CP331: Introduction to Parallel Computing

by Scott King

Dr. Ilias Kotsireas

James Reinders

Note: Processes on SHARCNET run 0...n.

Note: *procs* means processors.

Oh look...our first MPI program.

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

int main(int argc, char ** argv) {
  int num_procs;
  int ID;

  if (MPI_init(&argc, &argv) != MPI_SUCCESS) {
    printf("error");
  }

  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &ID);
  printf("hello world from process %d of %d \n", ID, num_procs);

  MPI_Finalize();
}
```

- The *Fair Share* algorithm is used in determining the queue for executing C code
- Command to run: `mpicc {name}.c -o {executable_name} -lmpi`
- Command to submit to queue: `sqsub -r 3m -q mpi -n 15 -o ofile.%J ./parallel_hello_world`
    - `sqsub`: Submit to queue
    - `-r 3m`: Time limit
    - `-q mpi`: Submit to MPI-specific queue
    - `-n 15`: How many cpu's we want
    - `-o ...`: The executable we compiled that we want to run

1 Overview of Parallel Computing

Parallel computing utilizes a parallel computer to reduce the time needed to solve a computational problem.

Applications:

- Galactic dynamics

- Climate modeling
- Molecular dynamics
- Protein folding (bioinformatics)
- Data mining
- Oil exploration
- Web search engines
- Medical imaging
- Financial modeling
- Graphics

Note: The first two applications involve numerically solving systems of PDE's.

1.1 Why do we use HPC?

1. To save time and money
2. Solve large scale problems (Computational Fluid Dynamics, CFD)
3. Provide concurrency
4. Distributed resources (SETI)
5. Limits to serial computing

1.2 Motivation for using HPC

1. Speed - large scale computation
    - Explicit numerical scheme - ie. finite differences
    - Discretize temporal and spatial variables, n points + boundary conditions
    - Typically, $O\left(10^{15}\right)$ flops on a 1 GFlop computer $\left(10^{9}\right) = 10^{6}$ secs = 280h
2. Memory
    - How much data can we get in 1GB of RAM?
    - 1GB can hold 512^3 double precision pairs
    - OR two double precision arrays of length $(n+1)^3$ each

2 Parallel Computers

Definition 1. *A computing system that allows multiple processors concurrently, in parallel, to solve a computational problem.*

2.1 Categories of Parallel Computers

2.1.1 Shared-Memory Systems

- The processors have no private memory, they access a single global memory
- Ex. SMP (symmetric multi-procs); memory access time is uniform across all procs
- Ex. NUMA (non-uniform memory access); allows us to incorporate more procs

{insert dia diagram}

2.1.2 Distributed-Memory Systems

- Each proc has a private local memory that is inaccessible by other procs
- The procs are interconnected
    1. Ethernet
    2. Dedicated high throughput, low-latency network

{insert dia diagram}

3 Parallel Computing

Parallel computing is meaningful when, a computational problem can be divided **evenly** among procs which will work in parallel, and will be coordinated in the form of information exchange and synchronization.

Information exchange is to provide private data of one proc to another proc.

Synchronization is keeping the procs at the same place when needed.

Both forms of coordination require communication among procs.

3.1 When is parallel computing beneficial?

1. All procs have sufficient workload
2. The extra overhead caused by parallelization is negligible

3.2 Work Division

Parallel computers are mainly used for task and data parallelism.

3.2.1 Task Parallelism

- Set of standalone computational tasks that exhibit clear distinction between each other
- Some of these tasks may need to follow the completion of other tasks

Example 2. Parameter analysis: $f^{(3)}(x) = a \times x\, f^{(2)}(x)$ where a is the parameter.

- Choose a range of values for a, the solve ODE numerically and study the solutions

Remark 3. All solving processes for distinct values of a are independent. Each of them can be assigned to a procs.

Math reasons. The behaviour of a solution for a^2, can be deduced and also depends on the behaviour of solutions for a. Setup a dependency graph. Parallel execution starts with the tasks that do not depend on other tasks and incrementally include more and more tasks that will become available for execution. When there are more tasks than procs, setup a dynamically allocated queue.

3.3 Data Parallelism

- When the number of tasks is small and the number of procs is large, then multiple processes have to share the work of one task → further work division
- Perform operations to different sections of the same data structure

Example 4. Evaluate a function, $f(x)$, for a set of x-values (1d array)

