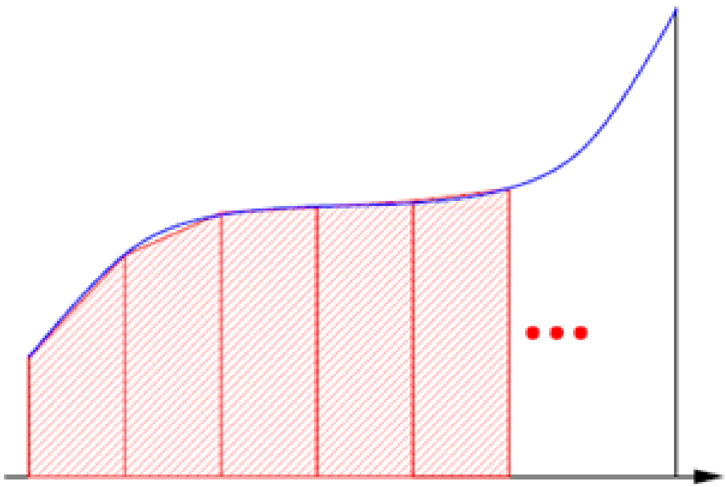
Parallel Scientific Computation

OpenMP 3

J.-H. Parq
IPCST
Seoul National University

Numerical Integration

- Trapezoid rule



$$\frac{\Delta x}{2}[f_0 + 2f_1 + 2f_2 + \cdots + 2f_{N-1} + f_N]$$

  - Ex.)

$$\int_0^1 \frac{4}{1+x^2}\, dx = \pi$$

Numerical Integration

Fortran

```fortran
dx = 1d0/dble(N)
S = S + 4d0/2d0
!$omp parallel private(x)
!$omp do reduction(+:S)
     do I = 1, N-1
          x = dble(I)*dx
          S = S + 4d0/(1d0 + x*x)
     end do
!$omp end do
!$omp end parallel
S = S + 2d0/2d0
ANSWER = S*dx
```

C

```c
dx = 1.0/(double)N
s = 4.0/2.0;
#pragma omp parallel private(i,x)
{
   #pragma omp for reduction(+:s)
   for (i=1; i<N; i++) {
          x = (double)i*dx;
          s += 4.0/(1.0 + x*x);
   }
}
s += 2.0/2.0;
answer = s*dx;
```

Loop Dependency

- Loop-independent dependence
  - Dependence exists within the same iteration
  - No hindrance in parallelization
- Loop-carried dependence
  - Dependence between different iterations
  - Ex.)

```
for (i=1; i<N; i++)  a[i] += a[i-1];
```

  - Difficult to parallelize

Loop Dependency

- Removing loop-carried dependence
  - Several cases

1. Loop skewing

- Serial example

```
/*Arrays have already been assigned.*/

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

- Parallel version

```
b[1] += a[0];
#pragma omp for private(i)
for (i=1; i<N-1; i++) {
   a[i] += c[i];
   b[i+1] += a[i];
}
a[N-1] += c[N-1];
```

Loop Dependency

- Removing loop-carried dependence
    2. Creating a clone array

- Serial example

```
/*Array a has already been assigned.*/

for (i=0; i<N-1; i++) {
   tmp = (b[i]+c[i])/2.0;
   a[i] = a[i+1]+tmp;
}
```

- Parallel version

```
#pragma omp for private(i)
for (i=0; i<N-1; i++) ta[i] = a[i+1];


#pragma omp for private(i,tmp)
for (i=0; i<N-1; i++) {
   tmp = (b[i]+c[i])/2.0;
   a[i] = ta[i]+tmp;
}
```

Loop Dependency

- Removing loop-carried dependence

  3. Loop fission

- Serial example

