Performance Challenges in Modular Parallel Programs

Umut Acar Vitaly Aksenov
Arthur Charguéraud Mike Rainey
aksenov.vitaly@gmail.com

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?

Evaluation

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

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

Application/input	PBBS (s)	Ours
blockradix-sort		
random	0.20	−7.4%
exponential	0.19	−8.4%
random kvp 256	0.49	−23.9%
random kvp 10^8	0.49	−27.7%
comparison-sort		
random	1.13	−36.4%
exponential	0.82	−31.3%
almost sorted	0.63	−18.8%
suffix-array		
trigrams	3.58	−6.3%
dna	1.29	−6.7%
text	4.11	−7.4%
wiki	3.66	−5.3%
convex-hull		
in circle	0.61	+5.8%
kuzmin	0.41	−6.9%
on circle	8.26	−32.4%
nearest-neighbours		
in square	5.75	−2.2%
kuzmin	22.00	−2.5%
in cube	7.90	−6.5%
on sphere	14.60	−31.2%
plummer	23.54	−2.5%
ray-cast		
in cube	7.90	−1.9%
on sphere	0.87	−0.2%
happy	0.50	−1.9%
xyz-rgb manuscript	9.46	+0.3%
turbine	4.10	−2.1%
delaunay		
in square	3.39	−4.1%
kuzmin	3.99	−4.4%
mis		
cube-grid	0.12	+1.2%
rMat24	0.07	+2.7%
rMat27	0.06	+2.7%
mst		
cube-grid	2.28	−9.9%
rMat24	2.21	−13.3%
rMat27	1.89	−16.3%
spanning		
cube-grid	0.62	−5.8%
rMat24	0.44	−0.6%
rMat27	0.33	−5.0%

References and Acknowledgements

[1] G. Blelloch et al., Brief-Announcement: The Problem Based Benchmark Suite. SPAA'2012, p. 68-70.

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

Motivating Example

We are given an array with elements of type `T`. Find the number of elements that satisfy `p`:
`p = [&] (T* x) { return hash(x) == 2017 }.`

```
template <T, P>
int match(T* lo, T* hi, P p)
  int result
  int n = hi - lo
  if n ≤ THRESHOLD
    result = 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)
    })
    result = result1 + result2
return result
```

Type T	Size	threshold	Comment
char	800M	1	100x slower
		10	17x slower
		5000 (TBB)	optimal
		ours	optimal
char[64]	200M	1	78% slower
		10 (TBB)	6% slower
		5000	optimal
		ours	optimal
char[2048]	0.4M	1 (TBB)	optimal
		10	optimal
		5000	16% slower
		ours	optimal
char[131072]	0.01M	1 (TBB)	optimal
		10	optimal
		5000	19x slower
		ours	optimal

Spguard to Control Granularity

- Arguments: complexity `c(x)`, parallel `pb(x)` and sequential bodies `sb(x)`;
- Maintains constant C that approximates the ratio between complexity and running time.
- Predicts the execution time as `C·c(x)`;
- if the prediction is less than κ then executes sequential body `sb(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
  spguard([&] { // complexity function
    return n
  }, [&] { // parallel body
    if n ≤ 1
      result = 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)
      })
      result = result1 + result2
  }, [&] { // sequential body
    result = match_seq(lo, hi, p)
  })
  return result
```

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()
  time work = if es.is_small(N)
                then measured_run(sb)
                else measured_run(pb)
  es.report(N, work)

const double κ  // parallelism unit
const double α  // growth factor

class estimator
  double C        // constant for estimations
  int Nmax = 0    // max complexity measure
  void report(int N, time T)
    atomic { if T ≤ κ and N > Nmax
               C = T / N
               Nmax = N }

  bool is_small(int N)
    return (N ≤ Nmax) or
           (N ≤ α · Nmax and N · C ≤ α · κ)
```

Challenges

1. In nested parallel programs a spguard with wrong initial constant can always choose parallel body and will never update a constant.

Solution. Report the time spent during the parallel call as the sum of sequential sub-computations.

2. A constant can differ significantly for different sizes: for example, if the data becomes unfit in the cache. So, we cannot use the obtained constant for the sizes arbitrarily bigger.

Solution. We allow to sequentialize only if the predicted work is small **and** the size is not much bigger than the current maximal known size.

Theoretical Result

Theorem. $T_p \le (1 + \frac{O(1)}{\kappa}) \cdot \frac{w}{P} + O(\kappa) \cdot s + \frac{1}{P} \cdot O(\log^2 \kappa)$,
where T_p is a parallel time of a nested parallel fork-join program including the constant time overhead per **`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 $T_p \le w/P + s$ which ignores task creation costs.

Assumptions

for any spguard g = **`spguard`**$(F(g), P(g), S(g))$, $P(g)$ = `[&]{`S_p, **`fork2join`**$(L(g), R(g))$, S_m`}`

time measurements do not differ much from work	$\frac{1}{E} \le M(S(g), I)/W_s(S(g), I) \le E$
sequential work in $P(g)$ is not much bigger than work in $S(g)$	$1 \le W_s(P(g), I)/W_s(S(g), I) \le D$
a task α times bigger induces no more than β more work	if $F(J) \le F(I) \le \alpha \cdot F(J)$ then $W_s(g, J) \le W_s(g, I) \le \beta \cdot W_s(g, J)$
there is a γ-balance between branches of the fork	$\frac{1}{\gamma} \le W_s(L(g), I)/W_s(S(g), I) \le \gamma$ and $\frac{1}{\gamma} \le W_s(R(g), I)/W_s(S(g), I) \le \gamma$
spguards are called sufficiently frequently in a call tree	if g' is an immediate outer spguard of g then $W_s(S(g'), I) \le \gamma W_s(S(g), I)$