```
for (i = 0; i < n; i++) {
  y[i] = f(x[i]);
}
```

Assumptions

- All function evaluations are equally expensive
- All procs are equally powerful
- Each proc is assigned a subset of the same number of x-values

We have:

- P: number of procs

Then:

$$n\,p = \left\lfloor \frac{n}{p} \right\rfloor + \begin{cases} 1\text{: } p < \mathrm{mod}\,(n,p) \\ 0\text{: otherwise} \end{cases}$$

where p is the process ID=0...$P-1$.

Property. The max difference between $n\,p$, for $p = 0...P-1$ is 1.

Example 5. Numerical case study

$P = 6$, $n = 63$

$$\left\lfloor \frac{n}{p} \right\rfloor = 10, \mathrm{mod}\,(n,p) = 3$$

$$n_0 = n_1 = n_2 = 11$$

$$n_3 = n_4 = n_5 = 10$$

Note: We know the number of x-values per proc.

The start position for proc, p, is:

$$\begin{aligned} i_{\mathrm{start},p} &= p \times \left\lfloor \frac{n}{p} \right\rfloor + \min\,(p, \mathrm{mod}\,(n,p)) \\ i_{\mathrm{start},0} &= 0 + \min\,(0,) = 0 \\ i_{\mathrm{start},1} &= 1 \times 10 + \min\,(1,3) = 11 \\ i_{\mathrm{start},p} &= 2 \times 10 + \min\,(2,3) = 22 \\ &\vdots \end{aligned}$$

Work Division. n_p, $i_{\text{start},p}$, ...

```
for (i = i_start_p; i < i_start_p + n_p; i++) {
    y[i] = f(x[i]);
}
```

Distributed Memory. Each proc has two local arrays, x_p, y_p

