CSCE 5160 Parallel Processing

Homework #3 Due Feb. 13, 2019: 4.5, 4.7, 4.12, 4.14, 4.19

Review (Chapter 5) Different Scalability Measures

Iso-Efficiency
Overhead in executing parallel algorithms $T_o(W,p) \to$ depends on $p$ and $W$

Execution time with $p$ processors $T_p = [W + T_o(W,p)] / p$

**Speed up** $S = W/T_p = (W*p)/[W + T_o(W,p)]$

**Efficiency** $E = S/p = 1 / [1 + T_o(W,p)/W] = W/[W+T_o(W,p)]$

$W = (E/(1-E)) * T_o(W,p) = K * T_o(W,p)$

$\to$ note $K = E/(1-E)$ is a constant for a given efficiency
For example if we want to maintain 80% efficiency $K = (0.8)/(0.2) = 4$

If we know the overhead function, we can find $W$ for a given Efficiency $E$ and $p$

---

CSCE 5160 Parallel Processing

Let us examine our previous example of adding $n$ numbers

serial complexity $= n = W$

Execution time with $p$ processors $= (n/p) + 2*p*log(p) = W/p + 2*p*log(p)$

$T_p = (W+2*p*log(p))/p$

What is the overhead part?

$T_o(W,p) = 2*p*log(p)$

Our iso-efficiency function will be $K*2*p*log(p)$
Note to maintain the same efficiency, Work will have to increase.

Other Scalability Measures

**Scaled Speedup** – Increase the Work for $p$ processors with number of processors

Memory Scaling: Increase work based on available memory
Consider applying Linear Scaledup work with Amdahl’s law
work done by p processors = c*p^W (c is some constant)

\[ T_{\text{pscaled}} = (c*pW)*f + (c*pW)*(1-f)/p \]

Scaled-Speedup = \[ W/T_{\text{pscaled}} = 1/[c*p*f + c*(1-f)] \]
If we set c=1 (scaling factor)

Speedup_{scaled} = 1/[1+(p-1)f]  

Speedup_{original} = \[ p/[1+(p-1)*f] \]

Note maximum scaled speedup is 1

<table>
<thead>
<tr>
<th>f</th>
<th>4-original</th>
<th>4-scaled</th>
<th>8-original</th>
<th>8-scaled</th>
<th>16-original</th>
<th>16-scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>3.478</td>
<td>0.8696</td>
<td>5.926</td>
<td>0.7407</td>
<td>9.143</td>
<td>0.571</td>
</tr>
<tr>
<td>0.1</td>
<td>3.077</td>
<td>0.7692</td>
<td>4.706</td>
<td>0.588</td>
<td>6.400</td>
<td>0.4</td>
</tr>
<tr>
<td>0.15</td>
<td>2.759</td>
<td>0.690</td>
<td>3.902</td>
<td>0.488</td>
<td>4.923</td>
<td>0.308</td>
</tr>
</tbody>
</table>

---

Memory Scaled: When we talked about scaling work proportional to p (processors), how do we come up with work.

Consider the following: We will assume that with p processors, the available memory will be p times as large as one processor. Can we scale the work proportional to available memory?

Example: 5.21 on page 223 of Textbook

Consider matrix * vector product \[ A*x = b \]

On a single processor: \[ T_1 = t_c*n^2 \]
where \( t_c \) is the computational time for a single Multiply and Accumulate computation

Assuming an algorithm that uses row striping of A (each processor receives n/p rows)
And each processor receives n/p elements of x (vector)

We need to broadcast x to all processors \( \rightarrow \) all to all communication
message size = n/p
Consider 2D Mesh. Cost of all to all communication in a $p^{1/2} \times p^{1/2}$ mesh is

$$T_{\text{comm}} = 2t_c(p^{1/2} - 1) + t_w \cdot m \cdot (p-1)$$

$$m = \frac{n}{p}$$

$$T_{\text{comm}} = 2t_c(p^{1/2} - 1) + t_w \cdot (n/p) \cdot (p-1)$$

$$= 2t_c(p^{1/2} - 1) + t_w \cdot n$$

Computation cost: Each processor computes $n/p$ results

$$T_{\text{comp}} = t_c \cdot n^2/p$$

Total cost $T_p = t_c \cdot n^2/p + 2t_c(p^{1/2} - 1) + t_w \cdot n$

Setting all times ($t_c, t_w, t_v$) = 1.

Also $W = n^2$

$$T_p = n^2/p + 2(p^{1/2} - 1) + n = W/p + W^{1/2} + 2(p^{1/2} - 1)$$

Speedup $S_{\text{original}} = W/T_p = W/[W/p + W^{1/2} + 2(p^{1/2} - 1)] = W^p/[W + W^{1/2} \cdot p + 2 \cdot p^{3/2} - 2]$

Memory Scaled speedup. Assume memory increases linearly with $p$.

