Today’s NP-Completeness Example: SUBSET-SUM

- Research Advertisement
  - Parallel Computing
  - Circuit Verification
SUBSET SUM

- Instance:
  - Let $S$ be a set,
  - Let $w : S \rightarrow \mathbb{Z}^+$ be a function that gives the “weight” of elements of $s$.
  - Let $t$ be an integer.

- Question: Is there a set $C \subseteq S$ such that the sum of the weights of the elements of $C$ is equal to $t$?

- SUBSET SUM is NP-complete
  - It is easy to see that SUBSET SUM in NP, proposing a set $C$ suffices as a certificate.
    - Such a subset is shorter than the original input, thus its size is polynomial in the length of the input.
    - Checking that $C \subseteq S$ and that $\sum_{m \in C} m = t$ are straightforward and polynomial time.
  - To show that SUBSET SUM is NP hard, we reduce one-in-three 3SAT to SUBSET SUM.
SUBSET SUM: Details
Verifying the Reduction
PARTITION is NP-Complete

- Problem instance: a finite set $S$ and a weight function $w : S \rightarrow \mathbb{Z}^+$. 
- Question: Can $S$ be partitioned into 2 disjoint sets, $S_1, S_2$ such that 
  $$\sum_{s \in S_1} s = \sum_{s \in S_2} s?$$ 
- PARTITION is NP-complete. Proof: by reduction from SUBSET SUM.
Dynamic Programming

- If there is some subset of $S$ whose sum equals $t$, we can perform that sum in order of increasing weights of the elements. Let $w_1, \ldots, w_m$ be this sequence of weights.
- This leads to a dynamic programming algorithm for solving SUBSET SUM.

```c
SubsetSum(Set<int> s, int t) {
    int[1 . . . t] x; /* initially all elements set to $m + 1$ */
    for int i = 1 . . . t do {
        for j = 1 . . . m do {
            if((w_j == i) | ((w_j < i) & (x[i-w_j] < w_j))) {
                x[i] = j;
                break; /* for m */
            }
        }
    }
    return(x[t] \leq m);
}
```

This algorithm runs in $O(t^2)$ time!
Weak vs. strong NP completeness

- A numerical problem has a pseudo-polynomial time complexity if it can be decided in time that is a polynomial in the values of the numbers occurring in the input.
  - SUBSET SUM has a pseudo-polynomial decision procedure.

- For a numerical problem with input $I$, let $\text{Length}(I)$ be the number of symbols in $I$ and $\text{Max}(I)$ be the largest (in absolute value) integer encoded by $I$.

- If there is a polynomial $p$ such that a problem, $X$, is NP-complete when restricted to inputs $I$ with

$$\text{Max}(I) \leq p(\text{Length}(I))$$

then we say that $X$ is strongly NP-complete.

- Less formally, $X$ is strongly NP-complete if there is no pseudo-polynomial decision procedure for $X$ (unless P = NP).
3-Partition is Strongly NP-Complete

- Problem instance: a finite set $S$ of $3m$ positive integers, a positive integer $b$, such that each $s \in S$ satisfies $b/4 < s < b/2$, and such that $\sum_{s \in S} s = mb$.

- Question: Can $S$ be partitioned into $m$ disjoint sets, $S_1, S_2, \ldots S_m$ such that for every $1 \leq i \leq m$, $\sum_{s \in S_i} s = b$?

- Note: by the constraints on $S$, each of the $S_i$ must have exactly three elements.

The material here on strong NP completeness is based on chapter 4 of Garey & Johnson, “Computers and Intractability.”
Multiprocessor Scheduling

- Problem instance:
  - a set, $T$, of tasks,
  - an integer-valued function $l(t)$ that for each $t \in T$ says how long task $t$ takes to run,
  - a positive integer, $p$, the number of processors,
  - a positive integer $d$, the deadline.

- Question: Can tasks be assigned to processors such that all tasks complete by the deadline?

- NP-completeness:
  - Multi-processor scheduling is NP-complete for any $p \geq 2$: proof by reduction from PARTITION.
  - Pseudo-polynomial time decision procedures exist for any fixed $p$ (but the degree of the polynomial grows with $p$).
  - Multi-processor scheduling is NP-complete in the strong sense when $p$ is allowed to be arbitrary: proof by reduction from 3-PARTITION.
Parallel Computing

Areas of interest:

- Parallel architectures
- Applications of parallel computing
- Energy-time trade-offs in computation
- Compilers and other system software for parallel computation
Superscalar Architectures: old ILP

- Fetch several (~4) instructions per cycles.

- **Rename** registers:
  - A logical register to physical register mapping, similar to virtual memory.
  - Keeps track of data dependencies between instructions.

- Send renamed instructions to issue queues of functional units.
  - When an instruction has its operands, it can execute.
  - Because the machine has multiple functional units, it can execute multiple instructions per cycle.

