DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Advanced MPI
OpenMP/MPI Hybrid Programming
Load Balancing; Termination Detection

Parallel and Distributed Computing

José Monteiro

Monday, 27 March 2023

Outline

- Advanced MPI
- Hybrid programming
  - Combining OpenMP and MPI
- Load balancing
- Termination detection

Asynchronous Communication

- Blocking operations
  - `MPI_Send` only returns when the send buffer can be reused
    - Data has been received by destination
    - Or MPI has saved the data somewhere
  - `MPI_Recv` only returns when the receive buffer has been filled with valid data

- Non-blocking versions
  - `MPI_Isend` / `MPI_Irecv`
    - Buffers transferred in the background
    - Cannot use buffers until communication complete
  - allow overlap of computation and communication
  - use `MPI_Wait` / `MPI_Test` to check comm complete

Asynchronous Collectives

- Recent versions of MPI also provide asynchronous collective operations:

```
MPI_Isomething(<usual arguments>, MPI_Request *req)
```

- They return an `MPI_Request` reqect, similar to nonblocking point-to-point operations

- The user must call `MPI_Test` / `MPI_Wait` or their variants to complete the operation

- Multiple nonblocking collectives may be outstanding, but they must be called in the same order on all processes

Asynchronous Collectives

- Blocking and non-blocking don't match!
  - either all processes call the non-blocking version or all call the blocking one
  - Thus the following code is incorrect:

```
if(rank == root)
  MPI_Reduce( &x /* ... */ root, comm );
else
  MPI_Ireduce( &x /* ... */ root, comm, &req);
```

- Note that in point-to-point you can match an `MPI_Irecv` with an `MPI_Send`

Persistent Communication

```
for(i = 1; i < BIGNUM; i++){
    MPI_Irecv(buf1, cnt, tp, src, tag, com, &recv_req);

    do_work(buf1, buf2);

    MPI_Isend(buf2, cnt, tp, dst, tag, com, &send_req);

    // Wait for send to complete
    MPI_Wait(&send_req, status);
    // Wait for receive to finish (no deadlock!)
    MPI_Wait(&recv_req, status);
}
```

Persistent Communication

```
//Step 1) Initialize send/request objects
MPI_Recv_init(buf1, cnt, tp, src, tag, com, &recv_req);
MPI_Send_init(buf2, cnt, tp, dst, tag, com, &send_req);
for(i = 1; i < BIGNUM; i++){
    //Step 2) Use start in place of recv and send
    //MPI_Irecv (buf1, cnt, tp, src, tag, com, &recv_req);
    MPI_Start(&recv_req);

    do_work(buf1, buf2);

    //MPI_Isend (buf2, cnt, tp, dst, tag, com, &send_req);
    MPI_Start(&send_req);

    //Wait for send to complete
    MPI_Wait(&send_req, status);
    //Wait for receive to finish (no deadlock!)
    MPI_Wait(&recv_req, status);
}
MPI_Request_free(&recv_req);    //Step 3) Clean up the requests
MPI_Request_free(&send_req);
```

OpenMP: Advantages / Disadvantages

- Advantages

OpenMP: Advantages / Disadvantages

- Advantages
  - Applications are relatively easy to implement

OpenMP: Advantages / Disadvantages

- Advantages
  - Applications are relatively easy to implement
  - Low latency, high bandwidth

OpenMP: Advantages / Disadvantages

- Advantages
  - Applications are relatively easy to implement
  - Low latency, high bandwidth
  - Allows run time scheduling, dynamic load balancing

OpenMP: Advantages / Disadvantages

- Advantages
  - Applications are relatively easy to implement
  - Low latency, high bandwidth
  - Allows run time scheduling, dynamic load balancing
  - Both fine and course grain parallelism are effective

OpenMP: Advantages / Disadvantages

- Advantages
  - Applications are relatively easy to implement
  - Low latency, high bandwidth
  - Allows run time scheduling, dynamic load balancing
  - Both fine and course grain parallelism are effective
  - Implicit communication

OpenMP: Advantages / Disadvantages

- Advantages
  - Applications are relatively easy to implement
  - Low latency, high bandwidth
  - Allows run time scheduling, dynamic load balancing
  - Both fine and course grain parallelism are effective
  - Implicit communication
- Disadvantages

OpenMP: Advantages / Disadvantages

- Advantages
  - Applications are relatively easy to implement
  - Low latency, high bandwidth
  - Allows run time scheduling, dynamic load balancing
  - Both fine and course grain parallelism are effective
  - Implicit communication
- Disadvantages
  - Concurrent access to memory may degrade performance

OpenMP: Advantages / Disadvantages

- Advantages
  - Applications are relatively easy to implement
  - Low latency, high bandwidth
  - Allows run time scheduling, dynamic load balancing
  - Both fine and course grain parallelism are effective
  - Implicit communication
- Disadvantages
  - Concurrent access to memory may degrade performance
  - Overheads can become an issue (e.g., size of the parallel loop too small)

OpenMP: Advantages / Disadvantages

- Advantages
  - Applications are relatively easy to implement
  - Low latency, high bandwidth
  - Allows run time scheduling, dynamic load balancing
  - Both fine and course grain parallelism are effective
  - Implicit communication
- Disadvantages
  - Concurrent access to memory may degrade performance
  - Overheads can become an issue (e.g., size of the parallel loop too small)
  - Coarse grain parallelism often requires a parallelization strategy similar to an MPI strategy, making the implementation more complicated

OpenMP: Advantages / Disadvantages

- Advantages
  - Applications are relatively easy to implement
  - Low latency, high bandwidth
  - Allows run time scheduling, dynamic load balancing
  - Both fine and course grain parallelism are effective
  - Implicit communication
- Disadvantages
  - Concurrent access to memory may degrade performance
  - Overheads can become an issue (e.g., size of the parallel loop too small)
  - Coarse grain parallelism often requires a parallelization strategy similar to an MPI strategy, making the implementation more complicated
  - Explicit synchronization is required

MPI: Advantages / Disadvantages

- Advantages

MPI: Advantages / Disadvantages

- Advantages
  - MPI involves explicit parallelism, often provides a better performance

MPI: Advantages / Disadvantages

- Advantages
  - MPI involves explicit parallelism, often provides a better performance
  - Parallel access to memory

MPI: Advantages / Disadvantages

- Advantages
  - MPI involves explicit parallelism, often provides a better performance
  - Parallel access to memory
  - A number of optimized collective communication routines are available

MPI: Advantages / Disadvantages

- Advantages
  - MPI involves explicit parallelism, often provides a better performance
  - Parallel access to memory
  - A number of optimized collective communication routines are available
  - In principle, communications and computation can be overlapped

MPI: Advantages / Disadvantages

- Advantages
  - MPI involves explicit parallelism, often provides a better performance
  - Parallel access to memory
  - A number of optimized collective communication routines are available
  - In principle, communications and computation can be overlapped
  - Communications often cause synchronization naturally

MPI: Advantages / Disadvantages

- Advantages
  - MPI involves explicit parallelism, often provides a better performance
  - Parallel access to memory
  - A number of optimized collective communication routines are available
  - In principle, communications and computation can be overlapped
  - Communications often cause synchronization naturally
- Disadvantages

MPI: Advantages / Disadvantages

- Advantages
  - MPI involves explicit parallelism, often provides a better performance
  - Parallel access to memory
  - A number of optimized collective communication routines are available
  - In principle, communications and computation can be overlapped
  - Communications often cause synchronization naturally
- Disadvantages
  - Decomposition, development and debugging of applications can be a considerable overhead

MPI: Advantages / Disadvantages

- Advantages
  - MPI involves explicit parallelism, often provides a better performance
  - Parallel access to memory
  - A number of optimized collective communication routines are available
  - In principle, communications and computation can be overlapped
  - Communications often cause synchronization naturally
- Disadvantages
  - Decomposition, development and debugging of applications can be a considerable overhead
  - Communications can often create a large overhead, which needs to be minimized

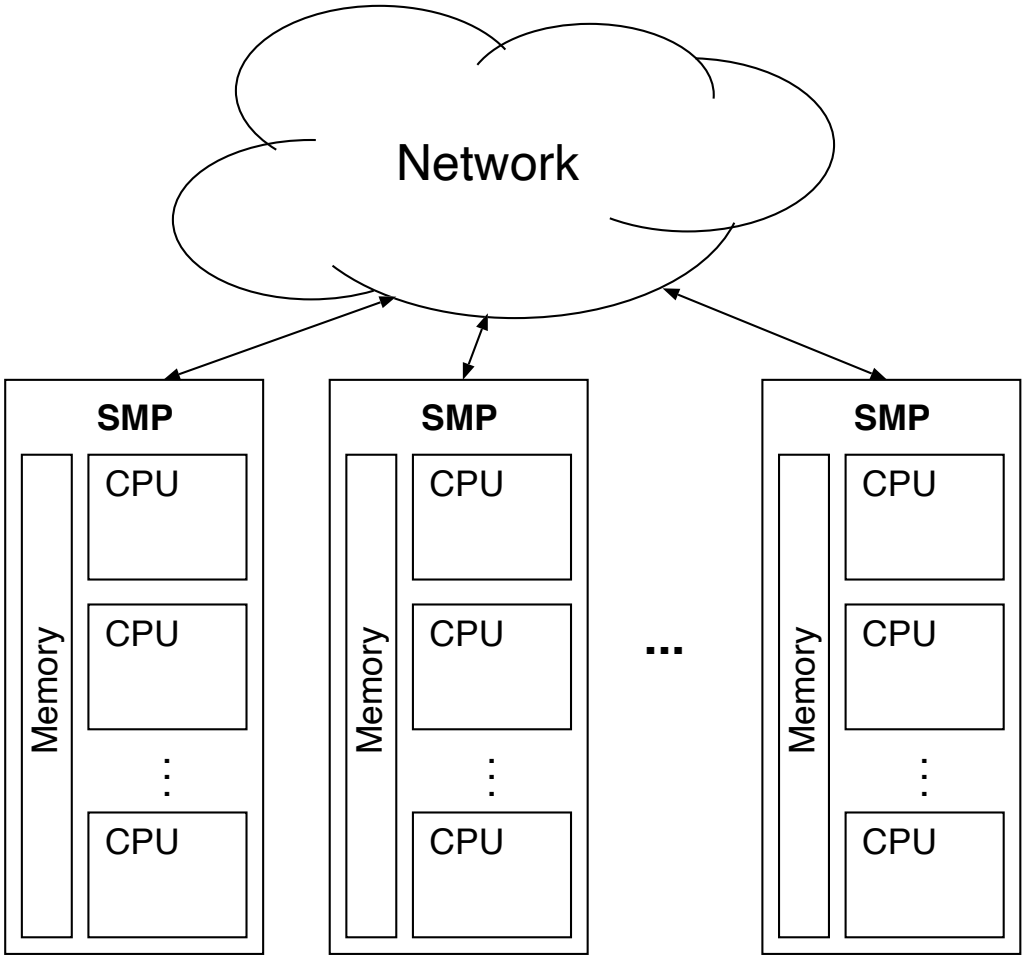
MPI: Advantages / Disadvantages

- Advantages
  - MPI involves explicit parallelism, often provides a better performance
  - Parallel access to memory
  - A number of optimized collective communication routines are available
  - In principle, communications and computation can be overlapped
  - Communications often cause synchronization naturally
- Disadvantages
  - Decomposition, development and debugging of applications can be a considerable overhead
  - Communications can often create a large overhead, which needs to be minimized
  - The granularity often has to be large, fine grain granularity can create a large quantity of communications

