HPC Programming Bootcamp

Day 3: Writing MPI programs

Abhinav Bhatele, Department of Computer Science

UNIVERSITY OF MARYLAND

OpenMP: reduction

```
double area, pi, x;
int i, n;
...
area = 0.0;


for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}

pi = area / n;
```

OpenMP: reduction

```
double area, pi, x;
int i, n;
...
area = 0.0;

#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}

pi = area / n;
```

OpenMP: reduction

```
double area, pi, x;
int i, n;
...
area = 0.0;

#pragma omp parallel for private(x) reduction(+:area)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}

pi = area / n;
```

schedule clause

- We can use the schedule clause too specify the allocation of iterations to threads
- Static scheduling
- Dynamic scheduling
- Guided scheduling
- Auto
- Runtime: based on the OMP_SCHEDULE flag

Writing MPI programs

Programming models

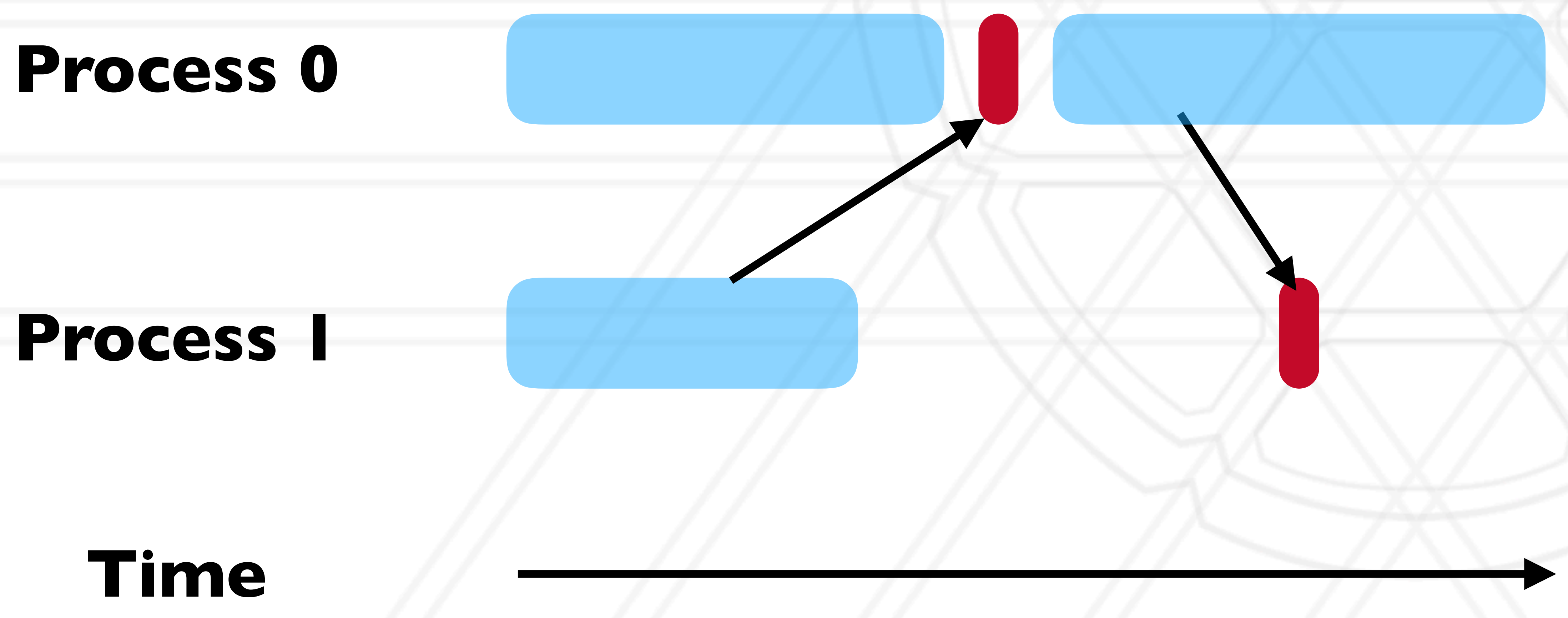
- Shared memory model: All threads/processes have access to all of the memory
  - Pthreads, OpenMP
- Distributed memory model: Each process has access to their own local memory
  - Also referred to as message passing
  - MPI, Charm++
- Hybrid models: Use both shared and distributed memory models together
  - MPI+OpenMP, Charm++ (SMP mode)

