Electrical Engineering and Computer Science

EECS 358 - INTRODUCTION TO PARALLEL COMPUTING

Lecture 11

Dist. Mem. Message Passing Programming - II

Outline

- Message Passing Programming intermediate concepts
- Loop parallelization
- Global versus local indices
- Loop scheduling
- READING: Foster, "Design and Building of Parallel Programs," Chapter 8

Review of MPI

- MPI_INIT: Initiate an MPI computation
- MPI_FINALIZE: Terminate an MPI computation
- MPI_COMM_SIZE: Determine number of processes
- MPI_COMM_RANK: Determine my process identifier
- MPI_SEND: Send a message
- MPI_RECV: Receive a message
- MPI_REDUCE, MPI_GATHER, MPI_BCAST: Collective communication

Loop Parallelization

- We will look at some simple and commonly occuring code fragments and consider schemes to parallelize them
- Techniques involve the use of:
  - Local variables to remove dependencies
  - Code transformations
  - Use of particular scheduling techniques

Fully Parallel Loop

- Data locality is important for performance
- Distributed memory machines are programmed by explicitly distributing program data when created so that each processor only operates on its local data
- For example, consider:

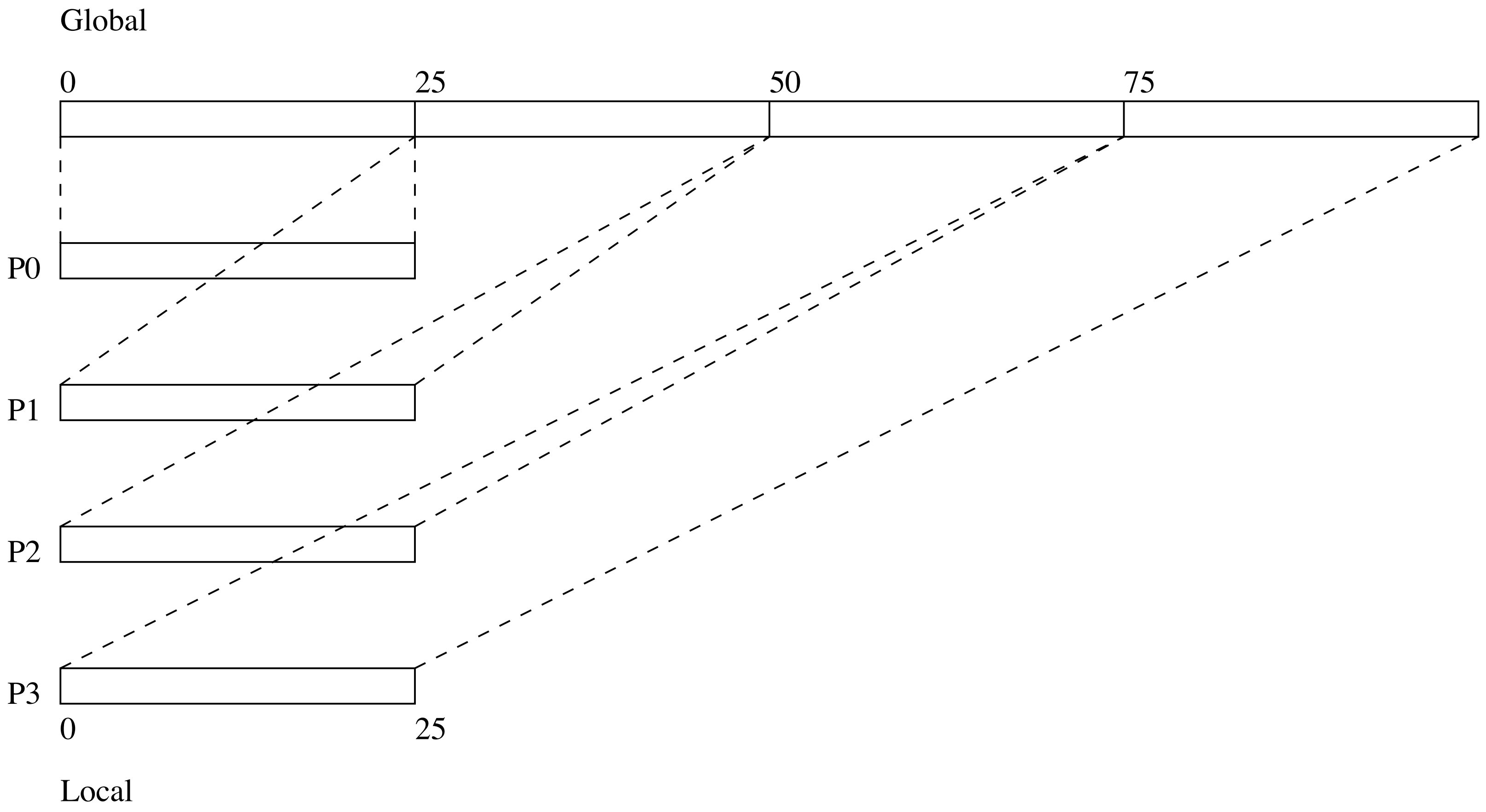
```
/* sequential program */

int a[100],b[100],c[100];

main()

{
  for(i=0; i < 100; i++)
    a[i] = b[i] * c[i];
}
```

