Parallelism (PAR)

Data-aware task decomposition strategies
(or ... how to reduce memory coherence traffic in your parallelization)

Eduard Ayguadé, José Ramón Herrero (Q1),
Daniel Jiménez and Gladys Utrera

Computer Architecture Department
Universitat Politècnica de Catalunya

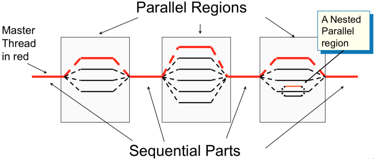
Course 2022/23 (Spring semester)

Learning material for this Unit

- Atenea: Unit 5 Data decomposition
  - Atenea quizz with motivation example
  - Going further: distributed-memory architectures video lesson (OPTIONAL)
- These slides to deep dive into the concepts in this Unit
- Collection of Exercises: problems in Chapter 5

Task creation in OpenMP (summary)

- `#pragma omp parallel`: One **implicit** task is created for each thread in the team (and immediately executed)



- `int omp_get_num_threads`: returns the number of threads in the current team. 1 if outside a parallel region
- `int omp_get_thread_num`: returns the identifier of the thread in the current team, between 0 and `omp_get_num_threads()-1`

Outline

Reducing memory coherence traffic: improving locality by data decomposition

Reducing memory coherence traffic: avoiding false sharing

Task vs. data decompositions

We can imagine[1] data to be distributed across the multiple memories in our NUMA multiprocessor system ...

... then, can we try to assign work so that tasks executed in a certain NUMA node access the data that is stored in the main memory of that NUMA node

- Use of implicit tasks created in `parallel` ...
- ... and the identifier of the thread they are running to decide what to execute

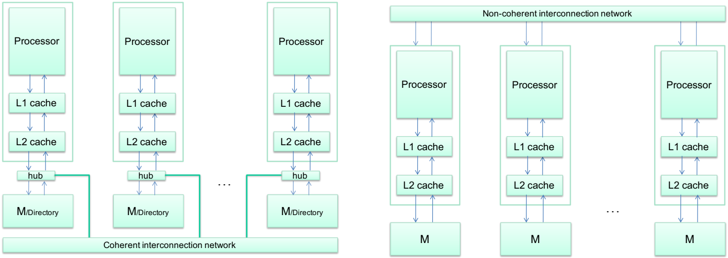
[1] Easy to imagine if we remember first touch, which brings data to the memory of the NUMA node that first touches it.

Why, when and how?

- Step 1: Identify the data used and/or produced in the computations
  - Output data, input data or both
- Step 2: Partition this data across various tasks
  - Linear or geometric decomposition
  - Recursive decomposition
- Step 3: Obtain a computational partitioning that corresponds to the data partitioning: owner-computes rule
- Step 4: In distributed–memory architectures, add the necessary data allocation and movement actions

Why, when and how? (cont.)

- Used to derive concurrency for problems that operate on large amounts of data focusing on the multiplicity of data
  - E.g. Elements in vectors, rows/columns/slices in matrices, elements in a list and subtrees in a tree
- ... for architectures in which memory plays a performance role



Guidelines for data decomposition

- Data can be partitioned in various ways – this may critically impact performance
  - Generate comparable amounts of work (for load balancing)
  - Maximize data locality (or minimize the need for task interactions)
    - Minimize volume of data involved in task interactions
    - Minimize frequency of interactions
    - Minimize contention and hot spots
  - Overlap computation with interactions to "hide" their effect
- Parametrizable data partition
  - number of data chunks, size, ...
- Simplicity

Example

Counting the instances of given itemsets in a database of transactions

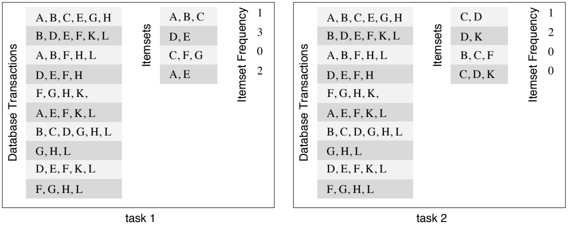
(a) Transactions (input), itemsets (input), and frequencies (output)

