

HPC Essentials

Part III : Message Passing Interface

Bill Brouwer

Research Computing and Cyberinfrastructure
(RCC), PSU

Outline

- Motivation
- Interprocess Communication
 - Signals
 - Sockets & Networks
- `procfs` Digression
- Message Passing Interface
 - Send/Receive
 - Communication
 - Parallel Constructs
 - Grouping Data
 - Communicators & Topologies

Motivation

- We saw last time that Amdahl's law implies an *asymptotic limit* to performance gains from parallelism, where parallel P and serial code ($1-P$) portions have **fixed** relative cost
- We looked at threads (“light-weight processes”) and also saw that performance depends on a variety of things, including good cache utilization and affinity
- For the problem size investigated, ultimately the limiting factor was disk I/O, there was no sense going beyond a single compute node; in a machine with 16 cores or more, there is no point when $P < 60\%$, should the process have sufficient memory
- *However, as we increase our problem size, the relative parallel/serial cost changes and P can approach 1*

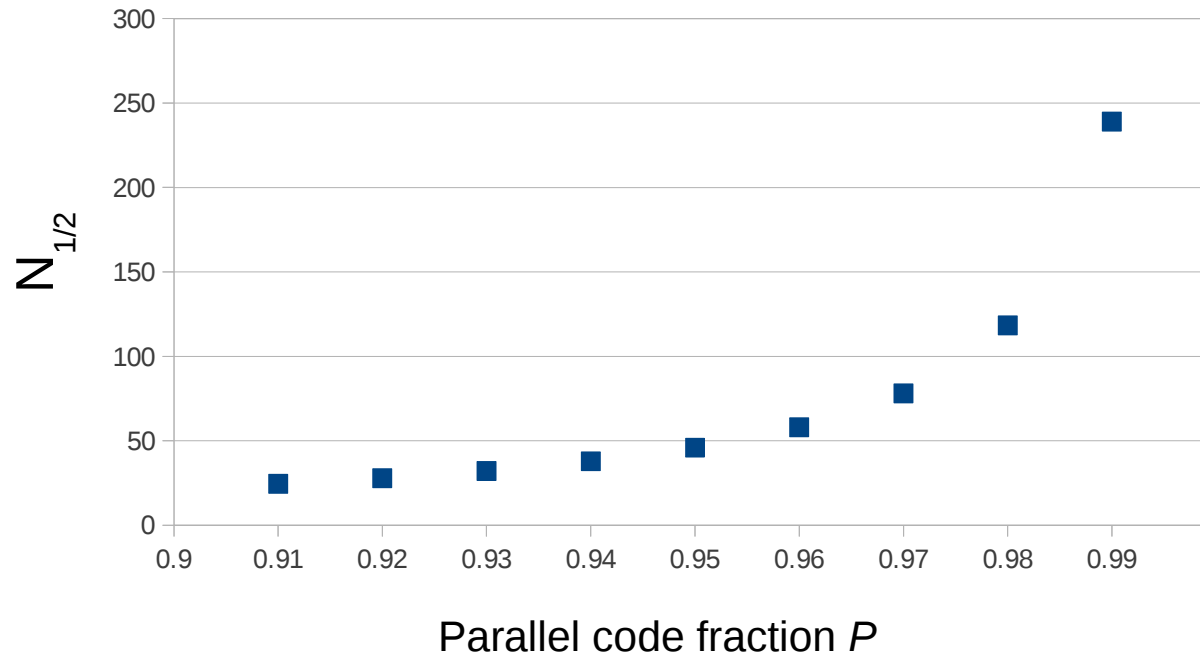
Motivation

• In the limit as processors $N \rightarrow \infty$ we find the maximum performance improvement :

$$1/(1-P)$$

• It is helpful to see the 3dB points for this limit ie., the number of processors $N_{1/2}$ required to achieve $(1/\sqrt{2}) * max = 1/(\sqrt{2} * (1-P))$; equating with Amdahl's law & after some algebra :

$$N_{1/2} = 1/((1-P) * (\sqrt{2}-1))$$



Motivation

- Points to note from the graph :
 - $P \sim 0.90$, we can benefit from ~ 20 cores
 - $P \sim 0.99$, we can benefit from a cluster size of ~ 256 cores
 - $P \rightarrow 1$, we approach the “embarrassingly parallel” limit
 - $P \sim 1$, performance improvement directly proportional to cores
 - $P \sim 1$ implies *independent* or batch processes
- Quite aside from considerations of Amdahl's law, as the problem size grows, we may simply exceed the memory available on a single node
- In this case, must move to a *distributed memory* processing model/multiple nodes (unless $P \sim 1$ of course)
- *How do we determine P ? \rightarrow PROFILING*

