Topics

- Introduction
- Towards a Scalable Execution Model
- Communicating Sequential Processes
- Performance Issues
- Distributed Programming with Unix
- Summary – Material for the Test
Topics

- Introduction
- Towards a Scalable Execution Model
- Communicating Sequential Processes
- Performance Issues
- Distributed Programming with Unix
- Summary – Material for the Test
Opening Remarks

• This week is about scalable application execution
  – Shared memory systems not scalable
  – Job stream parallelism does not accelerate single application

• A path to harnessing distributed memory computers
  – Dominant form of HPC systems

• Discuss the 3rd paradigm for parallel programming
  – Throughput computing (Segment 1)
  – Multithreaded shared memory (Segment 2)

• Dominant strategy
  – Arena of technical computing

• Embodiment of Cooperative Computing
  – Single application
  – Weak scaling
Topics

- Introduction
- Towards a Scalable Execution Model
- Communicating Sequential Processes
- Performance Issues
- Distributed Programming with Unix
- Summary – Material for the Test
Driving forces

• Technology
  – VLSI
    • Killer micros
    • High density DRAM
  – Emerging network

• Architecture
  – MPP
  – clusters

• Weak Scaling
  – Need for larger problems
  – Data parallelism
Scalability

• Strict scaling limits sustained performance
  – Fixed size problem to achieve reduced execution time with increased computing resources
  – Amdahl’s law
    • Sequential component limits speedup
  – Overhead imposes limits to granularity
    • therefore parallelism and speedup

• Weak scaling allows computation size to grow with data set size
  – Larger data sets create more concurrent processes
  – Concurrent processes approximately same size granularity
  – Performance increases with problem set size

• Big systems are big memories for big applications
  – Aggregates memories of many processing nodes
  – Allows problems far larger than a single processor could manage
Impact of VLSI

• Mass produced microprocessor enabled low cost computing
  – PCs and workstations
    • Economy of scale
  – Ensembles of multiple processors
    • Microprocessor becomes building block of parallel computers

• Favors sequential process oriented computing
  – Natural hardware supported execution model
  – Requires locality management
    • Data
    • Control
  – I/O channels (south bridge) provides external interface
    • Coarse grained communication packets

• Suggests concurrent execution at the process boundary level
  – Processes statically assigned to processors (one on one)
    • Operate on local data
  – Coordination by large value-oriented I/O messages
    • Inter process/processor synchronization and remote data exchange
Cooperative computing

• Between Capacity and Capability computing
  – Not a widely used term
  – But and important distinction wrt these others

• Single application
  – Partitioning of data into quasi independent blocks
  – Semi independent processes operate on separate data blocks
  – Limited communication of messages
    • Coordinate through remote synchronization
    • Cooperate through the exchange of some data

• Scaling
  – Primarily weak scaling
  – Limited strict scaling

• Programming
  – Favors SPMD style
  – Static scheduling mostly by hand
  – Load balancing by hand
  – Coarse grain
    • Process
    • Data
    • Communication
Heat equation:
\[
\frac{\partial u}{\partial t} = k \nabla^2 u
\]

In 2-D:
\[
u_t = k(u_{xx} + u_{yy})
\]

Implementation:
- Jacobi method on a unit square
- Neumann boundary condition
- Equal number of intervals along x and y axis
An Example Problem

• Partial Differential Equation (PDE)
  – Heat equation
  – 2-dimensions discrete point in a mesh to approximate a unit square
  – Static boundary conditions

• Stages of Code Development
  – Data decomposition
  – Concurrent sequential processes
  – Coordination through synchronization
  – Data exchange
Data Decomposition

- Partitioning the global data into major contiguous blocks
- Exploits spatial locality that assumes the use of a data element heightens the likelihood of nearby data being used as well
- Exploits temporal locality that assumes the use of a data element heightens the likelihood that the same data will be reused again in the near future
- Varies in form
  - in dimensionality
  - Granularity (size)
  - Shape of partitions
- Static mapping of partitions onto processor nodes
Distributed Concurrent Processes

• Each data block can be processed at the same time
  – Parallelism is determined by number of processes
  – More blocks with smaller partitions permit more processes
  – But …

• Processes run on separate processors on local data
  – Usually one application process per processor
  – Usually SPMD i.e., processes are equivalent but separate
    • Same code, different environments

• Execution of inner data elements of the partition block are done independently for each of the processes
  – Provides coarse grain parallelism
  – Outer loop iterates over successive application steps over the same local data
Data Exchange

• In shared memory, no problem, all the data is there
• For distributed memory systems, data needs to be exchanged between processes
• Ghost cells used to hold local copies of edges of remote partition data at remote processor sites
• Communication packets are medium to coarse grain and point to point for most data transfers
  – e.g., all edge cells of one data partition may be sent to corresponding ghost cells of the neighboring processor in a single message
• Multi-cast or broadcast may be required for some application algorithms and data partitions
  – e.g., matrix-vector multiply
Synchronize