```
for (i = 0; i < n_p; i++) {
    y_p[i] = f(x_p[i]);
}
```

We will need to know which x-values will be assigned to which proc. Assing to each proc a contiguous piece of memory (performance friendly). Index set {0,1,..,n-1} is divided in P pieces using n0 + n1 + ... np-1 = n

Example 6. Composite trapezoidal rule for numerical integration

$$\int_b^a f(x)\mathrm{dx} = h\left(\frac{1}{2}(f(a)+f(b)) + \sum_{i=1}^{n-1} f(a+i\,h)\right)$$

where $h = \frac{b-a}{n}$.

The two endpoints: $\frac{1}{2}$

$n-1$ inner points: 1

What we need to do is divide $n-1$ function evaluations at inner points among P procs, with each proc carrying out a partial summation.

$$S_p = \sum^{i_{\text{start},p}+n_p-1} f(x_i)$$

with $i = i_{\text{start},p}$ and $x_i = a + i\,h$, where $i = 1...n-1$. Also, $n_p = \lfloor \frac{n-1}{P} \rfloor + \begin{cases} 1, p < \mathrm{mod}\,(n-1,p) \\ 0, \text{otherwise} \end{cases}$

Ex. 2 $n = 200$, $P = 3$

$$\begin{aligned} n_0 &= 67 \quad & i_{\text{start},0} &= 1 \\ n_1 &= 66 \quad & i_{\text{start},1} &= 68 \\ n_2 &= 66 \quad & i_{\text{start},2} &= 134 \end{aligned}$$

And from that:

$$\begin{aligned} \text{proc } 0 &\rightsquigarrow x_1...x_{67} \\ \text{proc } 1 &\rightsquigarrow x_{68}...x_{133} \\ \text{proc } 2 &\rightsquigarrow x_{134}...x_{199} \end{aligned}$$

When all of the procs compute their S_p values in parallel, we need an additional computation:

$$h\left(\frac{1}{2}(f(a)-f(b)) + \sum_{p=0}^{P-1} S_p\right)$$

Local result, S_p, are available on each proc $\rightsquigarrow$ needed for communication.

There are **2 approaches:**

1. Designate a master proc (with slave procs)
    - Slave procs, pass on their S_p values to the master

- Master computes $\sum S_p + (...)$

2. All procs have equal role
   - Each proc passed its S_p value to every other proc
   - Each proc computes $\sum S_p + (...)$

3.3.1 MPI Terminology

1. *All-to-one* communication
2. *All-to-all*

These collective communications and their associated computations $\rightsquigarrow$ **reduction operations**. Ultimately, the time cost for these reduction operations is:

$$O(\log_2(P))$$

Example 7. ID-diffusion equation: $\frac{\delta u}{\delta t} = \frac{\delta^2 u}{d x^2} + f(x,t)$, where $u = u(x,t)$.

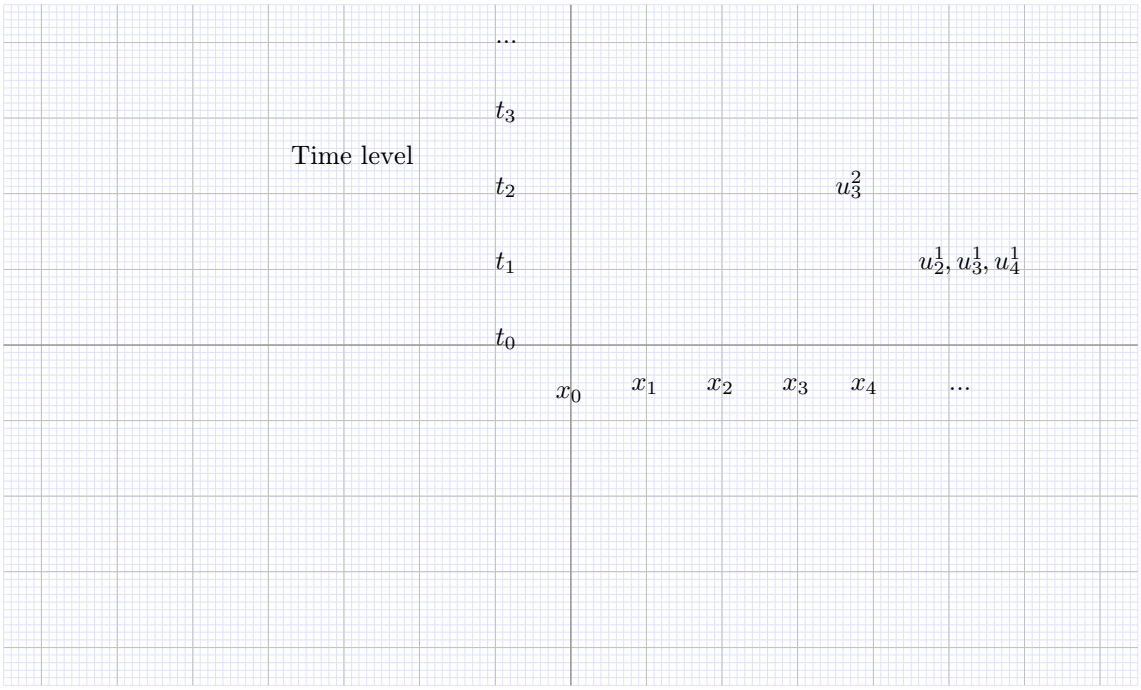
Inter-proc communication is required more often then **Ex. 2**, ie. during the parallel computations.

To solve a ID-diffused equation numerically, we employ a discretization scheme.

$$u_i^{l+1} = u_i^l + \frac{\Delta_t}{\Delta_{x^2}}(u_{i-1}^l - 2u_i^l + u_{i+1}^l) + \Delta_t f(x_i, t_e)$$

where we have $u_i^l = u(x_i, t_e)$ and x_i is the discretization of x and t_l is the discretization of t, Δ_t is the interval length and Δ_x is the interval length.

This means, that a point (x_i, t_{l+1}) we can compute the function value u_i^{l+1} from the known function values: $u_i^l, u_{i-1}^l, u_{i+1}^l$.



Comment. Boundary conditions provide initial values for the discretization scheme.

Parallelism in this example rises from the fact that the computations to find inner points u_i^{l+1} and u_j^{l+1} are independent.

The $n-1$ inner points can be computed in parallel for level $l+1$.

To compute u_i^{l+1}, we need only three values: $u_i^l, u_{i-1}^l, u_{i+1}^l$ from the previous level.

Work Division.

Partition inner points, $x_1, ..., x_{n-1}$, in P contiguous pieces. Each proc computes u_i^{l+1} for a subset of i indices in $\{1, ..., n-1\}$.

- The proc responsible for x_0, has to update u_0^{l+1}, using bondary conditions
- The proc responsible for x_n, has to update u_n^{l+1}, using the discretization scheme

Parallelism, in this example, assumes that **ALL** points at the same time level have been computed.

$\rightsquigarrow$ No proc will be allowed to proceed to level $t+1$ before all other processors have finished at level l.

$\rightsquigarrow$ Coordination between procs is achieved by a built-in synchronization mechanism, **barrier** (MPI term) which forces all pocs to wait for slower ones.

3.3.2 S-M Systems

Barrier operation is only needed for inter-proc communication.

3.3.3 D-M Systems

Proc, p, should operate on two local arrays u_p^l, u_p^{l+1} of length n_p.

4 Introduction to MPI

MPI is a variation of parallel hello world, each process different than 0, and sends a message to process 0.

Note: Remember that processes run from $0...P-1$. Procs $1...P-1$ will send messages to proc 0.

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

int main(int *argc, char **argv) {
  ...

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (my_rank != 0) {
    sprintf(message, "Greetings from process %d!", my_rank);
    dest = 0;
    // use strlen+1 so that it gets null terminated -- '\0'
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  } else {   // my_rank == 0
    for (source = 1; source < p; source++) {
      MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
```

```
      printf("%s\n", message);
    }
  }
  MPI_Finalize();
}
```