So we will also scale $W$ linearly with $p$.

$$T_p = (c \cdot t_c \cdot W)/p + (c \cdot t_w \cdot W)^{1/2} + 2(\sqrt{p} - 1)$$

$$= (c \cdot t_c \cdot W)/p + (c \cdot t_w \cdot W)^{1/2} + 2(\sqrt{p} - 1)$$

$$T_p = (c \cdot t_c \cdot W)/p + (c \cdot t_w \cdot W)^{1/2} + 2(\sqrt{p} - 1)$$

$$S_{\text{scaled}} = W/T_p = W/[c \cdot t_c \cdot W] + (c \cdot t_w \cdot W)^{1/2} + 2(\sqrt{p} - 1)]$$

If set $c = 1$

$$S_{\text{scaled}} = W/[1 + (\sqrt{p})^{1/2}]$$

Compare this with Speedup $S_{\text{original}} = W^p/[W + W^{1/2} \cdot p + 2 \cdot p^{3/2} - 2]$

No $p$ in the numerator in scaled version.

If we drop smaller terms, $S_{\text{scaled}}$ is closed to 1 $\rightarrow$ perfect scalability!
CSCE 5160 Parallel Processing

We can do the same with Matrix Multiplication

Remember the amount of memory needed is $O(n^2)$ same as Matrix*vector
However the communication cost for matrix multiplication is higher
Still need all-to-all: every processor needs send all its rows to all other processors
  Each processors needs to send $n^2/p$ elements
So
\[ T_{\text{comm}} = 2t_s(p^{1/2} - 1) + t_w m(p - 1) \text{ and } m = n^2/p \]
\[ 2 * (p^{1/2} - 1) + (p - 1) * (n^2/p) \] (setting $t_s$ and $t_w = 1$)
\[ T_{\text{computation}} = 2 * n^3 / p \]
\[ T_{p-\text{original}} = 2 * n^3 / p + 2 * (p^{1/2} - 1) + (p - 1) * (n^2/p) \]

Ignoring some terms that are smaller
\[ T_{p-\text{original}} = [n^3 + (n^2 * p)/2] / p = [W + W^{2/3} * p/2] / p \]
Speedup-original = $p / [1 + W^{2/3} * p/2]$

CSCE5160 February11, 2019

Note $W = n^3$ but memory grows $O(n^2)$. So memory scaled work increases
\[ p^4 W^{2/3} \]
\[ T_{p-\text{scaled}} = [p^4 W^{2/3} + (p^4 W^{2/3}^{2/3} + p/2) / p = W^{2/3} + p^4 W^{4/9} / 2 \]
Speedup-scaled = $W / T_{p-\text{scaled}} = W / [W^{2/3} + p^4 W^{4/9} / 2]$

We can approximate this to $W^{1/3} = n$

Note the speedup does not increase with $p$ but only with original $n$ in work
CSCE 5160 Parallel Processing

Now we are ready to get down to programming using MPI and OpenMP. Let us start with a simple example: dot product ($X^t Y = Z$)

In message passing environment, we will distribute $n/p$ elements of vector $X$ and $Y$ to each node:

- $p_0$ receives $x_0, \ldots, x_{(n/p-1)}$,
- $p_{i+1}$ receives, $x_{(i*n/p)}, x_{(i*n/p)+1}, \ldots, x_{(i*n/p-1)}$

We start at the root or master node which reads input data ($X$ and $Y$) from files. Then it needs send $n/p$ elements of $X$ and $Y$ to each node (including itself).

We use MPI_Scatter for this purpose.

Each node computes dot product with the elements of $X$ and $Y$ assigned

We can use code something like this

```c
float dot_product (float x[], float y[], int n)
{
    int i;
    float sum = 0.0;
    for (i=0; i<n; i++)
        sum = sum + x[i]*y[i];
    return sum;
}
```

Note is processor has a partial sum. We need to add these partial sums

Each processor can send its value to root using MPI_Gather
And root can add the values

Or, we can use MPI_Reduce to combine these two operations.
Now let us get more details about MPI functions

```c
float parallel_version ( float local_x[], local_y[], int my_n)
{
    float local_sum;
    float total_sum = 0.0;
    float dot_product (float x[], float y[], int n);

    local_sum = dot_product (local_x, local_y, my_n);
    MPI_Reduce(&local_sum, &total_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
    return total_sum;
}
```

Key MPI functions

- int MPI_Init (int * argc, char ** argv[]);
- int MPI_Finalize ();
- int MPI_Comm_Size (MPI_Comm comm, int * number_of_processes);
- int MPI_Comm_rank (MPI_Comm comm, int * my_process_id);
- int MPI_Send (void * message_data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator);
- int MPI_Recv (void * message_data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status * status);
- int MPI_Bcast (void * message, int count, MPI_Datatype data_type, int root, MPI_Comm comm);
```
Let us reexamine the MPI_Scatter used in our example

```c
int my_n = n/p;

