Parallel Programming Patterns

Pattern-Driven Parallel Application Development

Parallelism and Performance

- Automatic compiler optimizations have their limitations
  - Can only explore loop level parallelism
  - Cannot reason about multiple threads of execution
  - Cannot reason about synchronization overhead
- Explicit parallel programming is often required for performance scaling

Common Parallelization Patterns

- Matrix computations
  - Map and reduce
  - SPMD parallelization with reduction support
  - Memory bound vs. CPU bound computations
- Stencils and iterative stencils
  - Neighbor communications without or with recurrence
  - Tiling, skewing, and pipelining
- Recursion
  - Recursive/nested components or behaviors
  - Nested fork-join
- Other patterns
  - Scatter, gather, partial reduction

Map And Reduce

- The Map pattern
  - Replicate a function independently over every element of a set

```
for (i=0; i<n; ++i)
     f(A[i]);
```

  - **Examples**: gamma correction and thresholding in images; color space conversions; Monte Carlo sampling; ray tracing
- *The Reduction pattern*
  - combines every element in a collection into one value using an **associative** operator.

```
b = 0; for (i=0; i<n; ++i)
     b += f(B[i]);
```

  - **Examples:** averaging of Monte Carlo samples; convergence testing; image comparison metrics; matrix operations.
- Relevant optimizations (matrix operations)
  - SIMD parallelization, data partitioning, tiling, unroll&jam, unrolling

Example1: Dense Matrix Multiplication

```
void dgemm(double *a,double *b, double *c, double beta, int n)
{
          int i,j,k;
Nest1:  for (j = 0; j < n; j ++)
            for (i = 0; i < n; i ++)
                c[j*n+i] = beta*c[j*n+i];
Nest2:  for (k = 0; k < n; k ++)
Nest4:    for (j = 0; j < n; j ++)
Nest3:      for (i = 0; i < n; i ++)
                c[j*n+i] += a[k*n+i] * b[j*n+k];
}
```

- Which loops can be parallelized? Do the loops need to be rearranged via fusion/interchange?
  - **PermuteLoops:inner_loop=Nest3 target=Nest2 order="3,1,2"**
  - **FuseLoops:loop_to_fuse=Nest1 target=Nest2**
  - **FuseLoops:loop_to_fuse=Nest1 target=Nest4**

After Loop Fusion/Interchange

```
void dgemm(double * a,double * b,double * c,double beta,int n)
{
          int i,j,k;
Nest2:  for (j=0; j<n; j+=1)
          {
Nest4:     for (i=0; i<n; i+=1)
             {
Nest1:       c[j*n+i] = beta*c[j*n+i];
Nest3:        for (k=0; k<n; k+=1)
                 c[j*n+i] = c[j*n+i]+a[k*n+i]*b[j*n+k];
             }
          }
}
```

- Which loops can be fully parallelized using OpenMP? Is it a reduction?

After Loop Parallelization

```
#include <omp.h>
void dgemm(double * a,double * b,double * c,double beta,int n)
{
          int i,j,k;
          #pragma omp  parallel {
 Nest2:  #pragma omp  for private(j,i,k)
           for (j=0; j<n; j+=1)  {
 Nest4:      for (i=0; i<n; i+=1) {
 Nest1:        c[j*n+i] = beta*c[j*n+i];
 Nest3:        for (k=0; k<n; k+=1)
                     c[j*n+i] = c[j*n+i]+a[k*n+i]*b[j*n+k];
}  }   }  }
```

- May need to tune parameters to OMP for
  - statis/dynamic/guided scheduling, number of threads to use
  - Need blocking to further improve memory performance

After Loop Blocking

```
#include <omp.h>
void dgemm(double * a,double * b,double * c,double beta,int n)
{
        int i,j,k, i1,j1,j2,k1;
        #pragma omp  parallel {
          #pragma omp  for private(j,i,k,j1,j2,i1,k1)
           for (j_bk_1=0; j_bk_1<n; j1+=256) {
Nest2:    for (j=0; j<min(256,n-j1); j+=32)  {
Nest4:       for (i1=0; i1<n; i1+=32) {
Nest3:          for (k1=0; k1<n; k1+=32) }
            ……
} }   }  }
```