4.1 Inner Mechanisms

1. A *directive* is issued in the OS to place a copy of the executable on each proc.
2. Each proc begins execution of its copy.
3. Different procs can execute different pieces of code using branching, their ranks.

In general: Each proc runs a different program, *MIMD* (multiple instruction, multiple data).

In practice: (branching) *SPMD*, single program multiple data.

4.2 MPI/C Program Structure

- C statements and preprocess directives
- MPI is a library of definitions and functions that can be used in C/C++/Fortran programs

Things that are needed:

- We need to include `mpi.h`
- Consistent naming scheme
    - MPI identifier: `MPI_...`
- `MPI_Init` is called either: before any other MPI command can be called OR only once
- `MPI_Finalize` frees any memory allocated by MPI

4.2.1 Typical Layout

```
...
#include "mpi.h"

int main(...) {
  MPI_Init();
  // MPI function calls

  MPI_Finalize();
}
```

4.2.2 Communicators

The flow of control in an SPMD program depends on the rank of a process.

`MPI_Comm_rank`'s first argument is a communicator, ie. a collection of procs that can send messages to each other. The default, predefned communicator is `MPI_COMM_WORLD`; and it consists of all procs running when execution begins.

`MPI_Comm_size`'s first argument returns the number of processes in that communicator.

4.2.3 Messages: Data+Envelope

Message passing in MPI is carried out by `MPI_Send` and `MPI_Recv`.

- `MPI_Send` sends a message to a designated proc
- `MPI_Recv` receives a message from a proc

1. Suppose process A wants to send a message to process B
   - The message must be addressed
   - Determine the size of the message, or the end of the message
   - Destination + size
2. A sends a message to B, asking for data; C sends a message to B, containing values; D sends a message to B that should be printed
   - Add the address of the source process (A, C, D) so that B can act accordingly
3. B receives floats from the several procs $\begin{cases} \text{some to be printed} \\ \text{some sort in an array} \end{cases}$. How does B distinguish between two kinds of floats? MPI solution: use tags or message types!

