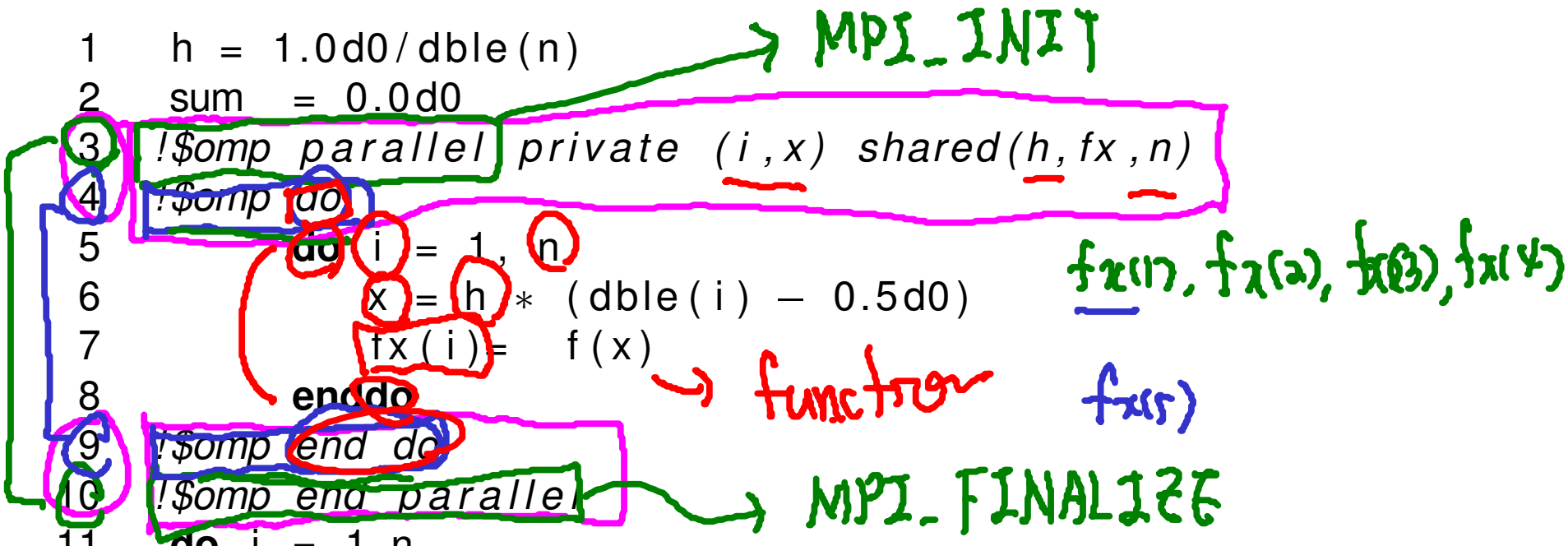
Bandwidth (↑)

- The capacity of a system, usually expressed as “items per second”. In parallel computing, the most common usages of the term bandwidth is in reference to
  [the number of bytes per second]
  that can be moved across a network link.
- A parallel program that generates relatively small numbers of huge messages may be limited by the bandwidth of the network in which case it is called a “**bandwidth limited**” program.

Outline

opi1.f90

```
1    h = 1.0d0/dble(n)
2    sum  = 0.0d0
3    !$omp parallel private (i,x) shared(h,fx,n)
4    !$omp do
5          do i = 1, n
6             x = h * (dble(i) - 0.5d0)
7             fx(i)=  f(x)
8          enddo
9    !$omp end do
10   !$omp end parallel
11   do i = 1,n
12        sum = sum + fx(i)
13   end do
14   pi = h * sum
```



1. Surround the do loop with **parallel** and **end parallel**.
2. To prevent that all threads execute this loop redundantly, the loop is further enclosed by **do** and **end do** directives in order to distribute the loop iterations to all processors (worksharing).

opi01.f90 (cont'd)

3. Since by default all variables are accessible by all threads (shared), the exceptions have to be taken care of. A first candidate for privatization is the loop index i. If the loop iterations shall be distributed, the loop index has to be **private**. This is realized through a private clause of the parallel directive.
4. As a second candidate for privatization there is the variable x, which is *used to temporally store* the node of the quadrature formula. This happens **independently** for each loop iteration and the variable contents is not needed after the loop.
5. If 'sum' is shared, all the processors compete each other to read the value and add newly calculated value of $f(x)$ stored in arry 'fx'. Indexes of 'fx' to be accessed by each processor are different, so shared. A value of 'sum' be messed up.