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,	C, D	1
A, E, F, K, L	D, K	2
B, C, D, G, H, L	B, C, F	0
G, H, L	C, D, K	0
D, E, F, K, L		
F, G, H, L		

Output data decomposition

- Partition of the output data structures across tasks. Input data structures may follow the same decomposition or require replication in order to avoid task interactions
- Example: the itemset frequencies are partitioned across tasks
  - The database of transactions needs to be replicated
  - The itemsets can be partitioned across tasks as well (reduce memory utilization)

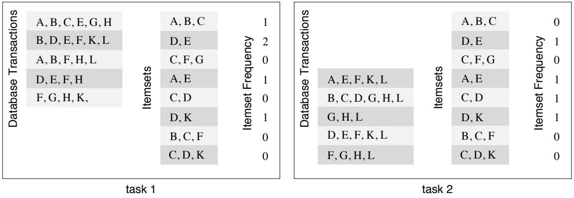
(b) Partitioning the frequencies (and itemsets) among the tasks



Input data decomposition

- Partition the input data structures across tasks. It may require combining partial results in order to generate the output data structures
- Example: the database transactions can be partitioned, but it requires the itemsets to be replicated. Final aggregation of partial counts for all itemsets

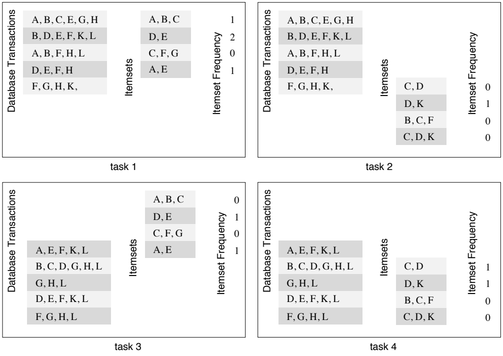
Partitioning the transactions among the tasks



Input *and* output data decomposition

- Input and output data decomposition could be combined
- Example: the database and itemsets (input) and counts (output) can be decomposed

Partitioning both transactions and frequencies among the tasks



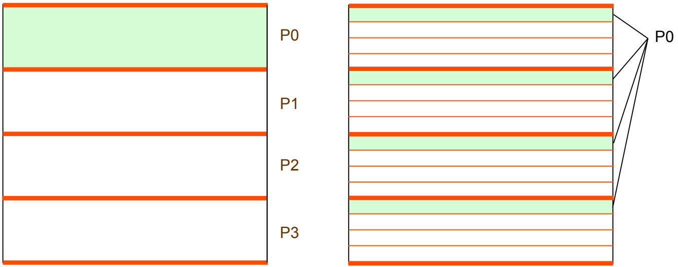
The Owner Computes rule

It defines who is responsible for doing the computations:

- In the case of output data decomposition, the owner computes rule implies that the output is computed by the task to which the output data is assigned.
- In the case of input data decomposition, the owner computes rule implies that all computations that use the input data are performed by the task to which the input is assigned.

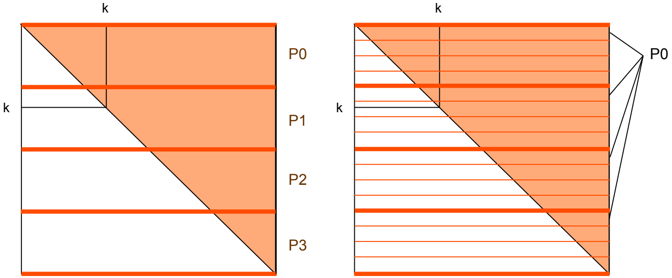
Data distributions for geometric decomposition (1)