   A tag is an integer from 0 to $2^{15}-1$, specified by the programmer and added to the message envelope.

   Floats for print: `tag 0`

   Floats for storage: `tag 1`
4. Suppose a program uses a library to solve a system of linear equations and that functions in the library need to do message passing.

Question 8. *How can we distinguish between messages a process A sends, and messages sent by a function in the library? (They might have the same tag)*

Solution. *The MPI solution is to add a communicator to the message environment.*

Note*: Functions under separate communicators cannot communicate with each other as the communicators are in separate memory spaces.*

Summary. The message environment contains:

1. Rank of sender/receiver
2. Tag
3. Communicator

4.2.4 Syntax for MPI_Send and MPI_Recv

MPI_Send:

```
int MPI_Send(
    void*         message,
    int           count,
    MPI_Datatype  datatype,
    int           dest,
```

```
    int           tag,
    MPI_Comm      comm)
```

MPI_Recv:

```
int MPI_Recv(
    void*         message,
    int           count,
    MPI_Datatype  datatype,
    int           source,
    int           tag,
    MPI_Comm      comm,
    MPI_Status    status)
```

1. `message` is a pointer to a block of memory where we store the message transmitted

2. The message contains a set of `count` values, each one has MPI type `datatype`

   Note: The amount of space allocated for receiving buffer does not match exactly the amount of space for the message being received.

3. `dest` is the rank of the receiving process and `source` is the rank of the sending process

   Note: MPI allows `source` to be a wildcard; predefined MPI constant: `MPI_ANY_SOURCE` $\rightarrow$ for a process to be able to receive a message from any sending process.

4. `tag` is an integer, (0 to $2^{15}-1$); `comm` is a communcator; predefined: `MPI_COMM_WORLD`

   Note: `MPI_Recv` can use the wildcard: `MPI_ANY_TAG`.

   For process A to send a message to process B, the argument(s) `comm` must be identical. A must use a tag to send information; and B must can use a either an identical tag or `MPI_ANY_TAG` to receive

5. `status` returns information on data actually received

   References a `struct`: $\begin{cases} \texttt{MPI\_SOURCE} \\ \texttt{MPI\_TAG} \\ \texttt{MPI\_ERROR} \end{cases}$

   `MPI_Get_Code` determines the size of the message received. If the source of the received message was `MPI_ANY_SOURCE` then status $\rightarrow$ `MPI_SOURCE` will contain the rank of the sending process.

4.3 Benchmarking Parallel Programs

Ignore time spent initializing MPI and performing I/O. What we care about is the wall clock time between the beginning of execution and termination.

The idea of benchmarking is to measure the efficiency of the parallel program against the serial counterpart. There are two commands that can help us out here: `MPI_Wtime` and `MPI_Wtick`.

- `MPI_Wtime` returns the number of seconds elapsed since some point in time in the past
- `MPI_Wtick` returns the precision of the result of `MPI_Wtime`

To benchmark a section of code, we put a pair of calls to `MPI_Wtime` before and after; then take the difference.

Problem 1. MPI processes executing on different procs may begin execution at different points in time.

Answer. Introduce a barrier synchronization before the first call to `MPI_Wtime`. No process can proceed beyond a barrier, until all processes have reacehed it. Usage:

```
int main (...) {
    double elapsed_time;
    MPI_Init...
    ...
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();
    ⋮
    elapsed_time += MPI_Wtime();
}
```

If we are measuring the time of a few processors, the time of each proc decreases: as the solid line tells us that adding procs decreases execution time. The dash line is the perceived perfect speed improvement.

Example 9. Numerical integration trapezoidal rule

$$\int_b^a f(x)dx$$

Subdivision in the interval $[a,b]$ in subintervals of length $h=\frac{b-a}{n}$ where n is the number of points in the subdivision.

i-th trapezoid has basis $[a+(i-1)h, a+ih]$ with $i=1,...,n$. Also, $x_i=a+ih$ has area:

$$\frac{h}{2}(f(x_{i-1})+f(x_i))$$

Thus:

$$I\approx\sum_{i=1}^{n}\text{area}_i=h\left[\frac{f(a)-f(b)}{2}+f(x_1)+f(x_2)+...\right]$$

This is still a serial program.

