Overview: Emerging Parallel Programming Models

- The partitioned global address space paradigm
  - The HPCS initiative; basic idea of PGAS
  - The Chapel language: design principles, task and data parallelism, sum array example, domains and locales

- Directed acyclic task graph programming models
  - Motivations, ideas
  - PLASMA: tiled Cholesky Factorization example, architecture
  - Intel CnC

Refs: Chapel home page, Chamberlain et al. *User-Defined Parallel Zippered Iterators in Chapel*, PGAS 2011; see embedded hyperlinks
The High Productivity Computer Systems Initiative

- DARPA project (2003–) to provide “a new generation of economically viable high productivity computing systems . . . that double in productivity (or value) every 18 months” HPCS Program
  - envisioned large-scale systems with aggressively multicore components
- P is for productivity (performance, programability, portability & robustness)
  - measure/predict the ease or difficulty of developing HPC applications
  - reduce the time to result: write code ⇒ run code ⇒ analyse results
- introduce a variety of experimental languages:
  - X10 (IBM)
  - Chapel (Cray)
  - Fortress (Sun - no longer supported by DARPA)

State until late 2008: implementation was in a poor state (very slow to compile & run, only support for shared memory targets; inadequate documentation).
Partitioned Global Address Space

- recall the shared memory model: multiple threads with pointers to a global address space
- in the partitioned shared memory model (PGAS) model:
  - have multiple threads, each with affinity to some partition of global address space
  - SPMD or fork-join thread creation
  - remote pointers to access data in other partitions
  - the model maps to a cluster with remote memory access
  - also can map to NUMA domains
Chapel: Design Principles

- object-oriented (Java-like syntax, but influenced by ZPL & HPF)
- supports exploratory programming
  - implicit (statically-inferable) types, run-time settable parameters (*config*), implicit *main* and module wrappings
- high-level abstractions for fine-grained parallel programming
  - *coforall*, *atomic* code sections; *sync* variables
- and for coarse grained parallel programming (*global-view* abstractions):
  - tasks (*cobegin* block)
  - *locales* (UMA places to run tasks)
  - (index) *domains* used to specify arrays, iteration ranges
  - distributions (mappings of domains to locales)
  - claims to drastically reduce code size over MPI programs
- more info on home page; current implementation is in C++
Chapel: Task Parallelism

- **task creation:**
  
  ```chapel
  begin doStuff();
  cobegin {
    doStuff1();
    doStuff2();
  }
  ```

- **synchronisation variables:**
  
  ```chapel
  var a$: sync int;
  begin a$ = foo();
  c = 2 * a$; // suspend until a$ is assigned
  ```

- **example: Fibonacci numbers**
  
  ```chapel
  proc fib(n: int): int {
    if (n < 2) then return 1;
    var t1$: sync int;
    var t2: int;
    begin t1$ = fib(n1);
    t2 = fib(n2);
    return t1$ + t2; //wait t1$
  }

  proc fib(n: int): int {
    if (n < 2) then return 1;
    var t1, t2: int;
    cobegin {
      t1 = fib(n1);
      t2 = fib(n2);
    }
    return t1 + t2; }
  ```
Chapel: Data Parallelism

- ranges:
  ```chapel
  var r1 = 0..9; // equiv: r1 = 0..#10;
  const MyRange = 0..#N;
  ```

- data parallel loops:
  ```chapel
  var A, B:[0..#N] double;
  forall i in 0..#N do // cf. coforall
      A(i) = A(i) + B(i);
  ```

- scalar promotion:
  ```chapel
  A = A + B;
  ```

- reductions and scans:
  ```chapel
  max = (max reduce A);
  A = (+ scan A); // prefix sum of A
  ```

- example: DAXPY config const N = 1000, alpha = 3.0;
  ```chapel
  proc daxpy(x:[MyRange] real, y:[MyRange] real):int {
      forall i in MyRange do
          y(i) = alpha * x(i) + y(i);
  }
  ```

  Alt., via promotion, the forall loop can be replaced by:
  ```chapel
  y = alpha * x + y;
  ```