MPI: Advantages / Disadvantages

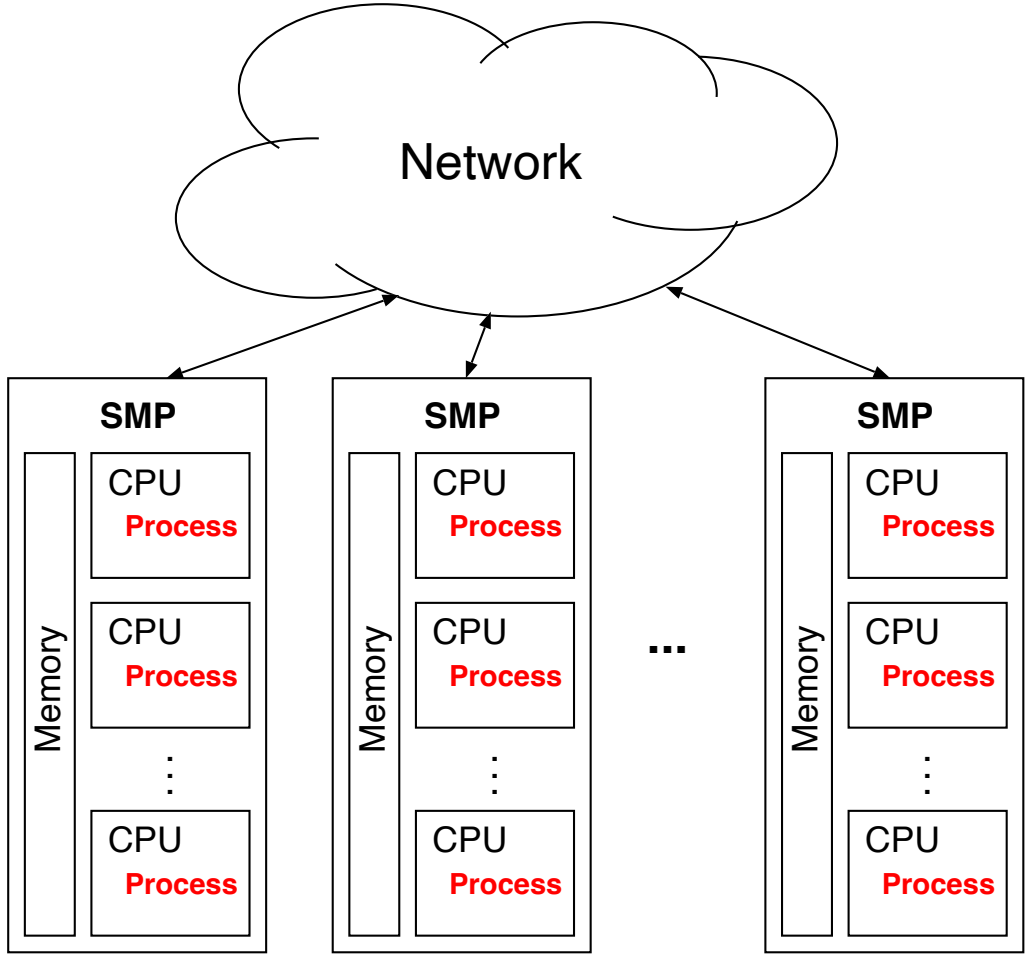
- Advantages
  - MPI involves explicit parallelism, often provides a better performance
  - Parallel access to memory
  - A number of optimized collective communication routines are available
  - In principle, communications and computation can be overlapped
  - Communications often cause synchronization naturally
- Disadvantages
  - Decomposition, development and debugging of applications can be a considerable overhead
  - Communications can often create a large overhead, which needs to be minimized
  - The granularity often has to be large, fine grain granularity can create a large quantity of communications
  - Dynamic load balancing is often difficult

Clusters of SMPs



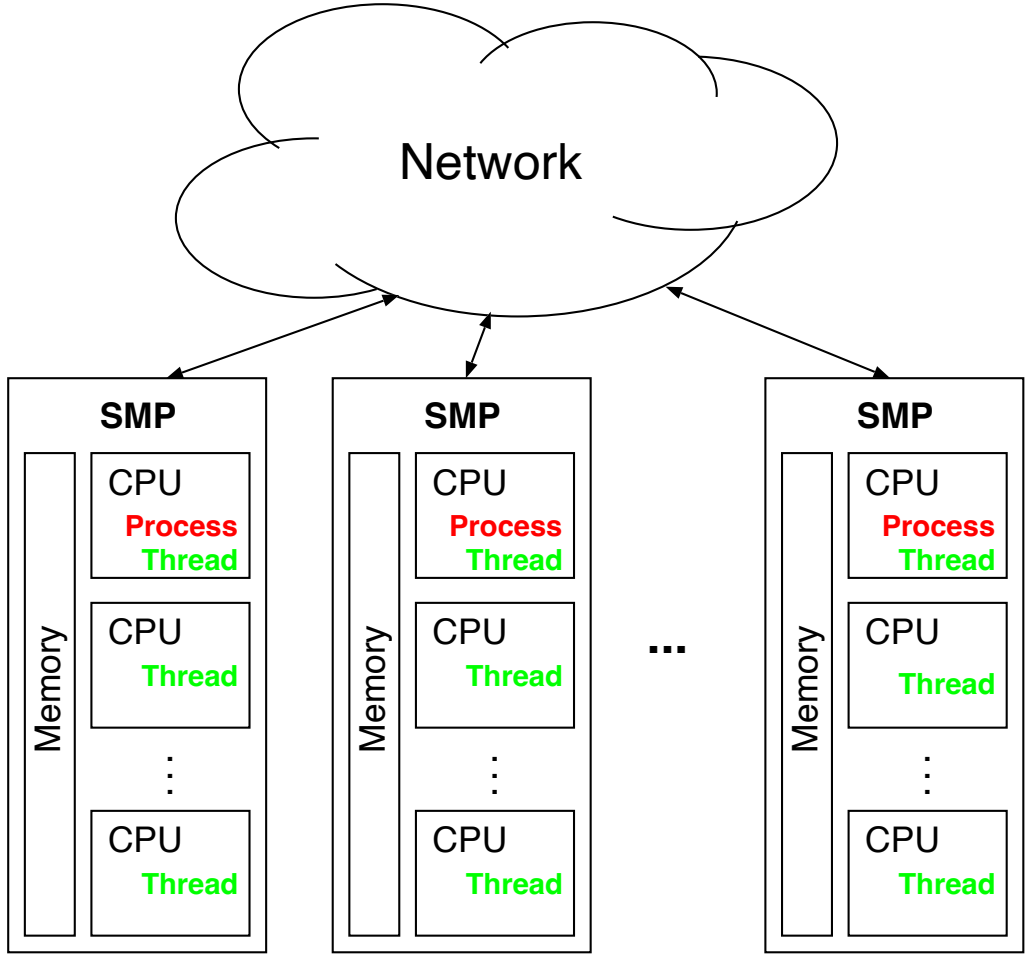
Clusters of SMPs

- Using only MPI



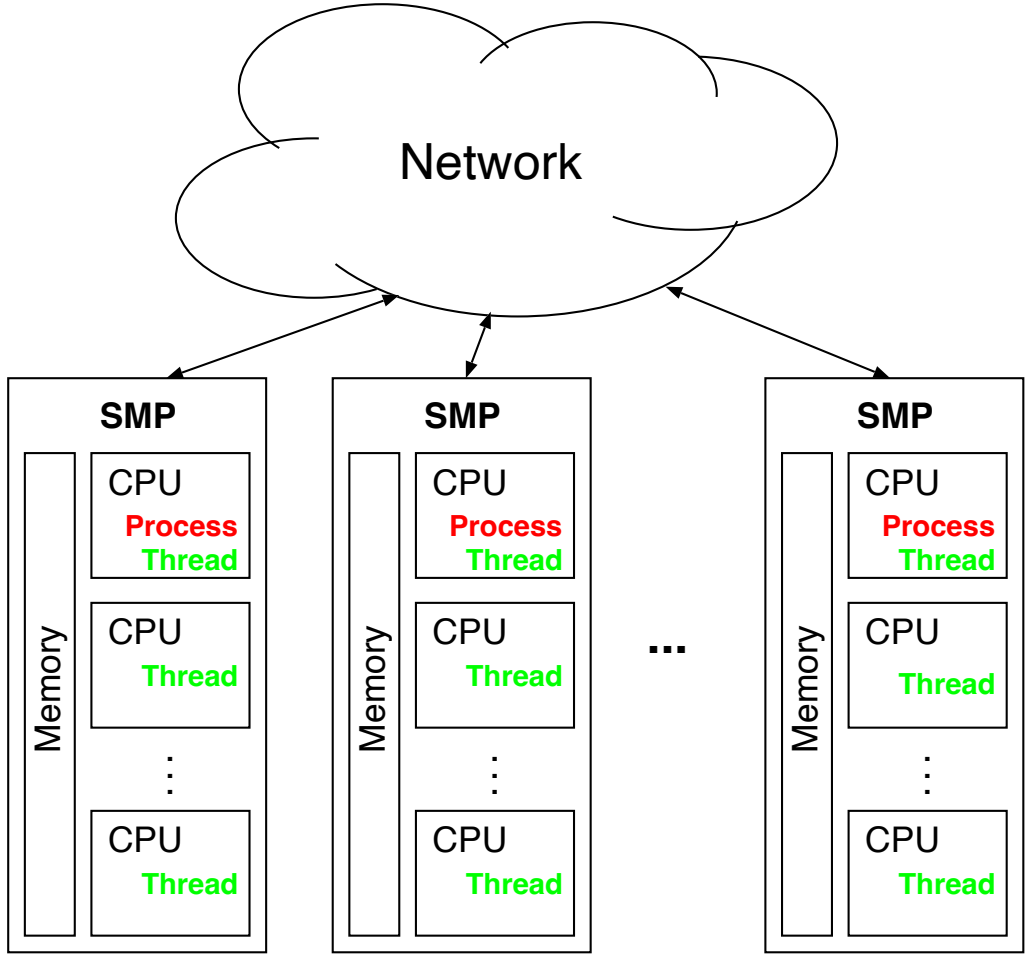
Clusters of SMPs

- Using MPI together with OpenMP



Clusters of SMPs

- Using MPI together with OpenMP



- Best of both worlds!

Hello World, in OpenMP+MPI

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

int main(int argc, char *argv[])
{
  int numprocs, rank, namelen, iam, nt;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  #pragma omp parallel private(iam, nt)
  {
    nt = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d from process %d out of %d on %s\n”,
           iam, nt, rank, numprocs, processor_name);
  }
  MPI_Finalize();
}
```

Hello World, in OpenMP+MPI

- Compiling

```
$ mpicc -fopenmp HellowWorld.c -o HelloWorld
```

Hello World, in OpenMP+MPI

- Compiling

```
$ mpicc -fopenmp HellowWorld.c -o HelloWorld
```

- Running

```
$ ./HelloWorld
Hello from thread 0 out of 2 from process 0 out of 1 on markov
Hello from thread 1 out of 2 from process 0 out of 1 on markov
```

Hello World, in OpenMP+MPI

- Compiling

```
$ mpicc -fopenmp HellowWorld.c -o HelloWorld
```

- Running

```
$ ./HelloWorld
Hello from thread 0 out of 2 from process 0 out of 1 on markov
Hello from thread 1 out of 2 from process 0 out of 1 on markov
$ mpirun -n 3 ./HelloWorld
Hello from thread 0 out of 2 from process 0 out of 3 on markov
Hello from thread 1 out of 2 from process 0 out of 3 on markov
Hello from thread 0 out of 2 from process 2 out of 3 on markov
Hello from thread 1 out of 2 from process 2 out of 3 on markov
Hello from thread 0 out of 2 from process 1 out of 3 on markov
Hello from thread 1 out of 2 from process 1 out of 3 on markov
```

A Common Execution Scenario

1. A single MPI process is launched on each SMP node in the cluster
2. Each process spawns N threads on each SMP node
3. At some global sync point, the master thread on each SMP communicate with one another
4. The threads belonging to each process continue until another sync point or completion

MPI Handling of Threads

- Replace `MPI_Init` by:

