



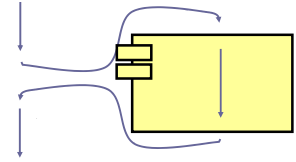
Optimized Composition of Parallel Components

Christoph Kessler, Linköping university
 Welf Löwe, Växjö university

Components



- Module / Container
 - encapsulates code and static data
 - unit of deployment and composition
 - can be light-weight
- Functional interface
 - for functional composition (i.e., call/return)
 - interoperability



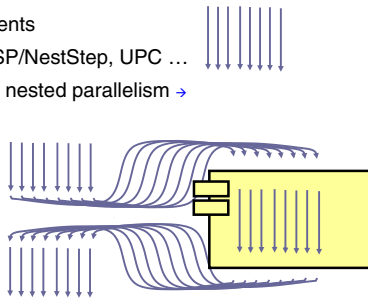
```
interface Sort {
    void sort ( float *arr, int n );
}
component QS implements Sort
{ ... }
```

2

Parallel Components (1)



- For SPMD environments
 - e.g. MPI, Fork, BSP/NestStep, UPC ...
 - Group splitting for nested parallelism →
- Source-code or binary components
 - Explicitly parallel
 - Called by groups of processors



► M-tasks, malleable tasks, ...

3

Nested SPMD Parallelism

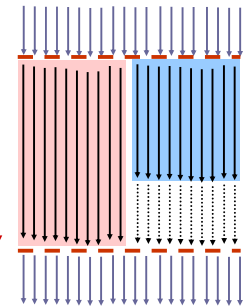


- Split a processor group in independent subgroups
 - split scope of sharing, memory consistency, barriers, ...
 - Statically
 - Dynamically e.g. parallel divide-and-conquer computations
- For task-level parallelism

```
/*@compose_parallel*/
/*@1*/ foo();
/*@2*/ bar();
/*@end_compose_parallel*/
```

- Longest-running subgroup determines parallel runtime $TIME_{par}(task@1, task@2)$

$TIME_{par}$



4

Nested SPMD Parallelism



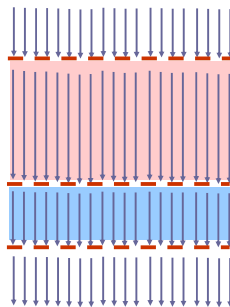
- Split a processor group in independent subgroups
 - split scope of Sharing, Speicherkonsistenz, Barriers, ...
 - Statically
 - Dynamically e.g. parallel divide-and-conquer computations
- For task-level parallelism

```
/*@compose_parallel*/
/*@1*/ foo();
/*@2*/ bar();
/*@end_compose_parallel*/
```

- Longest-running subgroup determines parallel runtime $TIME_{par}(task@1, task@2)$

- Alternative schedule: **Serialization**

$TIME_{par}$



Parallel Components (3)



- Component provider describes the **functionality**
 - in order to declare equivalence of different implementations (components)
 - by naming conventions for interface name.
- Composition tool
 - registers all components with same functionality
 - at deployment

```
interface Sort {
    void sort ( float *arr, int n );
}
component ParQS variantOf Sort
{ ... }
```

6

Parallel Components (4)

Component provider exports **execution time metadata** and **-code**:

- For each component function

```
foo (Type1 param1, ..., TypeK paramK) {...}
```

provide approximation of the **average-case execution time**

```
time_foo (int p, Type1 param1, ..., TypeK paramK) {...}
```

- Metafunction for static execution time prediction

```
/*@performance_aware*/
interface Sort {
  void sort ( float *arr, int n );
}
```

→ metafunction to be implemented by provider:

```
float time_sort ( int p, float *arr, int n );
```