- Global barriers
  - Coarse grain (in time) control of outer-loop steps
  - Usually used to coordinate transition from computation phase to communication phase

- Send/receive
  - Fine grain (in time) control of inner-loop data exchanges
  - Blocks on a send and receive
  - Computation at sender proceeds when data has been received
  - Computation at receiver proceeds when incoming data is available
  - Non-blocking versions of each exist but can lead to race conditions
Performance issues

- Parallelism speeds things up
- Data exchange slows things down
- Finer grain partitioning
  - provides more parallelism
    - Can use more processors
  - requires more fine grain messages
    - Overhead becomes more significant per datum
  - fewer operations per message
    - Overhead of communication becomes more significant per operation
- Synchronization is another source of overhead
- Computation and communication not overlapped
Topics

- Introduction
- Towards a Scalable Execution Model
- **Communicating Sequential Processes**
- Performance Issues
- Distributed Programming with Unix
- Summary – Material for the Test
Communicating Sequential Processes

- A model of parallel computing
  - Developed in the 1970s
  - Often attributed to Tony Hoare
  - Satisfies criteria for cooperative computing
    - Many would claim it as a means of capability computing
- Process Oriented
- Emphasizes data locality
- Message passing semantics
- Synchronization using barriers among other
- Distributed reduction operators added for purposes of optimization
CSP Processes

- Process is the body of state and work
- Process is the module of work distribution
- Processes are static
  - In space: assigned to a single processor
  - In time: exist for the lifetime of the job
- All data is either local to the process or acquired through incident messages
- Possible to extend process beyond sequential to encompass multiple threaded processes
  - Hybrid model integrates the two models together in a clumsy programming methodology
Locality of state

- Processes operate on memory within the processor
- Granularity of process iteration dependent on the amount of process data stored on processor node
- New data from beyond local processor node acquired through message passing, primarily by send/receive semantic constructs
Other key functionalities

• Synchronization
  – Barriers
  – messaging

• Reduction
  – Mix of local and global

• Load balancing
  – Static, user defined
Topics

• Introduction
• Towards a Scalable Execution Model
• Communicating Sequential Processes
• Performance Issues
• Distributed Programming with Unix
• Summary – Material for the Test
Parallelism: Operating System level

- Program (review): A program is a set of instructions usually stored in the memory. During execution a computer fetches the instruction stored in the memory address indicated by the program counter and executes the instruction.

- Process (review): Can be defined as a combination of program, memory address space associated with the program and a program counter.

- A program associated with one process cannot access the memory address space of another program.

- A multi-threaded process is one where a single memory address space is associated with multiple program counters.

- In this lecture we limit the discussion to single-threaded processes for the sake of simplicity.

Adapted from Ch. 7 Beowulf and Cluster Computing. Gropp, Lusk, Sterling
Unix processes: Overview

- New processes can be created using the `fork()-exec()` combination.
- `fork()`
  - efficient lightweight mechanism that copies the address space and creates a process with the same program. The process that invoked the `fork()` call is known as the parent process and the newly created process is called the child process.
  - For the child the `fork()` call returns a 0 whereas for the parent `fork()` call returns the process ID (PID).
- The child process then invokes the `exec()` system call.
- `exec()`
  - changes the program associated with the process
  - sets the program counter to the beginning of the program.
  - reinitializes the address space

Adapted from Ch. 7 Beowulf and Cluster Computing. Gropp, Lusk, Sterling
Image cropped from: http://sc.tamu.edu/
Parallelism using Unix utilities

- Usually a shell process waits for the child process to finish execution before prompting you to execute another command.
- By appending a new process invocation with the “&” character, the shell starts the new process but then immediately prompts for another command, this is called running a process in the “background”.
- This is the simplest form of master-worker model executed using basic unix utilities.

```bash
#!/bin/bash
export search_string=$1
echo searching for $search_string
for i in 20* 
do ( cd $i; grep $search_string * >> $search_string.out & 
) 
done
wait
cat 20*/$search_string.out > $1.all
```

Adapted from Ch. 7 Beowulf and Cluster Computing. Gropp, Lusk, Sterling
Remote Processes

• To create a new process on another machine, the initiator must contact an existing process and cause it to fork a new process. The contact is usually made over a TCP socket.