```
int MPI_Init_thread (
  int    argc,      /* command line arguments */
  char *argv[],
  int    desired,   /* desired level of thread support */
  int   *provided   /* level of thread support provided */
)
```

MPI Handling of Threads

- Thread support levels

MPI Handling of Threads

- Thread support levels
  - `MPI_THREAD_SINGLE`: Only one thread will execute

MPI Handling of Threads

- Thread support levels
  - `MPI_THREAD_SINGLE`: Only one thread will execute
  - `MPI_THREAD_FUNNELED`: Process may be multi-threaded, but only main thread will make MPI calls (all MPI calls are “funneled” to main thread) [Default]

MPI Handling of Threads

- Thread support levels
  - `MPI_THREAD_SINGLE`: Only one thread will execute
  - `MPI_THREAD_FUNNELED`: Process may be multi-threaded, but only main thread will make MPI calls (all MPI calls are "funneled" to main thread) [Default]
  - `MPI_THREAD_SERIALIZED`: Process may be multi-threaded, multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are "serialized")

MPI Handling of Threads

- Thread support levels
  - `MPI_THREAD_SINGLE`: Only one thread will execute
  - `MPI_THREAD_FUNNELED`: Process may be multi-threaded, but only main thread will make MPI calls (all MPI calls are "funneled" to main thread) [Default]
  - `MPI_THREAD_SERIALIZED`: Process may be multi-threaded, multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are "serialized")
  - `MPI_THREAD_MULTIPLE`: Multiple threads may call MPI, with no restrictions

Issues with `MPI_THREAD_MULTIPLE`

- An MPI implementation is not required to support levels higher than `MPI_THREAD_SINGLE`
  - It is not required to be thread safe
  - A fully thread-safe implementation will support `MPI_THREAD_MULTIPLE`

Issues with `MPI_THREAD_MULTIPLE`

- An MPI implementation is not required to support levels higher than `MPI_THREAD_SINGLE`
  - It is not required to be thread safe
  - A fully thread-safe implementation will support `MPI_THREAD_MULTIPLE`
- Correctness of multiple threads calling MPI functions in parallel implies the same outcome as if the calls are executed in any sequential order

Issues with `MPI_THREAD_MULTIPLE`

- An MPI implementation is not required to support levels higher than `MPI_THREAD_SINGLE`
  - It is not required to be thread safe
  - A fully thread-safe implementation will support `MPI_THREAD_MULTIPLE`
- Correctness of multiple threads calling MPI functions in parallel implies the same outcome as if the calls are executed in any sequential order
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions

Issues with `MPI_THREAD_MULTIPLE`

- An MPI implementation is not required to support levels higher than `MPI_THREAD_SINGLE`
  - It is not required to be thread safe
  - A fully thread-safe implementation will support `MPI_THREAD_MULTIPLE`
- Correctness of multiple threads calling MPI functions in parallel implies the same outcome as if the calls are executed in any sequential order
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions
- MPI calls by different threads must not share the same requests

Issues with `MPI_THREAD_MULTIPLE`

- An MPI implementation is not required to support levels higher than `MPI_THREAD_SINGLE`
  - It is not required to be thread safe
  - A fully thread-safe implementation will support `MPI_THREAD_MULTIPLE`
- Correctness of multiple threads calling MPI functions in parallel implies the same outcome as if the calls are executed in any sequential order
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions
- MPI calls by different threads must not share the same requests
- Collective operations using the same communicator must be correctly ordered among threads
  - Cannot call a broadcast on one thread and a reduce on another thread on the same communicator.

Hybrid MPI/OpenMP

- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures

Hybrid MPI/OpenMP

- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures
- Elegant in concept and architecture
  - Using MPI across nodes and OpenMP within nodes
  - Good usage of shared memory system resource (memory, latency, and bandwidth)
  - Avoids the extra communication overhead with MPI within node

Hybrid MPI/OpenMP

- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures
- Elegant in concept and architecture
  - Using MPI across nodes and OpenMP within nodes
  - Good usage of shared memory system resource (memory, latency, and bandwidth)
  - Avoids the extra communication overhead with MPI within node
- OpenMP adds fine granularity and allows increased and/or dynamic load balancing

Hybrid MPI/OpenMP

- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures
- Elegant in concept and architecture
  - Using MPI across nodes and OpenMP within nodes
  - Good usage of shared memory system resource (memory, latency, and bandwidth)
  - Avoids the extra communication overhead with MPI within node
- OpenMP adds fine granularity and allows increased and/or dynamic load balancing
- Some problems have two-level parallelism naturally

Hybrid MPI/OpenMP

- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures
- Elegant in concept and architecture
  - Using MPI across nodes and OpenMP within nodes
  - Good usage of shared memory system resource (memory, latency, and bandwidth)
  - Avoids the extra communication overhead with MPI within node
- OpenMP adds fine granularity and allows increased and/or dynamic load balancing
- Some problems have two-level parallelism naturally
- Some problems could only use restricted number of MPI tasks

Hybrid MPI/OpenMP

- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures
- Elegant in concept and architecture
  - Using MPI across nodes and OpenMP within nodes
  - Good usage of shared memory system resource (memory, latency, and bandwidth)
  - Avoids the extra communication overhead with MPI within node
- OpenMP adds fine granularity and allows increased and/or dynamic load balancing
- Some problems have two-level parallelism naturally
- Some problems could only use restricted number of MPI tasks
- Could have better scalability than both pure MPI and pure OpenMP

Hybrid Parallelization Strategies

- From sequential code
  - Decompose with MPI first, then add OpenMP

Hybrid Parallelization Strategies

- From sequential code
  - Decompose with MPI first, then add OpenMP
    - Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks

Hybrid Parallelization Strategies

- From sequential code
  - Decompose with MPI first, then add OpenMP
    - Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks
- From OpenMP code, treat as serial code
  - Not as straightforward as adding OpenMP to an MPI application because global program state must be explicitly handled with MPI
  - Requires careful thought about how each process will communicate amongst one another
  - May require a complete reformulation of the parallelization, with a need to possibly redesign it from the ground up

Hybrid Parallelization Strategies

- From MPI code, add OpenMP
  - Is the easiest of the two options because the program state synchronization is already handled in an explicit way
  - Benefits depend on how many simple loops may be work-shared
  - The number of MPI processes per SMP node will depend on how many threads one wants to use per process

Example: Conjugate Gradient Method

- Conjugate Gradient Method: iterative method for efficiently solving linear systems of equations

$$Ax = b$$

- It can be demonstrated that the function

$$q(x) = \frac{1}{2}x^T Ax - x^T b + c$$

has a unique minimum, x, that is the solution to Ax=b.

Example: Conjugate Gradient Method

- Algorithm for Conjugate Gradient Method

1. Compute gradient: $g(t) = A\,x(t-1) - b$
2. If $g(t)^T g(t) < \epsilon$, stop
3. Compute direction vector: $d(t) = -g(t) + \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$
4. Compute step size: $s(t) = -\frac{d(t)^T g(t)}{d(t)^T A d(t)}$
5. Compute new approximation of x: $x(t) = x(t-1) + s(t)d(t)$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

Example: Conjugate Gradient Method

$g(t-1)^T g(t-1)$

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

$g(t) = A\,x(t-1) - b$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

$g(t) = A\,x(t-1) - b$

$g(t)^T g(t)$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

$g(t) = A\,x(t-1) - b$

$g(t)^T g(t)$

$g(t)^T g(t) < \epsilon$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

$g(t) = A\,x(t-1) - b$

$g(t)^T g(t)$

$g(t)^T g(t) < \epsilon$

$d(t) = -g(t) + \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

$g(t) = A\,x(t-1) - b$

$g(t)^T g(t)$

$g(t)^T g(t) < \epsilon$

$d(t) = -g(t) + \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$

$d(t)^T g(t)$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

$g(t) = A\,x(t-1) - b$

$g(t)^T g(t)$

$g(t)^T g(t) < \epsilon$

$d(t) = -g(t) + \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$

$d(t)^T g(t)$

$A\,d(t)$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

$g(t) = A\,x(t-1) - b$

$g(t)^T g(t)$

$g(t)^T g(t) < \epsilon$

$d(t) = -g(t) + \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$

$d(t)^T g(t)$

$A\,d(t)$

$d(t)^T A\,d(t)$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

$g(t) = A\,x(t-1) - b$

$g(t)^T g(t)$

$g(t)^T g(t) < \epsilon$

$d(t) = -g(t) + \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$

$d(t)^T g(t)$

$A\,d(t)$

$d(t)^T A\,d(t)$

$s(t) = -\frac{d(t)^T g(t)}{d(t)^T A d(t)}$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

$g(t) = A\,x(t-1) - b$

$g(t)^T g(t)$

$g(t)^T g(t) < \epsilon$

$d(t) = -g(t) + \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$

$d(t)^T g(t)$

$A\,d(t)$

$d(t)^T A\,d(t)$

$s(t) = -\frac{d(t)^T g(t)}{d(t)^T A d(t)}$

$x(t) = x(t-1) + s(t)d(t)$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

$g(t) = A\,x(t-1) - b$

$g(t)^T g(t)$

$g(t)^T g(t) < \epsilon$

$d(t) = -g(t) + \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$

$d(t)^T g(t)$

$A\,d(t)$

$d(t)^T A\,d(t)$

$x(t) = x(t-1) + s(t)d(t)$

$s(t) = -\frac{d(t)^T g(t)}{d(t)^T A d(t)}$

Example: Conjugate Gradient Method

```
for(it = 0; it < n; it++) {
    denom1 = dot_product(g, g, n);
    matrix_vector_product(id, p, n, a, x, g);
    for(i = 0; i < n; i++) g[i] -= b[i];
    num1 = dot_product(g, g, n);

    /* When g is sufficiently close to 0, it is time to halt */
    if(num1 < EPSILON) break;

    for(i = 0; i < n; i++)
        d[i] = -g[i] + (num1/denom1) * d[i];
    num2 = dot_product(d, g, n);
    matrix_vector_product(id, p, n, a, d, tmpvec);
    denom2 = dot_product(d, tmpvec, n);
    s = -num2 / denom2;
    for(i = 0; i < n; i++) x[i] += s * d[i];
}
```

$g(t-1)^T g(t-1)$

$A\,x(t-1)$

$g(t) = A\,x(t-1) - b$

$g(t)^T g(t)$

$g(t)^T g(t) < \epsilon$

$d(t) = -g(t) + \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$

$d(t)^T g(t)$

$A\,d(t)$

$d(t)^T A\,d(t)$

$x(t) = x(t-1) + s(t)d(t)$

$s(t) = -\frac{d(t)^T g(t)}{d(t)^T A d(t)}$

Example: Conjugate Gradient Method

- MPI implementation: program based on rowwise block decomposition of matrix A, and replication of vector b

- Results of profiling:

Function	1 CPU	8 CPUs
`matrix_vector_product`	99,5%	97,5%
`dot_product`	0,2%	1,1%
`cg`	0,3%	1,4%

Example: Conjugate Gradient Method

```
void matrix_vector_product (int id, int p, int n,
                            double **a, double *b, double *c)
{
  int i, j;
  double tmp;

  /* Accumulates sum */
  for(i = 0; i < BLOCK_SIZE(id,p,n); i++) {
    tmp = 0.0;
    for(j = 0; j < n; j++)
      tmp += a[i][j] * b[j];
    piece[i] = tmp;
  }
  replicate_vector(id, p, piece, n, c);
}
```

Example: Conjugate Gradient Method

```
void matrix_vector_product (int id, int p, int n,
                            double **a, double *b, double *c)
{
  int i, j;
  double tmp;

  /* Accumulates sum */
  for(i = 0; i < BLOCK_SIZE(id,p,n); i++) {
    tmp = 0.0;
    for(j = 0; j < n; j++)
      tmp += a[i][j] * b[j];
    piece[i] = tmp;
  }
  replicate_vector(id, p, piece, n, c);
}
```

AllGather operation necessary to replicate the result vector

Example: Conjugate Gradient Method

- Adding OpenMP directives
  - Make outermost loop parallel
    - Outer loop may be executed in parallel if each thread has a private copy of `tmp` and `j`

