Automatic Discovery of Multi-level Parallelism in MATLAB

Arun Chauhan
Google Inc. and Indiana University

Pushkar Ratnalikar
Indiana University

February 7, 2015

Motivation

Multi-level Parallelism

Getting the premises right

Algorithm-level parallelism

Software-level parallelism

Hardware-level parallelism

Multi-level Parallelism

Getting the premises right

Algorithm-level parallelism

- Novel parallel algorithms
- Specialized for GPUs
- Specialized for FPGAs

Software-level parallelism

- Data parallelism
- Task parallelism

Hardware-level parallelism

- Superscalar
- Out of order execution
- Speculative execution
- Branch prediction

Multi-level Parallelism

Getting the premises right

No standard tools — **Algorithm-level parallelism**
- Novel parallel algorithms
- Specialized for GPUs
- Specialized for FPGAs

OpenMP, MPI, Intel TBB — **Software-level parallelism**
- Data parallelism
- Task parallelism

Largely automatic, compilers — **Hardware-level parallelism**
- Superscalar
- Out of order execution
- Speculative execution
- Branch prediction

Multi-level Parallelism

Getting the premises right

No standard tools	Algorithm-level parallelism	• Novel parallel algorithms • Specialized for GPUs • Specialized for FPGAs
OpenMP, MPI, Intel TBB	Software-level parallelism	• Data parallelism • Task parallelism
Largely automatic, compilers	Hardware-level parallelism	• Superscalar • Out of order execution • Speculative execution • Branch prediction

Multi-level Parallelism in Software

Getting the premises right

Software-level parallelism

Multi-level Parallelism in Software

Getting the premises right

Statement-level parallelism

Loop-level (data) parallelism

Function-level (task) parallelism

Component-level parallelism

Software-level parallelism

Multi-level Parallelism in Software

Getting the premises right

Statement-level parallelism

Loop-level (data) parallelism

Function-level (task) parallelism

Component-level parallelism

Software-level parallelism

- Multi-threaded builtin libraries
- Language constructs
  - E.g., parfor
- Parallel third-party libraries
  - E.g., GPUMat and StarP

Parallelism in MATLAB

- ILP for free, as always
- Carefully optimized libraries
  - Multi-threaded (for data parallelism)
  - Highly tuned (to utilize machine vector instructions)
- Language-level constructs
  - Programmer identifies parallel loops
  - Programmer identifies parallel tasks
  - Programmer identifies GPU-bound statements

Parallelism in MATLAB

- ILP for free, as always
- Carefully optimized libraries
  - Multi-threaded (for data parallelism)
  - Highly tuned (to utilize machine vector instructions)
- Language-level constructs
  - Programmer identifies parallel loops
  - Programmer identifies parallel tasks
  - Programmer identifies GPU-bound statements

Reliance on programmers untenable

Parallelism in MATLAB

- ILP for free, as always
- Carefully optimized libraries
  - Multi-threaded (for data parallelism)
  - Highly tuned (to utilize machine vector instructions)
- Language-level constructs
  - Programmer identifies parallel loops
  - Programmer identifies parallel tasks
  - Programmer identifies GPU-bound statements

