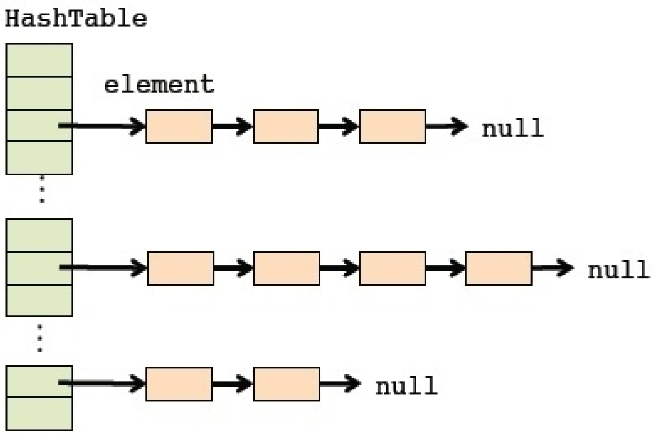
Data Decomposition: Problem 2



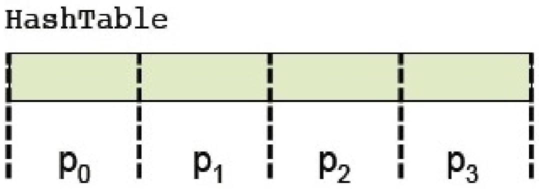

```
#define SIZE_TABLE 1048576
typedef struct {
    int data;
    element * next;
} element;

element * HashTable[SIZE_TABLE];
```

Taking as a starting point the following sequential code:

```
#define MAX_ELEM 1024
int main() {
    int ToInsert[MAX_ELEM], num_elem, index;
    ...
    for (i = 0; i < num_elem; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        insert_elem (ToInsert[i], index);
    }
    ...
```

we can see that the function `hash_function` returns the entry of the table (between 0 and `SIZE_TABLE`-1 where a determined element has to be inserted and the function `insert_elem` inserts that element in the corresponding position inside the chained list pointed by the entry `index` of the table `HashTable`.

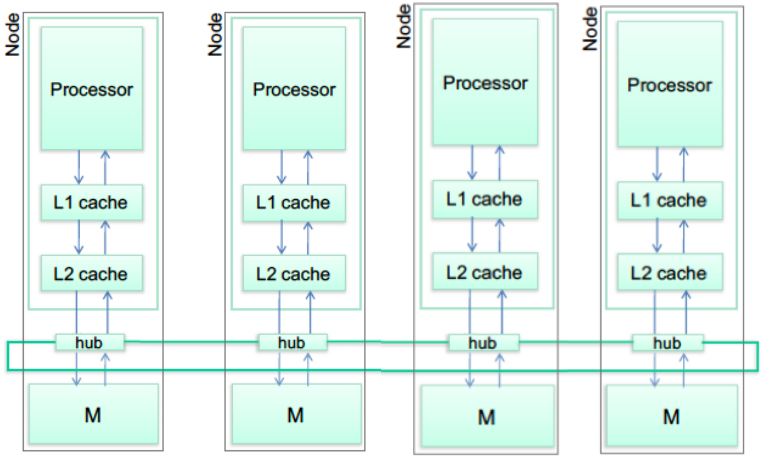
We ask: Redefine the data structures, if necessary, and write an OpenMP parallel version that obeys a data decomposition, where each thread has to do all the insertions that have to be done in `SIZE_TABLE/P` consecutive entries of the table `HashTable`, as shown in the next figure (being P the number of threads).



```c
#define MAX_ELEM 1024
typedef struct {
    int data;
    element *next;
} element *HashTable[SIZE_TABLE];

int main() {
    int ToInsert[MAX_ELEM], num_elem, index;
    …
    for (i = 0; i < num_elem; i++) {
        index = hash_function (ToInsert[i], SIZE_TABLE);
        insert_elem (ToInsert[i], index);
    }
    …
```