Profiling w/ Valgrind

```
[wjb19@lionxf scratch]$ valgrind --tool=callgrind ./psktm.x  
[wjb19@lionxf scratch]$ callgrind_annotate --inclusive=yes callgrind.out.3853
```

```
-----  
Profile data file 'callgrind.out.3853' (creator: callgrind-3.5.0)  
-----
```

```
I1 cache:  
D1 cache:  
L2 cache:  
Timerange: Basic block 0 - 2628034011  
Trigger: Program termination  
Profiled target: ./psktm.x (PID 3853, part 1)
```

Parallelizable worker
function is 99.5% of
total instructions
executed

```
-----  
20,043,133,545 PROGRAM TOTALS  
-----
```

Ir	file:function
20,043,133,545	???:0x0000003128400a70 [/lib64/ld-2.5.so]
20,042,523,959	???:0x0000000000401330 [/gpfs/scratch/wjb19/psktm.x]
20,042,522,144	???:(below main) [/lib64/libc-2.5.so]
20,042,473,687	/gpfs/scratch/wjb19/demoA.c:main
20,042,473,687	demoA.c:main [/gpfs/scratch/wjb19/psktm.x]
19,934,044,644	psktmCPU.c:ktmMigrationCPU [/gpfs/scratch/wjb19/psktm.x]
19,934,044,644	/gpfs/scratch/wjb19/psktmCPU.c:ktmMigrationCPU
6,359,083,826	???:sqrtf [/gpfs/scratch/wjb19/psktm.x]
4,402,442,574	???:sqrtf.L [/gpfs/scratch/wjb19/psktm.x]
104,966,265	demoA.c:fileSizeFourBytes [/gpfs/scratch/wjb19/psktm.x]

If we wish to scale outside a single node, we must use some form of *interprocess communication*

Inter-Process Communication

- There are a variety of ways for processes to exchange information, including:
 - Memory (~last week)
 - Files
 - Pipes (named/anonymous)
 - Signals
 - Sockets
 - Message Passing
- File I/O is too slow, and read/writes liable to race conditions
- Anonymous & named pipes are highly efficient but FIFO (first in, first out) buffers, allowing only unidirectional communication, and between processes on the same node
- Signals are a very limited form of communication, sent to the process after an interrupt by the kernel, and handled using a default handler or one specified using `signal()` system call
- Signals may come from a variety of sources eg., segmentation fault (`SIGSEGV`), keyboard interrupt Ctrl-C (`SIGINT`) etc

Signals

• `strace` is a powerful utility in UNIX which shows the interaction between a running process and kernel in the form of system calls and signals; here, a partial output showing mapping of signals to defaults with system call `sigaction()`, from `./psktm.x` :

```
rt_sigaction(SIGHUP, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigaction(SIGINT, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigaction(SIGQUIT, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigaction(SIGILL, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigaction(SIGABRT, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigaction(SIGFPE, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigaction(SIGBUS, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigaction(SIGSEGV, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigaction(SIGSYS, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigaction(SIGTERM, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigaction(SIGPIPE, NULL, {SIG_DFL, [], 0}, 8) = 0
```

UNIX signals

• Signals are crude and restricted to local communication; to communicate remotely, we can establish a *socket* between processes, and communicate over the network

