Programming Data and Task Parallelism with Chapel

Peiyi Tang
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR 72204

Abstract
Chapel is a new global-view parallel programming language developed by Cray Inc. that represents a new direction in programming parallel machines. In this paper, we present two data parallel and two task parallel algorithms written in Chapel to show the effectiveness of the language in specifying parallel algorithm and computation.

1 Introduction
Chapel [1] is a global-view parallel programming language developed by Cray Inc. Along with another two global-view parallel programming languages, X10 and Fortress, Chapel is a participation in DARPA’s High-Productivity Computing Systems program, all aiming at the parallel programming languages that support both high productivity and efficient code generated by compilers. The major difference between Chapel and X10 is that Chapel allows for user-defined distribution and layout [2] and makes less distinction between local and remote data by letting the compiler to figure out remote references [3].

According to [3], Chapel is a general parallel language and should allow to express any parallel algorithms without having to use other parallel programming models such as SPMD model. Chapel has four main feature areas: base language, data parallelism control, task parallel control and locality control. In this paper, we are going to present four parallel algorithms, two data parallel and two task parallel, in Chapel to demonstrate the expressiveness of the language. As demonstrated by Chapel codes in this paper, Chapel language is very effective to express data parallelism and task parallelism. For the data-parallel algorithms, the multi-locale control of Chapel allows programmers to express data distribution and locality of computation and derive realistic estimation of the time and memory space of the algorithms. For the task-parallel algorithms, the full/empty synchronization variable and the transaction atomic statement in Chapel proved to be very effective in expressing various synchronization required for parallel threads.

In Section 2, we present two data parallel algorithms: the Floyd-Warshall algorithm for the all-pairs shortest-paths problem and the dynamic programming algorithm to calculate optimal binary search tree. In Section 3, we present two task parallel algorithms: the bounded buffer problem and the dining philosopher problem. For each problem, we present the problem description, sequential algorithm, analysis of the parallelism and synchronization, and the Chapel code and its reasoning. Section 4 concludes the paper.

2 Data Parallel Algorithms in Chapel

2.1 All-Pairs Shortest-Paths Problem
Given a directed graph with weight \( W(e) \) for every edge \( e = (i, j) \) as direct distance from node \( i \) to node \( j \), the all-pairs shortest-paths problem is to find the length of the shortest path for every pair of nodes.
The Floyd-Warshall algorithm for the all-pairs shortest-paths problem [4] is as follows:

Initialize \( D(0) \) with direct distances between nodes;

for \( k = 1 \) to \( n \) do
   for \( i = 1 \) to \( n \) do
      for \( j = 1 \) to \( n \) do
         \( \dd_{ij}^{(k)} = \min(\dd_{ij}^{(k-1)}, \dd_{ik}^{(k-1)} + \dd_{kj}^{(k-1)}) \);

return \( O(n) \)

where \( D(k) = (\dd_{ij}^{(k)}) \) is the matrix of shortest path distances from \( i \) to \( j \) through nodes in \( \{1, \ldots, k\} \). \( D(0) \) is the matrix of the direct distance between the nodes, i.e.

\[
\dd_{ij}^{(0)} = \begin{cases} 
0 & \text{if } i = j \\
W(i, j) & \text{if there is an edge from } i \text{ to } j \\
\infty & \text{otherwise}
\end{cases}
\]

Since the \( k \)-th iteration of loop \( k \) only uses the data produced by the \( (k-1) \)-th iteration, there is no dependence between different iterations of the nested loop \( i \)
and \( j \) and, thus, they can be executed in parallel. Expressing this parallelism in Chapel, we can have the following Chapel multi-threading code:\(^1\)

```chapel
config const n = 10;
var probSpace: domain(2) = [1..n, 1..n];
var d: [probSpace] int = 0;
initDistance(d);
for k in 1..n do
  forall (i,j) in probSpace do
    d[i,j] = min(d[i,j], d[i,k] + d[k,j]);
printResults(n, d);
def min(a, b)
return if a <= b then a else b;
```

\(^1\)This program has been compiled and run correctly.