- the target of data parallelism could be SIMD, GPU or normal threads
Sum Array in Chapel

```chapel
use Time;
config const size: int = 5*1000*1000;
class ArraySum {
    var array: [0 .. size-1] int = 1;
    var currentSum: sync int = 0;
    def sum(numThreads: int) {
        var blockSize: int = size / numThreads;
        coforall i in [0 .. numThreads - 1] {
            var partSum: int = 0;
            forall j in [i*blockSize .. (i+1)*blockSize-1]
                do partSum += array(j);
            currentSum += partSum;
        }
    }
    var theArray: ArraySum = new ArraySum();
    var sumTimer: Timer = new Timer();
    for i in [0 .. 6] {
        theArray.currentSum.writeXF(0);
        sumTimer.start();
        theArray.sum(2**i);
        writeln(2**i, " threads ", sumTimer.elapsed(milliseconds), "ms");
        sumTimer.clear();
    }
}```
Chapel: Domains and Locales

- a rectangular domain is a tensor product of ranges, e.g.
  \[
  \text{const } D = \{0..\#M, 0..\#N\};
  \]

- also have sparse and associative domains

- locale: a unit of the target architecture: processing elements with (uniform) local memory

  \[
  \text{const } \text{Locales: } [0..\#\text{numLocales}] \ \text{locale} = ... \ ; // \\
  \text{built-in}
  \]

  \[
  \text{on } \text{Locales}[1] \ \text{do}
  \]

  \[
  \text{foo();}
  \]

  \[
  \text{coforall } (\text{loc, id}) \ \text{in } (\text{Locales, 1..}) \ \text{do} \ // \text{`zippered' iteration}
  \]

  \[
  \text{on } \text{loc do} \ // \text{migrates this task to loc}
  \]

  \[
  \text{forall } \text{tid in } 0..\#\text{numTasks do}
  \]

  \[
  \text{writeln("Task ", id, " thread ", tid, " executes on", loc)}
  \]

- use domain maps to map indices in a domain to locales:

  \[
  \text{const } \text{Dist} = \text{new dmap(}
  \]

  \[
  \text{new Cyclic(startIdx = 1, targetLocales = Locales[0..1]));}
  \]

  \[
  \text{const } D = [0..\#N] \ \text{dmapped Dist;}
  \]

  \[
  \text{var } x, y: [D] \ \text{real}; \ \text{const } a = -2.0;
  \]

  \[
  \text{forall } i \ \text{in } D \ \text{do}
  \]

  \[
  y(i) = \alpha \times x(i) + y(i) \ // \text{we could use } x(i-1) \ \text{too!}
  \]

  \[
  y = \alpha \times x + y \ // \text{same!}
  \]
DAG Execution Models: Motivations and Ideas

- In most programming models, serial or parallel, the algorithm is *over-specified*
  - Sequencing that is not necessary is often specified
  - Specifying what (sub-)tasks of a program can run in parallel is difficult and error-prone
  - The model may constrain the program to run on a particular architecture (e.g. a single address space)

- Directed acyclic task graph (DAG) programming models specify only the necessary semantic ordering constraints
  - We express an instance of an executing program as a graph of tasks
  - A node has an edge pointing to a second node if there is a (data) dependency between them

- The DAG run-time system can then determine when and where each task executes, with the potential to extract maximum parallelism
DAG Execution Models: Mechanisms

- in DAG programming models, we version data (say by iteration count)
  - it thus has declarative, write-once semantics
  - a node in a DAG will have associated with it:
    - the input data items (including version number) required
    - the output data items produced (usually with updated version number)
    - the function which performs this task
- running a task-DAG program involves:
  - generating the graph
  - allowing an execution engine to schedule tasks to processors
    - a task may execute when all of its input data items are ready
    - the function first ‘unpacks’ the input data items
    - the function informs the engine that its output data items have been produced before exiting
DAG Example: Tiled Cholesky Factorization

- **PLASMA** is a domain-specific programming model for dense linear algebra
- A (positive definite) symmetric matrix $A$ may be factored $A = LL^T$, where $L$ is triangular
- The ‘right-looking’ tiled algorithm executes in-place ($A$ may be stored in a (lower) triangular matrix), with $L$ overwriting $A$ (tile by tile)
- The (PLASMA) DAG pseudo-code is

```c
for k = 0..nb_tiles -1
    DPOTRF(A[k][k]) // A[k][k] = sqrt(A[k][k])
    for m = k+1..nb_tiles -1
        DTRSM(A[k][k], A[m][k]) // A[m][k] = A[k][k]^-1 A[m,k]
    for n = k+1..nb_tiles -1
        for (m = n+1..nb_tiles -1)
            DGEMM(A[m][k], A[n][k], A[m,n]) // A[m][n] -= A[n][k] A[m,n]^T
```

- Size of `nb_tiles` is a trade-off: ||ism / load balance vs amortize task startup & data fetch/store costs, cache performance etc
Tiled DAG Cholesky Factorization (II)

(courtesy Haidar et al, Analysis of Dynamically Scheduled Tile Algorithms for Dense Linear Algebra on Multicore Architectures)
Tiled DAG Cholesky Factorization (III)

- task graph with \( nb_{\text{tiles}} = 5 \)
- column on left shows the depth:width ratio of the DAG

(courtesy Haidar et al.)
Cholesky Factorization in PLASMA

- for each ‘core’ task, we define a function to insert it into PLASMA’s task-DAG scheduler and one to perform the task