```
for (i=1; i<N; i++) {
   a[i] = a[i-1]+a[i];
   tot += (b[i]+c[i])/2.0;
}
```

- Parallel version

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

 #pragma omp for reduction(+:tot)
private(i)
for (i=1; i<N; i++) {
   tot+= (b[i]+c[i])/2.0;
}
```

Loop Dependency

- Removing loop-carried dependence
  4. Using reduction clause
     - Already showed in the integration example
  5. Summation → direct calculation
     - Ex.) sum(0 to n) → n(n+1)/2

Time Check

- You must check execution time to see how parallelization performance change

1. Command ‘time’ in Unix or Linux
   - Total elapsed time (*i.e.*, master thread time)
   - Usage) time a.exe
2. Time library in C
   - Some Fortran versions also provide time library.
   - It is safe to use it only for the master thread
3. OpenMP routines
   - omp_get_wtime(): wall clock time on a thread

Enhancing Efficiency

- Efficiency of *OpenMP* parallelism depends on use of cache.

1. Number of threads
   - ✓ Just recommendation
   - **Even**: Use of L2 cache suppresses overhead increase.
   - **(# of cores per CPU)** × **n**: Use of L3 cache suppresses overhead increase.
2. Algorithm improvement
   - **Rearranging loops**: Loop fusion, loop tiling, ……
   - **Reducing** other **overhead** factors
   - **Balancing loads**: schedule clause, ……

Enhancing Efficiency

- Multi-dimensional array
  - In C, a more precedent index has a larger unit for memory address.

- Better access

```
#pragma omp for
for (i=0; i<N; i++) {
      sum = 0.0;
      for (k=0; k<M; k++)
         sum += A[i][k]*x[k];
      y[i] = sum;
}
```

- Worse access

```
#pragma omp for
for (i=0; i<N; i++) {
      sum = 0.0;
      for (k=0; k<M; k++)
         sum += x[k]*A[k][i];
      y[i] = sum;
}
```

Enhancing Efficiency

- Multi-dimensional array
  - In Fortran, usually, a more precedent index has a smaller unit for memory address.

- Better access

```
!$omp do
  do i = 1, N
    sum = 0d0
    do k = 1, M
       sum = sum + x(k)*A(k,i)
     end do
     y(i) = sum
  end do
!$omp end do
```

- Worse access

```
!$omp do
  do i = 1, N
    sum = 0d0
    do k = 1, M
       sum = sum + A(i,k)*x(k)
     end do
     y(i) = sum
  end do
!$omp end do
```

Enhancing Efficiency

- Multi-dimensional array
  - However, memory access depends on the system (OS, compiler, ……).
  - You should test a large array before arranging the order of access.

Enhancing Efficiency

- Loop fusion
  - Overheads should be reduced.
  - An array in cache should be reused if possible.

- Before

```
#pragma omp for
for (i=0; i<N; i++) a[i] += b[i];
#pragma omp for
for (k=0; k<M; k++) {
   y[k] *= y[k];
   z[k] = a[k]*k;
}
```

- After

```
#pragma omp for
for (k=0; k<M; k++) {
   y[k] *= y[k];
   if (k<N) a[k] += b[k];
   z[k] = a[k]*k;
}
```

Enhancing Efficiency

- Collapse clause
  - Fusing nested loops
  - Reduced overheads (and easier to balance)

- Fortran

```
!$omp do collapse(2)
  do J = 1, M
    do I = 1, N
      A(I,J) = A(I,J)*2d0
    end do
  end do
!$omp end do
```

- C/C++

```
#pragma omp for collapse(2)
for (i=0; i<N; i++) {
  for (j=0; j<M; j++)
    a[i][j] *= 2.0;
}
```

Enhancing Efficiency

- Loop fission
  - An array in cache should be reused if possible.

