Compositional C++

- A general purpose parallel programming language based on C++
- Focus on parallel programming paradigms through libraries
- Suitable for a wide range of parallel computing styles

New CC++ Concepts

- **Thread** for concurrency
- **Sync variable** for synchronization
- **Atomic functions** for mutual exclusion
- **Processor object** to control locality
- **Global pointer** for non-local references
- **Transfer functions** for data transfer

CC++ Extensions

- Keywords:
  - Par, parfor, spawn: thread creation
  - Sync: sync variables
  - Atomic: atomic functions
  - Global: processor objects, global pointers
- Predefined data-types
  - Ccvoid: data transfer functions
  - Proc_t: processor object placement

Introduction to CC++

- CC++ can express many different programming styles
  - Task parallelism, data parallelism, “active” objects
- Introduce CC++ with an example from computational chemistry
  - Compute the **Fock matrix**
- Presentation slightly different than in book

Computing the Fock Matrix

- The Fock matrix F is computed by the following integral equation:

$$F_{ij} = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} D_{kl} \left(I_{ijkl} - \frac{1}{2} I_{ikjl} \right)$$

Where:

F is the Fock Matrix

D is an *NxN* constant matrix

I is an integral over *A*, a 1-D constant vector

Fock Matrix Calculation

- Exploit symmetry to reduce operation count to $N^4/8$ **operations**

```
FockMatrix F;
for (i=0 ; i < N; i++)
  for (j=0; j < i; j++)
    for (k=0; k < j; k++)
      for (l=0; l < k; l++) {
        F.compute(i, j, k, l);
      }
```

FockMatrix Class Declaration

Constructor

```
class FockMatrix {
public:
   FockMatrix(int sz, float a; float **da);
   void compute(int, int, int, int);
private:
   void update(float, int, int, int, int);
   float integrate(int, int, int, int);
   float **d_matrix,  **fock_matrix;
   float * a_values;
   int block_size;
};
```

Implementation of FockMatrix::compute()

```
FockMatrix::compute(int i, int j, int k, int l)
{
   float integral = integrate(i, j, k, l);
   update(integral, i, j, k, l);
}
```

Implementation of FockMatrix::update()

```
FockMatrix::update(float integral,
                   int i, int j, int k, int l)
{
    fock_matrix[i][j] += d_matrix[k][l] * integral;
    fock_matrix[k][l] += d_matrix[i][j] * integral;
    fock_matrix[i][k] += d_matrix[j][l] * integral;
    fock_matrix[j][l] -= (1/2) d_matrix[i][k] * integral;
    fock_matrix[i][l] -= (1/2) d_matrix[j][k] * integral;
    fock_matrix[j][k] -= (1/2) d_matrix[i][l] * integral;
}
```

Observations on FockMatrix::update()

- Order of calls does not effect result
  - Ignore round-off error
- Concurrent calls can result in write/write conflict in `fock_matrix`
- Each call to FockMatrix::update()
  - Reads six different values of `d_matrix`
  - Reads/writes six different values of `fock_matrix`

CC++ Implementation of Fock Matrix Calculation

- Incremental approach
- First, sequential version (already given)
- Next, shared memory implementation
  - Data distribution is not an issue
  - CC++ constructs: parallel blocks, atomic functions
- Finally, distributed memory implementation
  - Functional decomposition
  - Global pointers, processor objects

Adding Parallelism in CC++

- Parallel blocks

```
par { S1; ...; Sn; }
```

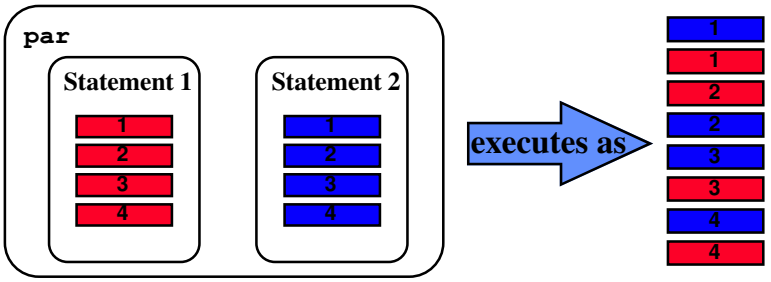
- Parallel loops

```
parfor(int i = 0; i < n; i++)
    Statement
```

- Function spawning

```
spawn F(...)
```

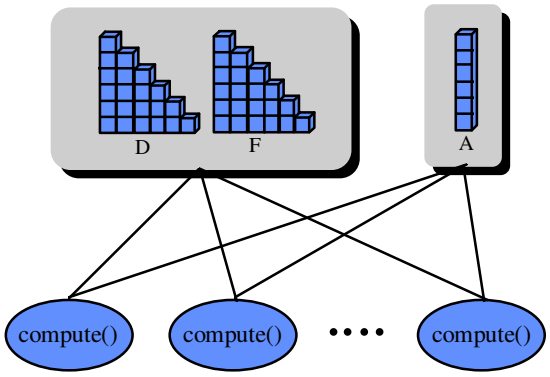
Parallel Statements use Fair Interleaving



