Task-Oriented Programming: Task-Graph Enhancements and Validation

Yitzhak (Tsahi) Birk
Israel Lenchner

Technion
The Hypercore Architecture

Nimrod Bayer + Ran Ginosar; later Plurality Ltd.; now adopted and adapted by Ramon Chips and incorporated in the RC-64 satellite-born accelerator.
HyperCore: Memory Architecture

- NO PRIVATE MEMORY
  - Any core is equally effective on any task
  - Dramatically simpler programming

- Numerous memory banks and “anti-local” address-to-bank mapping
- Low-latency (~1 cycle each way), high-bandwidth combinational NOC
  - No memory communication bottleneck

Resembles a PRAM machine
Programming Model: Task-Oriented Programming

- Programmer or tool identify possible parallelism
- Compile into
  - task-dependency graph
  - task codes
- Task graph loaded into scheduler
- Tasks loaded into memory

Task template:

```
Regular
Duplicable
Dummy (F/J)

{ task xxx( dependencies )

... INSTANCE ....

....
}
```
Task Types

• Regular task:
  • A single piece of sequential code
  • Can return True/False to scheduler
  • Execution of a dependent task may be conditioned upon the return value

• Duplicable task:
  • Multiple instances of same code running on different data (fixed stride)
  • Any subset may be executed concurrently
  • No return flag (other than completion)
  • Handled as a single vertex in the task graph
    (any number of instances may be dispatched simultaneously; task completed when all are done)

• Dummy task (Fork/Join):
  • No core allocated
  • Used to represent more complex dependencies.
Low latency parallel scheduling (task dispatching)

Multiple instances of a duplicable task may be run concurrently and can be dispatched simultaneously.

⇒ Enables efficient exploitation of fine grain parallelism that is often readily available but cannot be exploited due to prohibitive data movement and/or task dispatching overhead!
Program Example – Adding Two Vectors

• SET_QUOTA is a runtime function that sets the number of instances to run
• INSTANCE_NUMBER will get the values [0...length-1]

Task code
void init(void){
    SET_QUOTA(taskAdd,length);
}
void taskAdd(void){
    int id=INSTANCE_NUMBER;
    c[id]=a[id]+b[id];
}
void finish(void){
}
Example: Reduce (Task-Oriented)

- Sum of a vector

```c
void init(void){
}
void condition(void){
    if(length<=1){
        return false;
    }
    length/=2;
    SET_QUOTA(taskSum,length);
    return true;
}
void taskSum(void){
    int id=INSTANCE_NUMBER;
    arr[id] = arr[id] + arr[id + length];
}
void finish(void){
}
```

- Multiple calls to SET_QUOTA are allowed, the last value before triggering the task is relevant.
Architecture Benefits

• Any core can do any task equally well on short notice
  • scales automatically (code is agnostic to number of cores)
  • easy accommodation of core failure

• Many-bank shared cache + fast C-to-M NoC
  • low latency
  • No cache coherence issues
  • No communication bottleneck accessing shared memory

• Fast scheduling of tasks to free cores (many at once)
  • enables efficient exploitation of fine grain data parallelism
  • impossible in other architectures due to:
    • task scheduling overhead
    • data locality

• Programming model:
  • intuitive to programmers
  • easy for automatic parallelizing compiler
Shortcomings

• Limited precedence constraints, especially among duplicable tasks
  • Reason: “all or none” → serialization among such tasks
    → cache inefficiency when they share data and the cache cannot hold all of it
  • Solution: permit staggered lockstep with some slack

• Programming model exposure: accidental omission of edges in the ask graph leads to unpredictable execution results.
Our Goals:

• Ensure program correctness (verification that there are no data races among tasks that may execute concurrently)
  • No false approvals, but a low false rejection rate is permissible

• Aspirations:
  • Correctness + low false alarm rate without overly restricting the programmer
  • Support the extended precedence relationships among duplicable tasks
  • Scalability to a very large number of tasks.
Data Race Criterion

• A data race between two tasks exists iff both touch the same memory address, and at least one of them writes to it.
  \[\text{W-W} \quad \text{W-R} \quad \text{R-W} \quad \text{R-R}\]

• Given the memory footprints of two tasks, the computational complexity of testing is \(O(M)\), where \(M\) is the size of the footprint.
Main Components

• **Given the task graph, determine concurrently-runnable tasks**

• Determine task footprints
  • Program code analysis (source code or at any compilation stage)
  • Run task and record footprint

• **Compare footprints of concurrently runnable tasks to check for data races**
Determining Memory Footprints

• Two approaches:
  • Examination of the code
    • Offline code examination, but
    • May limit programming flexibility (use of pointers, etc.) due to address ambiguity
  • Run the tasks and compare the actual (data) memory footprints of concurrently runnable tasks
    • Requires code instrumentation or some other mechanism
    • Insensitive to addressing mode
    • Limitation: true only for specific run.
  • Also: combining the two

• Observation: determining that two tasks access the same address does not necessarily require knowing the actual address (E.g., a shared variable with same name)
Determining Concurrently Runnable Tasks

• Given: task graph $G(T,D)$ (a directed graph)
  • $T$: tasks
  • $D$: dependences

• Derive $G_{TC}(T,D)$, the transitive closure of $G$.