```
#pragma omp parallel for private(j,tmp)
  for(i = 0; i < BLOCK_SIZE(id,p,n); i++)
```

  - Specify number of active threads per process

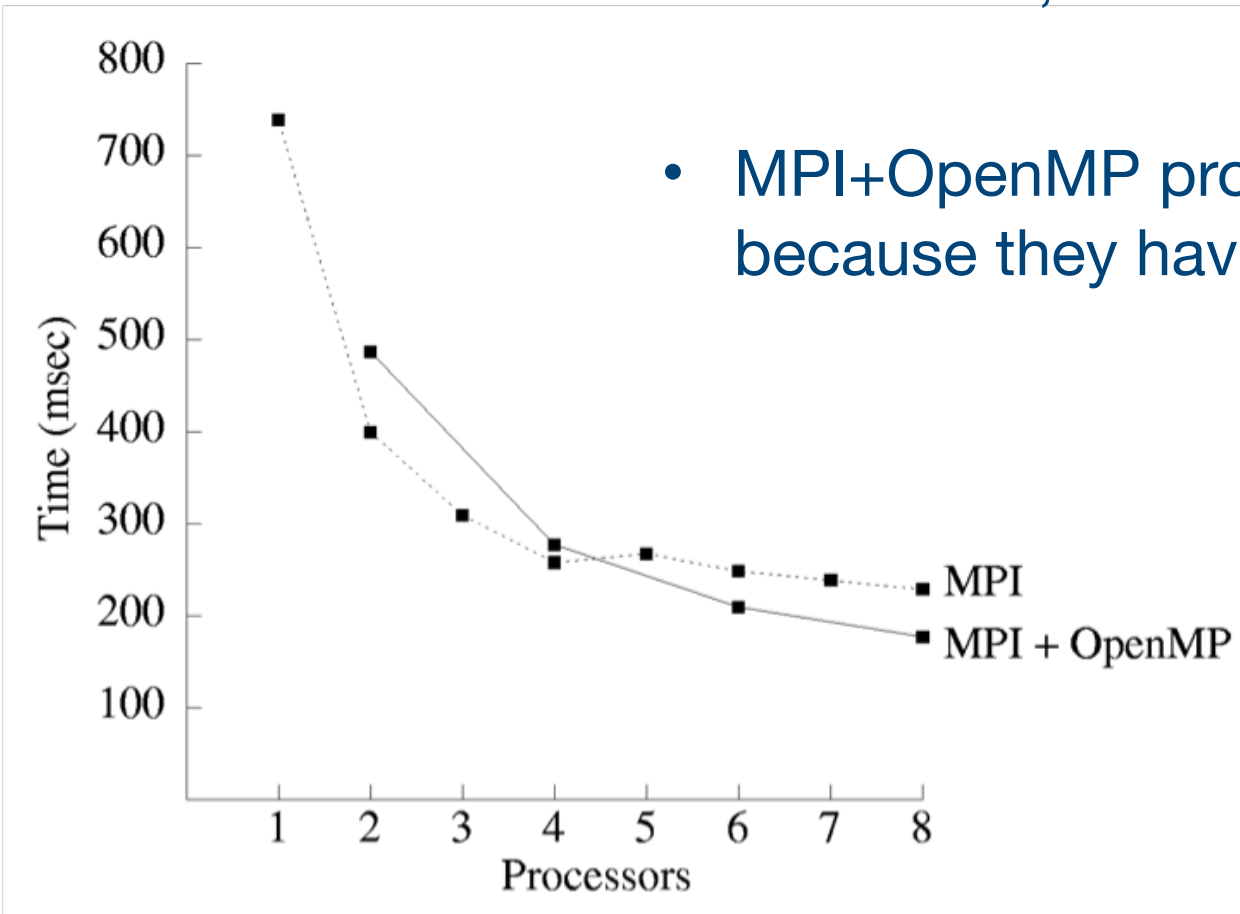
```
omp_set_num_threads(atoi(argv[3]));
```

Example: Conjugate Gradient Method

- Target system: a commodity cluster with four dual-processor nodes
- Pure MPI
  - Program executes on 1, 2, ..., 8 CPUs
  - On 1, 2, 3, 4 CPUs, each process on different node, maximizing memory bandwidth per CPU
- MPI + OpenMP
  - Program executes on 1, 2, 3, 4 processes, each process has two threads (program executes on 2, 4, 6, 8 threads)

Example: Conjugate Gradient Method

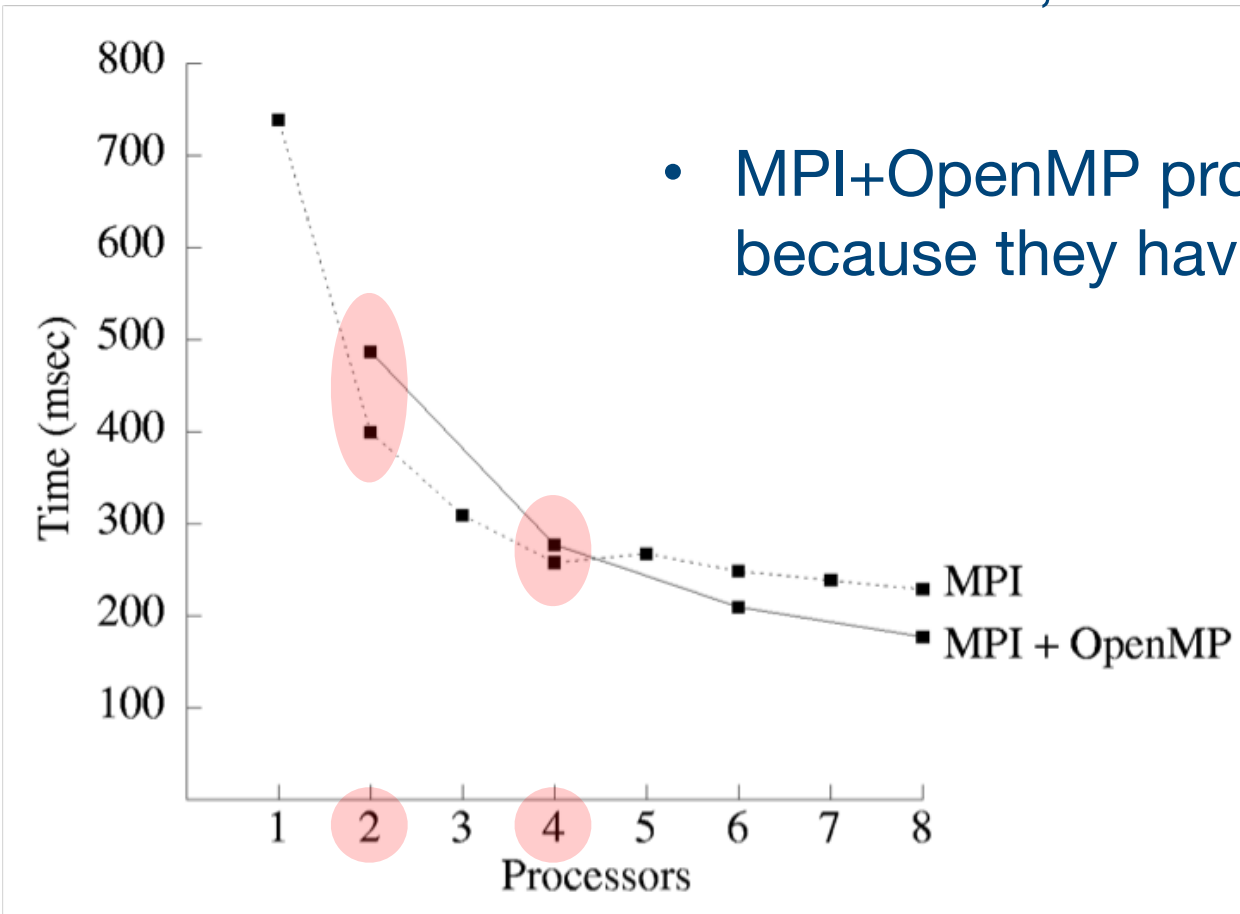
- MPI+OpenMP program slower on 2, 4 CPUs because threads are sharing memory bandwidth, while MPI processes are not



- MPI+OpenMP programs faster on 6, 8 CPUs because they have lower communication cost

Example: Conjugate Gradient Method

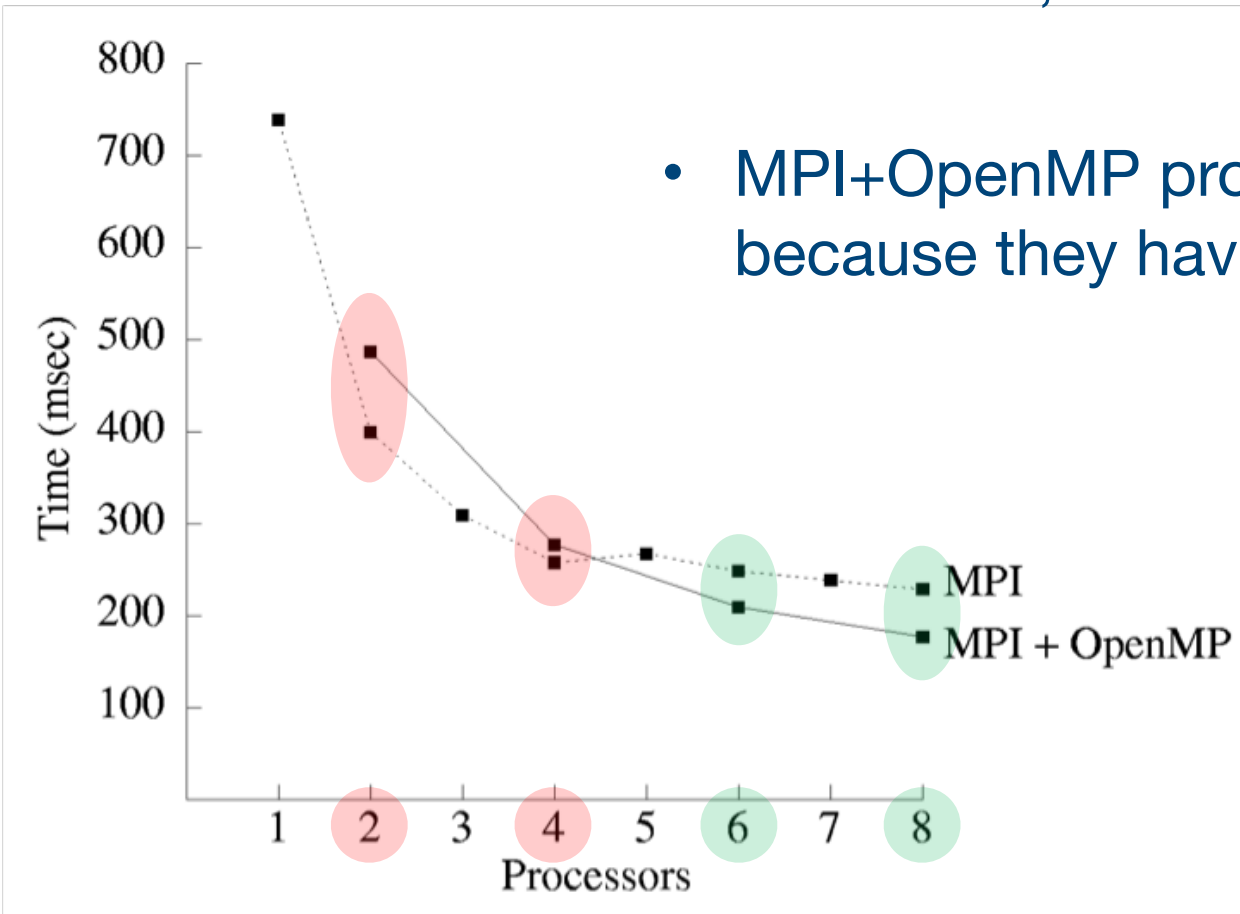
- MPI+OpenMP program slower on 2, 4 CPUs because threads are sharing memory bandwidth, while MPI processes are not



- MPI+OpenMP programs faster on 6, 8 CPUs because they have lower communication cost

Example: Conjugate Gradient Method

- MPI+OpenMP program slower on 2, 4 CPUs because threads are sharing memory bandwidth, while MPI processes are not
- MPI+OpenMP programs faster on 6, 8 CPUs because they have lower communication cost



Foster's Design Methodology

- Development of scalable parallel algorithms by delaying machine-dependent decisions to later stages

Four steps:

- Partitioning
- Communication
- Agglomeration
- Mapping

Mapping

- Examples discussed in previous classes:
  - Circuit satisfiability
  - Sieve of Eratosthenes
  - All pairs shortest paths
  - Matrix-vector multiplication

Mapping

- Examples discussed in previous classes:
  - Circuit satisfiability
  - Sieve of Eratosthenes
  - All pairs shortest paths
  - Matrix-vector multiplication
- So far, all primitive tasks require same amount of computation

Mapping

- Examples discussed in previous classes:
  - Circuit satisfiability
  - Sieve of Eratosthenes
  - All pairs shortest paths
  - Matrix-vector multiplication
- So far, all primitive tasks require same amount of computation
  - Agglomeration strategy: cluster primitive tasks evenly, taking into account communication

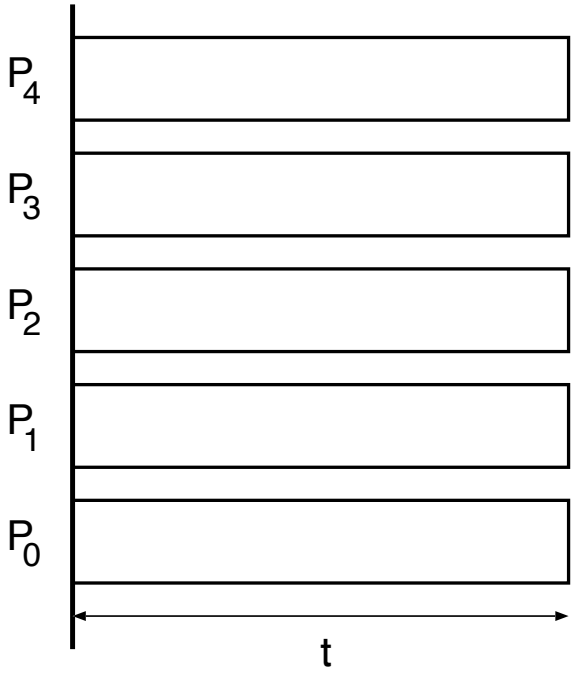
Mapping

- Examples discussed in previous classes:
  - Circuit satisfiability
  - Sieve of Eratosthenes
  - All pairs shortest paths
  - Matrix-vector multiplication
- So far, all primitive tasks require same amount of computation
  - Agglomeration strategy: cluster primitive tasks evenly, taking into account communication
  - Mapping strategy: create one task per processor, distributing tasks evenly.

Mapping

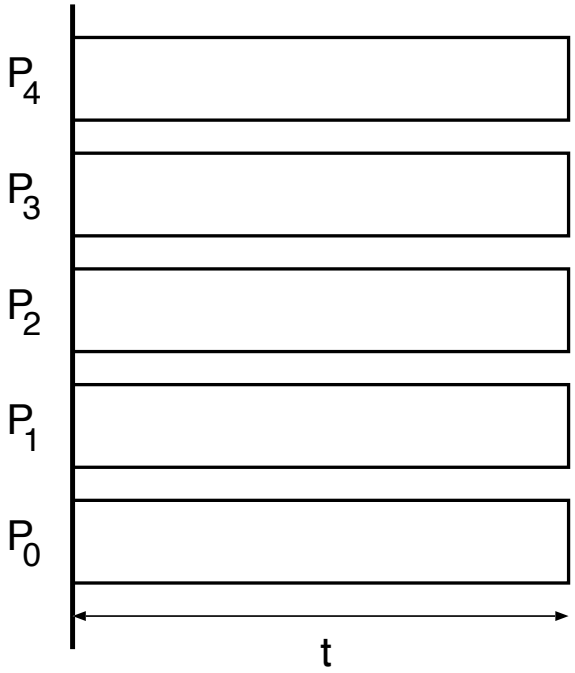
- Examples discussed in previous classes:
  - Circuit satisfiability
  - Sieve of Eratosthenes
  - All pairs shortest paths
  - Matrix-vector multiplication
- So far, all primitive tasks require same amount of computation
  - Agglomeration strategy: cluster primitive tasks evenly, taking into account communication
  - Mapping strategy: create one task per processor, distributing tasks evenly.
- What if we can not predict beforehand the amount of computation required per primitive task?

Load Balancing

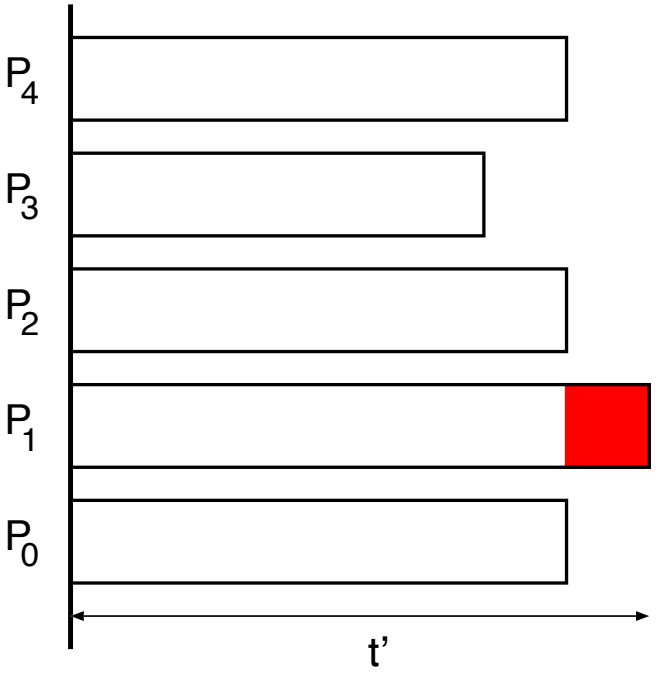


Perfect load balancing

Load Balancing

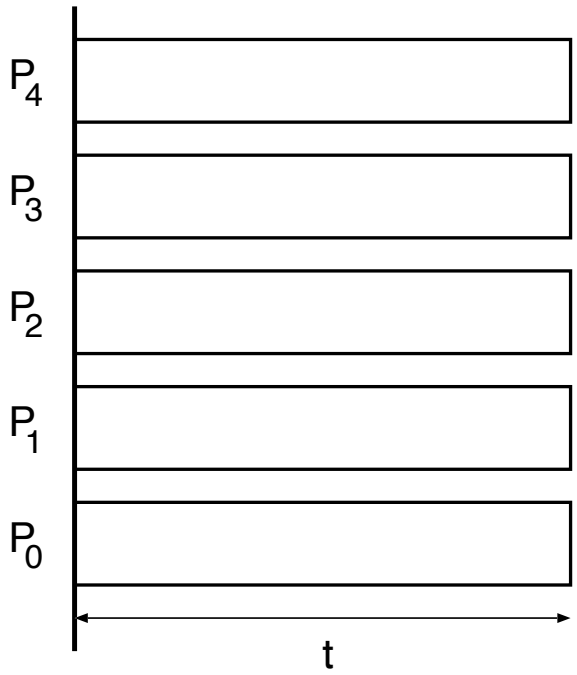


Perfect load balancing

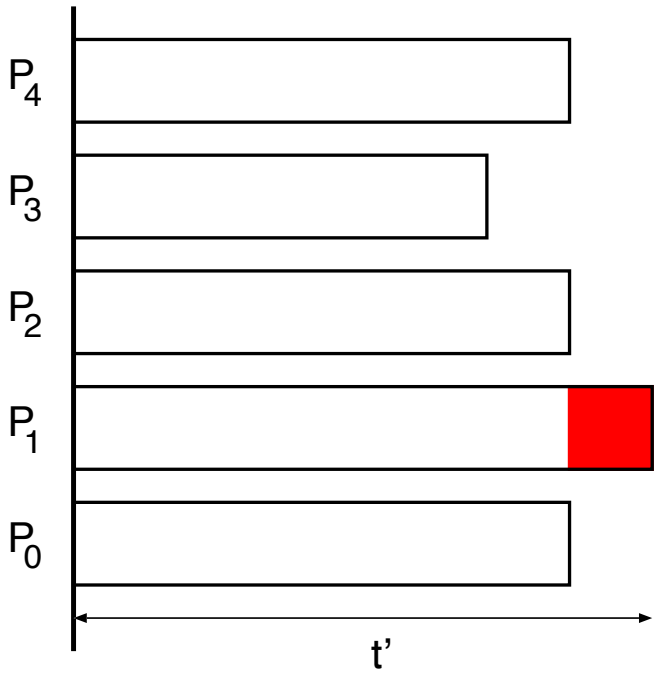


Imperfect load balancing

Load Balancing



Perfect load balancing



Imperfect load balancing

- Parallel execution time defined by last task to finish
- Parallel time minimized when load distributed evenly

Limitations of Static Load Balancing

- Computational effort of each task may not be known a-priori
  - Some problems have an indeterminate number of steps to complete

Limitations of Static Load Balancing

- Computational effort of each task may not be known a-priori
  - Some problems have an indeterminate number of steps to complete
- Programs are subject to variable communication delays

Limitations of Static Load Balancing

- Computational effort of each task may not be known a-priori
  - Some problems have an indeterminate number of steps to complete
- Programs are subject to variable communication delays
- Performance of each processor may vary, and may not be known beforehand

Limitations of Static Load Balancing

- Computational effort of each task may not be known a-priori
  - Some problems have an indeterminate number of steps to complete
- Programs are subject to variable communication delays
- Performance of each processor may vary, and may not be known beforehand
- Dynamic load balancing overcomes these issues by making the division of the load dependent on the actual runtimes
  - → Penalty: additional overhead due to task management

Dynamic Load Balancing Management

- Centralized management