Global versus Local Indices



Global versus Local Indices

- If above were programmed on a distributed memory machine with four processors, each processor would have array elements of size 25, and they would each perform 25 iterations

```
int a[25],b[25],c[25];

main()

{
  for(i=0; i < 25; i++)
    a[i] = b[i] * c[i];
}
```

Global versus Local Indices

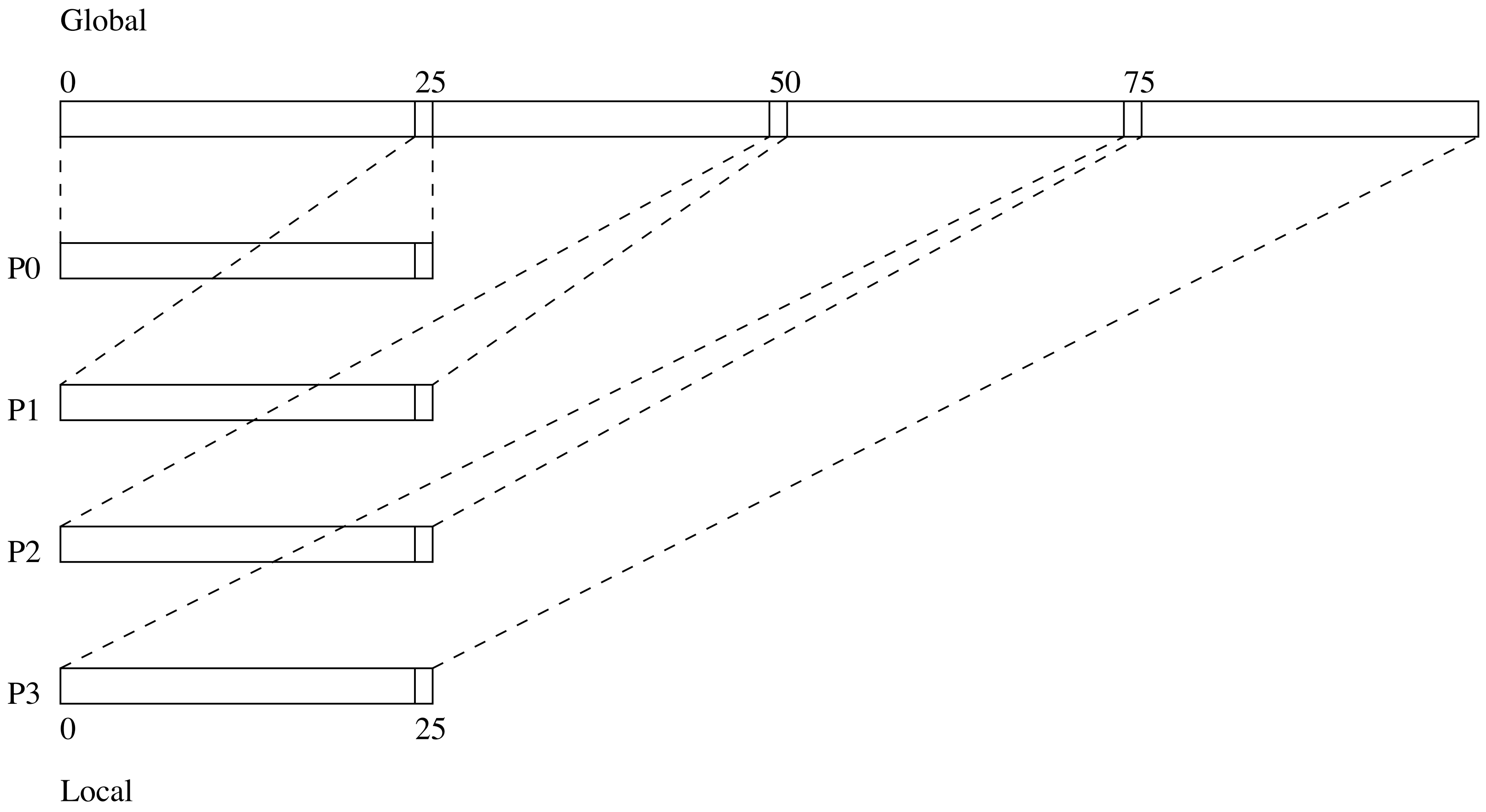
- If we were to change the example program slightly:

```
/* Sequential program */

int a[100],b[100],c[100];

main()
{
  for(i=0; i < 100; i++)
    a[i] = b[i+1] + b[i] * c[i];
}
```

