

# 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-pair 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
       $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)});$ 
return  $D^{(n)}$ 
```

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

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

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<sup>1</sup> :

```

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;

```

`forall` is a parallel loop such that all the iterations of index  $(i, j)$  from the domain `probSpace` are to be executed in parallel in parallel threads and the entire `forall` loop will not be completed until all the parallel threads are completed<sup>2</sup>.

To write the multi-locale Chapel code for the large size problem on scalable parallel machines, we need to distribute the data points and the tasks to calculate them in different nodes. Since the access of the data in the remote nodes costs significantly more than the local node, data and task distribution should minimize remote data access. For a fixed  $k$ , the computation of  $d[i, j]$  needs to access  $d[i, k]$  and  $d[k, j]$ . Distributing the data array `d` only along one dimension and flushing it along the other dimension will make one of these accesses local. The multi-local Chapel parallel code for the all-pairs shortest-paths problem is shown in Figure 1. The two-dimensional domain, `probSpace`, on which data array `d` is defined, is distributed on one-dimensional locales along the first dimension (row) using `Block` distribution. The second dimension (column) of the domain is not distributed, but rather flushed, as indicated by the `*` in the distribution specification<sup>3</sup>. Figure 2(a) shows the distribution of an  $8 \times 8$  two-dimensional domain with `(Block,`

```

config const n = 1000;
var probSpace: domain(2) distributed(Block,*)
  = [1..n,1..n];

var d: [probSpace] int = 0;
initDistance(d);
for k in 1..n do
  forall (i,j) in probSpace on d[i,j] 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;

```

Figure 1: The multi-locale Chapel code for Floyd-Marshall Algorithm.

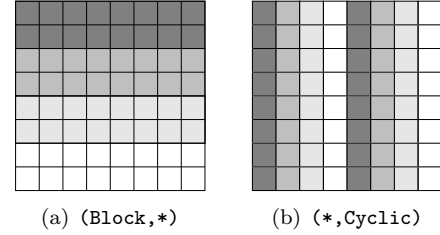


Figure 2: Distributions of 2D Domain on 1D Locale

`*`) on one-dimensional locales of four nodes. The `on` clause in the `forall` statement means that the computation of iteration  $(i, j)$  is to be carried by the locale that owns  $d[i, j]$ . For the locale computing  $d[i, j]$ ,  $d[i, k]$  is always a local access, because the second dimension is flushed and the locale owns the entire row  $d[i, 1..n]$ . The access of  $d[k, j]$  may be remote depending on the value of  $k$ .

## 2.2 Optimal Binary Search Tree Problem

Given  $n$  keys,  $k_1, \dots, k_n$ , and their probabilities of occurrence,  $p_1, \dots, p_n$ , the optimal binary search tree problem is to find the binary search tree of those keys with the minimum average search time.

If the optimal binary search tree for  $k_1, \dots, k_n$  has  $k_r$  ( $1 \leq r \leq n$ ) as its root, then its left sub-tree containing  $k_1, \dots, k_{r-1}$  and right sub-tree containing  $k_{r+1}, \dots, k_n$  must also be optimal. Let  $MST_{i,j}$  be its mean search time of the optimal binary search tree containing keys,  $k_i, \dots, k_j$  ( $j \geq i - 1$ ). Then,  $MST_{i,j}$  can be calculated by the following recursive equation:

$$MST_{i,j} = \min_{i \leq r \leq j} (MST_{i,r-1} + MST_{r+1,j}) + \sum_{k=i}^j p_k \quad (1)$$

The value of  $r$  that gives the minimum of the sums  $MST_{i,r-1} + MST_{r+1,j}$  determines  $k_r$  as the root of the optimal binary search tree for  $k_i, \dots, k_j$ .

<sup>1</sup>This program has been compiled and run correctly.

<sup>2</sup>There is a subtle point here. Since data points are updated by parallel threads in arbitrary order, the data points  $d[i, k]$  and  $d[k, j]$  can be updated before they are read to update  $d[i, j]$ . The original Floyd-Warshall algorithm says that we should use the  $d[i, k]$  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 no. Let us assume that we used the new  $d[i, k]$  calculated in the current iteration of loop  $k$  to calculate  $d[i, j]$ . The formula used to calculate  $d[i, k]$  must be

$$d[i, k] = \min(d[i, k], d[i, k] + d[k, k])$$

according to the code. Note that  $d[k, k]$  is always 0, because

$$d[k, k] = \min(d[k, k], d[k, k] + d[k, k])$$

always leaves  $d[k, k]$  0. Therefore, the  $d[i, k]$  calculated in 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  $d[i, j]$ .

<sup>3</sup>The distribution of multi-dimensional domains has not been

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

```
float cost[1..n+1,0..n]; int root[1..n+1,0..n];
float prob[1..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;
  } elseif (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  $cost[i, j]$  for  $MST_{i,j}$  we only need the data elements to its left (up to the diagonal),  $cost[i, (i-1)..(j-1)]$ , and the data elements below (up to the diagonal),  $cost[(i+1)..(j+1), j]$ . In Figure 3, the data element needed to compute  $cost[1, 4]$ ,  $cost[2, 5]$ ,  $cost[3, 6]$  and  $cost[4, 7]$  shown in the dark shadow are shown in the light shadow. Therefore, the computations of those  $cost[i, j]$  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  $cost[i, j] = 0$  and  $root[i, j] = -1$ , because they correspond to empty trees. For the sub-diagonal of  $j-i = 0$ , we should have  $cost[i, i] = p_i$  and  $root[i, i] = i$ . Then the parallel computations for remaining sub-diagonals of  $j-i = w$  should proceed