Shared Memory Solution

```
FockMatrix F;
parfor (i=0; i < N; i++) {
   parfor (j=0; j < i; j++) {
      parfor (k=0; k < j; k++) {
         parfor (l=0; l < k; l++) {
               F.compute(i, ,j, k, l);
         }
      }
   }
}
```

Parallelism in Functional Decomposition



Thread Safety and FockMatrix::compute()

- Parallel calls to `FockMatrix::update()` can result in errors

```
fock_matrix[i][j] += d_matrix[k][l] * integral;
```

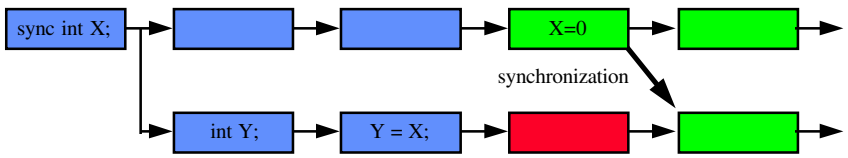
- Two alternatives:
  - Private array copies and merge results
  - Make updates atomic
- We show atomic update method

Synchronization in CC++

- Sync variables
  - Single assignment variables
  - Looks like ANSI **`const`** type modifier, but with delayed initialization
  - Used to construct arbitrary synchronization structures
- Atomic functions
  - Control interleaving of parallel statements

Sync variables

```
sync int x; // Create data object
...
x = 0;      // delayed initialization
```



Atomicity

- Interleaving is restricted by **atomic** function

```
class Queue {
   atomic void insert_into_queue(T);
}
```

- Any function can be made atomic
- Atomicity is maintained for each object instance
- Atomic functions can be called recursively

Thread-safe Versions of FockMatrix::compute()

- Look at two alternatives to enforce mutual exclusion
  - Atomic function
  - Lock class written in CC++
- Atomic function approach is best
- Lock based approach is instructional

Thread-Safe FockMatrix Class using Atomic Functions

```
class FockMatrix {
public:
   FockMatrix(int sz, float a; float **da);
   void compute(int, int, int, int);
private:
   atomic void update(float,int,int,int,int);
   float integrate(int, int, int, int);
   float **d_matrix,  **fock_matrix;
   float * a_values;
   int block_size;
};
```

Thread-Safe FockMatrix Class using Locks

- Write a lock class in CC++

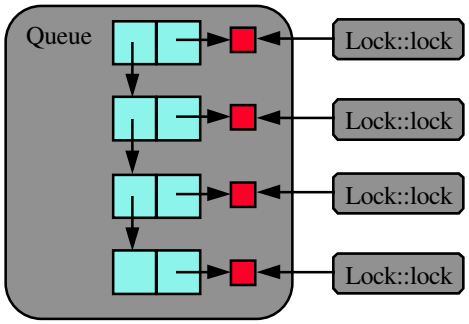
```
FockMatrix::compute(int i, int j, int k, int l)
{
     float integral = integrate(i, j, k, l);
     Lock lock;
     lock.lock();
     update(integral, i, j, k, l);
     lock.unlock();
}
```

Atomic Operations in Lock Class

- Use atomic functions to protect internal state of lock
- Unlock operation must be atomic
  - Change state of lock
  - Wake blocked threads
- Lock operation cannot be atomic
  - Need auxiliary atomic function

Blocking Threads

- Suspend execution of a lock call by keeping a queue of sync variables



Details of the Lock Class

```
class Lock {
public:
    void lock();
    atomic void unlock();
    Lock() : lock_state(unlocked) {};
private:
    enum { locked, unlocked} state;
    state lock_state;
    queue<sync state *> waiting_queue;
    atomic void check_lock(sync state *);
}
```

Obtaining a Lock

```
void Lock::lock() {
    sync state got_lock;
    check_lock(&got_lock);
    got_lock == locked;
}

atomic void Lock::check_lock(sync state * go)
{
    if (state == locked)
        waiting_queue.enqueue(go);
    else
        *go = lock_state = locked;
}
```

Releasing a lock

```
atomic Lock::unlock()
{
   if (waiting_queue.isempty())
      lock_state = unlocked;
   else
      *waiting_queue.dequeue() = 1;
}
```

Shared-Memory Fock Matrix Implementation

- The code developed will run with speedup on shared-memory computers
- Machines in this class include
  - SGI Challange
  - Convex Exemplar
  - Sparcserver
  - PC class machines running Windows NT or OS/2