• rsh (remote shell):
  – rsh command contacts the rshd process running on the remote machine and prompts it to execute a script/program.
  – The standard I/O for the remote machine are routed through rsh to the local machine’s standard I/O.
  – Due to severe security problems (plain text password handling) utilities like rsh and telnet are strongly discouraged and deprecated in many systems.
  – eg : rsh celeritas.cct.lsu.edu /bin/date

• ssh (Secure shell):
  – behaves much like rsh, but the authentication mechanism is based on public key encryption and encrypts all traffic between the local and remote machines.
  – since rsh does not have the encryption stage, rsh is substantially faster than ssh.
  – eg : ssh celeritas.cct.lsu.edu /bin/date
Sockets: Overview

- **socket**: a bidirectional communication channel between two processes that is accessed by the processes using the same `read` and `write` functions that are used for file I/O.

- **Connection Process**:
  - The initial connection process that two remote processes perform in order to establish a bidirectional communication channel is *asymmetric*.
    - The remote machine listens for a connection and accepts it.
    - One process initiates request for connection with the remote machine.
    - A bidirectional channel between the two machines is established.
  - Once a channel is established the communication between the two processes is *symmetric*.
  - In a client-server model, the process that waits for a connection is known as the server and the process that connects to it is known as the client.
TCP/IP: Overview

• Common Terms:
  – IP: Internet Protocol for communication between computers, and is responsible for routing IP packets to their destination.
  – TCP: Transmission Control Protocol for communication between applications
  – UDP: User Datagram Protocol for communication between applications
  – ICMP: Internet Control Message Protocol for detecting errors and network statistics

• TCP:
  – An application that wants to communicate with another application sends a request for connection.
  – The request is sent to a fully qualified address (more on this soon), and port.
  – After the “handshake” (SYN-ACK-SYN) between the two, a bidirectional communication channel is established between the two.
  – The communication channel remains alive until it is terminated by one of the applications involved.

• TCP/IP:
  – TCP breaks down the data to be communicated between applications into packets and assembles the data from packets when they reach the destination.
  – IP ensures routing of the data packets to their intended receiver.
TCP/IP : Overview

- Each computer on a network is associated with an IP address containing 4 numbers each holding a value between 0-255 eg: **130.184.6.128**
- Using Domain Name System (DNS) servers the numeric IP address is mapped to a domain name that is easier to remember, eg the Domain Name corresponding to **130.184.6.128** is **prospero.uark.edu**
- Analogy : Making a phone call
  - Caller – client
  - Receiver – server
  - Phone Number – IP address
  - Extension – Port number

Client :
- Picking up the receiver – socket ()
- locating the call recipient (from phone book / memory) – bind()
- Dialing the phone number – connect()
- Talking – read() / write()
- Hanging up – close()

Server :
- Connecting phone to the phone line – socket ()
- selecting an incoming line – bind ()
- Ringer ON – listen()
- Receiving the call – accept()
- Talking – read() / write()
- Hanging up – close()
Server: Create Socket

/* Create data structures to store connection specific info */
struct sockaddr_in sin, from;

/* The main call that creates a socket */
listen_socket = socket(AF_INET, SOCK_STREAM, 0);

Server
- Create a TCP socket
- Bind socket-port
- Listen for connections
- Loop:
  - accept connection
  - communicate
  - close connection

Client
- Create a TCP socket
- Connect to server
- Communicate
- Close connection
Server : Bind Socket-Port

/* Initializing data structures */
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
/* 0 - Allowing the system to do the selection of port to bind. 
This is user-configurable */
sin.sin_port = htons(0);

bind(listen_socket, (struct sockaddr *) &sin, sizeof(sin));

Server
- Create a TCP socket
- Bind socket-port
- Listen for connections
- Loop:
  - accept connection
  - communicate
  - close connection

Client
- Create a TCP socket
- Connect to server
- Communicate
- Close connection
listen(listen_socket, 5)

/*5 refers to the number of connection requests that the kernel should maintain for the application */

getsockname(listen_socket, (struct sockaddr *) &sin, &len);
print ("listening on port = %d\n", ntohs(sin.sin_port));

Server

- Create a TCP socket
- Bind socket-port
- Listen for connections
- Loop:
  - accept connection
  - communicate
  - close connection

Client

- Create a TCP socket
- Connect to server
- Communicate
- Close connection
Server: `accept()`

```c
#define ACCEPT

://Server::accept()

```

Server:

- Create a TCP socket
- Bind socket-port
- Listen for connections
- Loop:
  - accept connection
  - communicate
  - close connection

Client:

- Create a TCP socket
- Connect to server
- Communicate
- Close connection

```

```c
//talk_socket = accept(listen_socket, (struct sockaddr *) &from, &len);
/*accept() a blocking system call that waits until connection
from a client and then returns a new socket(talk_socket)using
which the server is connected to the client, so that it can
continue listening for more connections on the original server
socket (listen_socket).
*/
```
Client: Create Socket