Block (left) and cyclic (right) data decompositions



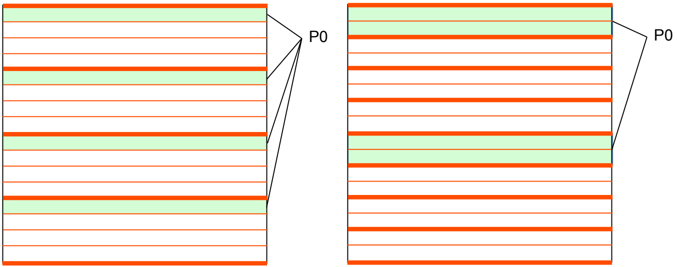
Data distributions for geometric decomposition (2)

Block (left) and cyclic (right) data decompositions in a triangular iteration space



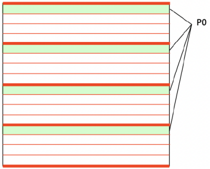
Data distributions for geometric decomposition (3)

Cyclic (left) and block-cyclic (right) data decompositions



Code transformations for data decompositions (1)

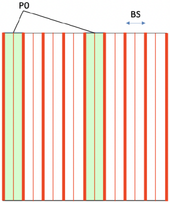
CYCLIC DATA DECOMPOSITION, by ROWS




```
#pragma omp parallel private (i, j)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (int i=my_id; i<N; i+= howmany)
        for (int j=0; j<N; j++)
           ... m[i][j] ... // Input or Output
    ...
}
```

BLOCK-CYCLIC DATA DECOMPOSITION, by COLUMNS

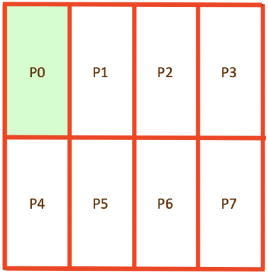



```
#pragma omp parallel private (i, j)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (int i=0; i<N; i++)
        for (int jj=my_id*BS; jj<N; jj+=howmany*BS)
             for (int j=jj; j<jj+BS; j++)
                 ... m[i][j] ... // Input or Output
        ...
}
```

Code transformations for data decompositions (2)

2D BLOCK / BLOCK DATA DECOMPOSITION




```
#pragma omp parallel private (i, j)
{
    int my_i = omp_get_thread_num()/4;
    int my_j = omp_get_thread_num()%4;
    int BSi = N/2;
    int BSj = N/4;
    int i_start = my_i * BSi;
    int i_end = i_start + BSi;
    int j_start = my_j * BSj;
    int j_end = j_start + BSj;

    for (int i=i_start; i<i_end; i++)
        for (int j=j_start; j<j_end; j++)
             ... m[i][j] ... // Input or Output
    ...
}
```

Code transformations for data decompositions (3)

BLOCK DATA DECOMPOSITION of vector v[M]

```
#pragma omp parallel private (i, j) num_threads(3)
{
    int my_id       = omp_get_thread_num();
    int howmany     = omp_get_num_threads();
    int index_start = my_id*M/howmany;
    int index_end   = (my_id+1)*M/howmany;

    for (int i=0; i<N; i++)
    {
        index = func(i);
        if (index>=index_start && index<index_end)
        {
           ... v[index] ... // Input or Output
        }
    }
}
```

Code transformations for data decompositions (4)

CYCLIC DATA DECOMPOSITION of vector v[M]

```
#pragma omp parallel private (i, j) num_threads(3)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (int i=0; i<N; i++)
    {
        index = func(i);
        if ((index%howmany)==my_id) // index%howmany => 0 1 2 0 1 2 0 1 2
        {
           ... v[index] ... // Input or Output
        }
    }
}
```

Code transformations for data decompositions (5)

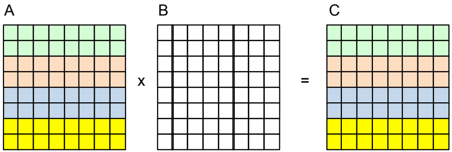
BLOCK-CYCLIC DATA DECOMPOSITION of vector v[M], BLOCK is BS elements

```
#pragma omp parallel private (i, j) num_threads(3)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (int i=0; i<N; i++)
    {
        index = func(i);
        // index/BS            0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
        // (index/BS)%howmany  0 0 0 1 1 1 2 2 2 0 0 0 1 1 1 2 2 2
        if (((index/BS)%howmany)==my_id)
        {
           ... v[index] ... // Input or Output
        }
    }
}
```

