Automatic Parallelism Management

Sam Westrick  
Carnegie Mellon University  
POPL 2024  
London, UK

Matthew Fluet  
Rochester Institute of Technology

Mike Rainey  
Carnegie Mellon University

Umut Acar  
Carnegie Mellon University

joint work with:
Fine-Grained Task Parallelism

- par: (unit -> 'a) * (unit -> 'b) -> 'a * 'b
- scheduler guarantees efficient execution on any number of processors

(* do body(i) for each i: lo <= i < hi *)
fun parfor(lo, hi, body) = 
  if lo >= hi then () else
  if lo+1 = hi then body(lo) else
  let val mid = lo + (hi-lo) div 2
  in par(fn () => parfor(lo, mid, body),
             fn () => parfor(mid, hi, body));
  ()
end
Parallelism Isn’t Free

(* do body(i) for each i: lo <= i < hi *)

fun parfor(lo, hi, body) = 
  if lo >= hi then () else 
  if lo+1 = hi then body(lo) else 
  let val mid = lo + (hi-lo) div 2 
  in par(fn () => parfor(lo, mid, body), 
      fn () => parfor(mid, hi, body)); 
  end

fun parfor(lo, hi, body) = 
  if hi-lo <= GRAIN_SIZE then 
    sequential_for_loop(lo, hi, body)
  let val mid = lo + (hi-lo) div 2 
  in par ... 
  end

up to 50x performance gap in practice
The Granularity Control Problem

- how much parallelism should I expose?
  (how “fine-grained” should my tasks be?)

(`* do body(i) for each i: lo <= i < hi *`)  
fun parfor(lo, hi, body) =  
  let val mid = lo + (hi-lo) div 2  
in par(fn () => parfor(lo, mid, body),  
  fn () => parfor(mid, hi, body));  
end

what grain size should you pick?

parfor(0, 1000, expensive_func)  
parfor(0, 100000000, cheap_func)  
parfor(0, N, fn i =>  
  let M = foo(i)  
  parfor(0, M, fn j => ...)  
)
The Granularity Control Problem

- how much parallelism should I expose?
  (how “fine-grained” should my tasks be?)
- can this be automated?

- lots of existing work
  (lazy scheduling, lazy binary splitting / lazy tree splitting, heartbeat scheduling, oracle-guided control, static cut-offs, cost annotations, profiling techniques...)

- we want...
  - fully general solution
  - provably efficient
  - implementable and effective in practice
Automatic Parallelism Management

Our Approach

static
programmer uses \texttt{par} liberally to express opportunities for parallelism
- \texttt{PCall}: new compilation technique for \texttt{par} with \textit{nearly zero cost}
- \texttt{PCall} behaves sequentially by default (avoids task creation by default)
- each \texttt{PCall} can be dynamically \textit{promoted} into an actual parallel task

dynamic
provably efficient scheduling of \textit{promotions}
- each promotion releases parallelism but also incurs a cost
- our algorithm guarantees...
  - \textit{work-efficiency} (cost of all promotions is amortized)
  - \textit{span-efficiency} (theoretical parallelism is preserved)

\textit{full implementation in MaPLe}
github.com/MPLLLang/mpl
Compilation

parallel source language (higher-order, polymorphic) → SSA IR (first-order, monomorphic) → optimize → machine code w/ pcall → runtime scheduler

implement parallelism with **pcall**: potentially parallel function calls

allocate registers, lay out memory, optimize, etc.

monomorphize, defunctionalize, optimize, etc.

optimize

execute with pcall promotion (dynamic)
**PCall Calling Convention**

\[ \text{Call}(\text{func, args, } \text{ret}) \]

\[ \text{PCall}(\text{func, args, ret_seq, ret_sync, ret_spwn}) \]

**IF NEVER PROMOTED...**
- behaves the same as normal \textit{Call}
- caller resumes at \textit{ret_seq}
- ret\_sync and ret\_spwn are discarded
PCall Promotion

\[\text{local vars}\]
- ret_spwn
- ret_sync
- ret_seq

\text{(caller)}

\text{new task}
- \text{local vars}
- ret_spwn
- ret_sync

\text{(caller)}

\text{promote}
Scheduling Promotions

- each promotion exposes parallelism but incurs a cost
- idea: amortize cost of promotion against “true” work

- algorithm
  - every N microseconds, each thread receives C tokens
  - any thread may spend one token to promote the *outermost* (oldest) outstanding PCall (in the thread’s own call-stack)

**Theorems:**
work-efficiency and span-efficiency
Parallelism Overhead (lower is better)

- *vanilla MaPLe*
- *(Ours) MaPLe + automatic parallelism management*

64-core performance
- two versions of each bench
  - **NoGran**: no granularity control
  - **Manual**: manual granularity control

overhead = $\frac{\text{Time}(\text{NoGran})}{\text{Time}(\text{Manual})}$
Automatic Parallelism Management

Summary

- **nearly zero cost** compilation technique for par (PCall + promotions)
- provable and practical efficiency, even without granularity control

see the paper for...

- SSA formalism, PCall semantics
- theorems: work- and span-efficiency
- description of changes to MLton/MaPLe compiler and run-time system
- in-depth empirical evaluation

github.com/MPLLang/mpl