  - One process (master) is responsible for assigning tasks to slave processes
- Decentralized management
  - All processes are equal and divide work among them cooperatively

Centralized Dynamic Load Balancing

- Work Pool or Processor Farm model

Centralized Dynamic Load Balancing

- Work Pool or Processor Farm model
  - Master holds all tasks of the application

Centralized Dynamic Load Balancing

- Work Pool or Processor Farm model
  - Master holds all tasks of the application
  - New tasks may be generated during execution

Centralized Dynamic Load Balancing

- Work Pool or Processor Farm model
  - Master holds all tasks of the application
  - New tasks may be generated during execution
  - When idle, slave process requests a task to the master

Centralized Dynamic Load Balancing

- Work Pool or Processor Farm model
  - Master holds all tasks of the application
  - New tasks may be generated during execution
  - When idle, slave process requests a task to the master
  - Master selects tasks among those ready to run and sends to slave
    - Tasks of larger size or importance are executed first
    - If tasks are all the same, a FIFO queue may be used

Centralized Dynamic Load Balancing

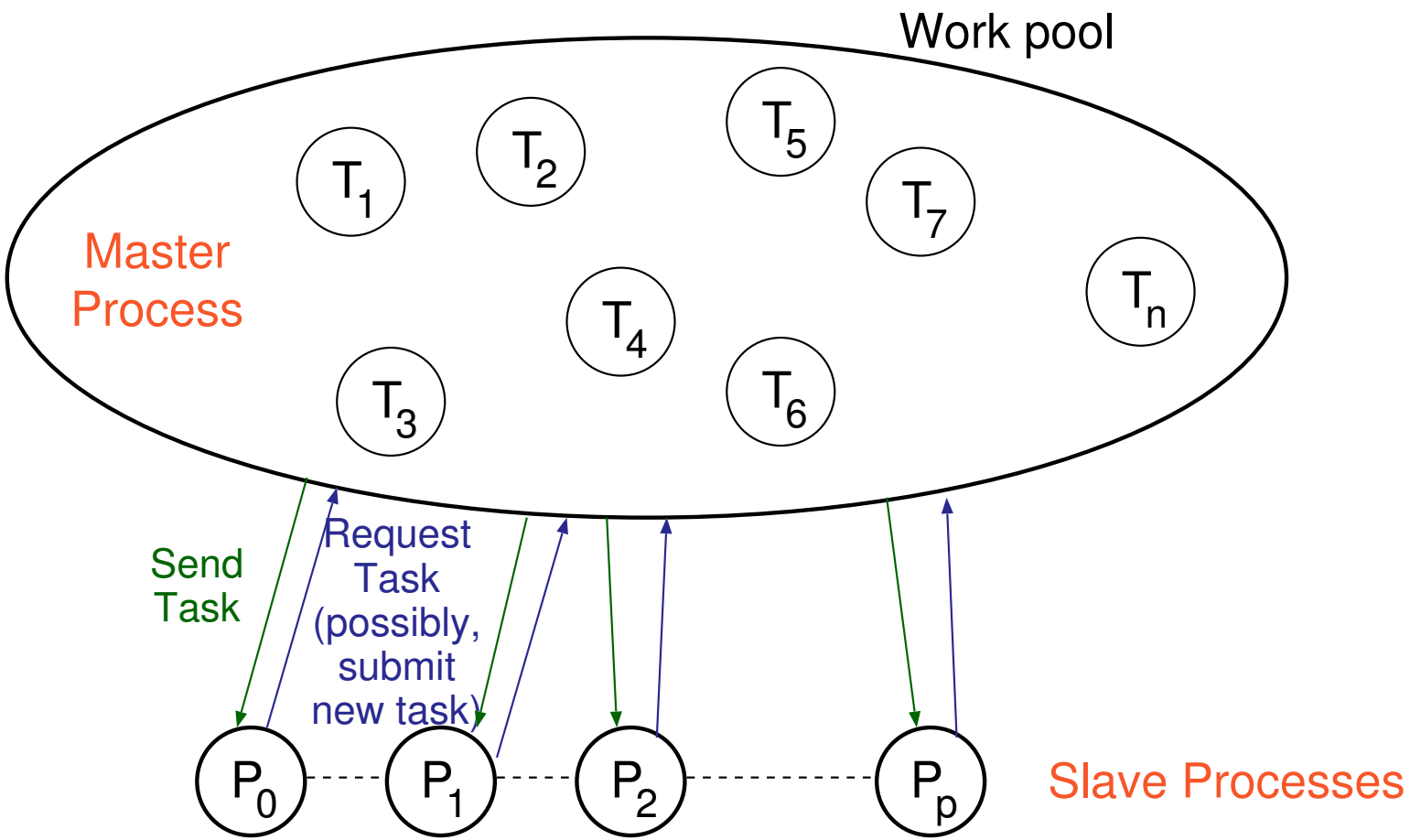
- Work Pool or Processor Farm model
  - Master holds all tasks of the application
  - New tasks may be generated during execution
  - When idle, slave process requests a task to the master
  - Master selects tasks among those ready to run and sends to slave
    - Tasks of larger size or importance are executed first
    - If tasks are all the same, a FIFO queue may be used
  - Specialized slaves can be considered

Centralized Dynamic Load Balancing

- Work Pool or Processor Farm model
  - Master holds all tasks of the application
  - New tasks may be generated during execution
  - When idle, slave process requests a task to the master
  - Master selects tasks among those ready to run and sends to slave
    - Tasks of larger size or importance are executed first
    - If tasks are all the same, a FIFO queue may be used
  - Specialized slaves can be considered
  - Master can also hold global data

Centralized Dynamic Load Balancing

- Work Pool or Processor Farm model



Centralized Dynamic Load Balancing

- Limitations of centralized dynamic load balancing

Centralized Dynamic Load Balancing

- Limitations of centralized dynamic load balancing
  - Master can become a bottleneck as it can only issue one task at a time

Centralized Dynamic Load Balancing

- Limitations of centralized dynamic load balancing
  - Master can become a bottleneck as it can only issue one task at a time
    - Not a significant problem if few slaves and/or computationally intensive tasks

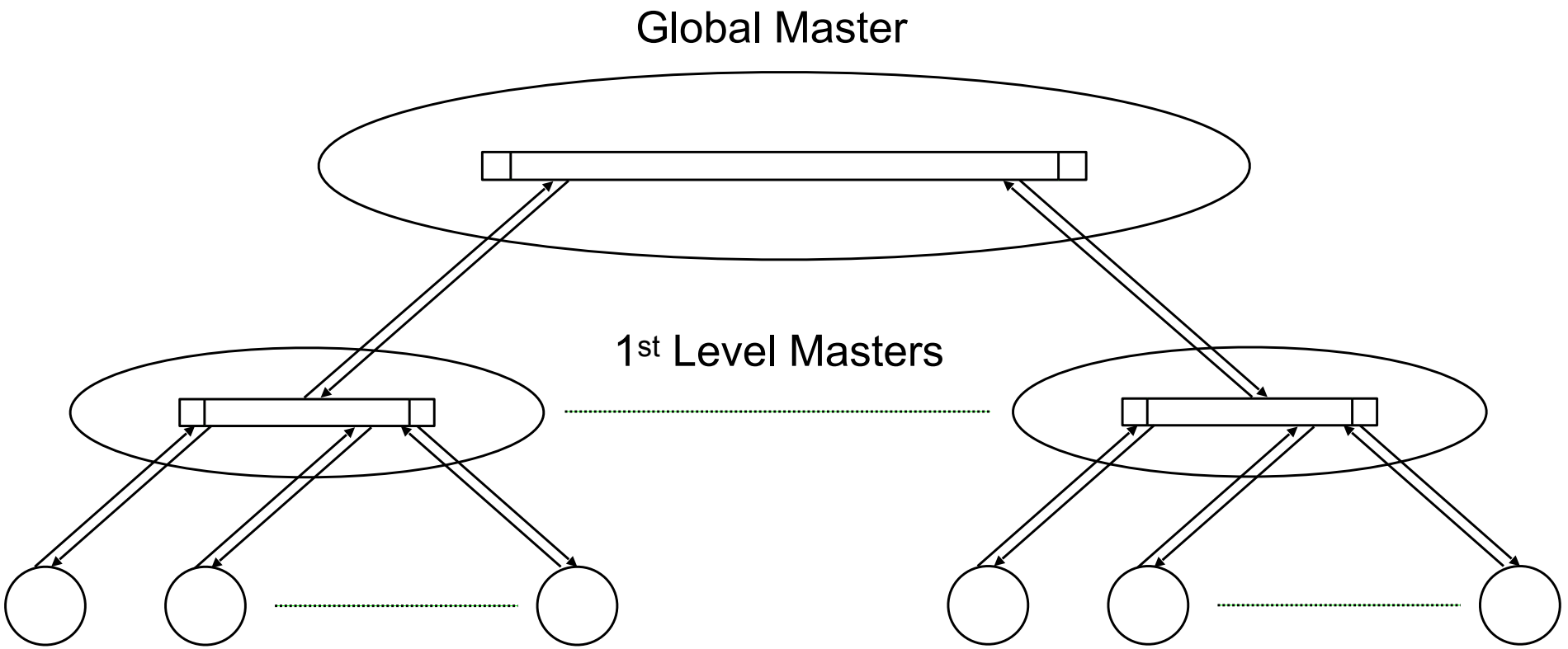
Centralized Dynamic Load Balancing

- Limitations of centralized dynamic load balancing
  - Master can become a bottleneck as it can only issue one task at a time
    - Not a significant problem if few slaves and/or computationally intensive tasks
    - For finer-grained tasks and many slaves, distribute task management

Decentralized Dynamic Load Balancing

- Initial approach can be simply to evolve the centralized system into a tree-like task management scheme
  - Main master distributes tasks to be managed by second-level masters
  - Each second-level master behaves as in the centralized method
- Naturally, this model can be extended to as many levels as desired

Decentralized Dynamic Load Balancing



Decentralized Dynamic Load Balancing

- Fully distributed work pool: each process starts with a given number of tasks, but may send or receive new tasks to handle

Decentralized Dynamic Load Balancing

- Fully distributed work pool: each process starts with a given number of tasks, but may send or receive new tasks to handle
  - Receiver-initiated method: process requests new tasks when it has few or no tasks to do

Decentralized Dynamic Load Balancing

- Fully distributed work pool: each process starts with a given number of tasks, but may send or receive new tasks to handle
  - Receiver-initiated method: process requests new tasks when it has few or no tasks to do
    - Better in high load systems

Decentralized Dynamic Load Balancing

- Fully distributed work pool: each process starts with a given number of tasks, but may send or receive new tasks to handle
  - Receiver-initiated method: process requests new tasks when it has few or no tasks to do
    - Better in high load systems
  - Sender-initiated method: process under heavy load sends tasks to processes willing to accept them

Decentralized Dynamic Load Balancing

- Fully distributed work pool: each process starts with a given number of tasks, but may send or receive new tasks to handle
  - Receiver-initiated method: process requests new tasks when it has few or no tasks to do
    - Better in high load systems
  - Sender-initiated method: process under heavy load sends tasks to processes willing to accept them
