Optimizing XL Unified Parallel C Programs

Contents

Compiler optimization. 1
Shared object access optimizations 1
    Shared object access privatization 1
    Shared object access coalescing 2
    Shared object remote updating 3
    Array idiom recognition 4
Parallel loop optimizations 5
    Loop reshaping 5
    Loop versioning 7

Compiler optimization

The XL Unified Parallel C compiler offers a comprehensive set of performance enhancing optimizations that you can use to develop high performance parallel applications.

The compiler uses a runtime system (RTS) that is designed for scalability in a large parallel computing environment. The RTS exposes an application programming interface (API) to the compiler, and the compiler calls the API functions in the RTS for efficient manipulation of the shared data distributed over a large number of threads.

In the partitioned global address space (PGAS) programming model, a thread can access a shared object either locally or remotely. A local access is an access to a shared object in a thread that maps to the same address partition as the accessing thread. A remote access is an access to a shared object in a thread that maps to a different address partition than the accessing thread.

RTS functions for remote accesses require a message to be sent to the owner thread to request or update the value of the shared object, and this requires extra time as compared to the local access. To reduce the communication time and thus improve application performance, the compiler can bypass RTS calls and directly access the shared data when it can prove that an access is local.

In the case of remote accesses, the compiler can, in some cases, analyze the locality of each access and use different optimization techniques to reduce the communication time of a remote network transfer.

Shared object access optimizations

This topic and the following subtopics describe the optimizations that the compiler can perform on shared object accesses.

In Unified Parallel C applications, shared objects are distributed into different threads. For example, elements of a shared array are distributed among all the threads of the application based on the blocking factor. When a thread needs to access a portion of an array, some of the accesses might be local to the issuing thread and some might be remote. Without optimizations, the compiler must call the RTS API to read the value of a shared array element or to assign a new value to it. Calling the RTS API to read or write a local shared object value impacts application performance significantly. In the case of a remote shared access, the processing time of an RTS call is compounded by the necessary network communication latency. The shared object access optimizations have the objective of eliminating unnecessary RTS calls for local shared accesses, and reduce the network communication time for remote shared accesses.

Shared object access privatization

When the compiler can prove that an access to a shared object is always local to the accessing thread, it can privatize the access and manipulate the data without calling RTS functions.

Typically, the compiler translates accesses to shared objects by generating appropriate RTS function calls. Any call to the RTS requires some processing time. When the underlying memory of the shared object is in the local address space of the accessing thread, the thread can access the object without issuing a runtime system call. The compiler achieves this by converting the references to the shared object to traditional C references, thus bypassing the RTS function calls.

Take the following code as an example:

```
shared [N/THREADS] int A[N], B[N], C[N];

void foo() {
  int i;
  upc_forall(i=0; i<N; ++i; &A[i]) {
    A[i] = B[i] + C[i];
  }
}
```

In this example, the thread that executes the i′th loop iteration is the thread that has affinity with the array element A[i]. The B and C arrays have the same blocking factor as A; therefore, all the shared accesses performed in the loop body are local to the issuing thread. The compiler can privatize all the 3 shared array accesses and it does not need to generate RTS calls for translating these accesses.

Data-parallel applications tend to spend most of the computation time in loops. Loop iterations and shared data are typically distributed into multiple threads. When the shared object access privatization optimization is enabled, each thread can access its local shared data directly without calling the RTS functions. This improves application performance in turn.

Note: When the compiler cannot prove that an access to a shared object is local, which means the compiler cannot prove that the accessing thread has affinity to the shared object, it must assume that the access is remote and call the RTS functions to determine the locality of the object.

Shared object access coalescing

The shared object access coalescing optimization is designed to replace multiple shared remote accesses to elements of a shared array having affinity to the same thread with a single remote access. This optimization can improve the performance of an application by reducing the number of communication calls necessary to transfer remote data between the accessing thread and the owner thread.

The compiler typically translates a remote access to a shared object into a function call to the RTS. For example, reading or writing multiple shared array elements that have affinity to a remote thread causes the compiler to generate an RTS call for each of the array elements read or modified. The compiler can combine multiple remote accesses into a single access when it can prove that the shared array elements being accessed satisfy the following conditions:

- The shared array elements have affinity to the same thread.
- The shared array elements accesses have constant stride.

When multiple remote accesses are combined, the compiler issues a single call to the RTS, thus reducing the number of the communication messages between the accessing thread and the owner thread.

Take the following code as an example:

```
#define ARRAY_SIZE 100
#define BF (ARRAY_SIZE/THREADS)

shared [BF] int A[ARRAY_SIZE];
shared [BF] int B[ARRAY_SIZE];

void loop() {
  int i;
  upc_forall(i=0; i<ARRAY_SIZE-BF; i++; &A[i]) {
    A[i] = B[i+BF];
  }
}
```

If the example code is run using 2 static threads on 2 nodes, then the access `B[i+BF]` is executed by thread 0 for all loop iterations. Elements `B[50]` through `B[99]` have affinity to thread 1 that is located on a remote node.