Sockets & Networks

- Davies/Baran first devised packet switching, an efficient means of communication over a channel; a computer was conceived to realize their design and ARPANET went online Oct 1969 between UCLA and Stanford
- TCP/IP became the communication protocol of ARPANET 1 Jan 1983, which was retired in 1990 and NFSNET established; university networks in the US and Europe join
- TCP/IP is just one of many protocols, which describes the format of data packets, and the nature of the communication; an analogous connection method is used by Infiniband networks in conjunction with Remote Direct Memory Access (RDMA)
- Unreliable Datagram Protocol (UDP) is analogous to a connectionless method of communication used by Infiniband high performance networks

Sockets : UDP host example

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <unistd.h> /* for close() for socket */
#include <stdlib.h>

int main(void)
{
    //creates an endpoint & returns file descriptor
    //uses IPv4 domain, datagram type, UDP transport
    int sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);

    //socket address object (sa) and memory buffer
    struct sockaddr_in sa;
    char buffer[1024];
    ssize_t recsize;
    socklen_t fromlen;

    //specify same domain type, any input address and port 7654 to listen on
    memset(&sa, 0, sizeof sa);
    sa.sin_family = AF_INET;
    sa.sin_addr.s_addr = INADDR_ANY;
    sa.sin_port = htons(7654);
    fromlen = sizeof(sa);
```

Sockets : host example cont.

```
//we bind an address (sa) to the socket using fd sock
if (-1 == bind(sock, (struct sockaddr *)&sa, sizeof(sa)))
{
    perror("error bind failed");
    close(sock);
    exit(EXIT_FAILURE);
}

for (;;)
{
    //listen and dump buffer to stdout where applicable
    printf ("recv test....\n");
    recsize = recvfrom(sock, (void *)buffer, 1024, 0, (struct sockaddr *)&sa, &fromlen);
    if (recsize < 0) {
        fprintf(stderr, "%s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
    printf("recsize: %z\n ", recsize);
    sleep(1);
    printf("datagram: %.*s\n", (int)recsize, buffer);
}
}
```

Sockets : client example

```
int main(int argc, char *argv[])
{
    //create a buffer with character data
    int sock;
    struct sockaddr_in sa;
    int bytes_sent;
    char buffer[200];

    strcpy(buffer, "hello world!");

    //create a socket, same IP and transport as before, address of host 127.0.0.1
    sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (-1 == sock) /* if socket failed to initialize, exit */
    {
        printf("Error Creating Socket");
        exit(EXIT_FAILURE);
    }

    memset(&sa, 0, sizeof sa);
    sa.sin_family = AF_INET;
    sa.sin_addr.s_addr = inet_addr("127.0.0.1");
    sa.sin_port = htons(7654);

    bytes_sent = sendto(sock, buffer, strlen(buffer), 0, (struct sockaddr*)&sa, sizeof sa);
    if (bytes_sent < 0) {
        printf("Error sending packet: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    close(sock); /* close the socket */
    return 0;
}
```

• You can monitor sockets by using the `netstat` facility, which takes its data from `/proc/net`

Outline

- Motivation
- Interprocess Communication
 - Signals
 - Sockets & Networks
- `procfs` Digression
- Message Passing
 - Send/Receive
 - Communication
 - Parallel Constructs
 - Grouping Data
 - Communicators & Topologies

procfs

- We mentioned the `/proc` directory previously in the context of `cpu` and `memory` information, which is frequently referred to as the `proc` filesystem or `procfs`
- *It is a veritable treasure trove of information*, written periodically by the kernel, and is used by a variety of tools eg., `ps`
- Each running process is assigned a directory, whose name is the process id
- Each directory contains text files and subdirectories with every detail of a running process, including context switching statistics, memory management, open file descriptors and much more
- Much like the `ptrace()` system call, `procfs` also gives user applications the ability to *directly manipulate* running processes, given sufficient permission; you can explore that on your own :)

procfs : examples

- Some of the more useful files :

- `/proc/PID/cmdline` : command used to launch process
- `/proc/PID/cwd` : current working directory
- `/proc/PID/environ` : environment variables for the process
- `/proc/PID/fd` : directory w/ symbolic link for each open file descriptor eg., streams
- `/proc/PID/status` : information including signals, state, memory usage
- `/proc/PID/maps` : memory map between virtual and physical addresses

- eg., contents of the fd firectory for running process `./psktm.x` :