  - A mixture of both approaches is possible

Decentralized Dynamic Load Balancing

- Fully distributed work pool: each process starts with a given number of tasks, but may send or receive new tasks to handle
  - Receiver-initiated method: process requests new tasks when it has few or no tasks to do
    - Better in high load systems
  - Sender-initiated method: process under heavy load sends tasks to processes willing to accept them
  - A mixture of both approaches is possible
  - Which process to send/request?

Decentralized Dynamic Load Balancing

- Fully distributed work pool: each process starts with a given number of tasks, but may send or receive new tasks to handle
  - Receiver-initiated method: process requests new tasks when it has few or no tasks to do
    - Better in high load systems
  - Sender-initiated method: process under heavy load sends tasks to processes willing to accept them
  - A mixture of both approaches is possible
  - Which process to send/request?
    - Round-robin, random

Decentralized Dynamic Load Balancing

- Fully distributed work pool: each process starts with a given number of tasks, but may send or receive new tasks to handle
  - Receiver-initiated method: process requests new tasks when it has few or no tasks to do
    - Better in high load systems
  - Sender-initiated method: process under heavy load sends tasks to processes willing to accept them
  - A mixture of both approaches is possible
  - Which process to send/request?
    - Round-robin, random
    - Send to neighbors (dependent on network topology)

Termination Detection

- When can we be sure that the computation has finished?

Termination Detection

- When can we be sure that the computation has finished?
  - Static load balancing
    - Typically termination is easy to detect as layout of application is fixed and well controlled

Termination Detection

- When can we be sure that the computation has finished?
  - Static load balancing
    - Typically termination is easy to detect as layout of application is fixed and well controlled
  - Centralized dynamic load balancing
    - Also easy for master to recognize termination
      - Task queue is empty
      - Every slave is idle, having requested another task, and without generating any new task
    - Alternatively, if a slave indicates that a solution has been found, master can terminate all slaves

Termination Detection

- When can we be sure that the computation has finished?
  - Static load balancing
    - Typically termination is easy to detect as layout of application is fixed and well controlled
  - Centralized dynamic load balancing
    - Also easy for master to recognize termination
      - Task queue is empty
      - Every slave is idle, having requested another task, and without generating any new task
    - Alternatively, if a slave indicates that a solution has been found, master can terminate all slaves
  - Decentralized dynamic load balancing
    - Not so trivial...

Termination Detection

- In general, termination at time *t* requires two conditions

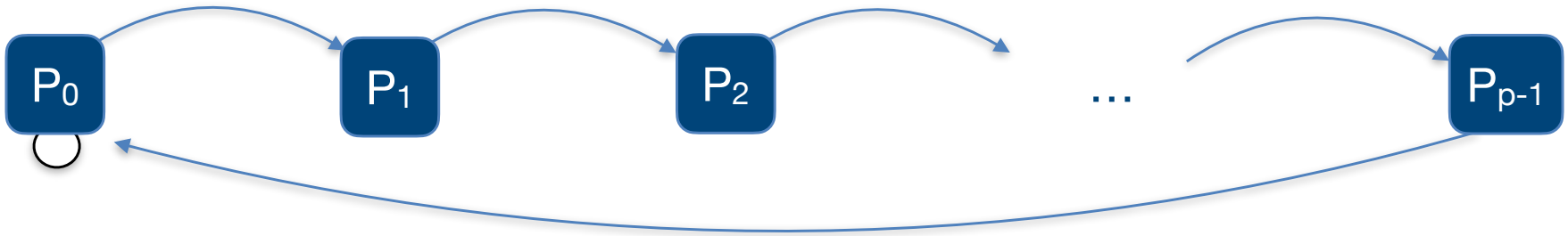
  (1) Application-specific local termination conditions exist for all processes at time *t*

  (2) There are no messages in transit at time *t*

- Second condition is what makes this problem difficult
  - Wait long enough before assuming program has finished?

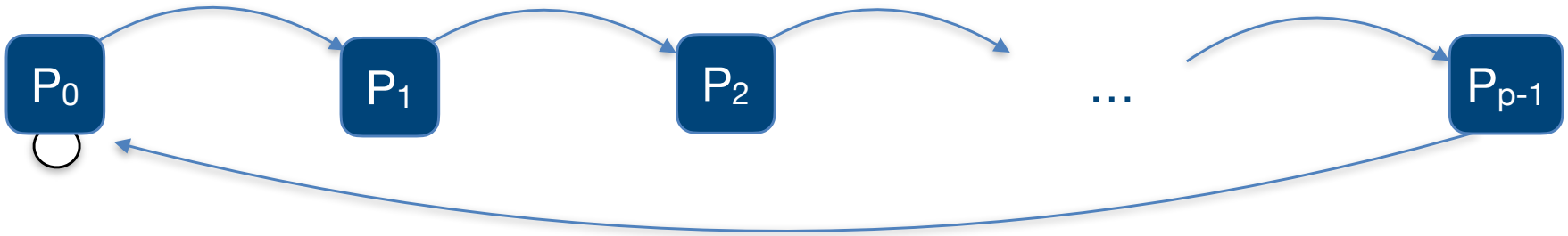
Termination Detection

- Ring Termination Algorithm



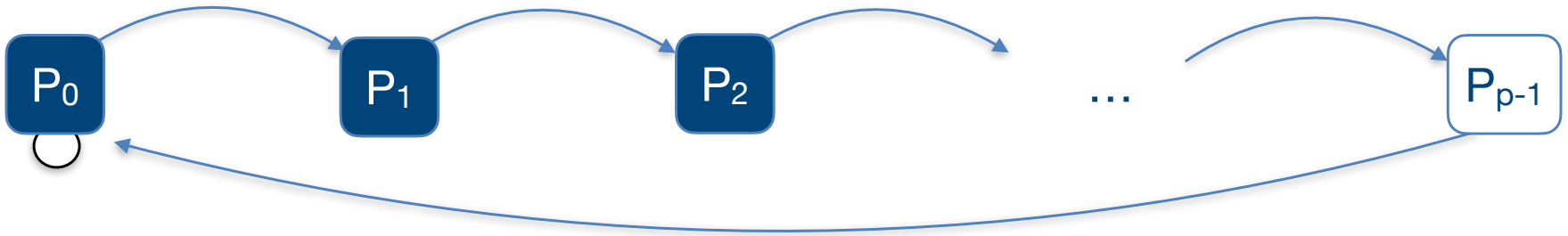
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate



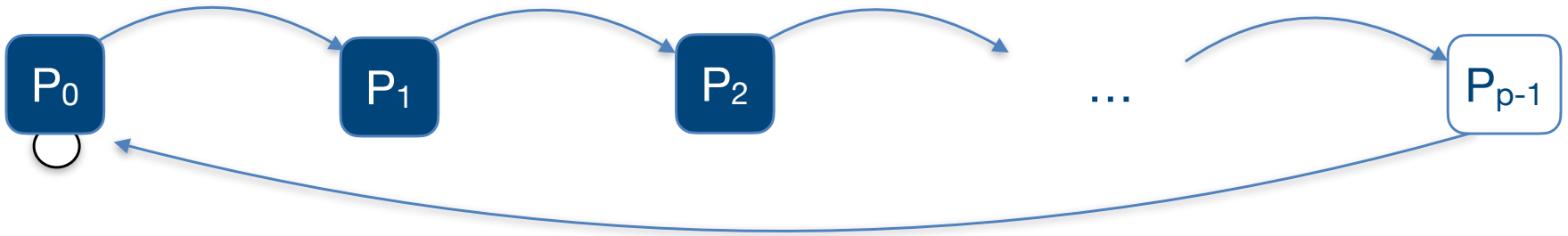
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate



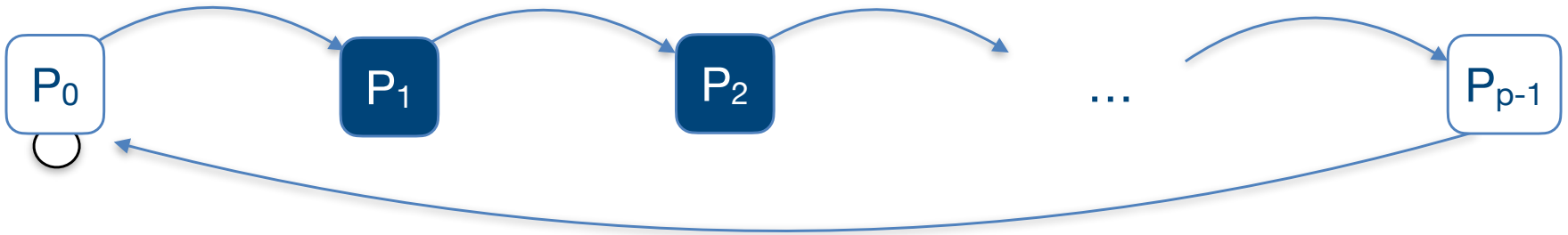
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1



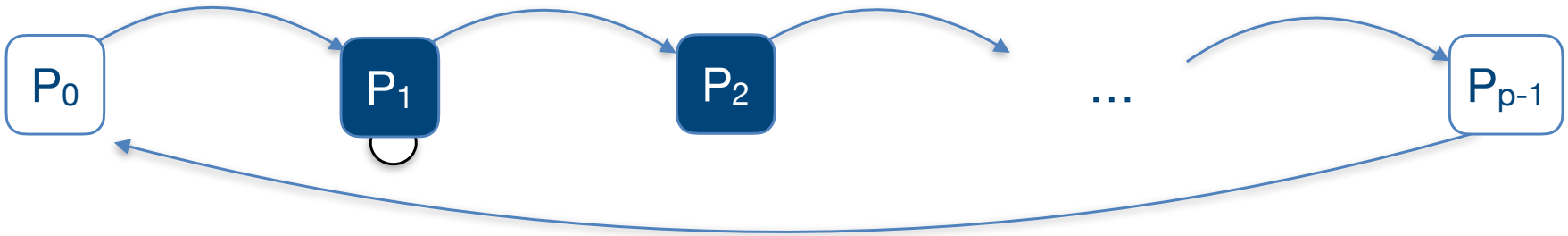
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1



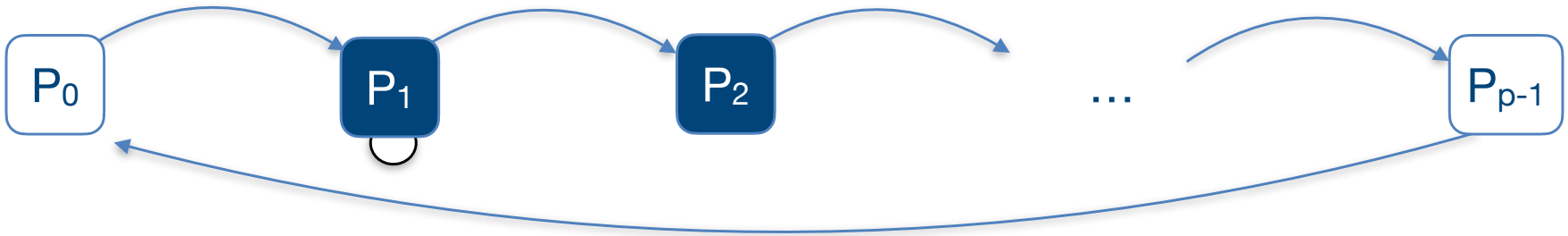
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1



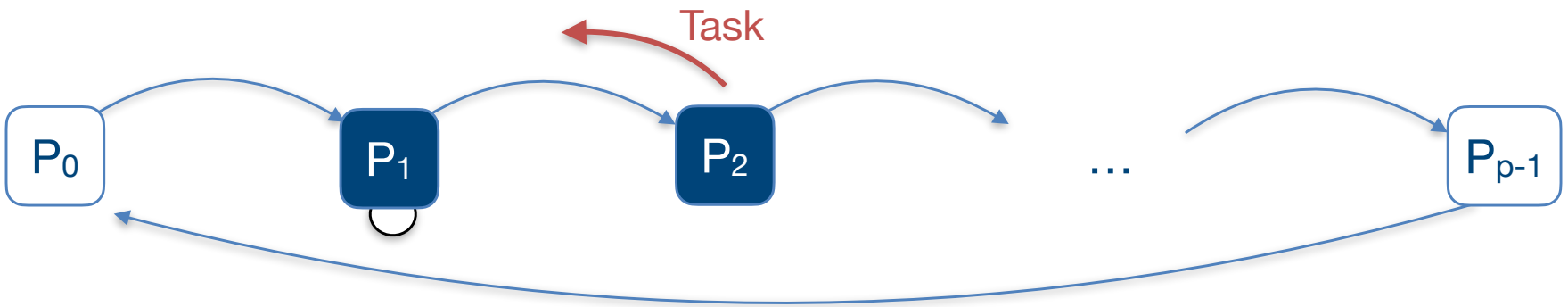
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$



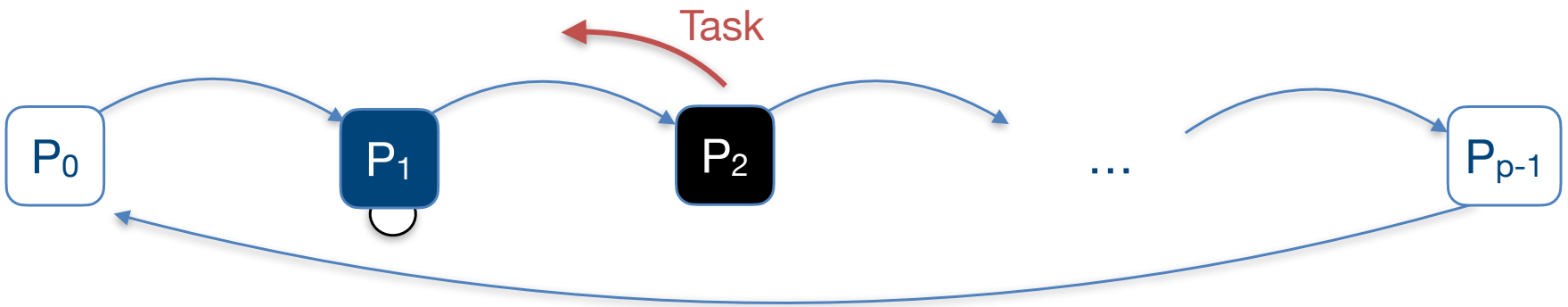
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$



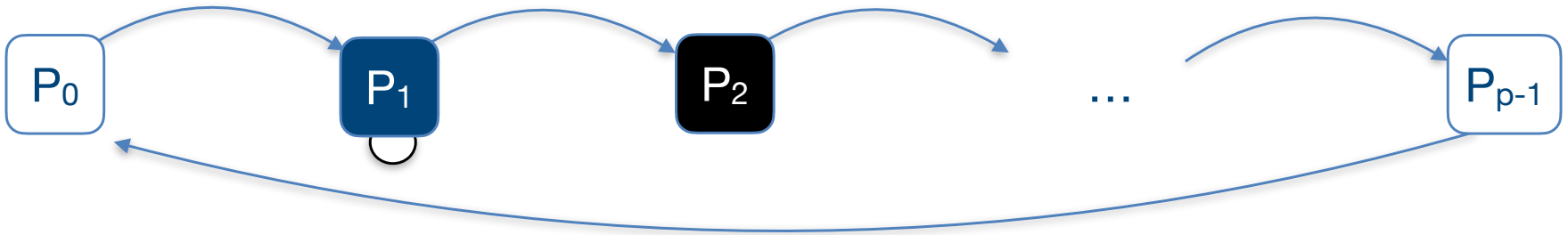
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$



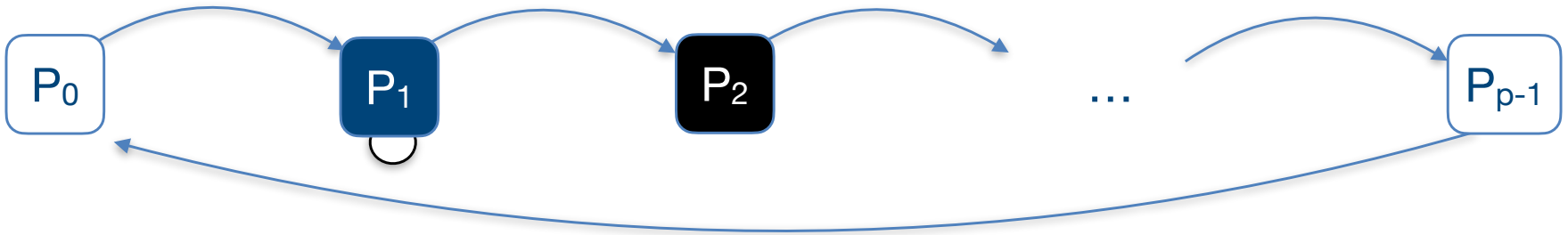
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$



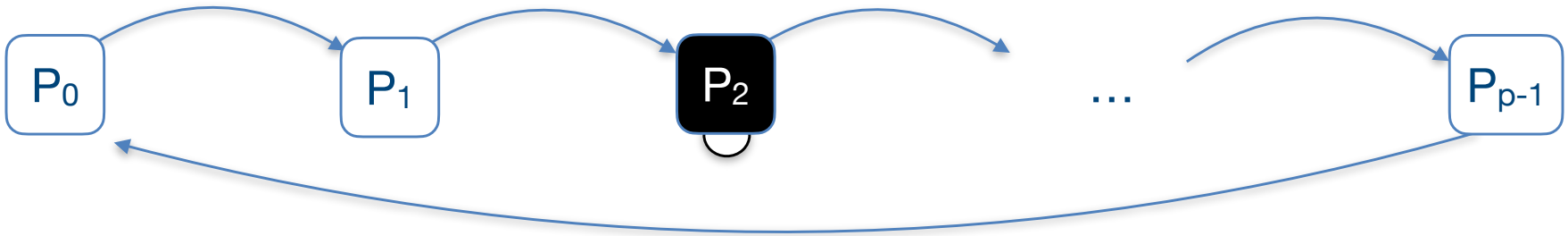
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$
  - the token is passed to the next process in the ring when the process finishes:



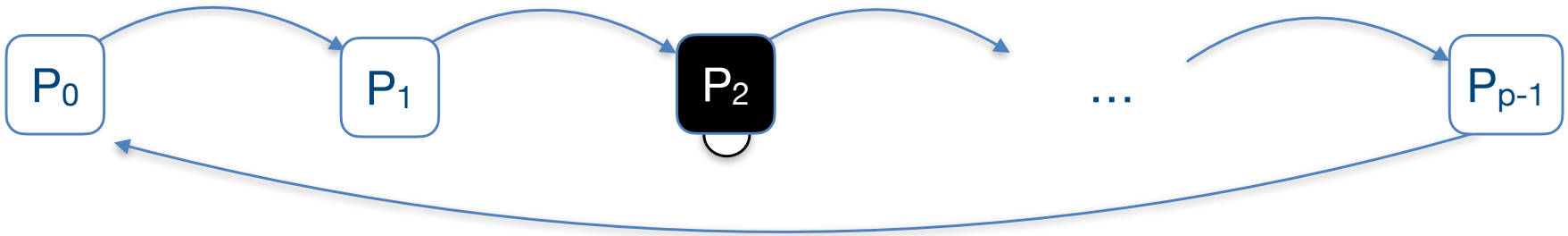
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$
  - the token is passed to the next process in the ring when the process finishes:



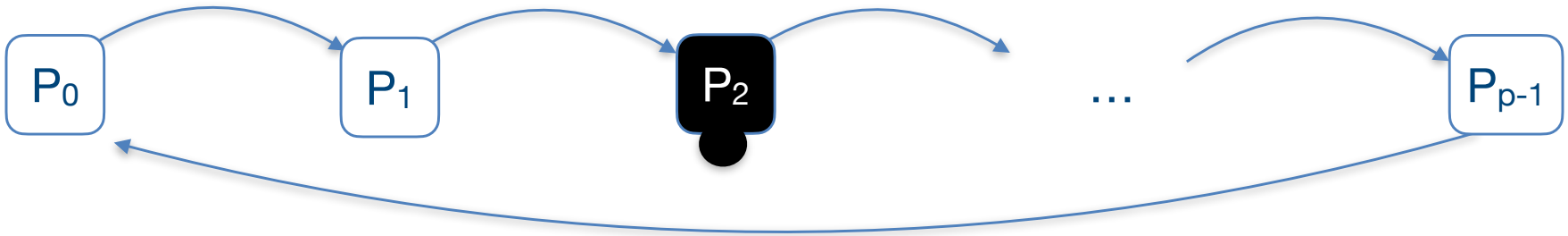
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$
  - the token is passed to the next process in the ring when the process finishes:
    - if the color of the process is black, the token becomes black otherwise, the token keeps the same color