```c
int Dsched_dpotrf(Dsched dsched, int nb, double *A, int lda)
{
    DSCHED_Insert_Task(dsched, TASK_core_potrf,
                      sizeof(int), &nb, VALUE,
                      sizeof(double)*nb*nb, A, INOUT | LOCALITY,
                      sizeof(int), &lda, VALUE, 0);
}
void TASK_core_dpotrf(Dsched dsched)
{
    int nb, lda; double *A;
    dsched_unpack_args_3(dsched, nb, A, lda);
    dpotrf("L", &n, A, &lda, ...);
}
```

- and these are inserted into the scheduler, which works out the implicit dependencies:

```c
for (k = 0; k < nb_tiles; k++) {
    Dsched_dpotrf(dsched, nb, A[k][k], lda);
    for (m = k+1; m < nb_tiles; m++) {
        Dsched_dtrsm(dsched, nb, A[k][k], A[m][k], lda);
        for (n = k+1; n < n_tiles; n++) {
            Dsched_dsyrk(dsched, nb, A[n][k], A[n][n], lda);
            for (m = n+1; m < nb_tiles; m++)
                Dsched_dgemm(dsched, nb, A[m][k], A[n][k], A[m,n]);
        }
    }
}
```
architecture for scheduler:
- inserted tasks go into an implicit DAG
- can be in NotReady, Queued, or Done states
- workers execute queued tasks
- descendants of Done tasks are examined to see if they are ready

execution trace of Cholesky: large vs small window sizes
- a full DAG becomes very large
- necessary only to expand a ‘window’ of the next tasks to be executed
- needs to reasonably large to get good scheduling
Intel Concurrent Collections (CnC)

- Intel CnC is based on two sources of ordering requirements:
  - producer / consumer (data dependence)
  - controller / controllee (control dependence)
- CnC combines ideas of streaming, tuple spaces and dataflow
- users need to supply a CnC dependency graph and code for step functions
- step functions get and put data into the CnC store

(figs courtesy Knobe&Sarker, CnC Programming Model)
### Cholesky in CnC

#### Dependency graph specification:

- **Cholesky** = dportf, **Trisolve** = dtrsm, **Update** = dsyrk + dgemm

- Note the iteration is an explicit part of the tag

(figure courtesy Knobe & Sarker, CnC | Programming Model)

#### Step function for the symmetric rank-k update:

```cpp
int
update::execute(const triple &t, cholesky_context &c) const
{
    const int iter = t[0], row = t[1], col = t[2];
    tile_const_ptr_type A_block, L1_block, L2_block;
    c.Lkji.get(triple(k, j, i), A_block); // Get input tile
    if (i == j) { // On diagonal -> both tiles are the same
        c.Array.get(triple(k+1, j, k), L1_block);
        cblas_dsyrk(...);
    } else { // Non-diagonal tile -> get 2 tiles
        c.Lkji.get(triple(k+1, i, k), L2_block); // Get 1. tile
        c.Lkji.get(triple(k+1, j, k), L1_block); // Get 2. tile
cblas_dgemm(...);
    }
    // Output (next time)
    c.Lkji.put(triple(k+1, j, i), A_block);
    return CnC::CNC_Success;
}
```

- The task graph is generated by executing a user-supplied harness

(figure courtesy Schlimbach, Brodman & Knob, Concurrent Collections on Distributed Memory)
DAG Task Graph Programming Models: Summary

- programmer needs only to break computation into tasks
- otherwise not too much more than specifying sequential computation
  - specify task ‘executor’ and task harness generator
  - becomes trickier when the task DAG is data-dependent
  - e.g. iterative linear system solvers

- advantages:
  - maximizing (optimizing) parallelism, transparent load balancing
  - arguably simpler programming model: no race hazards / deadlocks!
  - testing on a serial processor *should* reveal all bugs
  - abstraction over underlying architecture
  - permits fault-tolerance: tasks on a failed process may be re-executed (requires data items are kept in a resilient store)

- shows promise for both small and large scale parallel execution
  - distributed versions exist (CnC, DPLASMA, DARMA)