```
[wjb19@hammer1 fd]$ ls -lah
```

```
total 0
```

```
dr-x----- 2 wjb19 wjb19  0 Dec  7 12:13 .
dr-xr-xr-x 6 wjb19 wjb19  0 Dec  7 12:10 ..
lrwx----- 1 wjb19 wjb19 64 Dec  7 12:13 0 -> /dev/pts/28
lrwx----- 1 wjb19 wjb19 64 Dec  7 12:13 1 -> /dev/pts/28
lrwx----- 1 wjb19 wjb19 64 Dec  7 12:13 2 -> /dev/pts/28
lrwx----- 1 wjb19 wjb19 64 Dec  7 12:13 3 -> /gpfs/scratch/wjb19/inputDataSmall.bin
lrwx----- 1 wjb19 wjb19 64 Dec  7 12:13 4 -> /gpfs/scratch/wjb19/inputSrcXSmall.bin
lrwx----- 1 wjb19 wjb19 64 Dec  7 12:13 5 -> /gpfs/scratch/wjb19/inputSrcYSmall.bin
lrwx----- 1 wjb19 wjb19 64 Dec  7 12:13 6 -> /gpfs/scratch/wjb19/inputRecXSmall.bin
lrwx----- 1 wjb19 wjb19 64 Dec  7 12:13 7 -> /gpfs/scratch/wjb19/inputRecYSmall.bin
lrwx----- 1 wjb19 wjb19 64 Dec  7 12:13 8 -> /gpfs/scratch/wjb19/velModel.bin
```

procfs : status file extract

```
[wjb19@hammer1 30769]$ more status
```

```
Name:   psktm.x
State:  R (running)
SleepAVG:  0%
Tgid:    30769
Pid: 30769
PPid:   30687
TracerPid:  0
Uid: 2511 2511 2511 2511
Gid: 2530 2530 2530 2530
FDSize: 256
Groups: 2472 2530 3835 4933 5505 5732
VmPeak:   65520 kB
VmSize:   65520 kB
VmLck:    0 kB
VmHWM:   37016 kB
VmRSS:   37016 kB
VmData:  51072 kB
VmStk:    88 kB
VmExe:    64 kB
VmLib:   2944 kB
VmPTE:   164 kB
StaBrk: 1289a000 kB
Brk: 128bb000 kB
StaStk: 7ffffbd0a0300 kB
Threads: 5
SigQ:    0/398335
SigPnd:  0000000000000000
ShdPnd:  0000000000000000
SigBlk:  0000000000000000
SigIgn:  0000000000000000
SigCgt:  0000000180000000
```



Virtual memory usage



signals

Outline

- Motivation
- Interprocess Communication
 - Signals
 - Sockets & Networks
- `procfs` Digression
- **Message Passing Interface**
 - Send/Receive
 - Communication
 - Parallel Constructs
 - Grouping Data
 - Communicators & Topologies

Message Passing Interface (MPI)

- Classical von Neumann machine has single instruction/data stream (SISD) → single process & memory
- Multiple Instruction, multiple data (MIMD) system → connected processes are asynchronous, generally distributed memory (may also be shared where processes on single node)
- MIMD Processors are connected in some network topology; we don't have to worry about the details, MPI abstracts this away
- MPI is a standard for parallel programming first established in 1991, updated occasionally, by academics and industry
- It comprises routines for point-to-point and collective communication, with bindings to C/C++ and fortran
- Depending on underlying network fabric, communication maybe TCP or UDP-like in Infiniband networks

MPI : Basic communication

- Multiple, distributed processes are spawned at initialization, each process assigned a unique *rank* $0, 1, \dots, p-1$

- One may send information referencing process rank eg.,:

```
MPI_Send(&x, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
```

Buffer address

Rank of rcv

- This function has a receive analogue; both routines are *blocking* by default
- Send/receive statements generally occur in same code, processors execute appropriate statement according to rank & code branch
- Non-blocking functions available, allows communicating processes to continue with execution where able

MPI : Requisite functions

- Bare minimum → *initialize, get rank for process, total processes and finalize when done*