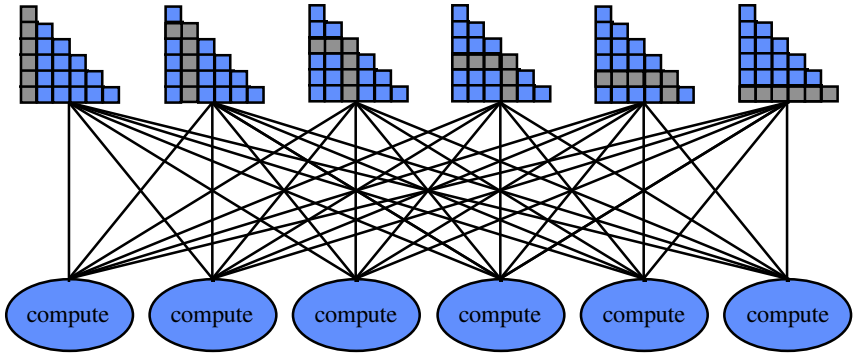
Steps to a Distributed-Memory Version of FockMatrix

- Increase granularity
- Perform data access through member function interface
- Distribute computation and data

Increasing Computation Granularity

```
FockMatrix F;
parfor (i=0 ; i < N ; i++) {
   parfor (j=0 ; j < i ; j++) {
       parfor (k=0; k < j ; k++) {
          // Execute sequentially
             F.compute(i, ,j, k);
       }
   }
}
```

Parallel Computation Structure



Creating Fock Matrix Data Servers

```
class FockMatrixSection {
pubic:
  FockMatrixSection(int n);
  // return Nth row and column of D as a 1-D vector
  comm_vector<float> get_D_section();
  // update Nth row and column of Fock matrix
  atomic void update(comm_vector<float>);
};
```

- Class `comm_vector` is part of standard CC++ library

Global Pointers

- CC++ pointers can span processors
- Keyword **global** indicates potentially expensive reference

```
int * global iPtr;
int * global * iPtr_array;
class C * global cPtr;
```

- Conversions

  – implicit local to global, explicit global to local

Incorporating Matrix Data Servers into FockMatrix

```
class FockMatrix {
public:
   FockMatrix(int sz, float a;
              FockMatrixSection * global *);
   void compute(int, int, int);
private:
   comm_vector<float> integrate(int,int,int);
   FockMatrixSection *global *matrix_sections;
   float * a_values;
   int block_size;
};
```

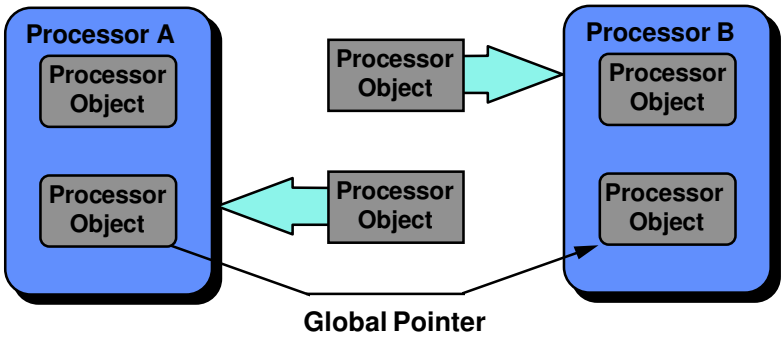
Data-Server Implementation of FockMatrix::compute()

```
FockMatrix::compute(int i, int j, int k)
{
   comm_vector<float> integral =
                      integrate(i, j, k);
   matrix_section[i]->update(integral);
   matrix_section[j]->update(integral):
   matrix_section[k]->update(integral);
}
```

Locality in CC++

- Processor objects are virtual processors
  - A whole program becomes an CC++ object
  - Each processor object is a separate address space
  - Global & static variables are local to processor object
- Processor objects can be created dynamically
- Interprocessor object references must be global
- Keyword **global** indicates classes to be used as processor objects

Resource Management With Processor Objects



Class Declaration of FockNode

```
global class FockNode {
    // Use processor object array declarations from
    // CC++ library
    declare2(pobj_array, FockNode, FockMatrix)
public:
    FockNode(const locations & l, const int sz):
                        array_components(l, sz) {}
    // Declare a parallel function
    void compute() { PFUNC(compute()); }
}
```

Implementation of FockNode::compute()

```
FockMatrix::compute()
{
    // Find out where I am in the processor object array
    int my_node = ::this->rank();
    parfor (j=0 ; j < my_node ; j++) {
        parfor (k=0; k < j ; k++) {
            F.compute(my_node, j, k);
        }
    }
}
```

Top level Distributed Fock Matrix Computation

```
main()
{
      locations l;
      // Create an array of FockMatrix objects
      FockNode fock(l,6);

      // Call compute in every FockMatrix object
      fock.compute();
}
```

Summary

- We have given a brief overview of CC++
  - Skipped discussion on procesor object placement and transfer functions
- Demonstrated stepwise refinement process
  - Sequential => Shared-memory => Dist-memory
- Introduced parts of standard CC++ library
  - Many additional functions are in library

Notes