Harmony Overview

- Harmony system based on feedback loop

![Harmony System Diagram](image)
Simplex Algorithms

Nelder-Mead

Parallel Rank Ordering
Tuning Granularity

- **Initial Parameter Tuning**
  - Application treated as a black box
  - Test parameters delivered during application launch
  - Application executes once per test configuration

- **Internal Application Tuning**
  - Specific internal functions or loops tuned
  - Possibly multiple locations within application
  - Multiple executions required to test configurations

- **Run-time Tuning**
  - Application modified to communicate with server mid-run
  - Only one run of the application needed
Example Application

- **SMG2000**
  - 6-dimensional space
    - 3 tiling factors
    - 2 unrolling factors
    - 1 compiler choice
  - 20 search steps

- **Performance gain**
  - 2.37x for residual computation
  - 1.27x for on full application
The Irony of Auto-Tuning

• Intensely manual process
  o High cost of adoption

• Requires application specific knowledge
  o Tunable variable identification
  o Value range determination
  o Hotspot identification
  o Critical section modification at safe points

• Can auto-tuning be more automatic?
Towards Automatic Auto-tuning

• Reducing the burden on the end-user

• Three questions must be answered
  o What parameters are candidates for auto-tuning?
  o Where are the best code regions for auto-tuning?
  o When should we apply auto-tuning?
Our Goals

• Maximize return from minimal investment
  o Use profiling feature as a model
    o Should be enabled with a runtime flag
  o Aim to provide auto-tuning benefits within one execution

• Minimize language extension
  o Applications should be used as originally written

• Non-trivial goals with C/C++/Fortran
  o Are there any alternatives?
Chapel Overview

- Parallel programming language
  - Led by Cray Inc.
  - “Chapel strives to vastly improve the programmability of large-scale parallel computers while matching or beating the performance and portability of current programming models like MPI.”

<table>
<thead>
<tr>
<th>Type of HW Parallelism</th>
<th>Programming Model</th>
<th>Unit of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-node</td>
<td>MPI</td>
<td>executable</td>
</tr>
<tr>
<td>Intra-node/multi-core</td>
<td>OpenMP/pthreads</td>
<td>iteration/task</td>
</tr>
<tr>
<td>Instruction-level vectors/threads</td>
<td>pragmas</td>
<td>iteration</td>
</tr>
<tr>
<td>GPU/accelerator</td>
<td>CUDA/OpenCL/OpenAcc</td>
<td>SIMD function/task</td>
</tr>
</tbody>
</table>

Content courtesy of Cray Inc.
Chapel Methodology

Chapel language concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine

"Why is everything so slow?"
"Why don't my programs be more flexible?"
"I have more control?"

Target Machine
High-Level Abstractions

Content courtesy of Cray Inc.
Chapel Data Parallelism

- Only domains and forall loop required
  - Forall loop used with arrays to distribute work
  - Domains used to control distribution
    - A generalization of ZPL’s region concept
Chapel Task Parallelism

- Three constructs used to express control-based parallelism
  - `begin` – “fire and forget”
  - `cobegin` – heterogeneous tasks
  - `coforall` – homogeneous tasks

```plaintext
begin writeln("hello world");

cobegin {
  begin producer();
  coforall 1 in 1..numConsumers {
    consumer(i);
  } // wait here for all consumers to return
```
Chapel Locales

- MPI (SPMD) Functionality

```chapel
writeln("start on locale 0");
onLocales(1) do
    writeln("now on locale 1");
writeln("on locale 0 again");
```

```chapel
proc main() {
    coforall loc in Locales do
        on loc do
            MySPMDProgram(loc.id, Locales.numElements);
}

proc MySPMDProgram(me, p) {
    println("Hello from node ", me);
}
```

Content courtesy of Cray Inc.
Chapel Config Variables

```chapel
config const numLocales: int;
const LocaleSpace: domain(1) = [0..numLocales-1];
const Locales: [LocaleSpace] locale;
```

```
% a.out --numLocales=4
Hello from node 3
Hello from node 0
Hello from node 1
Hello from node 2
```
Leveraging Chapel

• Helpful design goals
  o Expressing parallelism and locality is the user’s responsibility
    o Not the compiler’s

• Chapel source effectively pre-annotated
  o Config variables help to locate candidate tuning parameters
  o Parallel looping constructs help to locate hotspots
Current Progress

- Harmony Client API ported to Chapel
  - Uses Chapel’s foreign function interface
  - Chapel client module to be added to next Harmony release
- Achieves the current state of auto-tuning
  - What to tune
    - Parameters must determined by a domain expert
    - Manually register each parameter and value range
  - Where to tune
    - Critical loop must be determined by a domain expert
    - Manually fetch and report performance at safe points
  - When to tune
    - Tuning enabled once manual changes are complete
Improving the “What”

• Leverage Chapel’s “config” variable type
  o Helpful for everybody to extend syntax slightly
    ```
    config const someArg = 5 in 1..100 by 2;
    ```

• Not a silver bullet
  o False-positives and false-negatives definitely exist
  o Goes a long way towards reducing candidate variables
  o Chapel built-in candidate variables
    ```
    dataParTasksPerLocale
dataParIgnoreRunningTasks
dataParMinGranularity
    numLocales
    ```
Improving the “Where”

• Naïve approach
  o Modify all parallel loop constructs
    o Fetch new config values at loop head
    o Report performance at loop tail
  o Use PRO to efficiently search parameter space in parallel

• Poses open questions
  o How to know if config values are safe to modify mid-execution?
  o How to handle nested parallel loops?
  o How to prevent overhead explosion?

• Solutions outside the scope of this project
  o But we’ve got some ideas...
What’s Possible?

• Target pre-run optimization instead
  o Run small snippet of code pre-main
  o Determine optimal values to be used prior to execution
• Example: Cache optimization
  o Explore element size and stride
  o Pad array elements to fit size
  o Define domains
    o Automatically optimize for cache size and eviction strategy
    o Further increase performance portability
• Generate library of performance unit-tests
  o Bundle with Chapel for distribution
Improving the “When”

• Auto-tuning should be simple to enable
  o Use profiling as a model (just add –pg to the compiler flags)

• System should be self-reliant
  o Local server must be launched with application
Open Questions

- **Automatic hotspot detection**
  - Time spent in loop
  - Variables manipulated in loop
  - How to determine correctness-safe modification points
    - Static analysis?

- **Moving to other languages**
  - C/Fortran lacking needed annotations
  - More static analysis?

- **Why avoid language extension?**
  - Is it really so bad?