For the locale computing \( d[i,j] \) of the current iteration of loop \( k \) to update the \( d[i,j] \) of the current iteration, Should we use two data arrays \( d[i,k] \) and \( d[k,j] \) of the previous iteration of loop \( k \) to update the \( d[i,j] \) of the current iteration? Should we use two data arrays to distinguish the old and new data point sets? The answer is \( d \).

The formula used in the previous iteration. The same is true for the current iteration as above is no different from the \( d[i,k] \) from the previous iteration. The same is true for the \( d[k,j] \) used in calculating \( d[i,j] \).

\[\text{MST}_{i,j} = \min_{1 \leq r \leq j} (\text{MST}_{i,r-1} + \text{MST}_{r+1,j}) + \sum_{k=i}^{j} p_k \quad (1)\]

The value of \( r \) that gives the minimum of the sums \( \text{MST}_{i,r-1} + \text{MST}_{r+1,j} \) determines \( k_r \) as the root of the optimal binary search tree for \( k_i, \ldots, k_j \).
The optimal binary search tree can be found in linear time using the dynamic programming method. The working data structure for finding \( \text{MST}(1,n) \) is an \((n+1)\times(n+1)\) matrix \( \text{cost}[1..(n+1),0..n] \). The matrix element \( \text{cost}[i,j] \) is used to store \( \text{MST}_{i,j} \). Figure 3 shows the array \( \text{cost} \) for \( n = 7 \). \( \text{cost}[k,k] \) \((1 \leq k \leq n)\) and \( \text{cost}[k,k-1] \) \((1 \leq k \leq n+1)\) are \( p_k \) and 0, respectively. The root of the optimal binary search tree containing \( k_1, \ldots, k_j \) is stored in \( \text{root}[i,j] \) of another matrix \( \text{root}[1..(n+1),0..n] \). The dynamic programming algorithm for finding the optimal binary search tree [5] is as follows:

```chapel
float cost[1..n+1,0..n]; int root[1..n+1,0..n];
main() {
    for (i=n+1; i>1; i--)
        for (j=i-1; j<n; j++)
            MST(cost, prob, root, i, j);
}
MST(cost, prob, root, i, j) {
    if (j < i) {
        cost[i,j] = 0;
        root[i,j] = -1;
    } else if (i==j) {
        cost[i,j] = prob[i];
        root[i,j] = i;
    } else {
        psum = + reduce prob[i..j];
        bestCost = MAX;
        bestRoot = -1;
        for (int r=i; r<=j; r++) {
            rCost = psum + cost[i,r-1] + cost[r+1,j];
            if (rCost < bestCost) {
                bestCost = rCost;
                bestRoot = r;
            }
        }
        cost[i,j] = bestCost;
        root[i,j] = bestRoot;
    }
}
```