```
MPI_Init(&argc, &argv); //Start up
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); //My rank
MPI_Comm_size(MPI_COMM_WORLD, &p); //No. processors
MPI_Finalize(); //close up shop
```

- MPI_COMM_WORLD is a *communicator* parameter, a collection of processes that can send messages to each other.
- Messages are sent with *tags* to identify them, allowing specificity beyond using just a source/destination parameter

MPI : Datatypes

<code>MPI_CHAR</code>	signed char
<code>MPI_SHORT</code>	signed short int
<code>MPI_INT</code>	signed int
<code>MPI_LONG</code>	signed long int
<code>MPI_UNSIGNED_CHAR</code>	unsigned char
<code>MPI_UNSIGNED_SHORT</code>	unsigned short int
<code>MPI_UNSIGNED</code>	unsigned int
<code>MPI_UNSIGNED_LONG</code>	unsigned long int
<code>MPI_FLOAT</code>	float
<code>MPI_DOUBLE</code>	double
<code>MPI_LONG_DOUBLE</code>	long double
<code>MPI_BYTE</code>	
<code>MPI_PACKED</code>	

Minimal MPI example

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
    int rank, size, i;
    int buffer[10];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank > 0)
    {

        for (int i =0; i<10; i++)
            buffer[i]=i * rank;

        MPI_Send(buffer, 10, MPI_INT, 0, 0, MPI_COMM_WORLD);
    } else {

        for (int i=1; i<size; i++){
            MPI_Recv(buffer, 10, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
            printf("buffer element 0 : %i from proc : %i \n",buffer[0],i);
        }
    }
    MPI_Finalize();
    return 0;
}
```

MPI : Collective Communication

- A communication pattern involving all processes in a communicator is a *collective communication* eg., a broadcast
- Same data sent to every process in communicator, more efficient than using multiple p2p routines, optimized :

```
MPI_Bcast(void* message, int count, MPI_Datatype type,  
          int root, MPI_Comm comm)
```

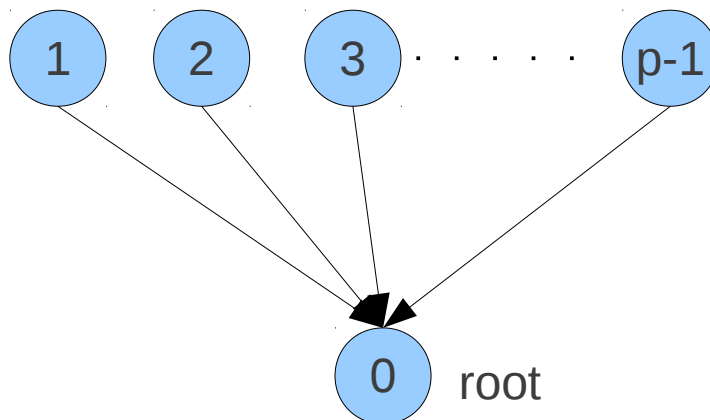
- Sends copy of data in `message` from `root` process to all in `comm`, a scatter/map operation
- Collective communication is at the heart of efficient parallel operations

Parallel Operations : Reduction

- Data maybe gathered/reduced after computation via :

```
MPI_Reduce(void* operand, void* result, int count,  
MPI_Datatype type, MPI_Op operator, int root, MPI_Comm  
comm)
```

- **Combines all operand, using operator and stores result on process root, in result**
- **A tree-structured reduce at all nodes == MPI_Allreduce, ie., every process in comm gets a copy of the result**