Example 10. Serialization of the trapzoid rule

Suppose that P, # of procs, divides n, ($\frac{n}{P}=p$). We can assign a subinterval to each proc. Each proc will compute the integral of f over its assigned subinterval whilst summing all interim results.

process #	**subinterval**
0	$[a, a+qh]$
1	$[a+qh, a+2qh]$
⋮	$[a+iqh, a+(i+1)qh]$
$P-1$	$[a+(P-1)qh, b]$

Each proc will need to know rank, $[a,b]$, n. Thus, hardcode a, b, and n. Proc 0 will be adding all the interim results.

Comments

- Branching based on rank of proc 0, SPMD
- There needs to be a careful distinction between local/global variables (a, b, n being global with local_a, local_b, local_n)

4.4 I/O on Parallel Systems

```
scanf(...,&a, &b, &n);          // 0  1  1024
```

Which proc will get the data.

Our assumption is that proc 0 will perform I/O.

The implication is that proc 0 has to send data to all other procs.

`get_data` function

1. Use different tags for messages containing a, b, n
2. Some systems allow each process to read from standard input and write to standard output

4.5 Performance Analysis

We can start with a general formula to bound the speedup achievable by a parallel program.

1. Amdahl's law
    - Decide whether a serial program merits parallelization
2. Gustafson-Barsis law
    - Evaluate the performance of a parallel program
3. Karp-Flatt metric
    - Decide whether the principal barrier to speedup is due to:
        - Amounter of inherentely sequential computation
        - Parallel overhead
4. Isoefficiency metric
    - Evaluate scalability of a parallel algorithm

We can say *speedup* is defined as:

$$\text{speedup} = \frac{\text{serial exec. time}}{\text{parallel exec. time}}$$

There are 3 categories of operations performed by a parallel program:

1. Serial computations
2. Parallel computations
3. Interproc communication + parallel overhead

Notation 11. *$\psi(n, p)$ can be denoted at the speedup achievable by solving a problem of size n on p procs.*

$\sigma(n)$ can be denoted as the serial portions of completion.

$\varphi(n)$ can be denoted as the parallel portions of completion.

$\kappa(n, p)$ can be denoted as the time for parallel overhead.

A serial program on 1 proc requires a time based off $\sigma(n) + \varphi(n)$.

Ideally, we can consider the best possible parallel executions on p procs.

The serial portion takes $\sigma(n)$ time and the parallel portion takes $\frac{\varphi(n)}{p}$ time.

IPC takes $\kappa(n,p)$ time, thus in summary:

$$\psi(n,p) \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\frac{\varphi(n)}{p}+\kappa(n,p)}$$

Next, we can be define efficiency as:

$$\text{efficiency} = \frac{\text{serial exec. time}}{p \times \text{parallel exec. time}}$$

where *efficiency* is the measure of proc utilization.

Thus, defined as:

$$\epsilon(n,p) \leq \frac{\sigma(n)+\varphi(n)}{p[\sigma(n)+\kappa(n,p)]+\varphi(n)}$$

Given a property that: $0 \leq \epsilon(n,p) \leq 1$.

4.5.1 Amdahl's Law

Let's assume that $\kappa(n,p) \geq 0$, and discard it:

$$\psi(n,p) \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\frac{\varphi(n)}{p}} \tag{1}$$

Denote, by f, the inherently sequential portion of the computation:

$$\begin{aligned} f = \frac{\sigma(n)}{\sigma(n)+\varphi(n)} &\rightsquigarrow \varphi(n) = \sigma(n)\left(\frac{1}{f}-1\right) \\ &\rightsquigarrow \sigma(n)+\varphi(n) = \frac{\sigma(n)}{f} \end{aligned}$$

Then we can plugin $\varphi(n)$ into (1).