Example: matrix multiply using implicit tasks (1)

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE]) {
   for (int i=0; i<MATSIZE; i++)
      for (int j=0; j<MATSIZE; j++)
         for (int k=0; k<MATSIZE; k++)
            C[i][j] += A[i][k]*B[k][j];
}
```



Let's write the code for a geometric block data decomposition by rows applied to both matrices A (input) and C (output)

Example: matrix multiply using implicit tasks (2)

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE]) {
int i, j, k;

#pragma omp parallel
   {
   int myid = omp_get_thread_num();
   int howmany = omp_get_num_threads();
   int i_start =  myid * (MATSIZE/howmany);
   int i_end = i_start + (MATSIZE/howmany);
   if (myid == howmany-1) i_end = MATSIZE;

   for (int i=i_start; i<i_end; i++)
      for (int j=0; j<MATSIZE; j++)
         for (int k=0; k<MATSIZE; k++)
            C[i][j] += A[i][k]*B[k][j];
   }
}
```

Load balancing problem: last implicit task may get up to `howmany-1` additional iterations!

Example: matrix multiply using implicit tasks (3)

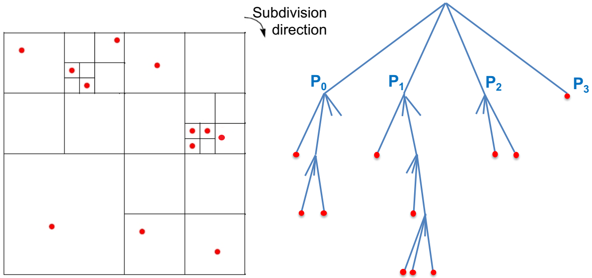
Let's reduce the load unbalance to 1 iteration at most ...

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE]) {
int i, j, k;

#pragma omp parallel
   {
   int myid = omp_get_thread_num();
   int howmany = omp_get_num_threads();
   int i_start =  myid * (MATSIZE/howmany);
   int i_end = i_start + (MATSIZE/howmany);
   int rem = MATSIZE % howmany;
   if (rem != 0) {
       if (myid < rem) {
           i_start += myid;
           i_end += (myid+1);
       } else {
           i_start += rem;
           i_end += rem;
       }
   }
   ...
   }
}
```

Data distributions for recursive decomposition (Optional)

Quadtree to represent particles in an N-body problem



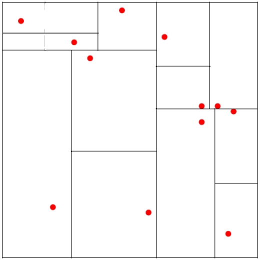
Particles in 2D space

Quatree representation

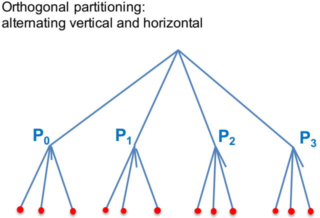
- Each leaf node stores position and mass for a body
- Other nodes store center of mass and total mass for all bodies below

Data distributions for recursive decomposition

Orthogonal distribution of the particles of an N-body, so that in each bi-partition the number of particles in each side is halved (load balancing)



Particles in 2D space



Quatree representation

Example: N-body computation (sequential)

Sequential code

```
void main() {
 // Initialize tree
 for (t=0; t<tmax; t++) {
   for (i=0;i<N;i++) doTimeStep(tree, node[i]);  // node[i] points to body i in the tree
   // Update the positions and velocities
   // Migrate bodies if required in the tree
 }
}
```

TreeNode structure

```
typedef struct {
   ...
   char     isLeaf
   TreeNode *quadrant[2][2];
   double   F; // force on node
   double   center_of_mass[3];
   double   mass_of_center;
   ...
} TreeNode;
```

Calculate forces implementation