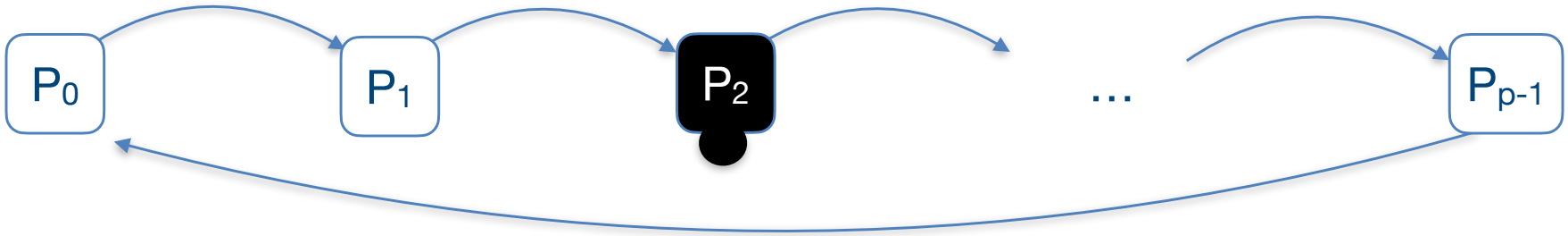
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$
  - the token is passed to the next process in the ring when the process finishes:
    - if the color of the process is black, the token becomes black otherwise, the token keeps the same color



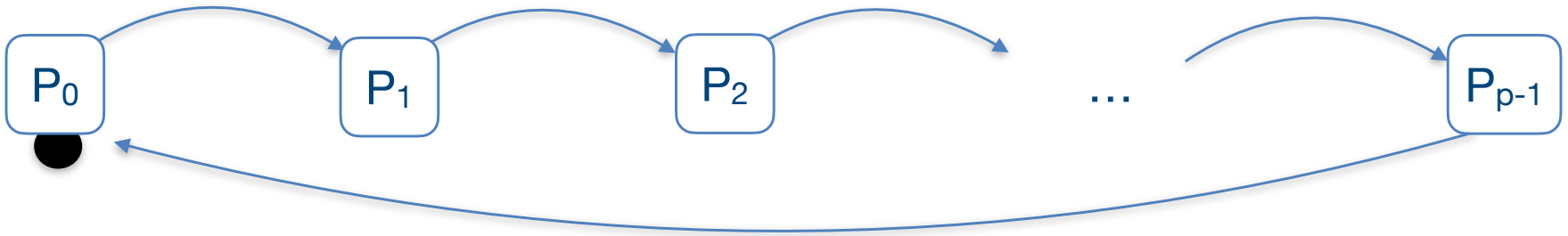
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$
  - the token is passed to the next process in the ring when the process finishes:
    - if the color of the process is black, the token becomes black otherwise, the token keeps the same color
  - a black process becomes white when it passes the token



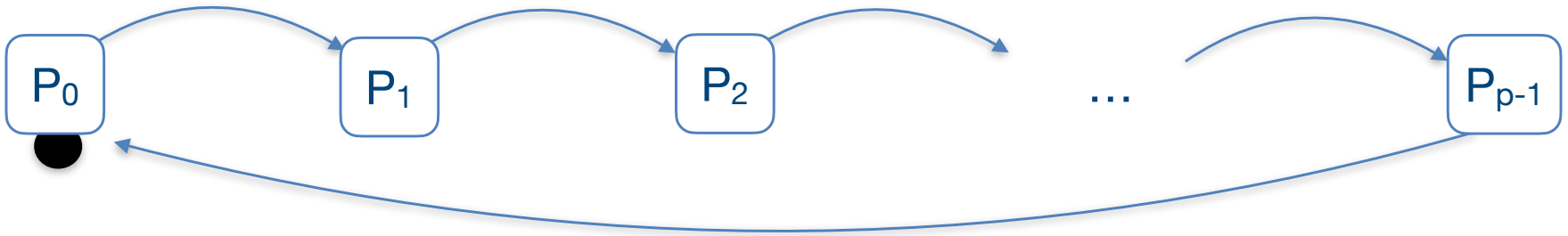
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$
  - the token is passed to the next process in the ring when the process finishes:
    - if the color of the process is black, the token becomes black otherwise, the token keeps the same color
  - a black process becomes white when it passes the token



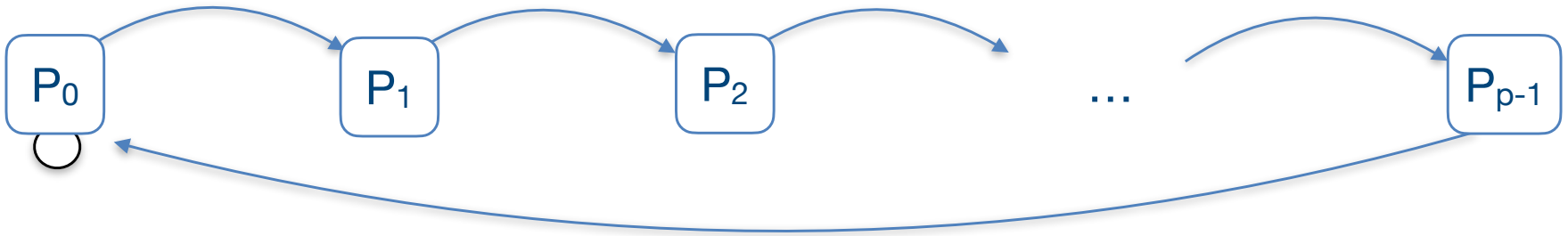
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$
  - the token is passed to the next process in the ring when the process finishes:
    - if the color of the process is black, the token becomes black otherwise, the token keeps the same color
  - a black process becomes white when it passes the token
  - when P_0 receives a black token, it passes a white token



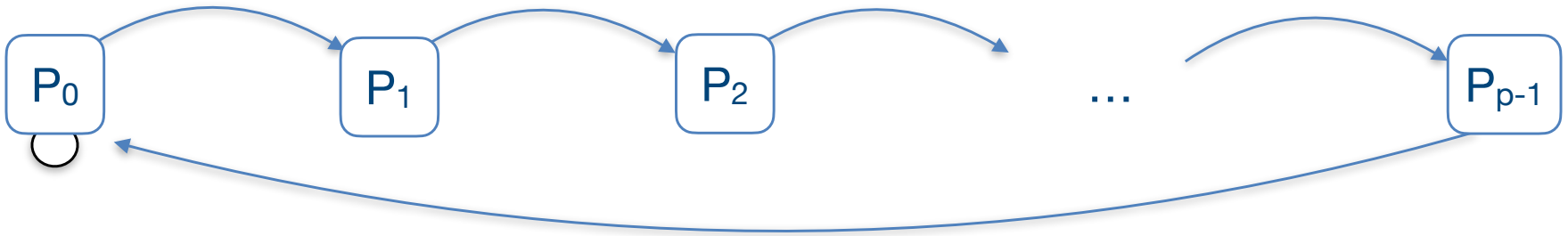
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$
  - the token is passed to the next process in the ring when the process finishes:
    - if the color of the process is black, the token becomes black otherwise, the token keeps the same color
  - a black process becomes white when it passes the token
  - when P_0 receives a black token, it passes a white token



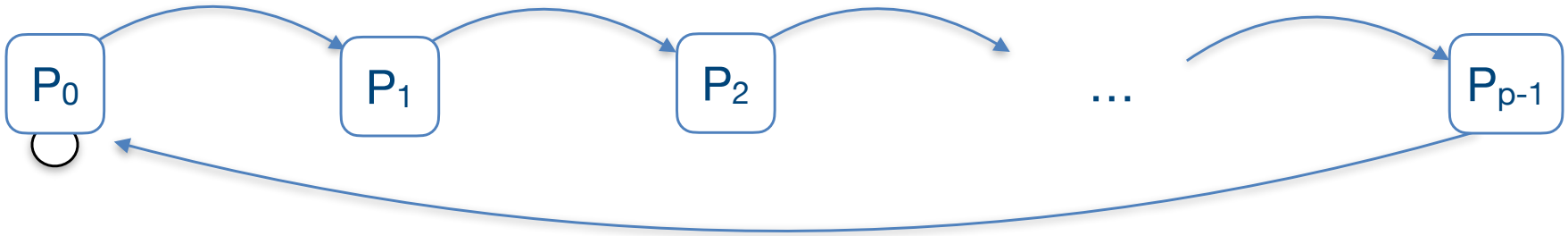
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$
  - the token is passed to the next process in the ring when the process finishes:
    - if the color of the process is black, the token becomes black otherwise, the token keeps the same color
  - a black process becomes white when it passes the token
  - when P_0 receives a black token, it passes a white token



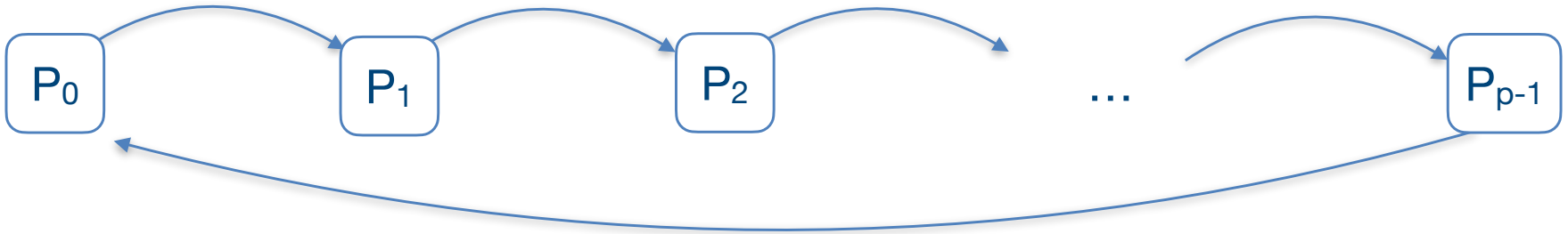
Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$
  - the token is passed to the next process in the ring when the process finishes:
    - if the color of the process is black, the token becomes black otherwise, the token keeps the same color
  - a black process becomes white when it passes the token
  - when P_0 receives a black token, it passes a white token
  - when P_0 receives a white token, computation can terminate



Termination Detection

- Ring Termination Algorithm
  - processes become white when they terminate
  - when P_0 becomes white, it sends a white token to P_1
  - a process P_i becomes black if it sends a message to process P_j and $j<i$
  - the token is passed to the next process in the ring when the process finishes:
    - if the color of the process is black, the token becomes black otherwise, the token keeps the same color
  - a black process becomes white when it passes the token
  - when P_0 receives a black token, it passes a white token
  - when P_0 receives a white token, computation can terminate



Debugging

- Most common tool for debugging of MPI programs: `printf`!

Debugging

- Most common tool for debugging of MPI programs: `printf`!
  - Redirection of `stdio`: MPI takes care of this

Debugging

- Most common tool for debugging of MPI programs: `printf`!
  - Redirection of `stdio`: MPI takes care of this
  - Don’t forget printf is buffered! (use `fflush`, `\n`)

Debugging

- Most common tool for debugging of MPI programs: `printf`!
  - Redirection of `stdio`: MPI takes care of this
  - Don’t forget printf is buffered! (use `fflush`, `\n`)
- Program runs slower. Difficulties for parallel debugging?

Debugging

- Most common tool for debugging of MPI programs: `printf`!
  - Redirection of `stdio`: MPI takes care of this
  - Don’t forget printf is buffered! (use `fflush`, `\n`)
- Program runs slower. Difficulties for parallel debugging?
  - By delaying some tasks, behavior of program may change

Debugging

- Most common tool for debugging of MPI programs: `printf`!
  - Redirection of `stdio`: MPI takes care of this
  - Don’t forget printf is buffered! (use `fflush`, `\n`)
- Program runs slower. Difficulties for parallel debugging?
  - By delaying some tasks, behavior of program may change
- Debuggers can be used to examine the execution of individual processes, however

Debugging

- Most common tool for debugging of MPI programs: `printf`!
  - Redirection of `stdio`: MPI takes care of this
  - Don't forget printf is buffered! (use `fflush`, `\n`)
- Program runs slower. Difficulties for parallel debugging?
  - By delaying some tasks, behavior of program may change