Global versus Local Indices



Global versus Local Indices

- The corresponding parallel program is:

```
/* Dist memory modified program on dist data */

int a[25] b[26] c[25];
main()
{
  MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs);
  MPI_COMM_RANK(MPI_COMM_WORLD,&id);
  /* all except the first processor send their first element to
     the left neighbor */
  if (id>0) MPI_SEND(&b[0], 1, MPI_FLOAT, id-1, 0, MPI_COMM_WORLD);
  /* all except the last processor recv an extra element at the end
     from the right neighbor */
  if (id<3) MPI_RECV(&b[25],1,MPI_FLOAT,id+1,0,MPI_COMM_WORLD);

  for(i=0; i < 25; i++)
      a[i] = b[i+1] + b[i] * c[i];
}
```

Local Variables

- Code fragment:

```
for (i=0;i<100;i++) {
  x=a[i]**3;
  b[i]=x*b[i];
}
```

- Problem: Variable x is dependent, however dependence confined to the same iteration
- Solution: Replicate x on each processor since each iteration will be completely executed by the same processor

Local Variables

```
main()

{
  float a[100],b[100];
  int i,p,lb,ub,id,nprocs;
  float x;

  MPI_COMM_RANK(MPI_COMM_WORLD,id);
  MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs);

    if (id==0) {
    /* send appropriate portions of arrays a and b to all other processors.
       explicit copying avoided by directly acessing appropriate portion */
    for (p=1;p<nprocs;p++) {
      lb=p*100/nprocs;
      MPI_SEND(&(a[lb]), 100/nprocs, MPI_FLOAT, p, ATYPE, MPI_COMM_WORLD);
      MPI_SEND(&(b[lb]), 100/nprocs, MPI_FLOAT, p, BTYPE, MPI_COMM_WORLD);
    }
  } else {
    /* receive portion of arrays from processor 0 */
      MPI_RECV(a, 100/nprocs, MPI_FLOAT, 0, ATYPE, MPI_COMM_WORLD);
      MPI_RECV(b, 100/nprocs, MPI_FLOAT, 0, BTYPE, MPI_COMM_WORLD);
  }
```

Local Variables (Contd)

```
  /* perform portion of work */
  for (i=0;i<100/nprocs;i++) {
    x=a[i]**3;
    b[i]=x*b[i];
  }
  if (id==0) {
    /* recv modified portion of array b from all other processors */
    for (p=1;p<nprocs;p++) {
      lb=p*100/nprocs;
      MPI_RECV(&(b[lb]), 100/nprocs, MPI_FLOAT, p, BTYPE, MPI_COMM_WORLD);
    }
  } else {
    /* send modified array b to processor */
      MPI_SEND(b, 100/nprocs, MPI_FLOAT, 0, BTYPE, MPI_COMM_WORLD);
  }
}
```

Loop-Carried Values

- Code fragment:

```
indx=0;
for (i=0;i<100;i++) {
  indx=indx + i;
  a[i]=b[indx];
}
```

- Problem: The value of $indx$ is carried over from iteration to iteration; such a variable is often called an induction variable

- Solution: Substitute a closed form expression for $indx$ in terms of the iteration counter and parallelize

- Note that we can replicate b to avoid complex packing and unpacking routines ($indx$ is non-linear)

