Task Parallelism

- Divide-and-conquer
  - Each task recursively creates $n$ tasks that divide the problem into subproblems
  - Each task $t$ then waits for all $n$ tasks to finish and then may ‘combine’ the responses
  - At some point the recursion stops (at the bottom of the tree), and some sequential kernel is executed
  - Then the result is propagated upward in the tree recursively
  - Examples: fibonacci, quick sort, ...
State-space search
- Each task recursively creates $n$ tasks to partition the search space
- If the problem is one-solution search, as soon as a task encounters a solution, the program may need to terminate
  - Kill-chasing problem

All-solution search may require behaviour much like divide-and-conquer where values are combined
- Example: all-solution nqueens
- Number of solutions are accumulated recursively up the tree
Each Fib chare is a task that performs one of two actions:

▶ Creates two new Fib chares to compute $fib(n - 1)$ and $fib(n - 2)$ and then waits for the response, adding up the two responses when they arrive
  ★ After both arrive, sends a response message with the result to the parent task
  ★ Or prints the value and calls CkExit() if it is the root
▶ If $n = 1$ or $n = 0$ (passed down from the parent) it sends a response message with $n$ back to the parent task
Fibonacci Example

```
mainmodule fib {
    mainchare Main {
        entry Main(CkArgMsg* m);
    };

    charare Fib {
        entry Fib(int n, bool isRoot, CProxy_Fib parent);
        entry void response(int value);
    };
};
```
struct Main : public CBase::Main {
   Main(CkArgMsg* m) {
      CProxy_Fib::ckNew(atoi(m->argv[1]), true, CProxy_Fib());
   }
};

struct Fib : public CBase::Fib {
   CProxy_Fib parent; bool isRoot; int result, count;

   Fib(int n, bool isRoot, CProxy_Fib parent) :
      parent(parent), isRoot(isRoot), result(0), count(n < 2 ? 1 : 2) {
      if (n < 2) response(n);
      else {
         CProxy_Fib::ckNew(n - 1, false, thisProxy);
         CProxy_Fib::ckNew(n - 2, false, thisProxy);
      }
   }

   void response(int val) {
      result += val;
      if (count == 0) {
         if (isRoot) {
            CkPrintf("Fibonacci number is: %d\n", result);
            CkExit();
         } else {
            parent.response(result);
            delete this;
         }
      }
   }
};
Fibonacci Execution

- \( \text{fib}(5) \)
- \( \text{fib}(4) \)
- \( \text{fib}(3) \)
Fibonacci Execution

fib(5)
  fib(4)
    fib(3)
    fib(2)
  fib(3)
  fib(2)
    fib(1)
Fibonacci Execution

```
 fib(5)
 /  \
/    \
fib(4)  fib(3)
 |      /  \
|     /    \
|    /      \
fib(3)  fib(2)  fib(2)
 |    / \
|   /    \
|  /      \
fib(2) fib(1) fib(1)
 |    |    |
|    |    |
fib(1) fib(0) fib(0)
```

Laxmikant V. Kale

Fibonacci Execution

```
fib(5)
fib(4)
fib(3)  fib(2)
fib(1)  fib(0)
fib(2)  fib(1)
fib(1)  fib(0)
```

Laxmikant V. Kale
Basic Charm++
September 6, 2012
Fibonacci Execution

$$\text{fib}(5)$$
$$\text{fib}(4)$$
$$\text{fib}(3)$$
$$\text{fib}(2)$$
$$\text{fib}(1)$$
$$\text{fib}(0)$$
$$\text{fib}(2)$$
$$\text{fib}(1)$$
$$\text{fib}(3)$$
$$\text{fib}(2)$$
$$\text{fib}(1)$$
$$\text{fib}(1)$$
$$\text{fib}(0)$$

