Granularity Problem

Given a multicore machine with shared memory and a nested parallel program how to execute this program efficiently?

Requirements:
- Online
- Handle templated code
- Hardware independent

• Too small tasks ⇒ too large overheads
• Too big tasks ⇒ not enough parallelism
• Ideal task size ⇒ how to select the threshold?

Motivating Example

We are given an array with elements of type T. Find the number of elements that satisfy p:

\[ p = \{x \mid T \leq x \} \]

\[ \text{return } \text{hash}(x) \approx 2017 \].

```
template <T, P>
int match(T* lo, T* hi, P p)
int result
int n = hi - lo
if n <= \text{THRESHOLD}
\text{result} = \text{match Seq}(lo, hi, p)
else
T mid = lo + (n / 2)
int result1, result2
fork2join() {
result1 = match(lo, mid, p)
result2 = match(mid, hi, p)
}
\text{result} = result1 + result2
\text{return }\text{result}
```

Spguard to Control Granularity

- Arguments: complexity c(x), parallel pb(x) and sequential bodies ab(x);
- Maintains constant C that approximates the ratio between complexity and running time.
- Predicts the execution time as C\cdot c(x);
- If the prediction is less than \alpha then executes sequential body ab(x), else executes parallel body pb(x);
- Measures the total sequential execution time for future predictions

```
template <T, P>
int match(T* lo, T* hi, P p)
int result
int n = hi - lo
\text{spguard}(\{i \mid \text{complexity function}
\text{return }\}
, \{i \mid \text{parallel body}
if n \leq i
\text{result} = \text{match Seq}(lo, hi, p)
\text{else}
T mid = lo + (n / 2)
int result1, result2
fork2join() {
result1 = match(lo, mid, p)
result2 = match(mid, hi, p)
result = result1 + result2
\text{result} = result2
```

Implementation of Spguards as a Library

```
template <Complexity, Par_body, Seq_body>
void spguard<estimator, es, Complexity, c, Par_body pb, Seq_body sb>
int N = c(0);
time work = if es.is_small(N)
\text{then measured_run(ab)}
\text{else measured_run(pb)}
ex.report(N, work)
const double n // parallelism unit
const double s // growth factor
class estimator C // constant for estimations
int Dax = 0 // satisfies complexity measure
void report(int N, time T)
atomic
\text{if } T \leq N \text{ and } N \geq Dax
C = T
N = \text{Dax}
bool is_small(int N)
\text{return } (N \leq Dax) \text{ or }
(N \leq \alpha \cdot \text{Dax} \text{ and } N \leq \alpha \cdot \text{Nax})
```

Evaluation

https://github.com/deepsea-inria/pctl

We compare against the manually tuned code from PBBS suite [1].

```
<table>
<thead>
<tr>
<th>Application/input</th>
<th>PBBS (%)</th>
<th>Ours (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>blockradix-sort</td>
<td>0.60</td>
<td>0.74</td>
</tr>
<tr>
<td>exponential</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>random</td>
<td>0.59</td>
<td>0.66</td>
</tr>
<tr>
<td>comparison-sort</td>
<td>0.40</td>
<td>0.50</td>
</tr>
<tr>
<td>suffix-array</td>
<td>0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>closest-neighbours</td>
<td>0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>ray-cast</td>
<td>0.19</td>
<td>0.20</td>
</tr>
<tr>
<td>delaunay</td>
<td>0.19</td>
<td>0.20</td>
</tr>
<tr>
<td>mix</td>
<td>0.19</td>
<td>0.20</td>
</tr>
<tr>
<td>rmat24</td>
<td>0.19</td>
<td>0.20</td>
</tr>
<tr>
<td>rmat27</td>
<td>0.19</td>
<td>0.20</td>
</tr>
<tr>
<td>spanning</td>
<td>0.19</td>
<td>0.20</td>
</tr>
</tbody>
</table>
```

Theoretical Result

**Theorem.** \( T_{p} \leq (1 + O(1)) \cdot \frac{w}{p} \cdot O(\kappa) \cdot s + \frac{1}{P} \cdot O(\log \log \kappa) \),

where \( T_{p} \) is a parallel time of a nested parallel fork-join program including the constant time overhead for fork2join, \( P \) is a number of cores, and \( w \) and \( s \) are work and span without considering overheads.

It is a generalization of Brent’s bound \( T_{p} \leq w/p + s \) which ignores task creation costs.

**Assumptions**

for any spguard \( g := \text{spguard}(\text{g}(g), P, g) \).

```
\text{time measurements do not differ much from work}
\frac{1}{P} \leq M(S(g), I)/W(S(g), I) \leq E
```

References and Acknowledgements


This work is partially supported by the National Science Foundation (CCF-1408940 and CCF-1629444) and European Research Council (ERC-2012-SVG-308246).