$$\begin{aligned} \psi(n,p) &\leq \frac{\frac{\sigma(n)}{f}}{\sigma(n)+\frac{\sigma(n)\left(\frac{1}{f}-1\right)}{p}} \\ &\leq \frac{\frac{1}{f}}{1+\frac{\left(\frac{1}{f}-1\right)}{p}} \\ &= \frac{1}{f+\frac{(1-f)}{p}} \end{aligned}$$

Remark 12. Amdahl's Law

If f is the fraction of operations in a computation that must be performed in serial, then the max speedup ψ achievable on p procs is:

$$\leq \frac{1}{f+\frac{(1-f)}{p}}$$

Usage:

- An upper bound on speedup achievable on p procs
- Determine the asymptotic speedup as p increases

****

Example 13. If we have a program that's 90% parallel and 10% serial, (max speedup up to 8 procs).

Define: $p=8,\ f=0.1$. We have:

$$\psi \leq \frac{1}{0.1+\frac{(1-0.1)}{8}} \approx 4.7$$

Example 14. We have program that's 25% serial. What's the max possible speedup?

$p \rightsquigarrow \infty,\ f=0.25$. Thus:

$$\psi \leq \frac{1}{0.25+\left(\frac{0.75}{p}\right)} = 4$$

4.5.2 Gustafson-Barsis Law

Treat time as a constant and let the problem size increase with p. We can define s as the fraction of time spent in the parallel computation performing inherently sequential operations and $1-s$ is the same but with parallel operations.

$$s=\frac{\sigma(n)}{\sigma(n)+\frac{\varphi(n)}{p}} \quad \rightsquigarrow \quad \sigma(n)=\left(\sigma(n)+\frac{\varphi(n)}{p}\right)\times s$$
$$1-s=\frac{\frac{\sigma(n)}{p}}{\sigma(n)+\frac{\varphi(n)}{p}} \quad \rightsquigarrow \quad \sigma(n)=\left(\sigma(n)+\frac{\varphi(n)}{p}\right)\times (1-s)\,p$$

Remark 15. Gustafson-Barsis Law

Given a parallel program solving a problem of size n on p procs, the max speedup is:

$$\psi \leq p+(p-1)\,s$$

Example 16. An application on 64 procs runs in 220s; where 5% of that is spent on serial computations, what is the max speedup?

$$\psi \leq 64+(0.05)\,63=60.85$$

Example 17. We want to demonstrate a new supercomputer with 16384 procs can achieve a speedup of 15000 on a specific problem. What is the max fraction that can be devoted to serial operations?

$$15000 \leq 16384-16383\times s \rightsquigarrow s=0.084$$

4.5.3 Isoefficiency Metric

Scalability: The measure of the ability of a parallel program to increase performance as the number of procs increases.

Suppose a parallel program exhibits efficiency: $\epsilon(n,p)$, we can define:

$$C = \frac{\epsilon(n,p)}{1-\epsilon(n,p)}$$

Also:

$$T_0(n,p) = (p-1)\,\sigma(n) + p\,\kappa(n,p)$$

where T_0 is the total amount of time spent by all procs doing work not done by a sequential algorithm.

To maintain a certain level of efficiency as the number of procs increases, n must be increased so that

$$T(n,1) \geq C \times T_0(n,p)$$

where $T(n,1)$ is the serial execution time. Need to maintain efficiency when increasing number of procs, thus increasing the size of the problem that can be solved.

4.6 Collective Communication

Recal the trapezoidal code:

Use trap on 8 procs.

a) Procs 1-7 stay idle while proc 0 does I/O

b) After proc 0 has collected output, higher order procs keep waiting until proc 0 distributes input to lower procs

c) Proc 0 computes and in addition, the sum of intermediate values

4.6.1 Tree Structure Communication

- Focus on input data distribution
- Divide work more evenly among procs
- Imagine a tree of procs with 0 at the root

$$\begin{aligned} 1^{\text{st}}\,\text{stage:}\,0 &\rightsquigarrow 1 \\ 2^{\text{nd}}\,\text{stage:}\,0 &\rightsquigarrow 2\,\text{and}\,1 \rightsquigarrow 3 \\ 3^{\text{rd}}\,\text{stage:}\,0 &\rightsquigarrow 4+1 \rightsquigarrow 5+2 \rightsquigarrow 6+3 \rightsquigarrow 7 \end{aligned}$$

- Reduced 7 stages to 3 stages
- In general, p procs; the data is distributed in $\lceil \log_2 P \rceil$ stages rather than $P-1$

Example 18. $p = 1024 \rightsquigarrow \lceil \log_2 P \rceil = 10$. That is a reduction by a factor of 100.