- Need to accommodate interactions among optimizations
  - Supported by POET
  - Will also need to accommodate register and CPU level optimizations

Combining The Optimizations

- Nests: Nest1, Nest2, Nest3, Nest4
- PermuteLoops:inner_loop=Nest3 target=Nest2 order="3,1,2"
- FuseLoops:loop_to_fuse=Nest1 target=Nest2
- FuseLoops:loop_to_fuse=Nest1 target=Nest4
- ParallelizeLoop: target=Nest2 private=j,i,k
- BlockLoops: inner_loop=Nest3 target=Nest2
- UnrollJam: inner_loop=Nest3 target=Nest2

```
void dgemm(double *a,double *b,
double *c, double beta, int n)
{
        int i,j,k;
Nest1: for (j = 0; j < n; j ++)
        for (i = 0; i < n; i ++)
            c[j*n+i] = beta*c[j*n+i];
Nest2: for (k = 0; k < n; k ++)
Nest4: for (j = 0; j < n; j ++)
Nest3: for (i = 0; i < n; i ++)
            c[j*n+i] += a[k*n+i] * b[j*n+k];
}
```

- ScalarRepl: aref=c[j*n+i] dim=Nest2,Nest4 data_type="double" target=Nest4[Nest.body] init_loc=Nest4[Nest.body] save_loc=Nest4[Nest.body]
- UnrollLoop: target=Nest3

Example2: Sparse Matrix Computation

```
    vmult(double *val, int* rowstart, int* col, double* src, double* dst) {
      int row;
      double *val_ptr = val + rowstart[row];
      int *col_ptr = col+rowstart[row];
Nest1: for (row=0; row<n_rows; ++row) {
          double s = *dst;
          double * val_end_of_row = val+rowstart[row+1];
          while (val_ptr != val_end_of_row) {
             s += (*val_ptr++) * src[*col_ptr++];
          }
          *dst++ = s;
   }  }
```

- Optimization: can we parallelize Nest1?
  - Need rewrite to eliminate carried dependences

After Parallelization

```
    vmult(double *val, int* rowstart, int* col, double* src, double* dst) {
      int row;
     omp_set_num_threads(2);
    #pragma omp  parallel
   {
Nest1: #pragma omp  for private(row)
       for (row=0; row<n_rows; row+=1)  {
         double * val_ptr=val+rowstart[row];
         int * col_ptr=col+rowstart[row];
         double s=dst[row];
         double * val_end_of_row=val+rowstart[row+1];
         while (val_ptr!=val_end_of_row)
             s = s+*val_ptr++*src[*col_ptr++];
         dst[row] = s;
     } }   }
```

An Alternative Version

```
    vmult(double *val, int* rowstart, int* col, double* src, double* dst)
    {
          int row, j;
Nest1: for (row=0; row<n_rows; ++row) {
Nest2:   for (j = rowstart[row]; j < rowstart[row+1]; j++) {
                dst[row] += val[j] * src[col[j]];
    } } }
```

- Optimizations
  - Outer loop can be parallelized (no dependence)
  - Inner loop is a reduction operation
- Code is much more readable than the handl-optimized version. Is it less efficient?

After Optimization

```
   vmult(double *val, int* rowstart, int* col, double* src, double* dst)
   {
        int row,j;  double _dst_0;
        omp_set_num_threads(2);
        #pragma omp  parallel  {
Nest1: #pragma omp  for private(row,j,_dst_0)
         for (row=0; row<n_rows; row+=1) {
           _dst_0 = dst[row];
Nest2:    for (j=rowstart[row]; j<rowstart[row+1]; j+=1)
             _dst_0 = _dst_0+val[j]*src[col[j]];
           dst[row] = _dst_0;
  } } }
```

- Optimization specification
  - ParallelizeLoop: target=Nest1 private=row,j
  - ScalarRepl: aref=dst[row] dim=Nest1 target=Nest2 data_type="double" init=1 save=1