- Before

```
#pragma omp for
for (i=0; i<N; i++) {
   y[i] = exp(-(double)i);
   for (k=0; k<M; k++) {
      a[k][i] += b[k][i] + c[i];
}
```

- After

```
#pragma omp for
for (i=0; i<N; i++)
   y[i] = exp(-(double)i);
#pragma omp for
for (k=0; k<M; k++)
   for (i=0; i<N; i++)
      a[k][i] += b[k][i] + c[i];
```

Enhancing Efficiency

- Loop tiling
  - 2-D array → tiles of data chunks fit for cache
  - Rearranging loops for tiles
  - Ex.) if cache size = Csize_x^2 × 2,

```
for (j1=0; j1<N; j1+=Csize_x)
    for (i=0; i<N; i++)
        for (j2=0; j2<N-j1 && j2<Csize_x; j2++)
            b[i][j1+j2] = a[j1+j2][i] ;
```

Enhancing Efficiency

- Relocating parallel regions

  1. Merging

- Before

```
#pragma omp parallel for
for (i=0; i<N; i++) {
        ……
}
#pragma omp parallel for
for (k=0; k<M; k++) {
        …….
}
```

- After

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i=0; i<N; i++) {
          ……
  }
  #pragma omp for
  for (k=0; k<M; k++) {
          …….
  }
}
```

Enhancing Efficiency

- Relocating parallel regions

2. Expanding

- Before

```
for (i=0; i<N; i++) {
#pragma omp parallel for
   for (k=0; k<M; k++) {
            …….
   }
}
```

- After

```
#pragma omp parallel
{
   for (i=0; i<N; i++) {
      #pragma omp for nowait
      for (k=0; k<M; k++) {
            …….
      }
   }
}
```

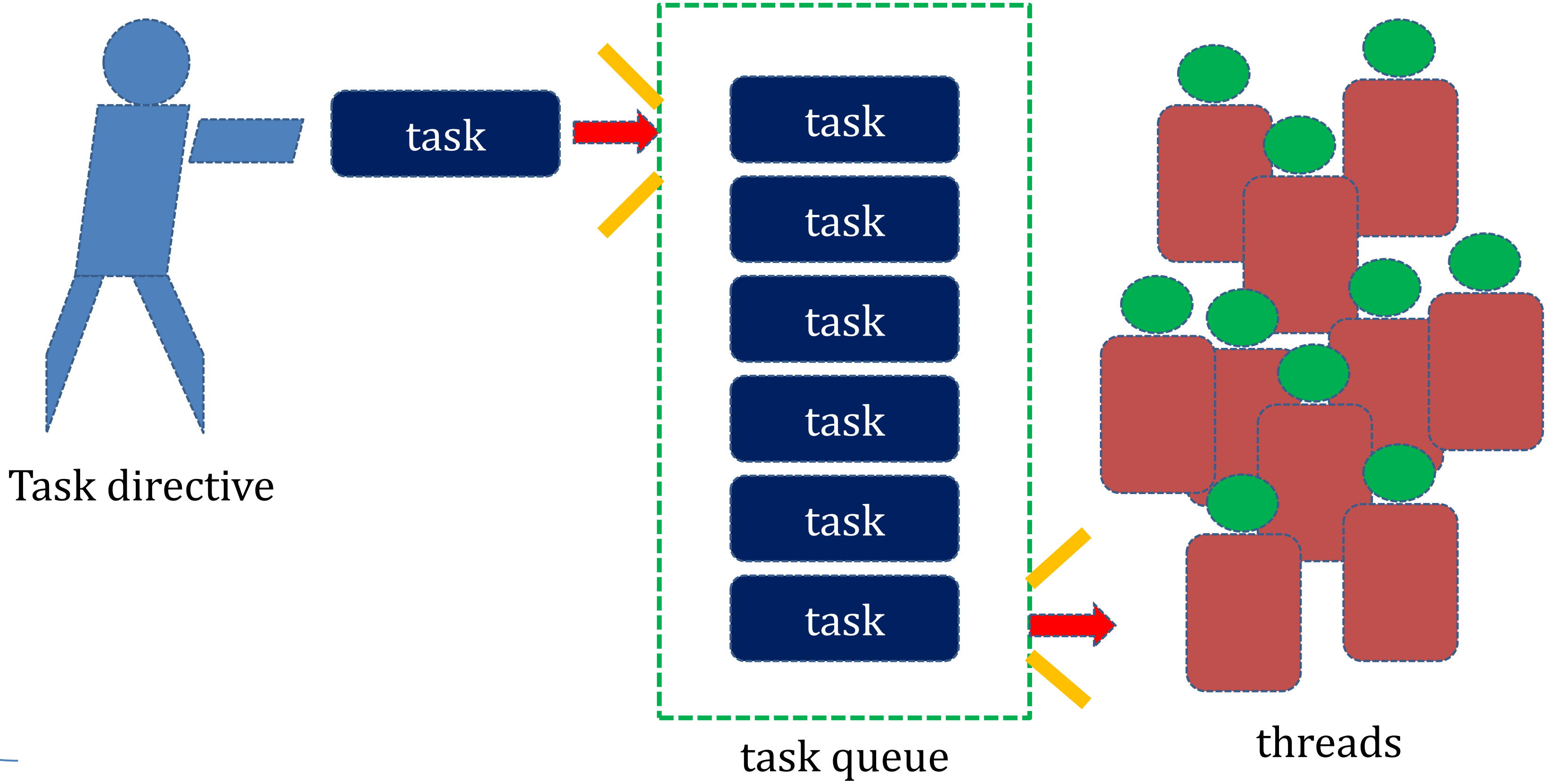
Enhancing Efficiency

- If clause
  - It is difficult to speed up by parallelizing loops of small size.
  - The if clause checks the loop size and determines if the loop will be parallelized.
  - Ex.)