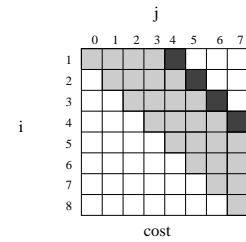


Figure 3: The parallelism on the sub-diagonal  $j-i = 3$ .

in the order  $w = 1, 2, \dots, n-1$ . Base on the analysis above, the multi-threading Chapel code for the optimal binary search problem is as follows<sup>4</sup>:

```
config const n = 10;
var probSpace: domain(2) = [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) {
    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);
```

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  $j-i = w$  ( $w = 1, \dots, n-1$ ) are expressed by the forall loop in the for loop with index  $w = 1, \dots, n-1$ . The iterator-expression  $(1..n-w, 1+w..n)$  is a zipper iterator over the two iterators. It returns indexes  $(1, 1+w), (2, 2+w), \dots, (n-w, n)$ . 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  $cost[i, r-1] + cost[r+1, j]$ , where  $r-1$  and  $r+1$  are the one-dimensional domains  $(i-1)..(j-1)$  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

<sup>4</sup>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  $0, P, 2P, \dots$  are allocated to locale 0, columns  $1, P + 1, 2P + 1, \dots$  allocated to locale 1, and so on, if there are  $P$  locales. Figure 2(b) shows the distribution `(*,Cyclic)` of an  $8 \times 8$  domain on four locales<sup>5</sup>. 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.

<sup>5</sup>The distribution of multi-dimensional domains has not been implemented in the current version (v0.780) of Chapel yet.

## 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<sup>6</sup>. The `main()` function creates `numProd` producers and `numCons` consumers using `cobegin` and `coforall` statements. The elements of the shared buffer `buff$` and index variables `in$` and `out$` are all full/empty synchronization variables<sup>7</sup> in Chapel. Synchronization variable `total$` 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 `buff[ind]`. Consumers use synchronization variable `out$` similarly. The full/empty buffer entries guarantee that no item

<sup>6</sup>The producer-consumer code example provided with the Chapel distribution at `chapel.cs.washington.edu` allows only one procedure and one consumer. We provide the code for bounded-buffer for multiple producers and consumers. This program has been compiled and run correctly.

<sup>7</sup>As a convention, the names of full/empty synchronization variables always end with `$` symbol.

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.

```

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,

```

producer(num:int){      consumer(num:int){
...
for i in 1..nT {        while ... {
  ind=in$;              ...
  in$=(ind+1)%size;    ind=out$;
}
}

```

```

buff$[ind]=(num,i);    out$=(ind+1)%size;
}                        var (pnum,item)=buff$[ind];
}                        ...
}                        }
}

```

to have more parallelism, allowing multiple producers and consumers to access different buffer entries at the same time. However, the items produced by the same producer may be fetched by a consumer out of the order they are produced, violating one of the requirements of the problem. See the technical report version of this paper [8] for an example of such case.

### 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<sup>8</sup>. 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

<sup>8</sup>This program has been compiled correctly. The atomic statement has not been implemented in the current version (v0.780) of Chapel yet.

```

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 algo-

rithms, 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

- [1] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *International Journal of High-Performance Computing Applications*, 21(3):291–312, August 2007.
- [2] Roxana E. Diaconescu and Hans P. Zima. An approach to data distributions in chapel. *International Journal of High-Performance Computing Applications*, 21(3):313–335, August 2007.
- [3] Bradford L. Chamberlain. Closing the parallelism gap with the chapel language. An interview with hpcwire for sc08, [http://www.hpcwire.com/topic/developertools/Closing\\_the\\_Parallelism\\_Gap\\_with\\_the\\_Chapel\\_Language\\_34793239.html](http://www.hpcwire.com/topic/developertools/Closing_the_Parallelism_Gap_with_the_Chapel_Language_34793239.html), 2008.
- [4] Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introductions to Algorithms, 2nd Edition*. McGraw-Hill Book Company, 2001.
- [5] Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis (3rd Ed)*. Addison-Wesley, 2000.
- [6] Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence Snyder. Global-view abstractions for user-defined reductions and scans. In *In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 40–47, March 2006.
- [7] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (8th Edition)*. John Wiley & Sons. Inc, 2009.
- [8] Peiyi Tang. Programming data and task parallelism with Chapel. Technical Report [titus.compsci.ualr.edu/~ptang/papers/ppchapel.pdf](http://titus.compsci.ualr.edu/~ptang/papers/ppchapel.pdf), Department of Computer Science, University of Arkansas at Little Rock, 2009.
- [9] Chapel Team. Chapel language specification 0.780. Technical Report <http://chapel.cs.washington.edu>, Cray Inc., 2008.
- [10] James R. Larus. *Transaction Memory*. Morgan & Claypool Publishers, 2007.