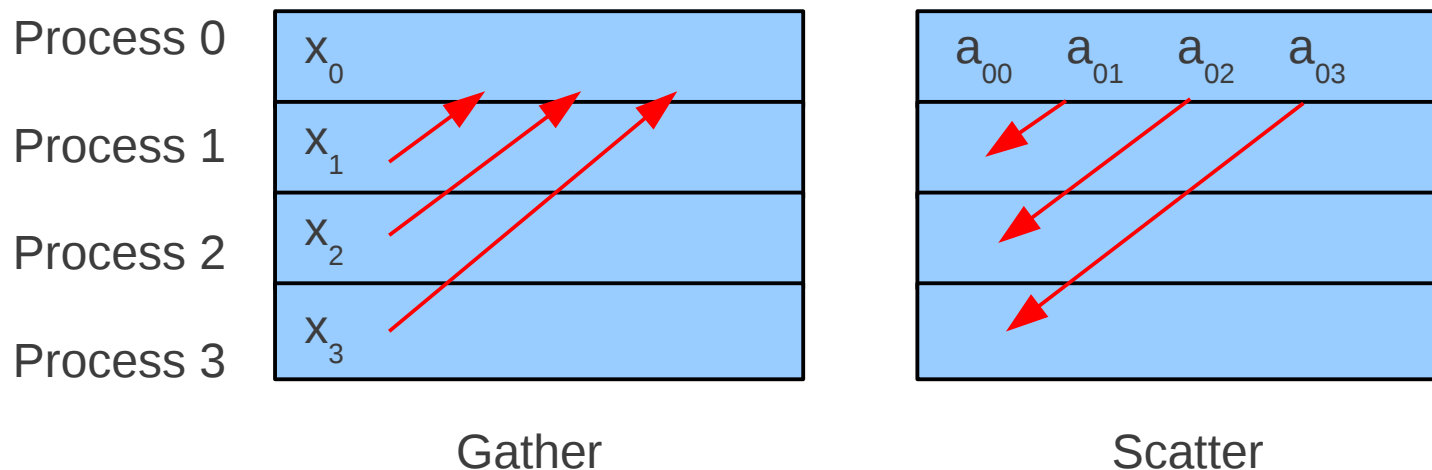


Reduction Ops

<code>MPI_MAX</code>	
<code>MPI_MIN</code>	
<code>MPI_SUM</code>	
<code>MPI_PROD</code>	
<code>MPI_LAND</code>	Logical and
<code>MPI_BAND</code>	Bitwise and
<code>MPI_LOR</code>	Logical or
<code>MPI_BOR</code>	Bitwise or
<code>MPI_LXOR</code>	Logical XOR
<code>MPI_BXOR</code>	Bitwise XOR
<code>MPI_MAXLOC</code>	Max w/ location
<code>MPI_MINLOC</code>	Min w/ location
<code>MPI_PACKED</code>	

Parallel Operations : Scatter/Gather

- Bulk transfers of many-to-one and one-to-many are accomplished by gather and scatter operations respectively
- These operations form the kernel of matrix/vector operations for example; they are useful for distributing and reassembling arrays



Scatter/Gather Syntax

- `MPI_Gather(void* send_data, int send_count, MPI_Datatype send_type, void* recv_data, int recv_count, MPI_Datatype recv_type, int root, MPI_Comm comm)`
- **Collects data referenced by `send_data` from each process in `comm` and stores data in process rank order on process w/ rank `root`, in memory referenced by `recv_data`**
- `MPI_Scatter(void* send_data, int send_count, MPI_Datatype send_type, void* recv_data, int recv_count, MPI_Datatype recv_type, int root, MPI_Comm comm)`
- **Splits data referenced by `send_data` on process w/ rank `root` into segments, `send_count` elements each, w/ `send_type` & distributed in order to processes**
- **For gather result to ALL processes → `MPI_Allgather`**

Grouping Data I

- Communication is expensive → bundle variables into single message
- We must define a derived type than can describe the heterogeneous contents of a message using type and displacement pairs
- Several ways to build this `MPI_Datatype` eg.,

```
MPI_Type_Struct(int count,  
int block_lengths[], //contains no. entries in each block  
MPI_Aint displacements[], //element offset from msg start  
MPI_Datatype typelist[], //exactly that  
MPI_Datatype* new_mpi_t //a pointer to this new type)
```

Allows for addresses > int

- A very general derived type, although arrays to struct must be constructed explicitly using other MPI commands
- Simpler when less heterogeneous eg., `MPI_Type_vector`, `MPI_Type_Contiguous`, `MPI_Type_indexed`