opi02.f90

```
1    sum  = 0.0d0
2   !$omp parallel
3   !$omp do private (i,x)
4    do i = 1, n
5      x = h * (dble(i) - 0.5d0)
6   !$omp critical
7      sum = sum + f(x)
8   !$omp end critical
9    enddo
10  !$omp end do
11  !$omp end parallel
```

1. Critical regions are segments of code which can only be executed by a single thread at a time.
2. This version however involves quite some overhead, because it introduces a synchronization with every iteration of the inner loop.

opi03.f90

```
1   !$omp parallel private (i,x, sum_local)
2    sum_local=0.0d0
3
4   !$omp do
5    do i = 1, n
6        x = h * (dble(i) - 0.5d0)
7        sum_local = sum_local + f(x)
8    enddo
9   !$omp end do
10
11  !$omp critical
12        sum = sum + sum_local
13  !$omp end critical
14  !$omp end parallel
```

1. The next version introduces an additional private variable, in which the individual threads sum up their contributions.
2. The total sum is then computed in a critical region after the parallel loop: one by one.

opi04.f90

```
1   !$omp parallel private (i,x)
2   !$omp do reduction (+:sum)
3    do i = 1, n
4         x = h * (dble(i) - 0.5d0)
5         sum = sum + f(x)
6    enddo
7   !$omp end do
8   !$omp end parallel
```

→ MPI_REDUCE

1. Analogous to the reduction function in MPI - a reduction clause of the do directive.

opi05.f90

```
1   !$omp parallel do private (i,x) reduction (+:sum)
2    do i = 1, n
3        x = h * (dble(i) - 0.5d0)
4        sum = sum + f(x)
5    enddo
```

ending point of omp PARALLEL.

1. Shortest form.

opi06.f90

```
1   !$OMP PARALLEL private (myid)
2    myid       = OMP_GET_THREAD_NUM()
3    numthreads = omp_get_num_threads()
4    write(*,*) myid, numthreads
5   !$OMP DO SCHEDULE(STATIC, numthreads) reduction(+:sum)
6    do i = 1, n
7        x = h * (dble(i) - 0.5d0)
8        sum = sum + f(x)
9    enddo
10  !$OMP end do
11  !$OMP END PARALLEL
```

opi06.f90

SCHEDULE(type, chunk): The iteration(s) in the work sharing construct are assigned to threads according to the scheduling method defined by this clause. The three types of scheduling are:

1. static: Here, all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default.
2. dynamic: Here, some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter chunk defines the number of contiguous iterations that are allocated to a thread at a time.
3. guided: A large chunk of contiguous iterations are allocated to each thread dynamically (as above). The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter chunk.

Outline

Matrix-Vector Multiplication: Basics

$$A = \begin{pmatrix} 1 & 3 & 1 \\ 1 & 1 & 2 \\ 2 & 3 & 4 \end{pmatrix}, x = \begin{pmatrix} 2 \\ 0 \\ -1 \end{pmatrix}, \tag{1}$$

$$b = A \cdot x \tag{2}$$

$$= \begin{pmatrix} 1 & 3 & 1 \\ 1 & 1 & 2 \\ 2 & 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \tag{3}$$

This means

$$\begin{pmatrix} 1 & 3 & 1 \\ 1 & 1 & 2 \\ 2 & 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{2} & 9 & -5 \\ \mathbf{0} & -2 & 1 \\ \mathbf{-1} & -3 & 2 \end{pmatrix} = \begin{pmatrix} \mathbf{1} & 0 & 0 \\ \mathbf{0} & 1 & 0 \\ \mathbf{0} & 0 & 1 \end{pmatrix} \tag{4}$$

```
 1  program MatMul01
 2  implicit none
 3  integer (kind=8) :: i,j
 4  real    (kind=16) :: A(3,3), b(3), x(3), sum
 5  !       Column-wise
 6  data x/2.,0.,-1./
 7  data A/1.,1.,2., 3.,1.,3., 1.,2.,4./
 8  !       Matrix multiplication b= A x
 9  do i = 1,3
10      sum = 0.d0
11      do j = 1,3
12          sum = sum + a(i,j) * x(j)
13      enddo
14      b(i) = sum
15  enddo
16  !       Display A, x, and b=Ax
17  open(10,file='matmul00.dat')
18  do i=1,3
19      write(10,"((5(2x,F12.6)))") (A(i,j),j=1,3), x(i), b(i)
20  enddo
21  close(10)
22  stop
23  end
```

$$A = \begin{pmatrix} 1 & 3 & 1 \\ 1 & 1 & 2 \\ 2 & 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 0 \\ -1 \end{pmatrix}$$