Stencils

- Update each elements using values of its neighbors
  - Neighborhoods are given by relative offsets.
  - Boundary conditions need to be considered
  - Recurrence occurs if input and output arrays are the same

```
for (int i = 1; i < N; i++)
    B[i] = f( A[i-1], A[i]);
```

vs.

```
for (int i = 1; i < N; i++)
    A[i] = f( A[i-1], A[i]);
```

- Examples
  - signal filtering including convolution, median,anisotropic diffusion,
  - Simulation of fluid flow, electromagnetic, and financial PDE solvers
  - Lattice QCD, sequence alignment and pattern matching

- Optimizations
  - Blocking (data partitioning)
  - Pipelining vs wave-front time-skewing
  - Task scheduling and load balancing

Example: Stencils With Timesteps (Jacabi7)

```
Nest1: for (t = 0; t < timesteps; t++) {
Nest2:   for (k = 1; k < nz - 1; k++)
Nest3:     for (j = 1; j < ny - 1; j++)
Nest4:       for (i = 1; i < nx - 1; i++)
       Anext[Index3D (nx, ny, i, j, k)] = A0[Index3D (nx, ny, i, j, k + 1)] +
         A0[Index3D (nx, ny, i, j, k - 1)] + A0[Index3D (nx, ny, i, j + 1, k)] +
         A0[Index3D (nx, ny, i, j - 1, k)] + A0[Index3D (nx, ny, i + 1, j, k)] +
         A0[Index3D (nx, ny, i - 1, j, k)] -  A0[Index3D (nx, ny, i, j, k)] *fac ;
  temp_ptr = A0;  A0 = Anext;  Anext = temp_ptr;
 }
```

- What optimizations are possible?
  - Parallelize the inner three loops?

Optimizing The Inner Loops

```
Nest1: for (t=0; t<timesteps; t+=1) {
      #pragma omp  parallel {
        #pragma omp  for private(k,j,i,k_bk_1,k_bk_2,j_bk_3,i_bk_4)
Nest2_par: for (k_bk_1=1; k_bk_1<nz-1; k_bk_1+=256)
Nest2:    for (k_bk_2=0; k_bk_2<min(256,nz-1-k_bk_1); k_bk_2+=32)
Nest3:      for (j_bk_3=1; j_bk_3<ny-1; j_bk_3+=32)
Nest4:       for (i_bk_4=1; i_bk_4<nx-1; i_bk_4+=32)
                ……
       }
      temp_ptr = A0;  A0 = Anext; Anext = temp_ptr;
   }
```

- Optimization specification
  - Nests: Nest1,Nest2,Nest3,Nest4
  - ParallelizeLoop: target=Nest2 private=k,j,i
  - BlockLoops: inner_loop=Nest4 target=Nest2
- Each thread need to sweep the entire stencil, no reuse of data

Alternative Solution: Enable Reuse Of Data

```
Nest1: for (t = 0; t < timesteps; t++) {
Nest2:   for (k = 1; k < nz - 1; k++)
Nest3:     for (j = 1; j < ny - 1; j++)
Nest4:       for (i = 1; i < nx - 1; i++)
       Anext[Index3D (nx, ny, i, j, k)] = A0[Index3D (nx, ny, i, j, k + 1)] +
         A0[Index3D (nx, ny, i, j, k - 1)] + A0[Index3D (nx, ny, i, j + 1, k)] +
         A0[Index3D (nx, ny, i, j - 1, k)] + A0[Index3D (nx, ny, i + 1, j, k)] +
         A0[Index3D (nx, ny, i - 1, j, k)] -  A0[Index3D (nx, ny, i, j, k)] *fac ;
  temp_ptr = A0;  A0 = Anext;  Anext = temp_ptr;
 }
```

- Can loop t be moved inside?
  - Need to loop skewing before permutation
- Would skewing itself allow safe permutation?
  - The array swapping operation would impose difficulty

Rewrite To Enable Parallelism

```
Nest1: for (t = 0; t < timesteps; t++) {
          if (t%2 == 0) { L0 = A0; Lnext = Anext; }
          else {Lnext = A0; L0 = Anext; }
Nest2:   for (k = 1; k < nz - 1; k++)
Nest3:     for (j = 1; j < ny - 1; j++)
Nest4:       for (i = 1; i < nx - 1; i++)
       lnext[Index3D (nx, ny, i, j, k)] = L0[Index3D (nx, ny, i, j, k + 1)] +
        L0[Index3D (nx, ny, i, j, k - 1)] + L0[Index3D (nx, ny, i, j + 1, k)] +
        L0[Index3D (nx, ny, i, j - 1, k)] + L0[Index3D (nx, ny, i + 1, j, k)] +
        L0[Index3D (nx, ny, i - 1, j, k)] -  l0[Index3D (nx, ny, i, j, k)] *fac ;
}
```