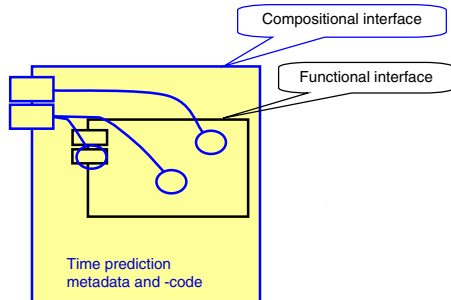
Example: Parallel Quicksort Component

```
/*@performance_aware*/
component ParQS variantOf Sort {
  ...
  float find_pivot ( float *arr, int n ) { ... }
  int partition ( float *arr, int n, /*@time_metadata
  ...
  /*@performance_aware*/
  void sort ( float *arr, int n )
  {
    float pivot = find_pivot ( arr,
    int n1 = partition ( arr, n, piv
    // recursive calls are indepe
    /*@compose_parallel*/
    /*@1*/ sort ( arr, n1 );
    /*@2*/ sort ( arr+n1, n-n
  }
  /*@end_compose_parallel
  ...
  } } @end_time_metadata */
```

Time-Metadata and -code – used in composition tool at deployment-time

Code markup for composition points

Performance-Aware Component



More Sorting Components ...

```
/*@performance_aware*/
component SeqQS variantOf Sort {
  // ... details omitted
  /*@performance_aware*/
  void sort( float arr, int n )
  {
    seq qsort( arr, n ); // done on 1 processor only
  }
  /*@time_metadata
  const float[] T_qsrt = ... // Table of seq. sorting times
  float time_sort ( int p, int n ) { return T_qsrt[n]; }
  @end_time_metadata */
}
```

and many other components implementing functionality Sort:

e.g. Parallel Mergesort ParMS, ...

To be registered in composition tool at deployment-time



Optimizing composition

For each call of a component function $foo(n, \dots)$:

- Automatic Choice** among components of same functionality foo

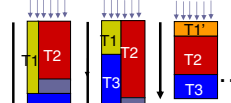
- the one with best (expected) exec. time
- Table-lookup and dynamic dispatch
- Also useful for composing sequential programs
- Generalization of OO dynamic dispatch for virtual methods: table now also indexed in problem sizes and #available processors

For each parallel composition point $compose_parallel(T_1, T_2, \dots)$

- Find best variant for each subtask-call

- AND simultaneously find an optimal schedule**

- Table-Lookup Schedule + Variants
- Largest potential for optimization
- Scheduling independent variant malleable tasks



Computing the tables

Composition tool computes at deployment time (off-line):

- Variant-Dispatch-Table** $V_{foo}[1..M][1..P]$
 - Contains function pointers to (expected) best variants of $foo()$ with p processors and problem size n
- Best-Time Table** $T_{foo}[1..M][1..P]$ for $V_{foo}[1..M][1..P]$
- Schedule-Lookup-Table** $S_{ParCp}[1..N-1][1..N-1] \dots [1..P]$ for each parallel composition point in a parallel component
 - contains tuples $S_{ParCp}[n_1][n_2] \dots [p]$
 - Branch address to code for (expected) best schedule with p processors and subtask sizes n_1, n_2, \dots
 - Processor allocation for each subtask

from the $time_...$ -metafunctions

by Dynamic Programming ...



DP-Algorithm computing the tables



Construct $V_{foo}[1..N][1..P]$, $T_{foo}[1..N][1..P]$, $S_{ParOp}[1..N][1..N][1..P]$, ...:

- For base problem size ($n=1$): no recursion!
obtain $T_{foo}[1][p]$ and $V_{foo}[1][p]$ directly from calling $time_foo(p, 1)$
 - For $p=1$: Schedule for $compose_parallel$ is always serial;
obtain $T_{foo}[n][1]$ and $V_{foo}[n][1]$ by minimizing over $time_foo(1, n)$ of all registered components implementing foo
 - For remaining columns $p = 2, 3, \dots, P$:
 - For problem sizes $n = 2, 3, \dots, N$:
 - NB: For all $n_1 < n, n_2 < n, \dots, p_1 \leq p, p_2 \leq p, \dots$:
Best variants $V_{\dots}[n_1][p_1], V_{\dots}[n_2][p_2], \dots$
with times $T_{\dots}[n_1][p_1], T_{\dots}[n_2][p_2], \dots$ already computed.
- Determine best schedule $S_{\dots}[n_1][n_2]..[p]$ for each $compose_parallel$ by approximation or brute-force-optimization over all $n_1 < n, n_2 < n, \dots$
 - $T_{foo}[n][p]$ und $V_{foo}[n][p]$ by
 - Executing all $time_foo(p, n)$ with table-lookup of best schedule's expected makespan (already computed) at each $compose_parallel$
 - Minimizing over all variants.