According to the algorithm above, to compute the \( \text{cost}[i,j] \) for \( \text{MST}_{i,j} \) we only need the data elements to its left (up to the diagonal), \( \text{cost}[i,(i-1)..(j-1)] \), and the data elements below (up to the diagonal), \( \text{cost}[(i+1)..(j+1),j] \). In Figure 3, the data element needed to compute \( \text{cost}[1,4] \), \( \text{cost}[2,5] \), \( \text{cost}[3,6] \) and \( \text{cost}[4,7] \) shown in the dark shadow are shown in the light shadow. Therefore, the computations of those \( \text{cost}[i,j] \) such that \( j-i \) is constant can be executed in parallel. That is, the parallelism lies along the sub-diagonals of the matrix and is one-dimensional. For the \((i,j)\) in the diagonal of \( j-i = 1 \), we should have \( \text{cost}[i,j] = 0 \) and \( \text{root}[i,j] = -1 \), because they correspond to empty trees. For the sub-diagonal of \( j-i = 0 \), we should have \( \text{cost}[i,i] = p_i \) and \( \text{root}[i,i] = i \). Then the parallel computations for remaining sub-diagonals of \( j-i = w \) should proceed

```
config const n = 10;
var probSpace: domain(2) = [1..n+1,0..n];
var cost: [probSpace] real = 0.0;
var root: [probSpace] int = -1;
initProb(prob);
forall i in 1..n {
    cost[i,i] = prob(i);
    root[i,i] = i;
}
for w in 1..n-1 {
    forall (i,j) in (1..n-w, 1+w..n) {
        var psum = + reduce prob(i..j);
        var r = i..j;
        var temp: [r] real = cost[i,r-1] + cost[r+1,j];
        var (minVal, minLoc) = minloc reduce-scan-temp(temp, temp.domain);
        cost[i,j] = psum + minVal;
        root[i,j] = minLoc;
    }
}
```

```
MST(cost, prob, root, i, j) {
if (j < i) {
    cost[i,j] = 0;
    root[i,j] = -1;
} else if (i==j) {
    cost[i,j] = prob[i];
    root[i,j] = i;
} else {
    psum = + reduce prob[i..j];
    bestCost = MAX;
    bestRoot = -1;
    for (int r=i; r<=j; r++) {
        rCost = psum + cost[i,r-1] + cost[r+1,j];
        if (rCost < bestCost) {
            bestCost = rCost;
            bestRoot = r;
        }
    }
    cost[i,j] = bestCost;
    root[i,j] = bestRoot;
}
```

The first `forall` loop is to initialize the `cost` and `root` arrays on the sub-diagonal \( j-i = 0 \). Then, the parallel computation for the other sub-diagonals \( j-i = w \) \((w = 1, \ldots, n-1)\) are expressed by the `forall` loop in the `for` loop with index \( w = 1, \ldots, n-1 \). The iterator-expression \((1..n-w, 1+w..n)\) is a zipper iterator over the two iterators. It returns indexes \((1,1+w), (2,2+w), \ldots, (n-w,n)\). These are exactly the indexes on the sub-diagonal \( j-i = w \). Array `temp` is an array of one-dimensional domain \( r = i..j \). It is initialized with \( \text{cost}[i,r-1] + \text{cost}[r+1,j] \), where \( r-1 \) and \( r+1 \) are the one-dimensional domains \((i-1)\_j(i-1)...(j+1)\) and \((i+1)...(j+1)\), respectively. `minloc` is a reduce-scan-operator in Chapel [6]. When used in a reduction expression, it returns the 2-tuple of the minimum value and its location (index) of the source values.

To map the parallel computations to multiple locales (nodes), we need to partition the domain

---

This program has been compiled and run correctly.
config const n = 10000;
var probSpace: domain(2) distributed(\*,Cyclic) = [1..n+1,0..n];
var cost: [probSpace] real = 0.0;
var root: [probSpace] int = -1;
var prob: [1..n] real;
initProb(prob);
forall i in 1..n {
    cost[i,i] = prob(i);
    root[i,i] = i;
}
for w in 1..n-1 {
    forall (i,j) in (1..n-w, 1+w..n) on cost[i,j] {
        var psum = + reduce prob(i..j);
        var r = i..j;
        var temp: [r] real = cost[i,r-1] + cost[r+1,j];
        var (minVal, minLoc) =
            minloc reduce (temp, temp.domain);
        cost[i,j] = psum + minVal;
        root[i,j] = minLoc;
    }
}
printResults(n, cost, root);

Figure 4: The multi-locale Chapel code for optimal binary search tree

probSpace, on which arrays cost and root are defined. The parallelism is one-dimensional on the sub-diagonals and, thus, the default one-dimensional array of locales is sufficient. The computation of cost[i,j] needs to access cost[i,(i-1)..(j-1)] and cost[(i+1)..(j+1),j]. Distributing the domain along the second dimension j will make the access of cost[(i+1)..(j+1),j] local.