Laxmikant V. Kale
Basic Charm++
September 6, 2012
Fibonacci Execution

Laxmikant V. Kale
Basic Charm++
September 6, 2012
Fibonacci Execution

fib(5)
fib(4)
fib(3)
Fibonacci Execution

fib(5)
Fibonacci Performance

- How much work/computation does each chare do in this example?
- What are some of the overheads of this approach?
- Is there way we can reduce/amortize the overhead?
Possible Solution

- Set a sequential threshold in the computational tree
  - Past this threshold (i.e. when $n < \text{threshold}$), instead of constructing two new chares, compute the fibonacci sequentially

$$\begin{align*}
\text{fib}(5) & \quad \text{fib}(4) \\
\text{fib}(3) & \quad \text{fib}(2) \\
\text{sequential fib}(3) & \quad \text{sequential fib}(2) \\
\text{sequential fib}(3) &
\end{align*}$$

- $\text{fib}(5), \text{fib}(4)$ are fine grains, $\text{fib}(3), \text{fib}(2)$ are coarser grains
- The coarser grains now amortize the cost of the fine-grained execution
Fibonacci w/Threshold Example

```cpp
#define THRESHOLD 10

struct Main : public CBase_Main { /* ... same as before ... */ };

struct Fib : public CBase_Fib {
    CProxy_Fib parent; bool isRoot; int result, count;

    Fib(int n, bool isRoot_, CProxy_Fib parent_) :
        parent(parent_), isRoot(isRoot_), result(0), count(n < THRESHOLD ? 1 : 2) {
        if (n < THRESHOLD) response(seqFib(n));
        else {
            CProxy_Fib::ckNew(n - 1, false, thisProxy);
            CProxy_Fib::ckNew(n - 2, false, thisProxy);
        }
    }

    int seqFib(int n) { return (n < 2) ? n : seqFib(n - 1) + seqFib(n - 2); }

    void response(int val) {
        result += val;
        if (−−count == 0) {
            if (isRoot) {
                CkPrintf("Fibonacci number is: %d\n", result);
                CkExit();
            } else {
                parent.response(result);
                delete this;
            }
        }
    }
};
```

Laxmikant V. Kale
Basic Charm++
September 6, 2012
Amdahl's Law and Grainsize

- **Original “law”:**
  - If a program has $K\%$ sequential section, then speedup is limited to $\frac{100}{K}$.
  - If the rest of the program is parallelized completely

- **Grainsize corollary:**
  - If any individual piece of work is $> K$ time units, and the sequential program takes $T_{seq}$,
  - Speedup is limited to $\frac{T_{seq}}{K}$

- **So:**
  - Examine performance data via histograms to find the sizes of remappable work units
  - If some are too big, change the decomposition method to make smaller units
Grainsize

(working) Definition: the amount of computation per potentially parallel event (task creation, enqueue/dequeue, messaging, locking...).
Grainsize and Overhead

- What is the ideal grainsize?
- Should it depend on the number of processors?

\[ T_1 = T \left( 1 + \frac{v}{g} \right) \]
\[ T_p = \max \left\{ g, \frac{T_1}{p} \right\} \]
\[ T_p = \max \left\{ g, \frac{T\left(1 + \frac{v}{g}\right)}{p} \right\} \]

\( v \): overhead per message,
\( T_p \): \( p \) processor completion time
\( g \): grainsize (computation per message)
Rules of thumb for grainsize

- Make it as small as possible, as long as it amortizes the overhead
- More specifically, ensure:
  - Average grainsize is greater than $kv$ (say $10v$)
  - No single grain should be allowed to be too large
    - Must be smaller than $\frac{T}{p}$, but actually we can express it as:
    - Must be smaller than $kmv$ (say $100v$)

- Important corollary:
  - You can be at close to optimal grainsize without having to think about $p$, the number of processors
How to determine/ensure grainsize

- Compiler techniques can help, but only in some cases
  - Note that they don’t need precise determination of grainsize, just one that will satisfy a broad inequality
    - $kv < g < mkv$ ($10v < g < 100v$)