HashTable : READ/ WRITE access

ToInsert : only READ access

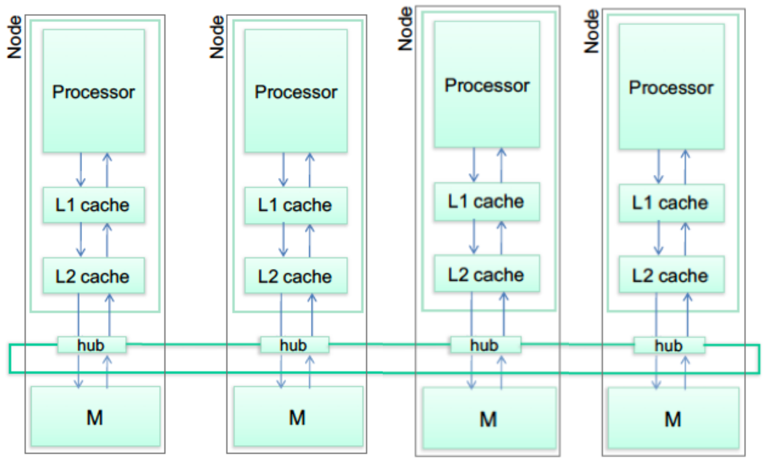


On a NUMA system, memory access costs are different (and coherence associated traffic) depending on where the data is allocated. Exploring possible scenarios we may have:

	ToInsert block distributed	ToInsert in single node
HashTable distributed	ToInsert[MAX_ELEM]: P_0 | P_1 | P_2 | P_3 HashTable[SIZE_TABLE]: P_0 | P_1 | P_2 | P_3	ToInsert[MAX_ELEM]: P_0 HashTable[SIZE_TABLE]: P_0 | P_1 | P_2 | P_3
HashTable in single node	ToInsert[MAX_ELEM]: P_0 | P_1 | P_2 | P_3 HashTable[SIZE_TABLE]: P_0	ToInsert[MAX_ELEM]: P_0 HashTable[SIZE_TABLE]: P_0

Parallelization: Task decomposition with implicit tasks

```
int main() {
int ToInsert[MAX_ELEM], index;
...



    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        insert_elem (ToInsert[i], index);
    }

...
```



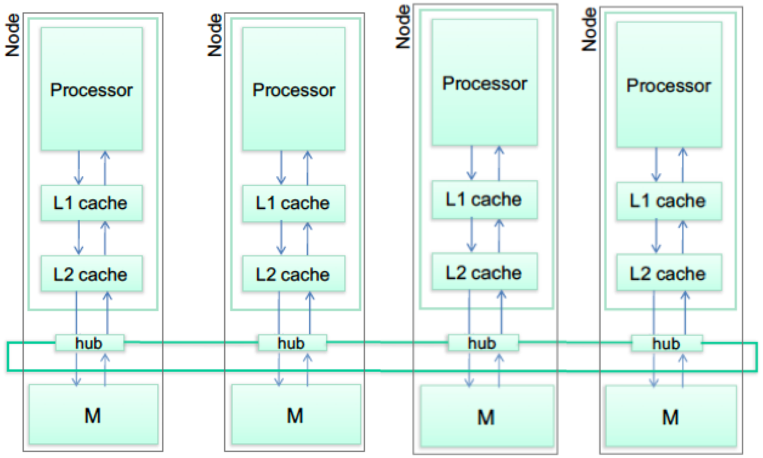
On a NUMA system, memory access costs are different (and coherence associated traffic) depending on where the data is allocated.
Exploring possible scenarios we may have:

	ToInsert block distributed	ToInsert in single node
HashTable distributed	ToInsert[MAX_ELEM]: P_0 | P_1 | P_2 | P_3 HashTable[SIZE_TABLE]: P_0 | P_1 | P_2 | P_3	ToInsert[MAX_ELEM]: P_0 HashTable[SIZE_TABLE]: P_0 | P_1 | P_2 | P_3
HashTable in single node	ToInsert[MAX_ELEM]: P_0 | P_1 | P_2 | P_3 HashTable[SIZE_TABLE]: P_0	ToInsert[MAX_ELEM]: P_0 HashTable[SIZE_TABLE]: P_0