```
#pragma omp parallel for if (M>threshold)
  for (k=0; k<M; k++) {
      ......
  }
```

Task Directive

- A task directive just pushes a task into a queue



Task Directive Example

```
 op = 1; i = 0;
#pragma omp parallel shared(a, coord, op, target)
 {
#pragma omp master
   {
     while (op && i<50) {
#pragma omp task firstprivate(i) private(j)
       {
        for (j=0; j<10000; j++) {
          if (a[i][j]==target) {
            coord[0] = i; coord[1] = j;  op = 0;  }   }
       }
       i++;
     }
   }
 }
```

Other Directives

- Single
  - Equivalent to ‘sections’ with one ‘section’
- Workshare
  - Only for Fortran
  - To cover f90 or higher (ex.: forall, where, ……)
- Threadprivate
  - Privatization of global variables or data
  - The ‘copyin’ clause initializes the variables.

Other Directives

- Flush
  - Identifying a synchronization point for memory access consistency
  - ‘Flush’ is virtually included in (both explicit and implicit) barriers and ‘critical’.
- Taskwait
  - Barrier for execution of all tasks set by ‘task’ directives
  - There are some restrictions in usage.

Implicit Barriers

- Directives with implicit barriers
  - Parallel
    - Indispensible barriers
  - Do/for, Sections, Single, Workshare
    - Removable barriers
- Directives without implicit barriers
  - Master, Task

Other Library Routines

- Rarely used but useful ones
  - omp_get_wtick
  - omp_in_parallel
  - omp_set_dynamic, omp_get_dynamic
  - omp_set_nested, omp_get_nested
  - omp_set_schedule
  - omp_init_lock, omp_set_lock, omp_unset_lock, omp_destroy_lock
  - Other lock routines

References

- C. Evangelinos,
  Parallel Programming for Multicore Machines Using OpenMP and MPI

- R. Chandra, L. Dagum, D. Kohr, D. Maydan *et al*.,
  Parallel Programming in OpenMP

- B. Barney,
  OpenMP

- L. R. Scott, T. Clark, and B. Bagheri,
  Scientific Parallel Computing