Distributed memory / message passing

- Each process can use its local memory for computation
- When it needs data from remote processes, it has to send messages
- PVM (Parallel Virtual Machine) was developed in 1989-1993
- MPI forum was formed in 1992 to standardize message programming models and MPI 1.0 was released around 1994
  - v2.0 - 1997
  - v3.0 - 2012

Message passing

- Each process runs in its own address space
  - Access to only their memory
- Use special routines to exchange data



Message Passing Interface (MPI)

- It is an interface standard — defines the operations / routines needed for message passing
- Implemented by vendors and academics for different platforms
  - Meant to be "portable": ability to run the same code on different platforms without modifications
- Two popular implementations are MPICH and MVAPICH

Hello World in MPI

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

int main(int  argc, char *argv) {
  int rank, size;
  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Hello world! I'm %d of %d\n", rank, size);

  MPI_Finalize();
  return 0;
}
```

Compiling and running an MPI program

- Compiling:

```
mpicc -o hello hello.c
```

- Running:

```
mpirun -np 2 ./hello
```

Process creation / destruction

- `int MPI_Init( int argc, char **argv )`
  - Initialize the MPI execution environment
- `int MPI_Finalize( void )`
  - Terminates MPI execution environment

Process identification

- `int MPI_Comm_size( MPI_Comm comm, int *size)`
  - Determines the size of the group associated with a communicator
- `int MPI_Comm_rank( MPI_Comm comm, int *rank)`
  - Determines the rank (ID) of the calling process in the communicator
- Communicator — a set of processes
  - Default communicator: `MPI_COMM_WORLD`

Send a message

```
int MPI_Send( const void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm )
```

`buf`: address of send buffer

`count`: number of elements in send buffer

`datatype`: datatype of each send buffer element

`dest`: rank of destination process

`tag`: message tag

`comm`: communicator

Receive a message

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

`buf`: address of receive buffer

`status`: status object

`count`: maximum number of elements in receive buffer

`datatype`: datatype of each receive buffer element

`source`: rank of source process

`tag`: message tag

`comm`: communicator

Simple send/receive in MPI

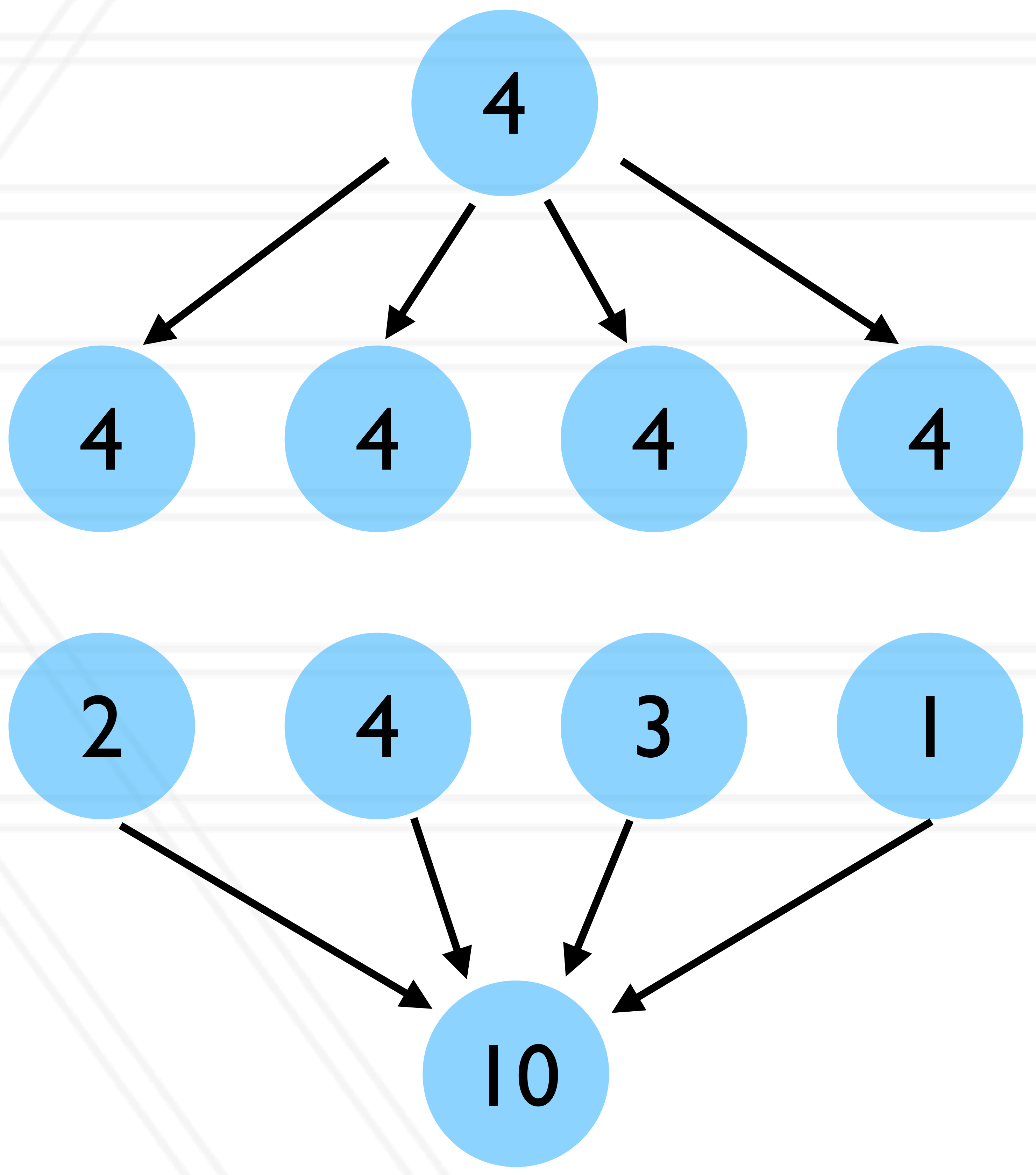
```
int main(int  argc, char *argv) {
  ...
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int data;
  if (rank == 0) {
    data = 7;
    MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  } else if (rank == 1) {
    MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received data %d from process 0\n", data);
  }