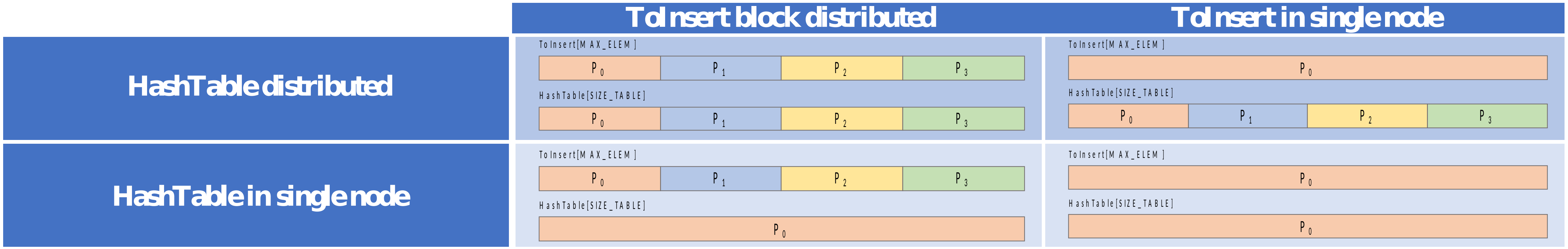
Parallelization: Task decomposition with implicit tasks

```
int main() {
int ToInsert[MAX_ELEM], index;
...
#pragma omp parallel private(index, i)
{
    #pragma omp for private(index)
    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        insert_elem (ToInsert[i], index);
    }
}
...
```

Any data race ??

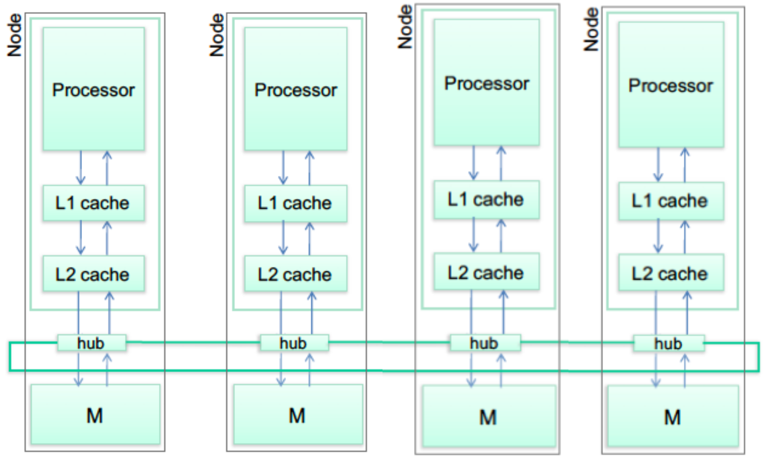


On a NUMA system, memory access costs are different (and coherence associated traffic) depending on where the data is allocated. Exploring possible scenarios we may have:



Parallelization: Task decomposition with implicit tasks

```
int main() {
int ToInsert[MAX_ELEM], index;
omp_lock_t locks[SIZE_TABLE];
...
#pragma omp parallel private(index, i)
{
    #pragma omp for private(index)
    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        omp_set_lock(&locks[index]);
        insert_elem (ToInsert[i], index);
        omp_unset_lock(&locks[index]);
    }
}
...
```