• Derive the \textbf{Independence Graph} $G_{Ind}=G_{TC}(T,D')$: there is an edge between two tasks iff they are runnable concurrently.
Example: Constructing the Independence Graph

$G$ - Dependency Graph

$G_{TC}$ - Transitive Closure

$G_{ind}$ - Independence Graph
Checking for Races (Given the Memory Footprints)

• Requirement: for any two tasks $A$ and $B$ s.t. $(A,B) \in D'$, check for a race

• If done in a straightforward manner, $O(T^2 \cdot M)$
Useful Observations

• Determining that two tasks access the same address does not necessarily require knowing the actual address

• If there is a race between tasks A and task B, there is also a race between A and the union of (memory footprints) of B and any other tasks.

• If there is a race between the union of one subset of tasks and that of another subset, then there is a race between at least one member of the first and at least one member of the other

• For a set of $n$ concurrently runnable tasks (a clique in $G_{\text{Ind}}$), can check in $O(n \cdot M)$

• If we keep a $W$ footprint and a $W|R$ footprint, then 2 tests suffice instead of 3.
Using Cliques in the Independence Graph

• Find all maximal cliques (maximal subsets of concurrently-runnable tasks)
• Derive all intersections of maximal cliques.
• Partition the result into elementary cliques (cliques whose member tasks all belong to the same subset of maximal cliques).

• For Each elementary clique, carry out the test among the member tasks:
  • Iteratively scan the tasks, checking each tasks against the cumulative union of the footprints
  • Keep the cumulative footprint for future use

• For each non-elementary clique, carry out the test among its member elementary cliques, similarly storing cumulative footprints along the way.

• Proceed until all cliques have been covered.
Example: Clique-Based Race Detection
Complexity of Footprint Comparisons (Crude approximation)

- **E** - Number of elementary cliques
- **R** - ~ mean number of maximal cliques including any given elementary clique. (“Reuse factor”)
- **M** - ~ Memory footprint (approximation: that of a union of tasks equals that of a single task)
- **T** - Number of tasks

### Table: Computational Costs

<table>
<thead>
<tr>
<th></th>
<th>Straightforward</th>
<th>Direct Maximal Cliques</th>
<th>Incremental</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computational</strong></td>
<td>M·T²</td>
<td>M·T·R</td>
<td>M·(T+E·R)</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>M·T</td>
<td>M·T</td>
<td>M·(Max(T, E+))</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>logM</td>
<td>logM·(Height of clique tree)</td>
<td>logM·(Height of clique tree)</td>
</tr>
</tbody>
</table>
Finding the Source of a Detected Race

• Find the smallest colliding cliques
• Use group testing for further partitioning.
Complication: Mutex Among Tasks

• Tasks conditioned upon the return value of their predecessor:

• All possible memory accesses inside a task are treated as if they actually occurred

• If we do not take into account the mutex between A and B we may get a false race detection

• Solution: in $T_{TC}$
  • Insert an edge between the mutex tasks
  • Insert an edge between each node in the subtree below A and every node in the subtree below B → no edges between mutually exclusive tasks in $G_{ind}$ → will not check for races → no false race detection.
Complication: OR dependency

- Edge between A and C in the task map doesn’t apply they cannot run concurrently.
- For example, A and B start running, B finishes before A.
- When B has finished task C become runnable, and start running.
- Tasks A and C are now running concurrently.
- Solution: remove edges crossing OR junction.
- Lowering the false positives rate: add dependency between C and the Most recent common ancestor of A and B.
Intra-Task Complications

• Example:

  Task A:
  ```c
  if (cond) {
    X = 5;
  } else {
    y = 5;
  }
  ```
  Task B:
  ```c
  if (!cond) {
    X = 4;
  } else {
    y = 4;
  }
  ```

  No mutex between tasks, but mutex between writes to the same address.

  Again, ignoring the control path may result in a false race detection.

• Approaches:
  • deeper analysis from the outset or
  • detailed exploration upon detection

• No false race detection if using actual program traces.
Conclusions

• An interesting problem with a real motivation
• We have a path to detecting and locating races
• Work required on program memory-access analysis

• Will be happy to hear ideas on:
  • program analysis
  • tips on using the Clang static analyzer
  • optimal clique combining

Thank You