** Modifying `GetData()` in trapezoidal method to use a tree structure.

```
for (stage = first; stage > last; stage++) {
  if I_receive(stage, my_rank, &source) {
    receive(data, source);
```

```
  } else {
    if I_send(stage, my_rank, p, &dest) {
      send(data, dest);
    }
  }
}
```

`I_receive` $\begin{cases} 1, \text{ calling proc receives data} \\ 0, \text{ otherwise} \end{cases}$

`I_send` $\begin{cases} 1, \text{ proc sends data} \\ 0, \text{ otherwise} \end{cases}$

Implementation. We need:

1. Whether a proc receives the source
2. Whether a proc send the dest

Deciding the optimal scheme requires some knowledge of the topology of the system.

- General tree scheme: number of stages $0, 1, ...$
  - If $2^{\text{stage}} \leq \text{my rank} < 2^{\text{stage}+1}$
    - Then `I_receive` from proc $(\text{my rank} - 2^{\text{stage}})$
  - If $\text{my rank} < 2^{\text{stage}}$
    - Then `I_send` to proc $(\text{my rank} + 2^{\text{stage}})$

```
int I_receive(int stage, int my_rank, int* source) {
  int power_2_stage = 1 << stage;     // Bit shift
  if (power_2_stage <= my_rank && my_rank < (2 * power_2_stage) {
    *source = my_rank - power_2_stage;
    return 1;
  } else {
    return 0;
  }
}


int I_send(int stage, int my_rank, int p, int* dest) {
  // Same as recv
}


void send(float a, float b, int n, int dest) {
  MPI_Send(&a, 1, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
  MPI_Recv(&b, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
  MPI_Send(&n, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
}


void GetData(a_ptr, b_ptr, n_ptr, my_rank, p) {
  int source;
  int dest;
```

```
  int stage;

  if (my_rank == 0) {
    // scanf to get input
  }

  for (stage = 0; stage < log2P; stage++) {
    if (I_receive(stage, my_rank, &source) {
      receive(a_ptr, b_ptr, n_ptr, &source);
    } else if (I_send(...)) {
      send(...);
    }
  }
}
```

- MPI provides a function to do all of this...`Broadcast`
- A communication pattern that involves all the procs in a communicator is a collective communicator
- A broadcast is a collective communicator, in which a proc, send the data to every process in the communicator

```
int MPI_Bcast(void* message, int count, MPI_Datatype data_type, int root,
MPI_Comm comm) { ... }
```

- `MPI_Bcast` sends a copy of the data in `message` on the proc with rank `root`, to each proc in the communicator `comm`
- It should be called by all procs in the communicator with the same arguments for `root` and `comm`
- `count`/`datatype` specify how much memory is needed for the message
- Revise `GetData()` using `MPI_Bcast`

```
void GetData(a_ptr, b_ptr, n_ptr, my_rank) {
  if (my_rank == 0) {
    // scanf(...)
  }

  MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Bcast(b_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Bcast(n_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

4.6.2 Tags, Safety, Buffering, Synchronization

Time	Proc A	Proc B
1	MPI send to B (tag = 0)	$\sim$
2	MPI send to B (tag = 1)	$\sim$
3	$\sim$	MPI recv from A (tag = 1)
4	$\sim$	MPI recv from A (tag = 0)

- This requires buffering

- Set aside memory for storing messages before a receive has been executed
- Until proc B calls `MPI_Recv`, the system does not know where the message, that A is sending, should be stored
- When B calls `MPI_Recv`, the system looks for any buffered message with an envelope matching the `recv` parameters
- If no message exists, it will wait until one arrives