In this example, the compiler attempts to coalesce the accesses to `B[i+BF]` together instead of issuing 50 calls to dereference individual elements of the shared array.

Shared object remote updating

A remote update of a shared object is the update of a memory location that has affinity to a thread running on a remote node. The compiler can update a remote shared object using an RTS function that requires less communication time.

When one thread updates a shared object that has affinity to a thread running on a remote node, the two threads performs the following operations:

1. The accessing thread makes a request to the owner thread to retrieve the value of the object and the owner thread responds with the requested value.
2. The accessing thread locally update the value received from the owner thread.
3. The accessing thread sends the new value to the owner thread.

This communication pattern requires 2 calls between the accessing thread and the owner thread. Communication calls impact the performance of an application, and the impact can be significant when the communication is carried over a network.

To mitigate such impact, the compiler can perform the following operations:

1. Recognize the update operation.
2. Use an RTS function to instruct the remote thread that owns the array element to perform the update locally.

In this case, the RTS function only makes a single communication call from the accessing thread to the owner thread, with the update instructions contained in the call. The update of the shared object is performed by the remote thread, and the number of communication calls is reduced from 2 to 1.

Consider the following example:

```
shared int histogram[N];
extern int my_poll();

int main() {
  upc_forall(int i=0; i<N; i++; i) {
    histogram[i] = 0;
  }
  upc_barrier;
  for(int i=0; i<K; i++) {
    histogram[my_poll()] += 1;
  }
}
```

In this example, we assume that the call to my_poll() produces a pseudo-random number ranging from 0 to N-1. Therefore the shared access to histogram[my_poll()] might be remote to the accessing thread, and in that case its value is updated remotely from another thread. When the shared object remote updating optimization is enabled, the number of communication calls required by this program is reduced and thus the performance of the program is improved.

Array idiom recognition

The compiler can recognize commonly used array manipulation code and transform the code into more effective Unified Parallel C string handing functions.

Unified Parallel C programs often include statements that manipulate the values of local or shared arrays. Consider the following code as an example:

```
shared [BF] int a[N];
int b[N];

int main () {
  int i;
  if (MYTHREAD==0) {
    for (i=0; i<N; i++)
      a[i]=b[i];
  }
}
```

In this example, the statement a[i]=b[i] copies a local array to a shared array. Each loop iteration copies one array element at a time. A naive translation of this code results in fine-grained communication. To reduce the number of communication calls, the compiler can perform the following steps:

1. Identify commonly used array manipulation statements (array idioms).
2. Classify the operations based on the access types.
3. Transform the identified array idiom pattern into a call to one of the Unified Parallel C string handling function calls (upc_memset, upc_memget, upc_memput, or upc_memcpy).

Compared with the operation of manipulating each array element in each loop iteration, the string handling functions perform block-based data manipulation, which can reduce the read/write latency and communication time.

With the array idiom recognition optimization enabled, the compiler transforms the loop in the preceding example as follows:

```
...
for (i=0; i<N; i+=BF) {
  upc_memput(&a[i], &b[i], BF*sizeof(b[i]));
}
...
```

Note: The array idiom recognition optimization only considers relaxed shared accesses.

Parallel loop optimizations

This section describes the optimizations that the compiler performs on the `upc_forall` loop.

The `upc_forall` loop is extensively used in XL Unified Parallel C applications to distribute work across all the threads, and thus the performance of the loops significantly affects the performance of the applications.

To improve the performance of the `upc_forall` loop, the compiler uses the following optimization techniques:

- Create multiple versions of the loop in preparation for locality analysis.
- Reshape the loop body so that no affinity test branch statement is used.

Loop reshaping

The compiler can optimize a `upc_forall` loop so that it can be executed more efficiently.

The compiler typically transforms a `upc_forall` loop into a `for` loop. It can insert a branch statement into the loop body to protect the execution of the loop so that each thread executes a subset of the loop iterations.

The following two examples illustrate the transformation of a `upc_forall` loop by inserting a branch statement.

Example 1: Transforming a `upc_forall` loop that has the integer affinity:

```
shared double a[N], b[N], c[N];

void foo_1 () {
  int i;
  upc_forall (i=0; i<N; i++, i)
    a[i]=b[i]+c[i];
}
```

The code in this example is transformed as follows if the compiler inserts a branch statement based on the integer affinity test:

```
shared double a[N], b[N], c[N];

void foo_1 () {
  int i;
  for (i=0; i<N; i++) {
    if ((i%THREADS)==MYTHREAD)
      a[i]=b[i]+c[i];
  }
}
```

Example 2: Transforming a `upc_forall` loop that has the pointer-to-shared affinity:

```
shared double a[N], b[N], c[N];

void foo_2 () {
  int i;
  upc_forall (i=0; i<N; i++, &a[i])
    a[i]=b[i]+c[i];
}
```

The code in this example is transformed as follows if the compiler inserts a branch statement based on the pointer-to-shared affinity test:

```
shared double a[N], b[N], c[N];

void foo_2 () {
  int i;
  for (i=0; i<N; i++) {
    if (upc_threadof(&a[i])==MYTHREAD)
      a[i]=b[i]+c[i];
  }
}
```

In these examples, the compiler inserts a branch statement into the loop body. The branch statement is evaluated in every loop iteration, which can impact the performance of the application and limit the scalability of the loop.