Grouping Data II

- Before these derived types can be used by a communication function, must be committed with `MPI_type_commit` function call
- In order for message to be received, type signatures at send and receive must be *compatible*; if a collective communication, signatures must be identical
- `MPI_Pack` & `MPI_Unpack` are useful for when messages of heterogeneous data are infrequent, and cost of constructing derived type outweighs benefit
- These methods also allow *buffering* in user versus system memory, and the number of items transmitted is in the message itself
- Group data allows for sophisticated objects; *we can also create more fined grained communication objects*

Communicators

- Process subsets or *groups* expand communication beyond simple p2p and broadcast communication, to create :
 - Intra-communicators → communicate among one other and participate in collective communication, composed of :
 - an ordered collection of processes (group)
 - a context
 - Inter-communicators → communicate between different groups
- Communicators/groups are *opaque*, internals not directly accessible; these objects are referenced by a handle

Communicators Cont.

- Internal contents manipulated by methods, much like private data in C++ class objects eg.,
 - `int MPI_Group_incl(MPI_Group old_group, int new_group_size, int ranks_in_old_group[], MPI_Group* new_group)` → **create a new_group from old_group, using ranks_in_old_group[] etc**
 - `int MPI_Comm_create(MPI_Comm old_comm, MPI_Group new_group, MPI_Comm* new_comm)` → **create a new communicator from the old, with context**
- `MPI_Comm_group` and `MPI_Group_incl` are *local* methods without communication, `MPI_Comm_create` is a collective communication implying *synchronization* ie,. to establish single context
- Multiple communicators may be created *simultaneously* using `MPI_Comm_split`

Topologies I

- MPI allows one to associate different addressing schemes to processes within a group
- This is a *virtual* versus real or physical topology, and is either a *graph structure* or a (Cartesian) *grid*; properties:
 - Dimensions, w/
 - Size of each
 - Period of each
 - Option to have processes `reordered` optimally within grid
- Method to establish Cartesian grid `cart_comm`:

```
int MPI_Cart_create(MPI_Comm old_comm, int
number_of_dims, int dim_sizes[], int wrap_around[],
int reorder, MPI_Comm* cart_comm)
```

- `old_comm` is typically just `MPI_COMM_WORLD` created at init

Topologies II

- `cart_comm` will contain the processes from `old_comm` with associated coordinates, available from `MPI_Cart_coords`:

```
int coordinates[2];
int my_grid_rank;
MPI_Comm_rank(cart_comm, &my_grid_rank);
MPI_Cart_Coords(cart_comm,
my_grid_rank, 2, coordinates);
```

- Call to `MPI_Comm_rank` is necessary because of process rank reordering (optimization)
- Processes in `cart_comm` are stored in row major order
- Can also partition in to sub-grid(s) using `MPI_Cart_sub` eg., for row:

```
int free_coords[2];
MPI_Comm row_comm;           //new sub-grid
free_coords[0]=0;           //bool; first coordinate fixed
free_coords[1]=1;           //bool; second coordinate free
MPI_Cart_sub(cart_comm, free_coords, &row_comm);
```

Writing Parallel Code

- Assuming we've profiled our code and decided to parallelize, equipped with MPI routines, we must decide whether to take a :
 - **Domain parallel** (divide tasks, similar data) or
 - **Data parallel** (divide data, similar tasks) approach
- Data parallel in general scales much better, implies lower communication overhead
- Regardless, easiest to begin by selecting or designing data structures, and subsequently their distribution using a constructed topology or scatter/gather routines, for example
- Program in modules, beginning with easiest/essential functions (eg., I/O), relegating 'hard' functionality to stubs initially
- Time code sections, look at targets for optimization & redesign
- Only concern yourself with the highest levels of abstraction germane to your problem, use parallel constructs wherever possible

A Note on the OSI Model

- We've been playing fast and loose with a variety of communication entities; sockets, networks, protocols like UDP, TCP etc
- The Open Systems Interconnection model separates these entities into 7 layers of abstraction, each layer providing *services* to the layer immediately above
- Data becomes increasingly fine grained going down from layer 7 to 1
- As application developers and/or scientists, we need only be concerned with layers 4 and above