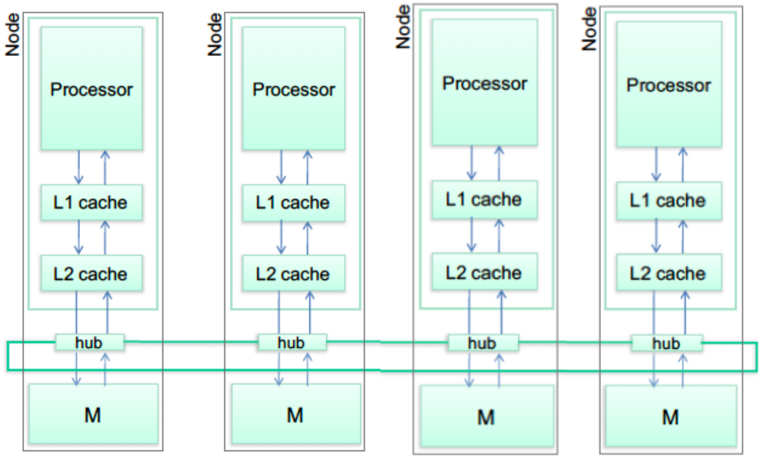


On a NUMA system, memory access costs are different (and coherence associated traffic) depending on where the data is allocated. Exploring possible scenarios we may have:

	ToInsert block distributed	ToInsert in single node
HashTable distributed	ToInsert[MAX_ELEM]: P_0, P_1, P_2, P_3 HashTable[SIZE_TABLE]: P_0, P_1, P_2, P_3	ToInsert[MAX_ELEM]: P_0 HashTable[SIZE_TABLE]: P_0, P_1, P_2, P_3
HashTable in single node	ToInsert[MAX_ELEM]: P_0, P_1, P_2, P_3 HashTable[SIZE_TABLE]: P_0	ToInsert[MAX_ELEM]: P_0 HashTable[SIZE_TABLE]: P_0

Parallelization: Task decomposition with implicit tasks

```
int main() {
int ToInsert[MAX_ELEM], index;
omp_lock_t locks[SIZE_TABLE];
...
#pragma omp parallel private(index, i)
{
    #pragma omp for private(index)
    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        omp_set_lock(&locks[index]);
        insert_elem (ToInsert[i], index);
        omp_unset_lock(&locks[index]);
    }
}
...
```



With this parallelization the **data distribution** may impact in the following ways:

READ ONLY ACCESS

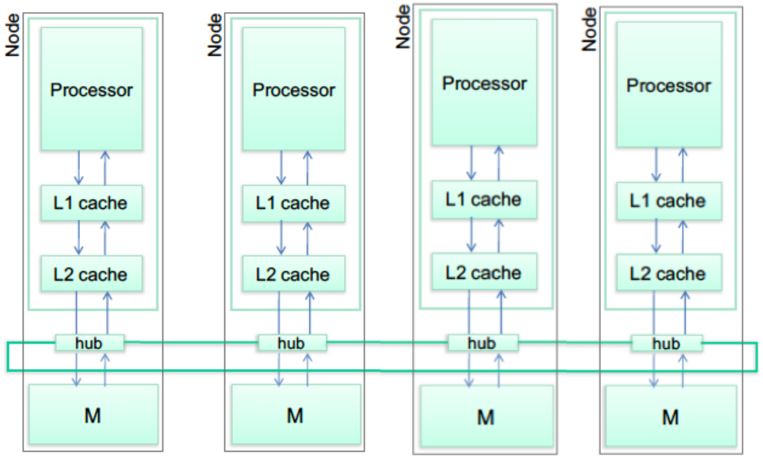
READ / WRITE ACCESS

	ToInsert block distributed	ToInsert in single node
HashTable distributed	ToInsert Local HashTable: Remote	ToInsert: Remote with bottleneck HashTable: Remote
HashTable in single node	ToInsert: Local HashTable: Remote with bottleneck	ToInsert: Remote with bottleneck HashTable: Remote with bottleneck.