Note that the length of sub-diagonals j - i = w is decreasing as w increases. In order to achieve load balance (i.e. every locale has roughly the same amount of computation), it is necessary to distribute the domain probSpace along the second dimension j cyclically. The multi-locale Chapel code for the optimal binary search problem, thus, is shown in Figure 4.

The clause distributed(\*,Cyclic) specifies that the columns of the domain probSpace are distributed cyclically. That is, columns 0,P,2P,... are allocated to locale 0, columns 1,P+1,2P+1,... allocated to locale 1, and so on, if there are P locales. Figure 2(b) shows the distribution (\*,Cyclic) of an 8 x 8 domain on four locales. The on clause in the forall statement specifies that the computation with index (i,j) is carried by the locale where cost[i,j] resides.

3 Task Parallel Algorithms in Chapel

3.1 Bounded Buffer Problem

In the classical bounded buffer problem, producers and consumers share a bounded buffer. Producers keep putting items to the bounded buffer, while consumers keep getting items from the buffer.

The correct solution of the problem must ensure that (1) every item produced by a producer must be fetched and consumed by a consumer exactly once and (2) the items produced by the same producer and consumed by the same consumer should be consumed in the order they are produced. A solution of the problem using monitor and conditional variables can be found in [7].

Chapel provides full/empty synchronization variables for synchronization between parallel threads. Reading an empty synchronization variable causes the thread to block until the variable becomes full by writing it. If there are more than one threads blocked on reading the empty synchronization variable, the thread to be waked up by a write is selected non-deterministically. Similarly, the threads writing a full synchronization variable will be blocked until the variable become empty by a read. The blocked writing thread to be waked up by a read is selected non-deterministically.

A solution of the bounded buffer problem in Chapel is shown in Figure 5. The main() function creates numProd producers and numCons consumers using cobegin and cofinal statements. The elements of the shared buffer buf$ and index variables in$ and out$ are all full/empty synchronization variables in Chapel. Synchronization variable total$ is used to monitor the total number of items consumed by all the consumers. Each producer generates numItems items of the form (num,i), where num is the producer id number, i a sequence number in 1..numItems. Each consumer creates a file and then keeps reading items from the buffer and writing corresponding messages into the file.

The read of synchronization variable in$ to the local index variable ind by a producer makes it empty, thus blocking the read by other producers, until in$ is written with (ind+1) % size after the producer writes the item in the buffer entry buf[ind]. Consumers use synchronization variable out$ similarly. The full/empty buffer entries guarantee that no item

\footnote{The distribution of multi-dimensional domains has not been implemented in the current version (v0.780) of Chapel yet.}
in a buffer entry will be overwritten nor it will be read twice. Thus, no items will be lost and every item produced is fetched and consumed exactly once.

```chapel
config const
size = 5, numItems = 25, numProd = 4, numCons = 3;
var buff$: [0..size-1] sync (int, int);
var in$: sync int = 0;
var out$: sync int = 0;
var total$: sync int = 0;
var notFinished: bool = true;
def main() {
    cobegin {
        coforall i in 1..numProd do producer(i);
        coforall j in 1..numCons do consumer(j);
    }
}
def producer(num: int) {
    var ind: int;
    for i in 1..numItems {
        var ind = in$;
        buff$[ind] = (num, i);
        in$ = (ind + 1) % size;
    }
}
def consumer(num: int) {
    var currTotal, ind: int;
    var filename: string = "tmp_" + num + ".out";
    var outfile = new file(filename,
        FileAccessMode.write);
    outfile.open();
    while notFinished {
        currTotal = total$;
        if currTotal == numProd * numItems - 1
            then notFinished = false;
        total$ = currTotal + 1;
        if currTotal < numProd * numItems {
            ind = out$;
            var (pnum, item) = buff$[ind];
            outfile.writeln("Producer ", pnum,
                " Item ", item, " through slot ", ind);
            out$ = (ind + 1) % size;
        }
    }
    outfile.close();
}
```

Figure 5: The Chapel code for Bounded Buffer Problem.