Layer	Granularity	Function	Example
7.Application	data	process accessing network	MPI
6.Presentation	data	encrypt/decrypt, data conversion	MPI
5.Session	data	management	MPI
4.Transport	segments	reliability & flow control	IB verbs
3.Network	packets	path	Infiniband
2.Data Link	frames	addressing	Infiniband
1.Physical	bits	signals/electrical	Infiniband

Conclusions

- We can determine the parallel portion of our code through profiling; as a rule of thumb a code with $P \sim 99\%$ can effectively utilize about 256 cores, code with $P \sim 90\%$ about 20 cores
- When the parallel portion of code approaches 90%, we can justify going outside the multi-core node and using some form of inter-process communication (IPC)
- IPC comes in a variety of forms eg., sockets connected over networks, signals between processes on a single machine
- The message passing interface (MPI) abstracts away details of IPC used over networks, providing language bindings to C, fortran etc
- MPI has a number of highly optimized collective communication and parallel constructs, sophisticated means of grouping objects, as well as computational topologies
- The OSI Model assigns various communication entities to one of seven layers, we need only be concerned with layer four and above

References

- Pacheco's excellent MPI text
 - <http://www.cs.usfca.edu/~peter/ppmpi/>
- Valgrind (no really, buy the manual)
 - <http://valgrind.org/>
- UNIX signals
 - <http://www.cs.pitt.edu/~alanjawi/cs449/code/shell/UnixSignals.htm>
- OpenMPI
 - <http://open-mpi.org/>
- procfs
 - <http://www.kernel.org/doc/man-pages/online/pages/man5/proc.5.html>
- Excellent article on ptrace
 - <http://linuxgazette.net/81/sandeep.html>
- Kernel vulnerabilities associated with ptrace/procfs
 - <http://www.kb.cert.org/vuls/id/176888>
- MPI tutorials
 - <http://www.mcs.anl.gov/research/projects/mpi/learning.html>
- Linux Gazette articles
 - <http://linuxgazette.net>
- Open Systems Interconnection
 - http://en.wikipedia.org/wiki/OSI_Reference_Model
- PBS reference
 - http://rcc.its.psu.edu/user_guides/system_utilities/pbs/

Exercises

- Build the supplied MPI code via `'make -f Makefile_'` and submit to cluster of your choice using the following PBS script
- Compare scaling with OpenMP example from last week, by varying both nodes and procs per node (ppn); differences? (NUMA vs good locality w/ MPI)
- Sketch how the gather function is collecting data, and the root process subsequently writes out to disk
- Similarly, sketch how the image data exists in memory; are the two pictures commensurate? (hint: no :))
- Re-assign image grid tiles to processes such that no file manipulation is required after program completion

Scheduling on clusters : PBS

- Basic submission script `foo.pbs`:

```
#PBS -l nodes=4:ppn=4
#PBS -l mem=1Gb
#PBS -l walltime=00:20:00
#PBS -o /gpfs/home/wjb19/scratch/ktm_stdout.txt
#PBS -e /gpfs/home/wjb19/scratch/ktm_stderr.txt
#PBS -V
```

```
cd /gpfs/home/wjb19/scratch
module load openmpi/gnu/1.4.2
```

```
mpirun ./psktm.x
```

- Submit to cluster :

```
[wjb19@lionxf scratch]$ qsub foo.pbs
```

- Check status:

```
[wjb19@lionxf scratch]$ qstat -u wjb19
```

- List nodes for running jobs :

```
[wjb19@lionxf scratch]$ qstat -n
```

Debug MPI applications

- MPI programs are readily debugged using serial programs like `gdb`, once you have :
 - compiled with `-g` & submitted your job
 - an assigned node from `qstat` query
 - process id on that node ie., ssh to node and
 - run `gdb pid==pid_for_the_proc`
- OpenMPI.org give a useful code block to use in conjunction with this technique :

```
{  
    int i = 0;  
    char hostname[256];  
    gethostname(hostname, sizeof(hostname));  
    printf("PID %d on %s ready for attach\n", getpid(), hostname);  
    fflush(stdout);  
    while (0 == i)  
        sleep(5);  
}
```

- Once attached and working with `gdb`, you can set some breakpoints and alter the parameter `i` (eg., set `var i=7`) to move out of the loop