Loop-Carried Values

```
main()

{
  float a[100],b[10000];
  int i,i1,id,nprocs,p,lb;

  MPI_COMM_RANK(MPI_COMM_WORLD,id);
  MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs);

  if (id==0) {
    /* send copies of b to all other processors */
    for (p=1;p<nprocs;p++) {
     MPI_SEND(&b, 10000, MPI_FLOAT, p, BTYPE, MPI_COMM_WORLD);
    }
  } else {
    /* recv a copy of b from processor 0 */
     MPI_RECV(&b, 10000, MPI_FLOAT, 0, BTYPE, MPI_COMM_WORLD);
  }

  offset=id*(100/nprocs);
  /* perform portion of work */
  for (i=0;i<100/nprocs;i++) {
    i1=i+offset;
```

Loop-Carried Values (Contd)

```
    indx=((i1*(i1+1))/2;
    a[i]=b[indx];
    }
  }
  if (id==0) {
    /* recv modified portion of array a from all other processors */
    for (p=1;p<nprocs;p++) {
      lb=p*100/nprocs;
      MPI_RECV(&(a[lb]), 100/nprocs, MPI_FLOAT, p, ATYPE, MPI_COMM_WORLD);
    }
  } else {
    /* send modified array a to processor 0 */
      MPI_SEND(a, 100/nprocs, MPI_FLOAT, 0, ATYPE, MPI_COMM_WORLD);
  }
}
```

Sum Reduction

- Code fragment:

```
total=0.0
for(i=0;i<100;i++) {
  total= total + a[i];
}
```

- Problem: Value of total is carried over from iteration to iteration
- Solution: Create local sub_total variables for each processor and do a global sum at the end

Sum Reduction

```
main()

{
  float a[100],total;
  int i,id,nprocs,p,lb;
  float sub_total;


  MPI_COMM_RANK(MPI_COMM_WORLD,id);
  MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs);

  if (id==0) {
    total=0.0;
    /* send appropriate portions of array a to all other processors */
    for (p=1;p<nprocs;p++) {
      lb=p*100/nprocs;
      MPI_SEND(&(a[lb]), 100/nprocs, MPI_FLOAT, p, ATYPE, MPI_COMM_WORLD);
     }
  } else {
    /* receive portion of arrays from processor 0 */
     MPI_RECV(&a, 100/nprocs, MPI_FLOAT, 0, ATYPE, MPI_COMM_WORLD);
  }
```

Sum Reduction (Contd)

```
  /* perform portion of reduction */

  sub_total=0.0;
  for (i=0;i<100/nprocs;i++) {
    sub_total= sub_total + a[i];
  }

  MPI_REDUCE(&sub_total, &total, 1, TTYPE, MPI_SUM, 0, MPI_COMM_WORLD);

}
```

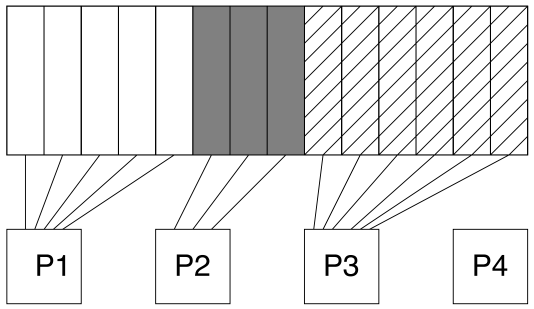
Loop Scheduling

- We will now look at scheduling code for following loop

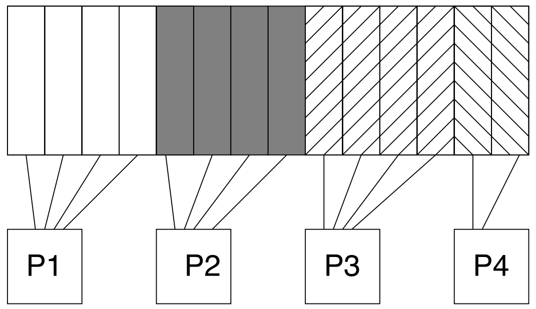
```
for (i=0; i < 100; i++) {
  a[i] = b[i] * c[i];
}
```

- Prescheduling
- Static Blocked Scheduling
- Static Interleaved Scheduling
- Dynamic Scheduling