We could move the write and read of the item out of the critical sections controlled by `in$` and `out$` as follows,

```chapel
buff$[ind]=(num,i);  out$=(ind+1)%size;
}

```

This program has been compiled correctly. The atomic statement has not been implemented in the current version (v0.780) of Chapel yet.

### 3.2 Dining Philosopher Problem

The classical dining philosopher program requires that the neighboring philosophers on the round dining table should never be eating at the same time, as they share the same fork between them. Each philosopher needs to pick up the two forks on his/her right and left before eating. After eating, he/she puts down the forks. Since each philosopher needs two forks for eating, the simple implementation using low-level semaphore or lock for the exclusive use of each fork could lead to a deadlock. A deadlock-free solution of the problem using monitor and condition variables can be found in [7].

Chapel provides *atomic* statement which has the atomic transaction semantics. Atomic statements of parallel threads appear to be serialized in their execution, although they are executed concurrently. That is, the execution of an atomic statement will not be affected by the execution of other atomic statement. This is called *isolation* in the transaction semantics. Each atomic statement is executed entirely and no variable assignment is visible until the statement is completed, or in the case of failure, it appears not to have executed at all. This is called *atomicity* of the transition semantics. The atomic statement in Chapel allows us to have an elegant deadlock-free solution of the dining philosopher problem shown in Figure 6.

The state of each philosopher can be changed only by itself. The changing from the hungry state to the eating state is done in the atomic statement only if its two neighboring philosophers are not in the eating state. The atomic statements of the philosophers not neighboring with each other are executed concurrently as they do not share any variables. For the neighboring philosophers, the execution of their atomic statements are serialized as if there was a mutual exclusion lock to allow only one of them to proceed. Thus, only one
config const numPhil = 5, numTimes = 5;
enum states {thinking, hungary, eating};
var state: [0..numPhil-1] states = states.thinking;
def main() {
    coforall i in 0..numPhil-1 do philosopher(i);
}
def philosopher(i: int) {
    for j in 1..numTimes {
        state[i] = states.hungary;
        while state[i] == states.hungary {
            atomic {
                if (state[(i-1)%numPhil] != states.eating
                    && state[(i+1)%numPhil] != states.eating)
                    state[i] = states.eating;
            }
        }
        state[i] = states.thinking;
    }
}

Figure 6: Chapel code for Dining Philosopher Problem.

of them can change to the eating state, which will prevent the other from entering the eating state. The while statement is used to re-try if the philosopher finds that it is still in the hungry state after the atomic statement. The isolation of the transaction semantics, thus, ensures the mutual exclusion of the eating state of the neighboring philosophers. Since the atomic statements are serialized and there is not blocking or waiting, deadlock is not possible.

The current Chapel has not decided whether to adapt the strong or weak atomicity semantics [9], or more precisely, the strong or weak isolation semantics [10]. The weak atomicity (isolation) only guarantees the isolation between the codes of within atomic statement, but not between the codes within and outside of the atomic statement. But, even with the weak atomicity, the code in Figure 6 is still correct. In particular, the read by philosopher i of the states of its neighboring philosophers can be concurrent with their writes outside the atomic statement. But, both writes are only to change to the hungry or thinking states. They do not change the outcome of the if statement in the atomic statement of philosopher i.

4 Conclusion

We have presented two data-parallel algorithms and two task-parallel algorithms using Chapel. As demonstrated by these codes, Chapel language is very effective to allow both data parallelism and task parallelism to be expressed. For the data-parallel algorithms, the multi-locale control of the language allows programmers to express data distribution and locality of computation clearly. For the task-parallel algorithms, the full/empty synchronization variable and the transaction atomic statement proved to be very effective in expressing various synchronization required for parallel threads.

References