- Optimization specification
  - Nests: Nest1,Nest2,Nest3,Nest4
  - SkewLoops: inner_loop=Nest2 target=Nest1 **factor=2**
  - ParallelizeLoop: target=Nest1 private=t,j,i,l0,lnext

Wavefront Parallelization

```
Nest2:  for (k=1; k<nz-1+2*timesteps; k+=1) {
        omp_set_num_threads(2);
        #pragma omp  parallel  {
Nest1: #pragma omp  for private(t,j,i,l0,lnext)
            for (t=max(0,(2+(k-(nz-1)))/2); t<min(timesteps,(1+k)/2); t+=1)
Nest3: for (j=1; j<ny-1; j+=1)
Nest4: for (i=1; i<nx-1; i+=1)
              {
                if (t%2 == 0) { L0 = A0; Lnext = Anext; }
                else {Lnext = A0; L0 = Anext; }
                lnext[Index3D(nx,ny,i,j,k-2*t)] = l0[Index3D(nx,ny,i,j,k-2*t+1)]
+l0[Index3D(nx,ny,i,j,k-2*t-1)]+l0[Index3D(nx,ny,i,j+1,k-2*t)]
+l0[Index3D(nx,ny,i,j-1,k-2*t)]+l0[Index3D(nx,ny,i+1,j,k-2*t)]
+l0[Index3D(nx,ny,i-1,j,k-2*t)]-l0[Index3D(nx,ny,i,j,k-2*t)]*fac;
              }
        }
```

Combining Blocking With Skewing

```
Nest2_group3: for (k_bk_1=1; k_bk_1<nz-1+16*timesteps; k_bk_1+=8) {
      omp_set_num_threads(2);
      #pragma omp  parallel {
Nest1: #pragma omp  for private(t,k,j,i,l0,lnext,j_bk_2,i_bk_3)
    for (t=max(0,(16+(k_bk_1-(nz-1)))/16); t<min(timesteps,(15+k_bk_1)/16); t+=1)
Nest3:  for (j_bk_2=1; j_bk_2<ny-1; j_bk_2+=32)
Nest4:  for (i_bk_3=1; i_bk_3<nx-1; i_bk_3+=32)
          for (k=0; k<min(8,nz-1-(k_bk_1-16*t)); k+=1)
          for (j=0; j<min(32,-j_bk_2+(-1+ny)); j+=1)
          for (i=0; i<min(32,-i_bk_3+(-1+nx)); i+=1)  {
           ……
        }  }  }
```

- Transformation specification
  - Nests: Nest1,Nest2,Nest3,Nest4
  - BlockLoops: inner_loop=Nest4 target=Nest2 factor=8,32,32
  - SkewLoops: inner_loop=Nest2 target=Nest1 factor=16
  - ParallelizeLoop: target=Nest1 private=t,k,j,i,l0,lnext

Recursion

- An universal programming pattern
  - Structural recursion: nesting of components
    - Recursive data partitioning a.k.a. cache oblivious algorithms
    - E.g., tree traversal and recursive matrix partitioning
  - Dynamic recursion: nesting of behaviors
    - Dynamic behavior at different nesting levels
    - E.g., dynamic nested fork-join of threads (how cilk_for and tbb::parallel_for are implemented
- Optimizations
  - Recursive parallelism using nested fork-join
  - Recursive partitioning of data or tasks
  - Dynamic adaptation of computation

Summary

- Compilers can only parallelize/optimize a small set of computational patterns
  - Dense matrix computations
  - Stencil computations
- Developers can help with annotations, but only to a limited extent
  - E.g., sparse matrix codes, transforming kernels to invoke exiting HPC libraries
  - May need to add new program transformations
    - But can they be automated?
- Alternative parallelization schemes
  - Pipelining
  - Recursive divide and conquer (nested parallelism)