  ...
}
```

Collective operations

- `int MPI_Barrier( MPI_Comm comm)`
  - Blocks until all processes in the communicator have reached this routine
- `int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )`
  - Send data from root to all processes
- `int MPI_Reduce( const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )`
  - Reduce data from all processes to the root



Collective operations

- `int MPI_Scatter( const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
    - Send data from root to all processes
- `int MPI_Gather( const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
    - Gather data from all processes to the root
- MPI_Scan

Calculate the value of $\pi = \int_0^1 \frac{4}{1+x^2}$

```
int main(int argc, char *argv[])
{
    ...

    n = 10000;

    h   = 1.0 / (double) n;
    sum = 0.0;

    for (i = 1; i <= n; i += 1) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }
    pi = h * sum;

    ...
}
```

Calculate the value of $\pi = \int_0^1 \frac{4}{1+x^2}$

```
int main(int argc, char *argv[])
{
    ...

    n = 10000;
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    h   = 1.0 / (double) n;
    sum = 0.0;

    for (i = myrank + 1; i <= n; i += numranks) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }
    pi = h * sum;

    MPI_Reduce(&pi, &globalpi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    ...
}
```

MPI communicators

- Communicator is a group or set of processes numbered 0, … , n-1
- Every program starts with MPI_COMM_WORLD
- Several MPI routines to create sub-communicators
  - MPI_Comm_split
  - MPI_Cart_create
  - MPI_Group_incl

Non-blocking point-to-point calls

- MPI_Isend and MPI_Irecv
- Two parts:
  - post the operation
  - Wait for results: need to call MPI_Wait or MPI_Test
- Can help with overlapping computation with communication

Other MPI Calls

- MPI_Wtime
- MPI profiling interface: PMPI_*

Performance Tools

Performance analysis

- Parallel performance of a program might not be what we expect
- How do we find performance bottlenecks?
- Two parts to performance analysis: measurement and analysis/visualization