→ **Not efficient: Remote access (read and update) with regular synchronizations and access bottleneck…**

Parallelization: Let's think on DATA …

```
#pragma omp parallel private(index, i)
{
// each thread "self-assigns" the range of elements to process
    int myid = omp_get_thread_num();
    int numprocs = omp_get_num_threads();
    int lower = myid * (MAX_ELEM/numprocs);
    int upper = lower + (MAX_ELEM/numprocs);
    if (myid == numprocs-1) upper = MAX_ELEM;
    for (i = lower; i < upper; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        omp_set_lock(&locks[index]);
        insert_elem (ToInsert[i], index);
        omp_unset_lock(&locks[index]);
    }
}
```



With this parallelization the **data distribution** may impact in the following ways:

READ ONLY ACCESS

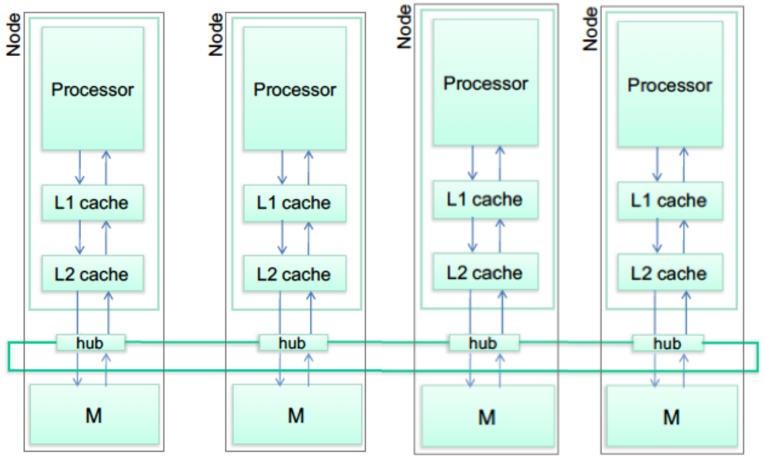
READ / WRITE ACCESS

	ToInsert block distributed	ToInsert in single node
HashTable distributed	ToInsert Local HashTable: Remote	ToInsert: Remote with bottleneck HashTable: Remote
HashTable in single node	ToInsert: Local HashTable: Remote with bottleneck	ToInsert: Remote with bottleneck HashTable: Remote with bottleneck.

→ Not efficient: Remote access (read and update) with regular synchronizations and bottleneck..

Parallelization: Let's think on DATA…. OUTPUT – BLOCK GEOMETRIC DATA DECOMPOSITION

```
#pragma omp parallel private(index, i)
{
// each thread "self-assigns" the range of elements to process

    // Instead of distributing ToInsert between threads
    // Let's distribute HashTable ...



    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        insert_elem (ToInsert[i], index);
    }
}
```



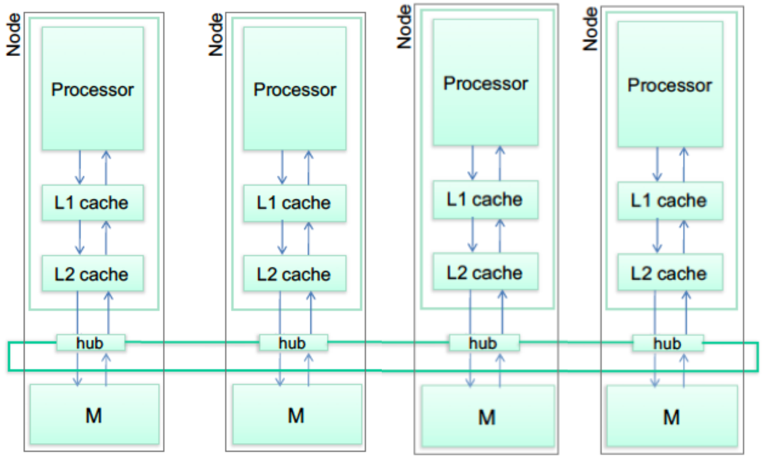
READ ONLY ACCESS

READ / WRITE ACCESS	ToInsert block distributed	ToInsert in single node
HashTable distributed	ToInsert Remote HashTable: Local	ToInsert: Remote with bottleneck HashTable: Local
HashTable in single node	ToInsert: Remote HashTable: Remote with bottleneck	ToInsert: Remote with bottleneck HashTable: Remote with bottleneck.