- **Key issues:**
  - Need to wait to **commit** an instruction until all previous instructions commit.
  - To find instructions that can execute in parallel, need to fetch beyond pending branches.
    - This requires **branch speculation**.
Merging on a Superscalar

while (x < xtop && y < ytop) {
    if (*x < *y) *z++ = *x++;
    else *z++ = *y++;
}

- Data dependent branch in each iteration
- Any branch predictor wrong 50% of time for random data
- High mispredict penalties leads to low performance
Tiny Processing Elements (TPEs)

- Divide a “core” into many small processors
- Each has its own instruction store and fetch
- Heterogeneous in the small,
  - homogeneous in the large(?)
- Communication through registers with FIFO semantics
Merging on a TPE cluster

- Separate cache-TPEs fetch \( x \) and \( y \) streams.
- Another TPE merges the two streams.
- A fourth TPE writes the result back to cache/memory.
- The fetch and store TPEs make progress regardless of the branch outcome for the compare TPE.
TPE Research Questions

- How to take advantage of multiple instruction streams for executing a single thread.

- How to write a compiler for TPEs:
  - Compilers for traditional (C, C++, Java) languages.
  - Compilers for languages with explicit parallelism.

- On-chip network design.

- Power vs. speed trade-offs.

- ...
Why do Circuit-Level Verification?

- Digital design has become relatively low error:
  - Systematic design flows.
  - Lots of simulation.
  - Equivalence checking.
  - Model checking.

- Circuit-level bugs remain a problem:
  - SPICE is still the main validation tool, and it doesn’t scale.
  - Deep-submicron circuit effects undermine digital abstractions.
  - Hard/impossible to simulate bugs.
Arbiters

- **Specification**
  - Initially: $\neg r_1 \land \neg r_2 \land \neg g_1 \land \neg g_2$.
  - Assume: $\square r_i U g_i$, $\square \neg r_i U \neg g_i$.
  - Guarantee:
    - Handshake: $\square \neg g_i U r_i$, $\square g_i U r_i$.
    - Mutual Exclusion: $\square \neg (g_1 \land g_2)$.
    - Liveness: $\square (r_1 \oplus r_2) \Rightarrow \Diamond (g_1 \oplus g_2) \lor (r_1 \land r_2)$, $\square \neg r_i \Rightarrow \Diamond \neg g_i$.
      Note: because metastability is unavoidable, no arbiter can guarantee $\square (r_1 \land r_2) \Rightarrow \Diamond (g_1 \lor g_2)$.

- **Why Verify an Arbiter?**
  - Exercise in modeling concurrent events from the environment.
  - Requires handling a non-trivial circuit behavior: metastability.
Specifying signal behavior – Brockett’s annulus:

- Region 1 represents a logical low signal. The signal may wander in a small interval.
- Region 2 represents a monotonically rising signal.
- Region 3 represents a logical high signal.
- Region 4 represents a monotonically falling signal.
- Brockett’s annulus allows entire families of signals to be specified.
Coho: Reachability Using Projections

- Coho projects high dimensional polyhedron onto two-dimensional subspaces.

- A projectagon is the intersection of a collection of prisms, back-projected from the projection polygons.

- Coho computes reachable sets by integrating over a series of timesteps:
  - A bounding projectagon is obtained by moving each face forward in time.
  - Projectagon faces correspond to projection polygon edges; thus, Coho works on one edge at a time.

from: http://pond.dnr.cornell.edu/
Results

- Safety Properties
  - Mutual Exclusion
  - Handshake Protocol
  - Brockett Annuli

- Liveness Properties

Mutual Exclusion
Results

- Safety Properties
  - Mutual Exclusion
  - Handshake Protocol
  - Brockett Annuli

- Liveness Properties

Handshake
Results

- Safety Properties
  - Mutual Exclusion
  - Handshake Protocol
  - Brockett Annuli

- Liveness Properties

\[ \dot{x} \text{ vs. } x \quad \dot{g} \text{ vs. } g. \]

Brockett Annuli
Results

- Safety Properties
  - Mutual Exclusion
  - Handshake Protocol
  - Brockett Annuli

- Liveness Properties:
  - Initialization: stable within 200ps
  - Uncontested Requests: grant the client within 350ps
  - Contested Requests: metastability within hyper-rectangle

\[ r_1 \in B_3 \quad x_1 \in [0.55, 1.3] \quad g_1 \in B_1 \]
\[ r_2 \in B_3 \quad x_2 \in [0.55, 1.3] \quad g_2 \in B_1 \]

- Reset: withdraw grants within 270ps
- Fairness: grant the other client within 420ps
The last slide

- Dec. 1: HW 11 due at 4pm.
- Dec. 1, 3, 5: office hour from 1-2pm.
- Dec. 6: final exam, 3:30-6:30pm, CHBE 103

Thanks to all of you for a good term! 😊