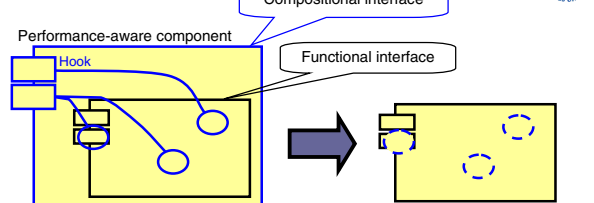
The Dispatch Table V_{sort}

- computed for implementations of
 - ParQS (1),
 - ParMS (2),
 - SeqQS (3),
 - ParIS (4)
- in Fork on SB-PRAM

	P=	1	2	3	4	5	6	7	8	9	...	16
N= 1:		3	3	3	3	3	3	3	3	3	3	3
N= 2:		3	3	3	3	3	3	3	3	3	3	3
N= 3:		3	3	3	3	3	3	3	3	3	3	3
N= 4:		3	3	3	3	3	3	3	3	3	3	3
N= 5:		3	4	4	4	4	4	4	4	4	4	4
N= 6:		3	2	2	2	2	2	2	2	2	2	2
N= 7:		3	2	2	2	2	2	2	2	2	2	2
N= 8:		3	2	2	2	2	2	2	2	2	2	2
N= 10:		3	2	2	2	2	2	2	2	2	2	2
N= 12:		3	2	2	2	2	2	2	2	2	2	2
N= 14:		3	2	2	2	2	2	2	2	2	2	2
N= 16:		3	2	2	2	2	2	2	2	2	2	2
N= 20:		3	2	2	2	2	2	2	2	2	2	2
N= 24:		3	2	2	2	2	2	2	2	2	2	2
N= 28:		3	2	2	2	2	2	2	2	2	2	2
N= 32:		3	2	2	2	2	2	2	2	2	2	2
N= 40:		3	2	1	1	1	1	1	1	1	1	1
N= 48:		3	2	1	1	1	1	1	1	1	1	1
N= 56:		3	2	1	1	1	1	1	1	1	1	1
N= 64:		3	2	1	1	1	1	1	1	1	1	1
N= 80:		3	2	1	1	1	1	1	1	1	1	1
N= 384:		3	2	1	1	1	1	1	1	1	1	1
N= 448:		3	1	1	1	1	1	1	1	1	1	1
N= 512:		3	1	1	1	1	1	1	1	1	1	1
N= 640:		3	1	1	1	1	1	1	1	1	1	1
N= 768:		3	1	1	1	1	1	1	1	1	1	1
N= 896:		3	1	1	1	1	1	1	1	1	1	1
N=1024:		3	1	1	1	1	1	1	1	1	1	1

Quad-logarithmic axis for N-dimension(s)
Truncation beyond tabulated indices

Composition



- Bind hooks (patching marked-up code locations)
- Functional interface remains.
- Technically: e.g. Invasive Software Composition [U. Aßmann, 2003]