Parallelization: Let's think on DATA…. OUTPUT – BLOCK GEOMETRIC DATA DECOMPOSITION

```
#pragma omp parallel private(index, i)
{
// each thread "self-assigns" the range of elements to process
    int myid = omp_get_thread_num();
    int numprocs = omp_get_num_threads();
    int lower = myid * (SIZE_TABLE/numprocs);
    int upper = lower + (SIZE_TABLE/numprocs);
    if (myid == numprocs-1) upper = SIZE_TABLE;

    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        insert_elem (ToInsert[i], index);
    }
}
```

Just index belonging to the range [lower .. upper)

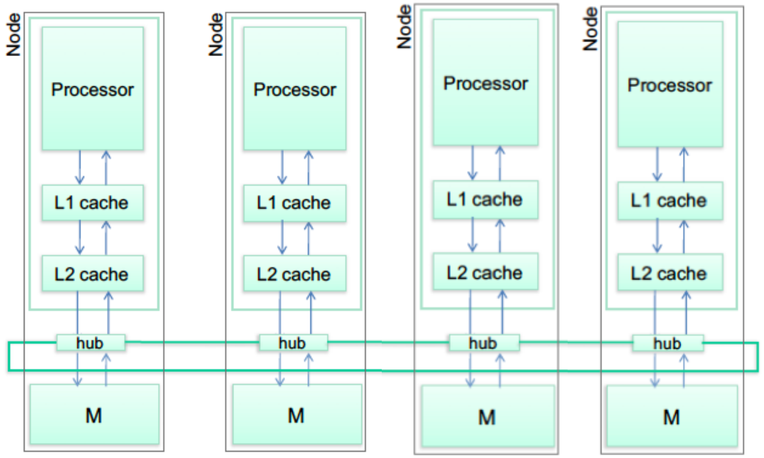


READ ONLY ACCESS

READ / WRITE ACCESS	ToInsert block distributed	ToInsert in single node
HashTable distributed	ToInsert Remote HashTable: Local	ToInsert: Remote with bottleneck HashTable: Local
HashTable in single node	ToInsert: Remote HashTable: Remote with bottleneck	ToInsert: Remote with bottleneck HashTable: Remote with bottleneck.

Parallelization: Let's think on DATA…. OUTPUT – BLOCK GEOMETRIC DATA DECOMPOSITION

```
#pragma omp parallel private(index, i)
{
// each thread "self-assigns" the range of elements to process
    int myid = omp_get_thread_num();
    int numprocs = omp_get_num_threads();
    int lower = myid * (SIZE_TABLE/numprocs);
    int upper = lower + (SIZE_TABLE/numprocs);
    if (myid == numprocs-1) upper = SIZE_TABLE;

    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        if ((index>=lower) && (index<upper)) insert_elem (ToInsert[i], index);
    }
}
```



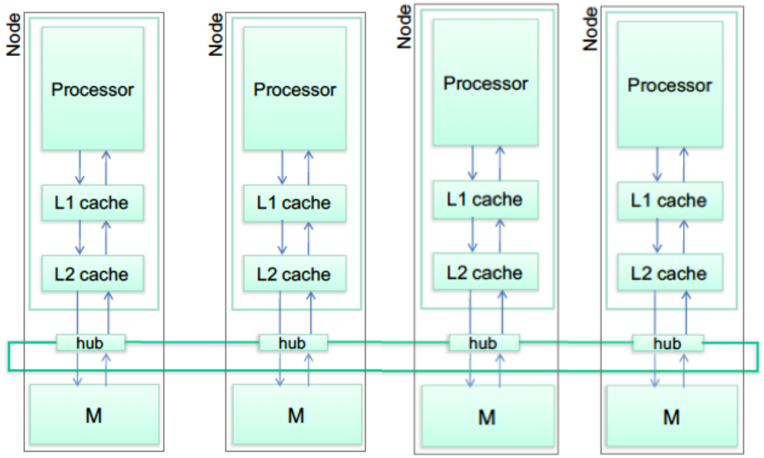
// WITHOUT SYNCHRONIZATION

READ ONLY ACCESS

READ / WRITE ACCESS	ToInsert block distributed	ToInsert in single node
HashTable distributed	ToInsert Remote HashTable: Local	ToInsert: Remote with bottleneck HashTable: Local
HashTable in single node	ToInsert: Remote HashTable: Remote with bottleneck	ToInsert: Remote with bottleneck HashTable: Remote with bottleneck.