3×3 $\quad$ 3×1

$$b_i = \sum_{j=1}^{3} A_{ij} \cdot x_j$$

```
 1  program MatMul01
 2  implicit none
 3  integer (kind=8) :: i,j
 4  real    (kind=16) :: A(3,3), b(3), x(3)
 5  !       Column-wise
 6  data x/2.,0.,-1./
 7  data A/1.,1.,2., 3.,1.,3., 1.,2.,4./
 8  !       Matrix multiplication b= A x
 9
10  b = matmul(A,x)
11
12  !       Display A, x, and b=Ax
13  open(10,file='matmul01.dat')
14  do i=1,3
15      write(10,"((5(2x,F12.6)))") (A(i,j),j=1,3), x(i), b(i)
16  enddo
17  close(10)
18  stop
19  end
```

Outline

$\mathbf{A} \cdot \mathbf{B} = \mathbf{C} = ?$

B = transpose(A)

$B = A^T$

$$\mathbf{A} \cdot \mathbf{B} = \mathbf{C} = ? \tag{5}$$

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix}, \mathbf{B} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\ 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\ 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \end{pmatrix}$$

where $\mathbf{A}$ and B are $n \times n$ square matrixes where $n = 8$. What is C?

$$C_{ij} = \sum_k A_{ik} B_{kj} = 1^2 + 2^2 + \cdots + 8^2 = 204$$

```
1   program MatMul01
2   implicit none
3   integer (kind=8) :: i,j,k
4   real    (kind=16) :: A(8,8), B(8,8), c(8,8), sum, Csum
5   do i = 1,8
6     do j = 1,8
7       A(i,j) = dble(j)
8     enddo
9   enddo
10  B=transpose(A)
11  do i = 1,8
12    do j = 1,8
13      sum = 0.d0
14      do k = 1, 8
15        sum = sum + A(i,k) * B(k,j)
16      enddo
17      C(i,j) = sum
18    enddo
19  enddo
20  Csum=0.d0
21  do i=1,8
22    do j = 1,8
23      Csum = Csum + C(i,j)
24    enddo
25  enddo
26  write(*,*) Csum
27
28  !     Display C
29  open(10,file='matmul02.dat')
30  do i=1,8
31     write(10,"((8(2x,F8.2)))") (C(i,j),j=1,8)
32  enddo
33  close(10)
34  stop
35  end
```

A: 1 2 3 4 5 6 7 8
, 2 3 4

no matter what i
$A_{ij} = j$

$C = A \cdot B$ $\quad A_{ij} \cdot B_{ij}$

$\Rightarrow A_{ij} \cdot B_{jk}$

$\Rightarrow A_{ik} \cdot B_{kj}$

ΣC_{ij}

204×64

Outline

PBS practice – MPI

- After you login fractal, do the followings:

```
$ mkdir␣samples
$ cd␣samples
$ cp␣-r␣/opt/cee618s13/samples/*␣./
$ cd␣pbs
$ cd␣mpintel
$ make
$ make␣que
$ watch␣qstat
$ ls␣-l
$ cat␣PBS-mypi*
```

PBS practice – MPI (cont'd)

- There are two pbs files "PRLsample16.pbs" and "PRLsample32.pbs".
  1. "PRLsample16.pbs" is for runs using as many as 16 or less processors.
  2. "PRLsample32.pbs" is for runs using as many as 32 or less processors (more than 16).

- A new customized script " mpirun.sh" will be used.
  1. Usage for 8 processors: **mpirun.sh␣-np␣8␣./fpi.x**

PBS practice – OpenMP

- After the MPI practice, do the followings:

  $ cd␣../openmp
  $ make
  $ make␣run

- A new customized script “ omprun.sh" will be used. This script automatically connects compute-1-16 node and run OpenMP program. The number of threads is specified using “-n” option. The default number of threads is 1, if -n option is not used.
  1. Usage: **omprun␣-n␣<thread number>␣./myprogram**
  2. Example: omprun␣-n␣8␣./omp7.x

PBS practice – Serial

- After the OpenMP practice, do the followings:

```
$ cd␣../serial
$ qsub␣SRLsample1.pbs
```

- “SRLsample1.pbs” can be used to submit serial jobs after the command on the last line is modified