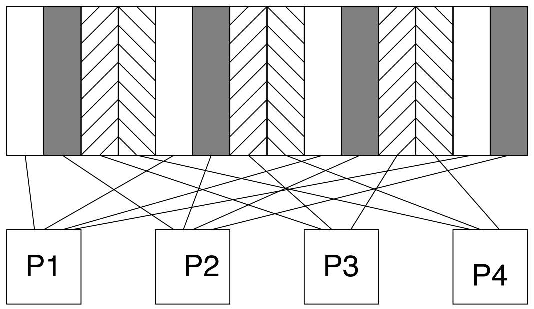
Loop Scheduling



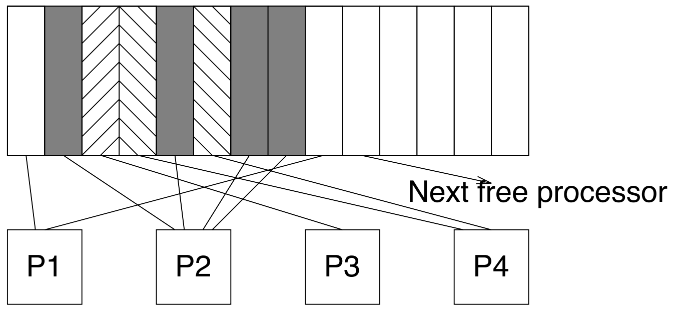
(a)



(b)



(c)



(d)

Prescheduling

```
MPI_INIT();
MPI_COMM_RANK(comm,id);
if (id == 0) {
  /* compute iterations 0 to 29 itself */
  for (i=0; i < 30; i++) {
     a[i] = b[i] * c[i];
  }

  /* send data for iterations 30 to 69 to node 1 */
  MPI_SEND(&b[30], 40, MPI_INT, 1, BTYPE, MPI_COMM_WORLD);
  MPI_SEND(&c[30], 40, MPI_INT, 1, CTYPE, MPI_COMM_WORLD);

  /* send data for iterations 70 to 99 to node 2 */
  MPI_SEND(&b[70], 30, MPI_INT, 2, BTYPE, MPI_COMM_WORLD);
  MPI_SEND(&c[70], 30, MPI_INT, 2, CTYPE, MPI_COMM_WORLD);
```

Prescheduling (Contd)

```
/* receives result for iterations 30 to 69 from node 1 */
MPI_RECV(&a[30], 40, MPI_INT, 1, ATYPE, MPI_COMM_WORLD);

/* receive result for iterations 70 to 99 from node 2 */
MPI_RECV(&a[70], 30, MPI_INT, 2, ATYPE, MPI_COMM_WORLD);

}

if (id == 1) {
/* node 1 receives data for iterations 30-69 from node 0 */
   MPI_RECV(&b, 40, MPI_INT, 0, BTYPE, MPI_COMM_WORLD);
   MPI_RECV(&c, 40, MPI_INT, 0, CTYPE, MPI_COMM_WORLD);
   for(i=0;i <40; i++) {
      a[i] = b[i] * c[i];
   }
/* node 1 sends result for iterations 30-69 to node 0 */
   MPI_SEND(&a, 40, MPI_INT, 0, ATYPE, MPI_COMM_WORLD);
}
```

Prescheduling (Contd)

```
   if (id == 2) {
   /* node 2 receives data for iterations 70-99 from node 0 */
      MPI_RECV(&b, 30, MPI_INT, 0, BTYPE, MPI_COMM_WORLD);
      MPI_RECV(&c, 30, MPI_INT, 0, CTYPE, MPI_COMM_WORLD);
      for(i=0;i < 30; i++) {
         a[i] = b[i] * c[i];
      }
   /* node 2 sends result for iterations 70-99 to node 0 */
      MPI_SEND(&a, 30, MPI_INT, 0, ATYPE, MPI_COMM_WORLD);
   }
}
```

Static Blocked Scheduling

```
MPI_COMM_RANK(MPI_COMM_WORLD,id);
MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs);

if (id == 0) {
  /* compute iterations 0 to (100/nprocs) itself */
  for (i=0; 100/nprocs; i++) {
     a[i] = b[i] * c[i];
  }

  for (p=1; p <= nprocs; p++) {
   /* send iterations lb to ub to to node p */
     lb = p * 100 / nprocs;
     ub = (p +1) * 100 / nprocs;

     MPI_SEND(&b[lb], 100/nprocs, MPI_INT, p, BTYPE, MPI_COMM_WORLD);
     MPI_SEND(&c[lb], 100/nprocs, MPI_INT, p, CTYPE, MPI_COMM_WORLD);

  } /* for (p=1.. */
```

Static Blocked Scheduling

```
   for (p=1; p <= nprocs; p++) {
      /* compute lower and upper bounds of iterations */

      lb = p * 100 / nprocs;
      ub = (p+1) * 100 / nprocs;
      MPI_RECV(&a[lb], 100/nprocs, MPI_INT, p, ATYPE, MPI_COMM_WORLD);

   }
   }
   else {
      /* (id != 0 */
      MPI_RECV(&b, 100/nprocs, MPI_INT, 0, BTYPE, MPI_COMM_WORLD);
      MPI_RECV(&c, 100/nprocs, MPI_INT, 0, CTYPE, MPI_COMM_WORLD);

      for(i=0;i <100/nprocs; i++) {
         a[i] = b[i] * c[i];
      }
      MPI_SEND(&a, 100/nprocs, MPI_INT, 0, ATYPE, MPI_COMM_WORLD);
   }
}
```

Alternate Form: Static Blocked Scheduling

```
 MPI_COMM_RANK(MPI_COMM_WORLD,id);
 MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs);

 /* Scatter the values of arrays b and c to various processors */

 MPI_SCATTER(&b,100/nprocs,MPI_INT,&btmp,100/nprocs,MPI_INT,0,MPI_COMM_WORLD);

 MPI_SCATTER(&c,100/nprocs,MPI_INT,&ctmp,100/nprocs,MPI_INT,0,MPI_COMM_WORLD);

 /* compute iterations 0 to (100/nprocs) itself */

 for (i=0; 100/nprocs; i++) {
      atmp[i] = btmp[i] * ctmp[i];
 }
 MPI_GATHER(&atmp,100/nprocs,MPI_INT,&a,100/nprocs,MPI_INT,0,MPI_COMM_WORLD);
}
```

Dynamic Scheduling

```
MPI_COMM_RANK(MPI_COMM_WORLD,id);
MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs);

/* chunk number of iterations given to each processor each
   time a processor asks for work */

chunk = 10;

if (id == 0) {
  thereiswork = TRUE;
  global_i = 0;
  while (thereiswork) {
     for (p=1; p <= nprocs; p++) {
     /* send iterations lb to ub to to node p */
        if (global_i > 100) {
          thereiswork = FALSE;
        }
        MPI_SEND(&thereiswork, 1, MPI_INT, p, TERMTYPE, MPI_COMM_WORLD);
```

Dynamic Scheduling

```
        lb = global_i;
        ub = global_i + chunk;
        global_i = global_i + chunk;

/* send work to processor p */
        MPI_SEND(&b[lb], chunk, MPI_INT, p, BTYPE, MPI_COMM_WORLD);
        MPI_SEND(&c[lb], chunk, MPI_INT, p, CTYPE, MPI_COMM_WORLD);

        /* get result back from processor p */
        MPI_RECV(&a[lb], chunk, MPI_INT, p, ATYPE1, MPI_COMM_WORLD);
      }
```

Dynamic Scheduling

```
   }
 }
 else {
      /* (id != 0) */
      thereiswork = TRUE;
      while (thereiswork) {

         /* find from node 0 if there is work left */
         MPI_RECV(&thereiswork, 1, MPI_INT, 0, TERMTYPE, MPI_COMM_WORLD);

         /* receive work, perform work, and send result */
         MPI_RECV(&b, chunk, MPI_INT, 0, BTYPE, MPI_COMM_WORLD);
         MPI_RECV(&c, chunk, MPI_INT, 0, CTYPE, MPI_COMM_WORLD);

         for(i=0;i < chunk; i++)
            a[i] = b[i] * c[i];
         MPI_SEND(&a, chunk, MPI_INT, 0, ATYPE1, MPI_COMM_WORLD);
       }
  }
}
```

Summary

- Message Passing Programming intermediate concepts
- Loop parallelization
- Global versus local indices
- Loop scheduling
- NEXT LECTURE: Dist. Memory Message Passing Prog. III: Intermediate MPI Concepts and Examples
- READING: Foster, "Design and Building of Parallel Programs", Chapter 8