MPI_Scatter (&x, my_n, MPI_FLOAT, &local_x, my_n, MPI_FLOAT, root, MPI_COMM_WORLD);
```

... /* similarly, scatter Y elements. 

Note all processes execute this call. Root contains the complete vectors X and Y. All processes (including root) will receive n/p elements.

There are other useful functions and software packages for useful functions encountered in scientific computing.
All processes execute the same command
The result (or accumulated data) is only at the root
Count refers to amount sent by each node
Result stays with only root

We can view MPI_Reduce as
for (j=1;….) {
    MPI_Recv();
    perform_accumulation;}

Operations with MPI_Reduce
MPI_MAX, MPI_MIN
MPI_MAXLOC
MPI_MINLOC
MPI_PROD
MPI_LAND (logical AND)
MPI_BAND (Bitwise AND)
MPI_LOR, MPI_BOR
MPI_LXOR, MPI_BXOR

int MPI_Reduce (void * recv_data, /* data source */
              void* accumulated_data, /* result */
              int count, /* per process count */
              MPI_Datatype send_type, /* data type sent */
              MPI_Op operation, /* operation like, sum, min */
              int receiver, /* receiver or root */
              MPI_Comm comm /* communication group */);

Let us now think of doing the same using OpenMP
Remember here we use shared memory
That is X and Y vectors are shared
No need to send out n/p elements of X and Y to each processor

What about result which is the sum of all partial results?
We need to use "locks" so that one processor at a time adds its value to current result
Tutorial on OpenMP & MPI

What is an OpenMP?
OpenMP is an API that can be used with FORTRAN, C and C++ for programming shared address space Machines. It is a standard for directive based parallel programming.

Scope for OpenMP
OpenMP API covers user-directed parallelization.
The programmer has to explicitly specify the action to be performed by the compiler and runtime system in order to execute the program parallel.

Thread: An execution entity with a stack and associated static memory.
OpenMP thread: A thread that is managed by openMP runtime system.

- Compiler Directives (44)
- Runtime Library Routines (35)
- Environment Variables (13)
Tutorial on OpenMP & MPI

OpenMP program structure:

```c
#include<omp.h>
main()
{
//serial segment…..
  // Beginning of parallel section
#pragma omp parallel private(var1) shared(var2)
{
  // parallel segment
}
//Resume serial code
```

Notes:
1) A team of threads are created when thread reaches parallel directive.
2) Master thread has 0 as the thread number. Statements in parallel block are executed in parallel by every thread.
3) All the threads are synchronized and join in the master thread in the end.
4) openMP work as fork/join model. Fork a team of threads and join them in the end
Tutorial on OpenMP & MPI

How to create, compile and run openMP programs?

1) SSH into arch.cse.unt.edu from putty .
2) Log in with ur EUID and given password. You can change password after log in.
3) Using text editor such as vi/vim, pico etc. create an openMP C or C++ files.
   Command: pico your_file_name.c
             vi your_file_name.c
4) If you using vi command save the program by pressing esc and then :wq. pico text editor asks you to save when you exit the program.
5) Use the command gcc –fopenmp omp_hello.c –o hello to compile
6) Open the object file you created ./hello to see the output.
7) You can set how many threads you want, to compile the program paralelly.
   Command: export OMP_NUM_THREADS=4
Tutorial on OpenMP & MPI

Using `vi` text editor to open exiting or new c file.

This figure shows a sample c program using openMP API. It prints `hello world` from different threads.
Tutorial on OpenMP & MPI

Compiling openMP program and creating an object file.

This picture shows the output of the Hello world program using openMP. 8 threads were created here. The output from these threads are shown in the figure. You can specifically mention the number of threads you need using export OMP_NUM_THREADS.
Tutorial on OpenMP & MPI

What is an MPI?
MPI defines a standard library for message passing that can be used to develop portable message passing programs with C or Fortran.

MPI contains over 125 routines.
6 routines are essential.

Routines are used to initialize and terminate the MPI library, to get information about parallel computing environments and to send and receive messages.

- **MPI_Init**
  - Initialize MPI

- **MPI_Finalize**
  - Terminates MPI

- **MPI_Comm_size**
  - Determines the number of processes

- **MPI_Comm_rank**
  - Determine the label of calling processes

- **MPI_send**
  - Sends a message

- **MPI_Recv**
  - Receives a message
Tutorial on OpenMP & MPI

How to create, compile and run MPI programs?

1) SSH into arch.cse.unt.edu from putty.
2) Log in with ur EUID and given password. You can change password after log in.
3) Using text editor such as vi/vim, pico etc. create an MPI C or C++ files.
   Command: pico your_file_name.c
             vi your_file_name.c
4) If you using vi command save the program by pressing esc and then :wq. pico text editor asks you to save when you exit the program.
5) Use the command mpicc –o mpi mpi_example.c to compile
6) Open the object file you created ./mpi to see the output.
7) You can set how many processes you want, to compile the program parallelly.
   Command: mpirun –n 4 ./mpi
This figure shows hello world program With MPI library.

This figure shows the output of hello world Program. The number of processes are 4.