ParQS Component after Composition



```

component ParQS {
  float find_pivot(float *arr, int n) { ... }
  int partition(float *arr, int n, float pivot) { ... }
  extern const int[][] V_sort; // The V table for sort
  const int[][] S_ParQS = ... // The S table for ParQS
  const void (*Sched_ParQS[2])(float *, int, int, int, int) = { s1_ParQS, s2_ParQS };

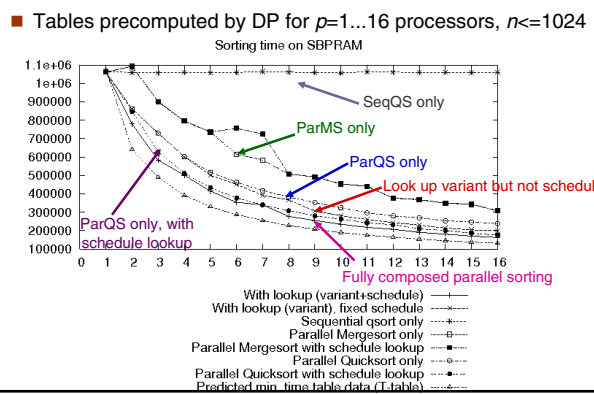
  void sort(float *arr, int n) {
    if (n==1) return;
    float pivot = find_pivot(arr, n);
    int n1 = partition(arr, n, pivot);
    int p = groupsize(); // # executing procs
    Sched_ParQS[ S_ParQS[n1][n-1][p][0] ]
      ( arr, n1, n-n1, S_ParQS[n1][n-1][p][1], S_ParQS[n1][n-1][p][2] );
  }

  void s1_ParQS(float *arr, int n1, int n2, int p1, int p2) { // serialized schedule:
    V_sort[n1][p1] ( arr, n1, p1 );
    V_sort[n-1][p2] ( arr+n1, n2, p2 );
  }

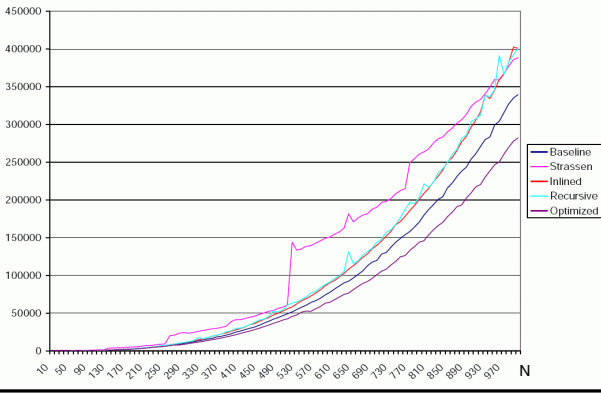
  void s2_ParQS(float *arr, int n1, int n2, int p1, int p2) { // parallel schedule:
    split_group(p1, p2) { V_sort[n1][p1](arr, n1); } { V_sort[n2][p2](arr+n1, n2); }
  }
}
    
```

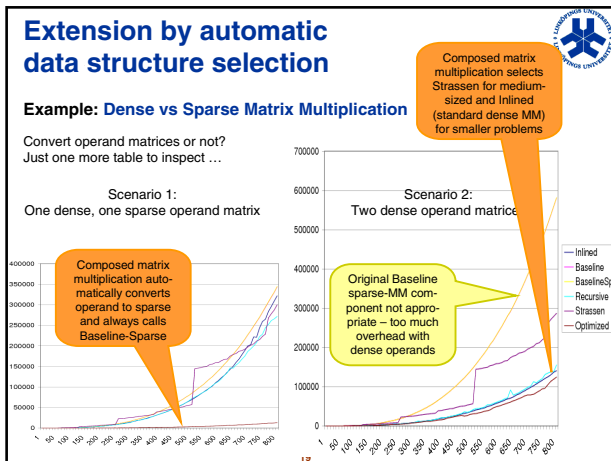
Schedule dispatch - look up function pointer in S_ParQS and call:

Sorting N=1023 numbers on SBPRAM



Matrix-Matrix-Multiplication, P=1





Summary

- Basic idea: Delay composition decisions (callee binding, scheduling / resource allocation, parameter conversions) to run-time (even if callee statically known)
 - Use run-time information about currently available resources to guide optimized composition
- Time prediction and dispatch code is computed off-line (at deployment time) from training data on the target machine
- Use time prediction (and possibly further metadata) to select at run-time exp. best variant, schedule, conversions
- Tabled prediction/dispatch information
 - low run-time overhead
- Writing components and libraries becomes easier
 - Focus on one algorithm per component. Code dealing with performance-related special cases is generated automatically.

Ongoing Projects and Future Work

- **Further domains**, e.g. matrix operations, FFT, image processing, ...
- **Further metrics** beyond ACET: Power, Code length, Chip area, ...
- **Further platforms**
 - Linux-Cluster
 - Multicore e.g. Cell BE
 - MPSoC / FPGA for Hardware-Software Co-Design
- **Modeling shared architectural resources** e.g. shared memory access
- **Composition tool**
 - Stronger table compression
 - Static flow analysis of call context properties
 - Static agglomeration of composition units
- **Adaptivity**
 - Instrumentation for dynamic updating of time parameter values
 - Occasional re-optimization