To address this issue, the XL Unified Parallel C compiler optimizes the `upc_forall` loop without inserting the branch statement. The following subtopics provide details about the loop reshaping optimization.

Reshaping loops with integer affinity

The compiler can optimize the `upc_forall` loop with the integer affinity without inserting a branch statement.

To avoid inserting a branch statement into the loop body, the compiler can the change the lower bound of the loop to `MYTHREAD` and the increment of the loop to `THREADS` in each thread.

Consider the `upc_forall` loop in the following example:

```
shared double a[N], b[N], c[N];

void foo () {
  int i;
  upc_forall (i=0; i<N; i++; i)
    a[i]=b[i]+c[i];
}
```

With the loop reshaping optimization enabled, the compiler transforms the loop in the example as follows:

```
shared double a[N], b[N], c[N];

void foo () {
  int i;
  for (i=MYTHREAD; i<N; i+=THREADS)
    a[i]=b[i]+c[i];
}
```

Without any branch statement evaluation, the loop can be executed more efficiently and is more scalable.

Reshaping loops with pointer-to-shared affinity

The compiler can optimize the `upc_forall` loop with the pointer-to-shared affinity without inserting a branch statement.

To avoid inserting a branch statement into the loop body, the compiler can reshape a `upc_forall` loop into two loops – the outer loop iterates over blocks of the shared array owned by each thread and the inner loop iterates through each block.

Consider the following code as an example:

```
shared [2] int a[N], b[N], c[N];
int i;

upc_forall (i=0; i<N; i++; &a[i]) {
  a[i]=b[i]+c[i];
}
```

In this example, the pointer-to-shared affinity test is used in the loop. With the loop reshaping optimization enabled, the compiler transforms the preceding code as follows:

```
shared [2] int a[N], b[N], c[N];
int i, j;

for (i=MYTHREAD*2; i<N; i+=THREADS*2) {
  for (j=i; j<i+2; j++) {
    a[j]=b[i]+c[i];
  }
}
```

The transformed loop does not contain any affinity test branch statement, and the loop can be executed more efficiently and is more scalable.

Note: If the compiler cannot determine how to optimize a `upc_forall` loop, it must insert a branch statement into the loop body based on the affinity test.

Loop versioning

The compiler can create multiple versions of the `upc_forall` loop so that it can perform locality analysis on indirect accesses to a shared object through a pointer-to-shared.

Locality analysis is an important optimization technique. It helps the compiler identify opportunities for shared object privatization and coalescing.

To prepare a `upc_forall` loop containing pointer-to-shared dereferences for locality analysis, the compiler performs the following steps:

1. Create two control flow paths, the *true* path and the *false* path.
2. Compute the expression that, when evaluates to *true*, leads the program control branch into the *true* path. The condition the compiler generates guarantees that all the shared accesses through a pointer-to-shared in the *true* path have known locality, that is, the pointer-to-shared points to the first array element in a block, and the pointer is loop invariant.
3. Create two copies of the original loop and place each copy of the loop in one of the two control paths.

In this way, the compiler can perform locality analysis on the indirect accesses to shared objects through a pointer-to-shared in the *true* path.

Consider the following example:

```
...
void foo(shared [BF] int *p1, shared [BF] int *p2) {
  int i;
  upc_forall (i=0; i<N; ++i; &p1[i])
    p1[i]=p2[i+1];
}
```

When the loop versioning optimization is enabled, the compiler transforms the code in this example as follows:

```
...
void foo(shared [BF] int *p1, shared [BF] int *p2) {
  int i;
  _Bool ver_p1=(upc_phaseof(p1)==0)?1:0;
  _Bool ver_p2=(upc_phaseof(p2)==0)?1:0;
  if (ver_p1 && ver_p2) {
    upc_forall (i=0; i<N; i++; &p1[i])
      p1[i]=p2[i+1];
  } else {
    upc_forall (i=0; i<N; i++; &p1[i])
      p1[i]=p2[i+1];
  }
}
```

In the transformed code, p1 and p2 are considered for locality analysis in the *true* path, whereas they are not considered in the *false* path. In this particular example, note that in the *false* path, p1 has the exact same index as the affinity expression and the compiler can still prove that it is always local, so the compiler can always privatize p1.

The loop versioning transformation helps the compiler prepare the indirect accesses to shared object through a pointer-to-shared in the upc_forall loop for locality analysis, and can consequently contribute to the improvement of application performance.