```
void doTimeStep(TreeNode* subTree, TreeNode* body) {
  if(subTree) {
    if(!subTree->isLeaf && !distant(subTree, body)) {
      for(int i=0; i<2; i++)
       for(int j=0; j<2; j++)
           doTimeStep(subTree->quadrant[i][j], body);
    }
    else // subtree is a leaf
      calcForces(subTree, body); // update F field for body
  }
}
```

A distant subtree is approximated as a single body with mass/center

Example: N-body computation (data decomposition)

Each thread computes the forces in each node caused by the sub-tree assigned to it

```
void main() {
   // initialize tree
   ...
   #pragma omp parallel private(subtree) num_threads(4)
   {
      // Each thread will get a subtree
      subtree = partition(tree, omp_get_thread_num(), omp_get_num_threads());
      for (int t=0; t<tmax; t++) {
          for (int i=0;i<N;i++) doTimeStep(subtree, node[i]);
          // Update the positions and velocities
          ...
          if (...) {   // Migrate bodies if required in the quad-tree
              ...
              subtree = partition(tree, omp_get_thread_num(), omp_get_num_threads());
          }
      }
   }
}
```

Outline

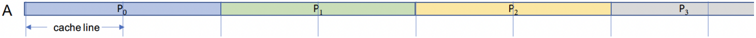
Reducing memory coherence traffic: improving locality by data decomposition

Reducing memory coherence traffic: avoiding false sharing

Examples/situations of false sharing ... (1)

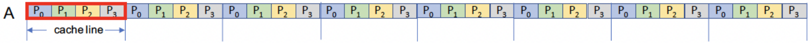
```
#pragma omp parallel
{
    int myid = omp_get_thread_num();
    int BS = n / omp_get_num_threads();
    for (i=myid*BS; i<(myid+1)*BS; i++) A[i] = foo(i*23);
}
```

Possible solution: introduce some load unbalance, so that BS corresponds with a number of elements that fit in a number of complete cache lines

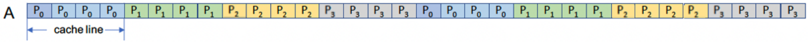


Examples/situations of false sharing ... (2)

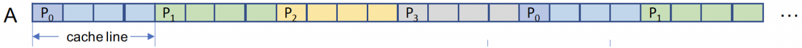
```
#pragma omp parallel
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    for (i=myid; i<n; i+=howmany) A[i] = foo(i*23);
}
```



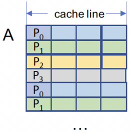
Possible solution: make larger chunk size (p.e. 4) → block-cyclic



Alternative solution: Add padding – i.e. one element per cache line



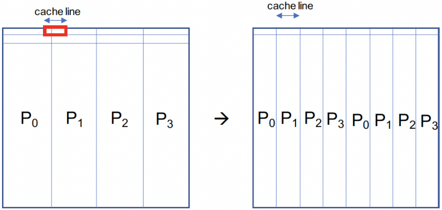
How? `int A[100];` → `A[100][4];`
And the access needs to change ...
`A[i][0] = foo(i*3);`



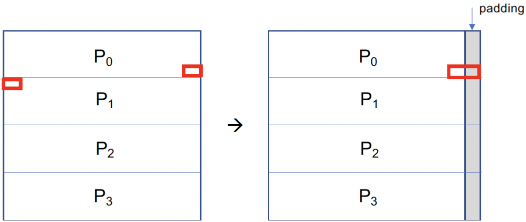
Examples/situations of false sharing ... (3)

In 2D matrices we can also have false sharing problems ... solutions ?

- block → block-cyclic



- Add some padding



Parallelism (PAR)

Data-aware task decomposition strategies
(or ... how to reduce memory coherence traffic in your parallelization)

Eduard Ayguadé, José Ramón Herrero (Q1),
Daniel Jiménez and Gladys Utrera

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2022/23 (Spring semester)