- Debuggers can be used to examine the execution of individual processes, however
  - They change program execution times significantly

Debugging

- Most common tool for debugging of MPI programs: `printf`!
  - Redirection of `stdio`: MPI takes care of this
  - Don’t forget printf is buffered! (use `fflush`, `\n`)
- Program runs slower. Difficulties for parallel debugging?
  - By delaying some tasks, behavior of program may change
- Debuggers can be used to examine the execution of individual processes, however
  - They change program execution times significantly
  - Cannot capture relevant aspects of parallel computations such as timing of events

Debugging

- Common bugs

Debugging

- Common bugs
  - Deadlocks
    - Receive before send: use asynchronous routines

Debugging

- Common bugs
  - Deadlocks
    - Receive before send: use asynchronous routines
    - Wrong tag and/or id of sending or receiving process

Debugging

- Common bugs
  - Deadlocks
    - Receive before send: use asynchronous routines
    - Wrong tag and/or id of sending or receiving process
  - Incorrect results
    - Type mismatch between send and receive

Debugging

- Common bugs
  - Deadlocks
    - Receive before send: use asynchronous routines
    - Wrong tag and/or id of sending or receiving process
  - Incorrect results
    - Type mismatch between send and receive
    - Mixed-up parameters in MPI function calls

Debugging

- Common bugs
  - Deadlocks
    - Receive before send: use asynchronous routines
    - Wrong tag and/or id of sending or receiving process
  - Incorrect results
    - Type mismatch between send and receive
    - Mixed-up parameters in MPI function calls
- Advantageous to use collective communications
  - Usually invoke the function from the same line of the source code, hence all the arguments are the same
    - Either all calls are correct or all wrong

Debugging

- Common bugs
  - Deadlocks
    - Receive before send: use asynchronous routines
    - Wrong tag and/or id of sending or receiving process
  - Incorrect results
    - Type mismatch between send and receive
    - Mixed-up parameters in MPI function calls
- Advantageous to use collective communications
  - Usually invoke the function from the same line of the source code, hence all the arguments are the same
    - Either all calls are correct or all wrong
  - Point-to-point communications are more error prone
    - Different MPI function calls, different arguments, etc

Practical Debugging Strategies

- Get single-process version working correctly

Practical Debugging Strategies

- Get single-process version working correctly
- Test with the smallest number of processes and smallest problem size necessary for all of the program's functionality to be exercised

Practical Debugging Strategies

- Get single-process version working correctly
- Test with the smallest number of processes and smallest problem size necessary for all of the program's functionality to be exercised
- For point-to-point communications print the data being sent and being received to make sure the values match

Practical Debugging Strategies

- Get single-process version working correctly
- Test with the smallest number of processes and smallest problem size necessary for all of the program's functionality to be exercised
- For point-to-point communications print the data being sent and being received to make sure the values match
- Prefix debug messages with the process rank

Practical Debugging Strategies

- Get single-process version working correctly
- Test with the smallest number of processes and smallest problem size necessary for all of the program's functionality to be exercised
- For point-to-point communications print the data being sent and being received to make sure the values match
- Prefix debug messages with the process rank
- First debug initialization phase, to ensure that all data structures have been setup correctly

Practical Debugging Strategies

- Get single-process version working correctly
- Test with the smallest number of processes and smallest problem size necessary for all of the program's functionality to be exercised
- For point-to-point communications print the data being sent and being received to make sure the values match
- Prefix debug messages with the process rank
- First debug initialization phase, to ensure that all data structures have been setup correctly
- Check calculation of local indices

Review

- Advanced MPI
- Hybrid programming
  - Combining OpenMP and MPI
- Load balancing
- Termination detection

Next Class …

- Methods for Solving Linear Systems
  - Direct Methods: solution is sought directly, at once
    - Gaussian Elimination
    - LU Factorization
  - Iterative Methods: solution is sought iteratively, by improvement
    - Relaxation Methods
    - Krylov Methods
    - Preconditioning
- Linear Second-order Partial Differential Equations (PDEs)
  - Finite difference methods
  - Example: steady-state heat distribution
  - Ghost points