```c
/* Create data structures to store connection specific info */
struct sockaddr_in sin;
struct hostent *hp;

/* The main call that creates a socket */
talk_socket = socket(AF_INET, SOCK_STREAM, 0);
```

Server
- Create a TCP socket
- Bind socket-port
- Listen for connections
- Loop:
  - accept connection
  - communicate
  - close connection

Client
- Create a TCP socket
- Connect to server
- Communicate
- Close connection
/* initialize data structures*/
hp = gethostbyname(HOST_NAME)
bzero((void *)&sin, sizeof(sin));
bcopy((void *) hp->h_addr, (void *) &sin.sin_addr, hp->h_length);
sin.sin_family = hp->h_addrtype;
sin.sin_port = htons(atoi(PORT_NUM));

/* connect to the server */
connect(talk_socket,(struct sockaddr *) &sin, sizeof(sin));
Client : send msg. write()

n = write(talk_socket, buf, strlen(buf)+1);
if (n < 0)
    error("ERROR writing to socket");
bzero(buf,256);

/*Client initiates communication with server using a write() call*/

Server
- Create a TCP socket
- Bind socket-port
- Listen for connections
- Loop:
  - accept connection
  - communicate
  - close connection

Client
- Create a TCP socket
- Connect to server
- Communicate
- Close connection
Server recv/send read()/write()

Server
- Create a TCP socket
- Bind socket-port
- Listen for connections
- Loop:
  - accept connection
  - communicate
  - close connection

Client
- Create a TCP socket
- Connect to server
- Communicate
- Close connection

n = read(talk_socket, buf, 1024);
if (n < 0)
    error("ERROR reading from socket");
else
    write(talk_socket, buf, n);
/* simple echo; content stored in buf*/
Client: `recv`: `read()`

```
n = read(talk_socket, buf, 1024);
if (n < 0)
    error("ERROR reading from socket");
else
    printf("received from server: %s \n", buf);
/*receives messages sent by the server stored in buf*/
```

Server

- Create a TCP socket
- Bind socket-port
- Listen for connections
- Loop:
  - accept connection
  - communicate
  - close connection

Client

- Create a TCP socket
- Connect to server
- Communicate
- Close connection
Close Socket

close(talk_socket)

/* Ends the socket connection corresponding to one particular client. The control goes back to the loop and server continues to wait for more client connections at listen_socket */

Server

− Create a TCP socket
− Bind socket-port
− Listen for connections
− Loop:
  • accept connection
  • communicate
  • close connection

Client

− Create a TCP socket
− Connect to server
− Communicate
− Close connection

close(talk_socket)

/* Ends the client socket connection */
Demo: CSP Example
Socket Programming: Problems

- Limited portability (not all interconnect interfaces support sockets)
- Limited scalability (number of ports available on a node)
- Tedious and error-prone hand-coding (unless somebody did it before)
- Tricky startup process (assumed port availability is not guaranteed)
- Only point-to-point communication supported explicitly; no implementation of collective communication patterns
- Frequently used communication topologies not available (e.g., Cartesian mesh), have to be coded from scratch
- Direct support only for data organized in continuous buffers, forcing writing own buffer packing/unpacking routines
- Suffer from the overhead of protocol stack (TCP), or require designing algorithms to manage reliable, in-order, free of duplicates arrival of complete messages (e.g. datagram-oriented)
- Basic data transfer calls (read/write) do not guarantee returning or sending the full requested number of bytes, requiring the use of wrappers (and possibly resulting in multiple kernel calls per message)
- Complicated writing of applications with changing/unpredictable communications (it's only easy when reads are matched to writes and you know when both of them occur)
- On some OS’s sockets may linger long after application exits, preventing new startups using the same configuration
- If used, asynchronous management of socket calls adds another layer of complexity (either through select() or multiple threads)
Topics

- Introduction
- Towards a Scalable Execution Model
- Communicating Sequential Processes
- Performance Issues
- Distributed Programming with Unix
- Summary – Material for the Test
Summary – Material for the Test

• Basics: slides 6 – 9, 16
• CSP: slides 19
• Unix: slides 24, 28 - 30
The LSU *Code of Student Conduct* defines plagiarism in Section 5.1.16:

- "Plagiarism is defined as the unacknowledged inclusion of someone else's words, structure, ideas, or data. When a student submits work as his/her own that includes the words, structure, ideas, or data of others, the source of this information must be acknowledged through complete, accurate, and specific references, and, if verbatim statements are included, through quotation marks as well. Failure to identify any source (including interviews, surveys, etc.), published in any medium (including on the internet) or unpublished, from which words, structure, ideas, or data have been taken, constitutes plagiarism;"

Plagiarism will not be tolerated and will be dealt with in accordance with and as outlined by the LSU Code of Student Conduct: