Electrical and Computer Engineering C97-20

SPECIAL TOPICS IN ECE

INTRODUCTION TO PARALLEL COMPUTING

Lecture 5

Shared Memory Programming II

Outline

- Loop scheduling
  - Static
  - Dynamic
- Loop Parallelization
  - Dependence free
  - Dependence situations
- READING: B. Bauer, "Practical Parallel Programming", Ch. 5 and 6

Loop Scheduling

- Difficult problem in shared memory machines because of load variations on different processors
- A varying number of processors become available over time for on a particular user
- The computation involved in each iteration can itself be widely variant

Loop Scheduling Algorithms

- Consider the following loop:

```
do i=1,100
  a(i) = b(i] * c(i);
enddo
```

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

- No dependencies across iterations
- Same amount of work is performed in each iteration

Prescheduling or Indirect Scheduling

- Assign work to each process using a schedule array

```
integer low(3),high(3)
data low /1 30 70/
data high /29 69 100/
common /shared/ low,high,a,b,c

subroutine parallel_func()

integer id,i
common /shared/ low,high,a,b,c

id=m_get_myid()
do i=low(id+1),high(id+1)
  a(i) = b(i) * c(i);
enddo
```

```
float *a,*b,*c;
int low[]={0,30,70};
int high[]={30,70,100};

void parallel_func() {
  int id,i;

  id=m_get_myid();
  for (i=low[id];i<high[id];i++) {
    a[i]=b[i]+c[i];
  }
}
```


Static Blocked Scheduling

- Assign a contiguous chunk of iterations based on process id

```
common /shared/ a,b,c

subroutine parallel_func()

integer id,i,nprocs,low,high
common /shared/ a,b,c

id=m_get_myid()
nprocs=m_get_numprocs()
low=id*100/nprocs+1
high=(id+1)*100/nprocs
do i=low,high
  a(i) = b(i) * c(i);
enddo
```

```
float *a,*b,*c;

void parallel_func() {
  int id,i,nprocs;
  int low,high;

  id=m_get_myid();
  nprocs=m_get_numprocs();
  low=id*100/nprocs;
  high=(id+1)*100/nprocs;
  for (i=low;i<high;i++) {
    a[i]=b[i]+c[i];
  }
}
```


Static Interleaved Scheduling

- Assign iterations in a round-robin fashion based on process id

```
common /shared/ a,b,c

subroutine parallel_func()

integer id,i,nprocs,low,high
common /shared/ a,b,c

id=m_get_myid()
nprocs=m_get_numprocs()
do i=id,100,nprocs
  a(i) = b(i) * c(i);
enddo
```

```
float *a,*b,*c;

void parallel_func() {
  int id,i,nprocs;
  int low,high;

  id=m_get_myid();
  nprocs=m_get_numprocs();
  for (i=id;i<100;i+=nprocs) {
    a[i]=b[i]+c[i];
  }
}
```


Dynamic Scheduling (Self Scheduling)

- Processes execute iterations using a shared counter

```
integer i
common /shared/ a,b,c,i

subroutine parallel_func()

integer chunk,i1,i2
parameter (chunk=2)
common /shared/ a,b,c,i

i1=1
while (i1.le.100) do
  call m_lock()
    i1=i
    i=i+chunk
  call m_unlock
  do i2=i1,min(i1+chunk-1,100)
    a(i2)=b(i2)+c(i2)
  enddo
end while
```

```
float *a,*b,*c;
int i;

void parallel_func() {
  int i1,i2;
  int chunk=2;

  i1=0;
  while (i1<100) {
     m_lock();
        i1=i;
        i+=chunk;
     m_unlock();
     for (i2=i1;i2<min(i1+chunk,100);i2++) {
       a[i2]=b[i2]*c[i2];
     }
  }
}
```

Local Variables

- Code fragment:

```
do i=1,n
  x=a(i)*3
  b(i)=x*b(i)
enddo
```

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

- Problem: Variable x is dependent, however dependence confined to the same iteration
- Solution: Make x local to each process since each iteration will be completely executed by the same process

Local Variables

```
common /shared/ a,b,n

subroutine parallel_func()

integer i,id,nprocs
real x
common /shared/ a,b

id=m_get_myid()
nprocs=m_get_numprocs()
do i=id+1,n,nprocs
  x=a(i)*3
  b(i)=x*b(i)
enddo
```

```
float *a,*b;
int n;

void parallel_func(n) {
  int i,id,nprocs;
  float x;

  id=m_get_myid();
  nprocs=m_get_numprocs();
  for (i=id;i<n;i+=nprocs) {
    x=a[i]*3;
    b[i]=x*b[i];
  }
}
```


Dynamic Scheduling (GSS)

- Processes execute iterations using a shared counter

```
integer i
common /shared/ a,b,c,i

subroutine parallel_func()

integer chunk,i1,i2,nprocs
common /shared/ a,b,c,i

nprocs=m_get_numprocs()
i1=1
while (i1.le.100) do
  call m_lock()
    i1=i
    chunk=(100-i1+1)/(2*nprocs)+1
    i=i+chunk
  call m_unlock
  do i2=i1,min(i1+chunk-1,100)
    a(i2)=b(i2)+c(i2)
  enddo
end while
```

```
float *a,*b,*c;
int i;

void parallel_func() {
  int i1,i2,chunk,nprocs;

  nprocs=m_get_numprocs();
  i1=0;
  while (i1<100) {
     m_lock();
        i1=i;
        chunk=(100-i1+1)/(2*nprocs)+1;
        i+=chunk;
     m_unlock();
     for (i2=i1;i2<min(i1+chunk,100);i2++) {
       a[i1]=b[i1]*c[i1];
     }
  }
}
```


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

Loop-Carried Values

- Code fragment:

```
indx=0
do i=1,n
  indx=indx+i
  a(i)=b(indx)
enddo
```

```
indx=0;
for (i=0;i<n;i++) {
  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

Loop-Carried Values

```
common /shared/ a,b,n

subroutine parallel_func()

integer i,id,nprocs,indx
common /shared/ a,b,n

id=m_get_myid()
nprocs=m_get_numprocs()
do i=id+1,n,nprocs
  indx=(i*(i+1))/2
  a(i)=b(indx)
enddo
```

```
float *a,*b;
int n;

void parallel_func() {
  int i,id,nprocs;

  id=m_get_myid();
  nprocs=m_get_numprocs();
  for (i=id;i<n;i+=nprocs) {
    indx=(i*(i+1))/2;
    a[i]=b[indx];
    }
}
```


Local Variables

- Code fragment:

```
do i=1,n
  x=a(i)*3
  b(i)=x*b(i)
enddo
val=foo(x)
```

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

- Problem: Variation of previous problem, now x is used outside the loop, need the value of x from the last iteration
- Solution: Make x local as before, just ensure the process executing the last iteration of the loop preserves the value of x for further use

Local Variables

```
common /shared/ a,b,x,n

subroutine parallel_func()

integer i,id,nprocs
real local_x
common /shared/ a,b,x,n

id=m_get_myid()
nprocs=m_get_numprocs()
do i=id+1,n,nprocs
  local_x=a(i)*3
  b(i)=local_x*b(i)
  if (i.eq.n) then
    x=local_x
  end if
enddo
```

```
float *a,*b,x;
int n;

void parallel_func() {
  int i,id,nprocs;
  float local_x;

  id=m_get_myid();
  nprocs=m_get_numprocs();
  for (i=id;i<n;i+=nprocs) {
    local_x=a[i]*3;
    b[i]=local_x*b[i];
    if (i==n-1) x=local_x;
  }
}
```

Sum Reduction

- Code fragment:

```
total=0.0
do i=1,n
  total=total+a(i)
enddo
```

```
total=0.0
for(i=0;i<n;i++) {
  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

```
common /shared/ a,total,n

total=0.0

subroutine parallel_func()

integer i,id,nprocs
real sub_total
common /shared/ a,total,n

id=m_get_myid()
nprocs=m_get_numprocs()
sub_total=0.0
do i=id+1,n,nprocs
  sub_total=sub_total+a(i)
enddo

call m_lock()
  total=total+sub_total
call m_unlock()
```

```
float *a,total;
int n;

void parallel_func() {
  int i,id,nprocs;
  float sub_total;

  id=m_get_myid();
  nprocs=m_get_numprocs();
  sub_total=0.0;
  for (i=id;i<n;i+=nprocs) {
    sub_total+=a[i];
  }

  m_lock();
    total+=sub_total;
  m_unlock();
}
```


Indirect Indexing

- Code fragment:

```
do i=1,n
  ix=indexx(i)
  iix=ixoffset(ix)
  total(iix)=total(iix)+delta
enddo
```

```
for (i=0;i<n;i++) {
  ix=indexx[i];
  iix=ixoffset[ix];
  total[iix]=total[iix]+delta;
}
```

- Problem: The value of iix is dependent on the values of the arrays $indexx$ and $ixoffset$ and cannot be guaranteed to be different for each iteration
- Solution: Remove the data dependent element to a non-parallelized loop; thus creating a parallel and a non-parallel loop

Indirect Indexing

```
do i=1,n
  ix=indexx(i)
  iix(i)=ixoffset(ix)
enddo
do i=1,n
  total(iix(i))=total(iix(i))+delta
do i=1,n
```

```
for (i=0;i<n;i++) {
  ix=indexx[i];
  iix[i]=ixoffset[ix];
}
for (i=0;i<n;i++) {
  total[iix[i]]=total[iix[i]]+delta;
}
```

False Reduction Dependence

- Code fragment:

```
do i=1,n
  total=0.0
  do j=1,m
    total=total+a(i,j)
  enddo
  b(i)=c(i)*total
enddo
```

```
for (i=0;i<n;i++) {
  total=0.0;
  for (j=0;j<m;j++) {
    total+=a[i][j];
  }
  b[i]=c[i]*total;
}
```

- Reduction in inner loop tempts one to parallelize it; however, a better scheme is to make $total$ local and parallelize the outer loop

Recurrence

- Code fragment:

```
do i=1,n
  a(i)=a(i-1)+b(i)
enddo
```

```
for (i=0;i<n;i++) {
  a[i]=a[i-1]+b[i];
}
```

- Problem: Computation of a depends on value in previous iterations
- Solution: No easy transformation, special algorithms exist for solving recurrences in parallel

Stride Not 1

- Code fragment:

```
do i=1,n,2
  a(i)=a(i-1)+b(i)
enddo
```

```
for (i=0;i<n;i+=2) {
  a[i]=a[i-1]+b[i];
}
```

- Recurrence example with non-unit stride removes the dependence and makes the loop parallel

Inconsequential Data Dependence

- Code fragment:

```
index=select(n)
do i=1,n
  a(i)=a(index)
enddo
```

```
index=select(n);
for(i=1;i<n;i++) {
  a[i]=a[index];
}
```

- At first glance seems like loop cannot be parallelized; however it can be seen the dependence is inconsequential since all the values of a will be the same in any case. Loop can thus be parallelized

Loop Distribution

- Code fragment:

```
do i=2,n
  a(i)=b(i)+c(i)*d
  c(i)=a(i-1)
enddo
```

```
for (i=1;i<n;i++) {
  a[i]=b[i]+c[i]*d;
  c[i]=a[i-1];
}
```

- Problem: Dependence carried on a
- Solution: Distribute loop into two loops, both are now i parallelizable

Loop Distribution

```
common /shared/ a,b,c,d,n

subroutine parallel_func()

integer i,id,nprocs,start
common /shared/ a,b,c,d,n

id=m_get_myid()
nprocs=m_get_numprocs()
if (id.eq.0) then
  start=nprocs+1
else
  start=id+1
endif

do i=start,n,nprocs
  a(i)=b(i)+c(i)*d
enddo
call m_barrier()
do i=start,n,nprocs
  c(i)=a(i-1)
enddo
```

```
float *a,*b,*c,d;
int n;

void parallel_func() {
  int i,id,nprocs,start;

  id=m_get_myid();
  nprocs=m_get_numprocs();
  if (id==0) start=nprocs;
  else start=id;

  for (i=start;i<n;i+=nprocs) {
    a[i]=b[i]+c[i]*d;
  }
  m_barrier();
  for (i=start;i<n;i+=nprocs) {
    c[i]=a[i-1];
  }
}
```


Loop Reordering

- Code fragment:

```
do k=1,n
  do i=1,n
    do j=1,n
      a(i,j)=a(i,j)+b(i,k)+c(k,j)
    enddo
  enddo
enddo
```

```
for (k=0;k<n;k++) {
  for (i=0;i<n;i++) {
    for (j=0;j<n;j++) {
      a[i][j]+=b[i][k]+c[k][j];
    }
  }
}
```

- Outer loop cannot be parallelized due to dependence carried on a. Can parallelize i loop, but very little work inside the parallel loop
- Interchanging k and i loops does not alter program semantics; however, parallelizing the outer i loop creates more work inside the parallel loop

Loop Reordering

```
common /shared/ a,b,c

subroutine parallel_func()

integer i,id,nprocs
common /shared/ a,b,c

id=m_get_myid()
nprocs=m_get_numprocs()
do i=id+1,n,nprocs
  do k=1,n
    do j=1,n
      a(i,j)=a(i,j)+b(i,k)+c(k,j)
    enddo
  enddo
enddo
```

```
float *a,*b,*c;

void parallel_func() {
  int i,id,nprocs;

  id=m_get_myid();
  nprocs=m_get_numprocs();
  for (i=id;i<n;i+=nprocs) {
    for (k=0;k<n;i++) {
      for (j=0;j<n;j++) {
        a[i][j]+=b[i][k]+c[k][j];
      }
    }
  }
}
```

Summary

- Loop scheduling
- Loop Parallelization
- NEXT LECTURE: Shared Memory Parallel Programming Examples
- READING: P. Banerjee, "Parallel Algorithms for VLSI CAD", chapter 2.