Parallelization: Let's think on DATA…. OUTPUT – BLOCK GEOMETRIC DATA DECOMPOSITION

```
#pragma omp parallel private(index, i)
{
// each thread "self-assigns" the range of elements to process
    int myid = omp_get_thread_num();
    int numprocs = omp_get_num_threads();
    int lower = myid * (SIZE_TABLE/numprocs);
    int upper = lower + (SIZE_TABLE/numprocs);
    if (myid == numprocs-1) upper = SIZE_TABLE;

    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);   // WITHOUT SYNCHRONIZATION
        if ((index>=lower) && (index<upper)) insert_elem (ToInsert[i], index);
    }
}
```



READ ONLY ACCESS

READ / WRITE ACCESS	ToInsert block distributed	ToInsert in single node
HashTable distributed	ToInsert Remote HashTable: Local	ToInsert: Remote with bottleneck HashTable: Local
HashTable in single node	ToInsert: Remote HashTable: Remote with bottleneck	ToInsert: Remote with bottleneck HashTable: Remote with bottleneck.

→ with NO synchronizations … Local access for read/write, Remote access for read only, without congestion

```
#pragma omp parallel private(index, i)
{
    int myid = omp_get_thread_num();
    int numprocs = omp_get_num_threads();
    int lower = myid * (MAX_ELEM/numprocs);
    int upper = lower + (MAX_ELEM/numprocs);
    if (myid == numprocs-1) upper = MAX_ELEM;
    for (i = lower; i < upper; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        omp_set_lock(&locks[index]);
        insert_elem (ToInsert[i], index);
        omp_unset_lock(&locks[index]);
    }
}
```

Distribute **ToInsert** (input vector) between threads

INPUT DATA DECOMPOSITION

Distribute **HashTable** (output vector) between threads

OUTPUT DATA DECOMPOSITION

```
#pragma omp parallel private(index, i)
{
    int myid = omp_get_thread_num();
    int numprocs = omp_get_num_threads();
    int lower = myid * (SIZE_TABLE/numprocs);
    int upper = lower + (SIZE_TABLE/numprocs);
    if (myid == numprocs-1) upper = SIZE_TABLE;
    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(element[i], SIZE_TABLE);
        if ((index>=lower) && (index<upper))
            insert_elem (element[i], index);
    }
}
```

Parallelization: Let’s think on DATA…. OUTPUT – **CYCLIC** GEOMETRIC DATA DECOMPOSITION

```
#pragma omp parallel private(index, i)
{
// each thread “self-assigns” the range of elements to process
    int myid = omp_get_thread_num();
    int nt   = omp_get_num_threads();
    // index%nt determines which is the thread that should
    // address this output
    // Assume nt=3
    // 0 1 2 3 4 5 6 7 8 9 10 - index
    // 0 1 2 0 1 2 0 1 2 0 1  - index%3 → my_id

    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        If ((index%nt)==my_id) insert_elem (ToInsert[i], index);
    }
}
```

// WITHOUT SYNCHRONIZATION

Parallelization: Let’s think on output **BLOCK-CYCLIC** GEOMETRIC DATA DECOMPOSITION (**block=2**)

```
#pragma omp parallel private(index, i)
{
// each thread “self-assigns” the range of elements to process
    int myid = omp_get_thread_num();
    int nt   = omp_get_num_threads();
    // Assume nt=3, block=2
    // 0 1 2 3 4 5 6 7 8 9 10 11- index
    // 0 0 1 1 2 2 3 3 4 4 5  5 – (index/2)
    // 0 0 1 1 2 2 0 0 1 1 2  2 – ((index/2)%nt) → my_id

    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        If ((index/2)%nt==my_id) insert_elem (ToInsert[i], index);
    }
}
```

// WITHOUT SYNCHRONIZATION