- Simplest tool: timers in the code and printf

Performance Tools

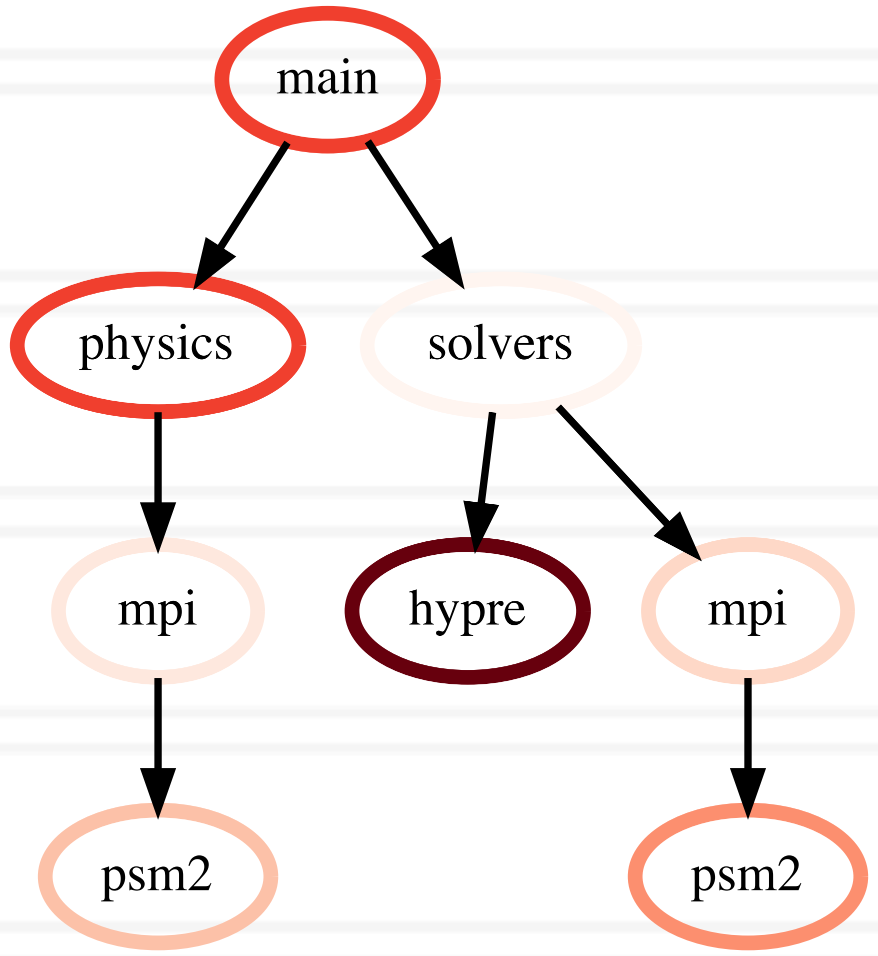
- Tracing tools
  - Capture entire execution trace
  - Vampir, Score-P
- Profiling tools
  - Typically use statistical sampling
  - Gprof
- Many tools can do both
  - TAU, HPCToolkit, Projections

Metrics recorded

- Counts of function invocations
- Time spent in code
- Hardware counters

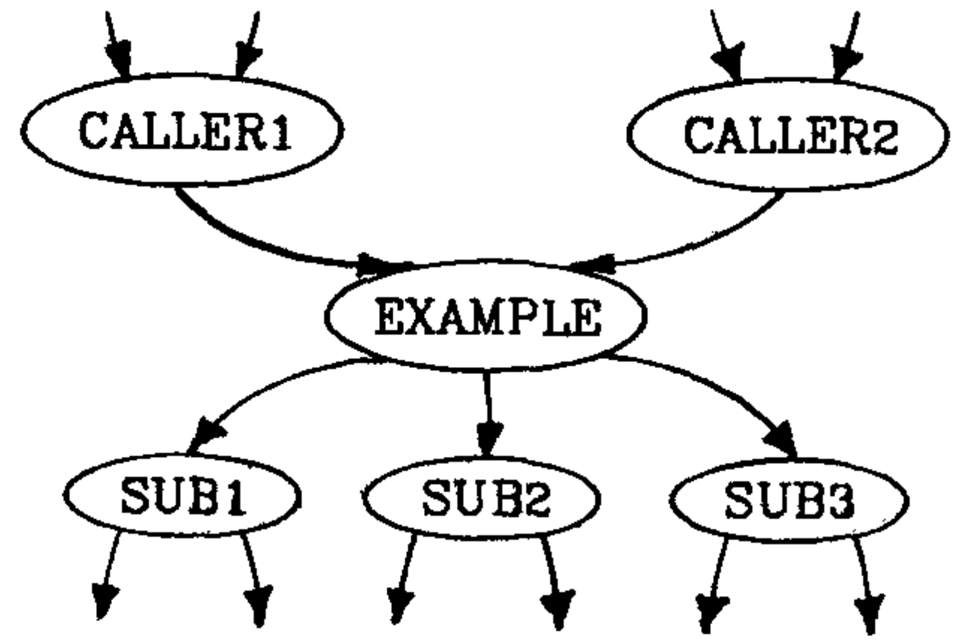
Calling contexts, trees, and graphs

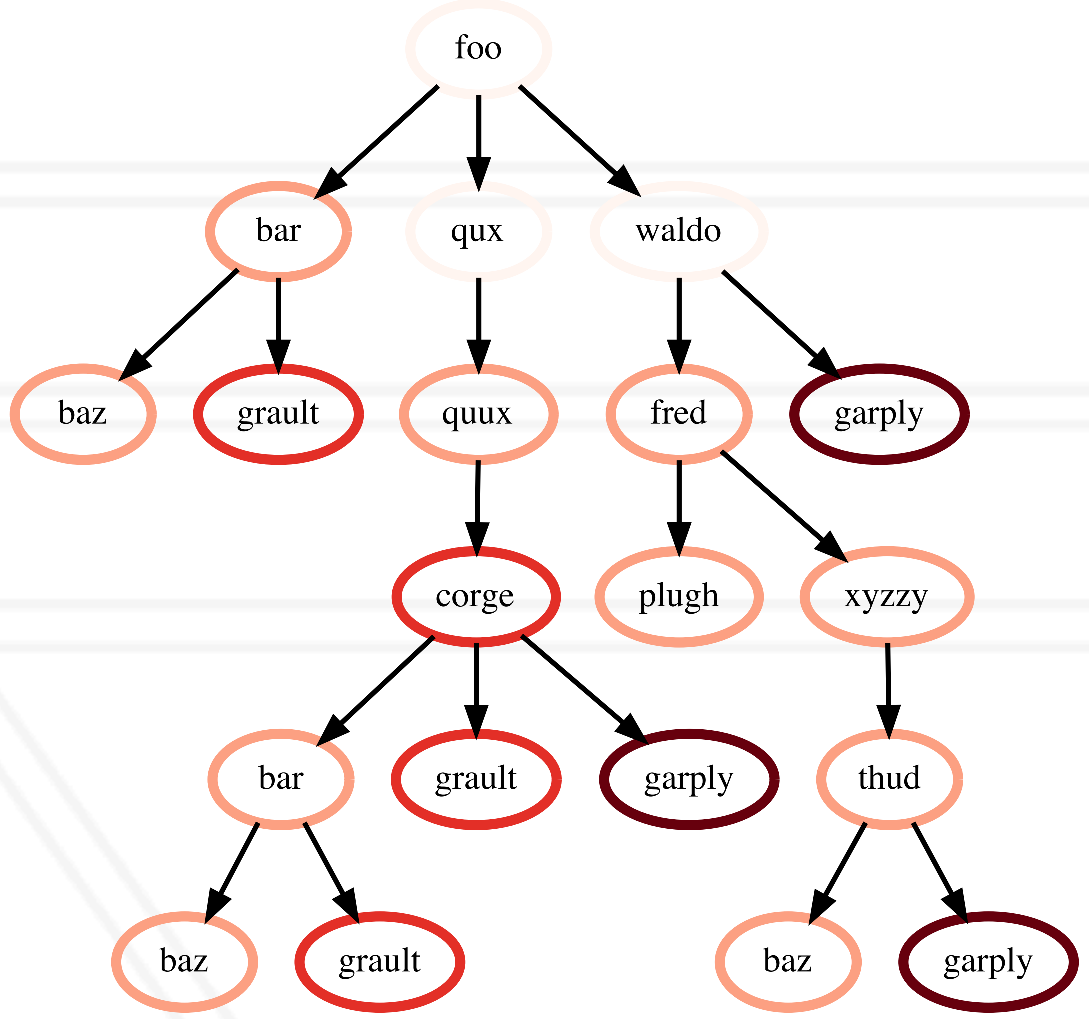
- Calling context or call path: Sequence of function invocations leading to the current sample
- Calling context tree: dynamic prefix tree of all call paths in an execution
- Call graph: keep caller-callee relationships as arcs



Output

- Flat profile: Listing of all functions with counts and execution times
- Call graph profile
- Calling context tree





Questions?

UNIVERSITY OF MARYLAND

Abhinav Bhatele

5218 Brendan Iribe Center (IRB) / College Park, MD 20742

phone: 301.405.4507 / e-mail: bhatele@cs.umd.edu