Wish to automate

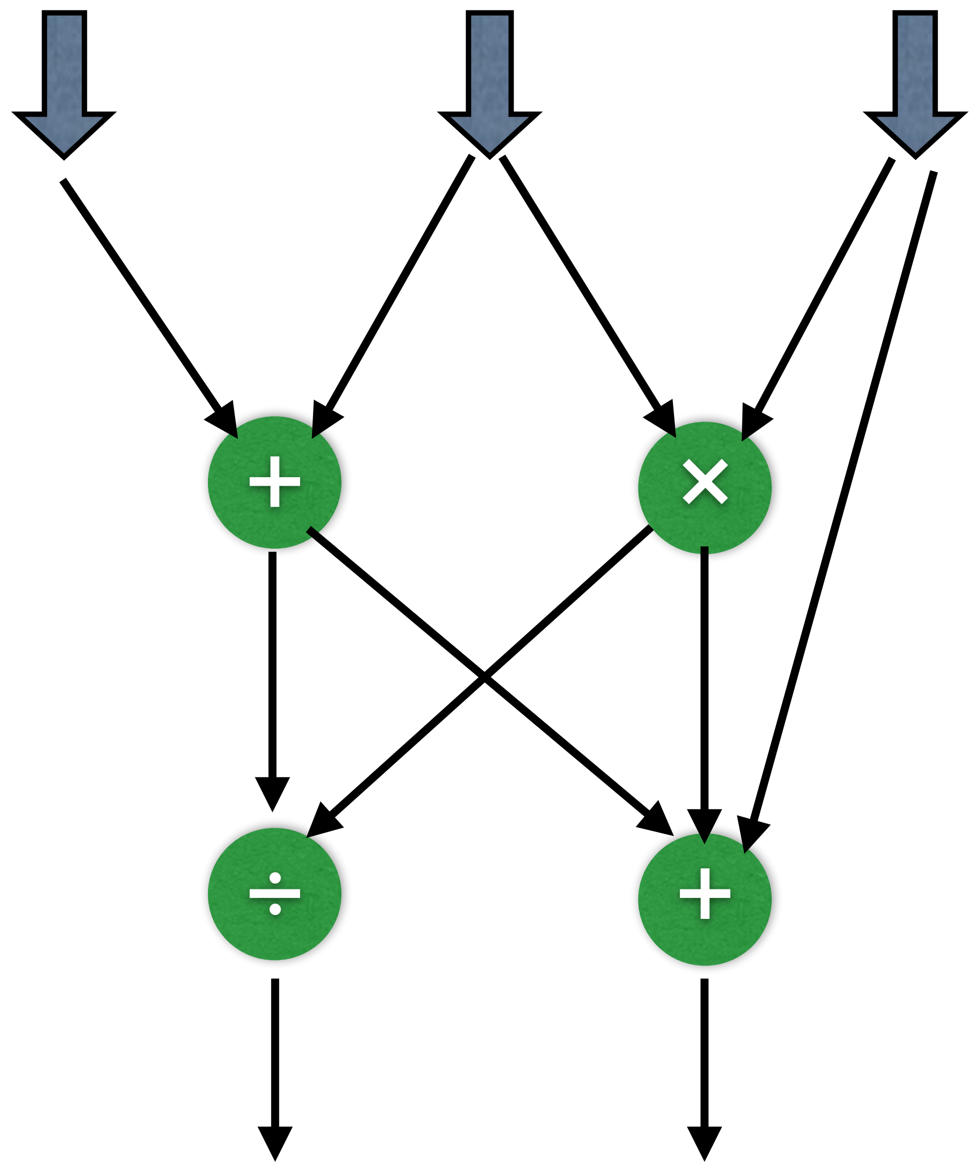
Reliance on programmers untenable

What is the Right Model of Parallelism?

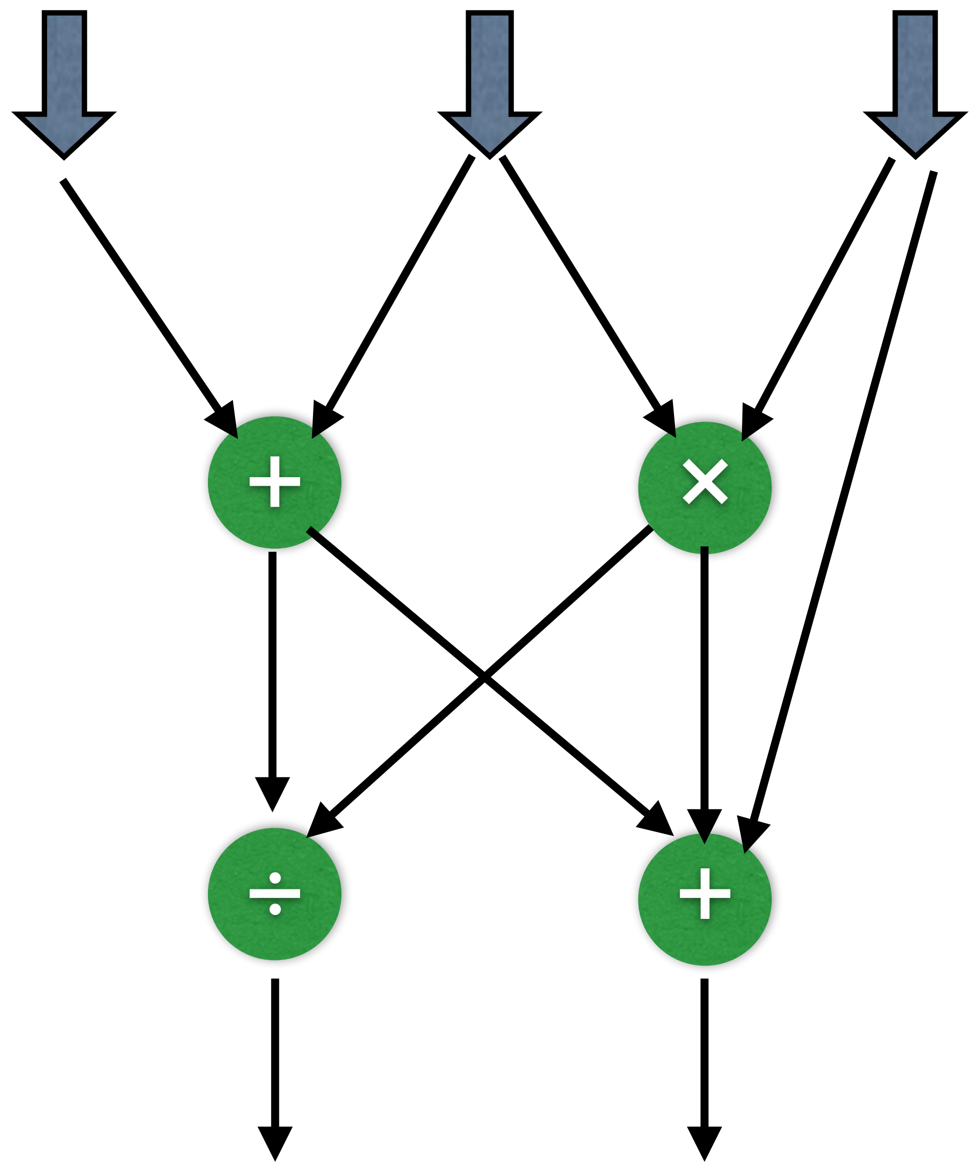
- One that does not require programmers to write parallel code at all!
- But, at the system level:
  - Need to exploit parallelism at all levels of hardware and software
  - Need to match the parallelism in the application to the underlying hardware

Data-flow Model of Computing

Data-flow computing …

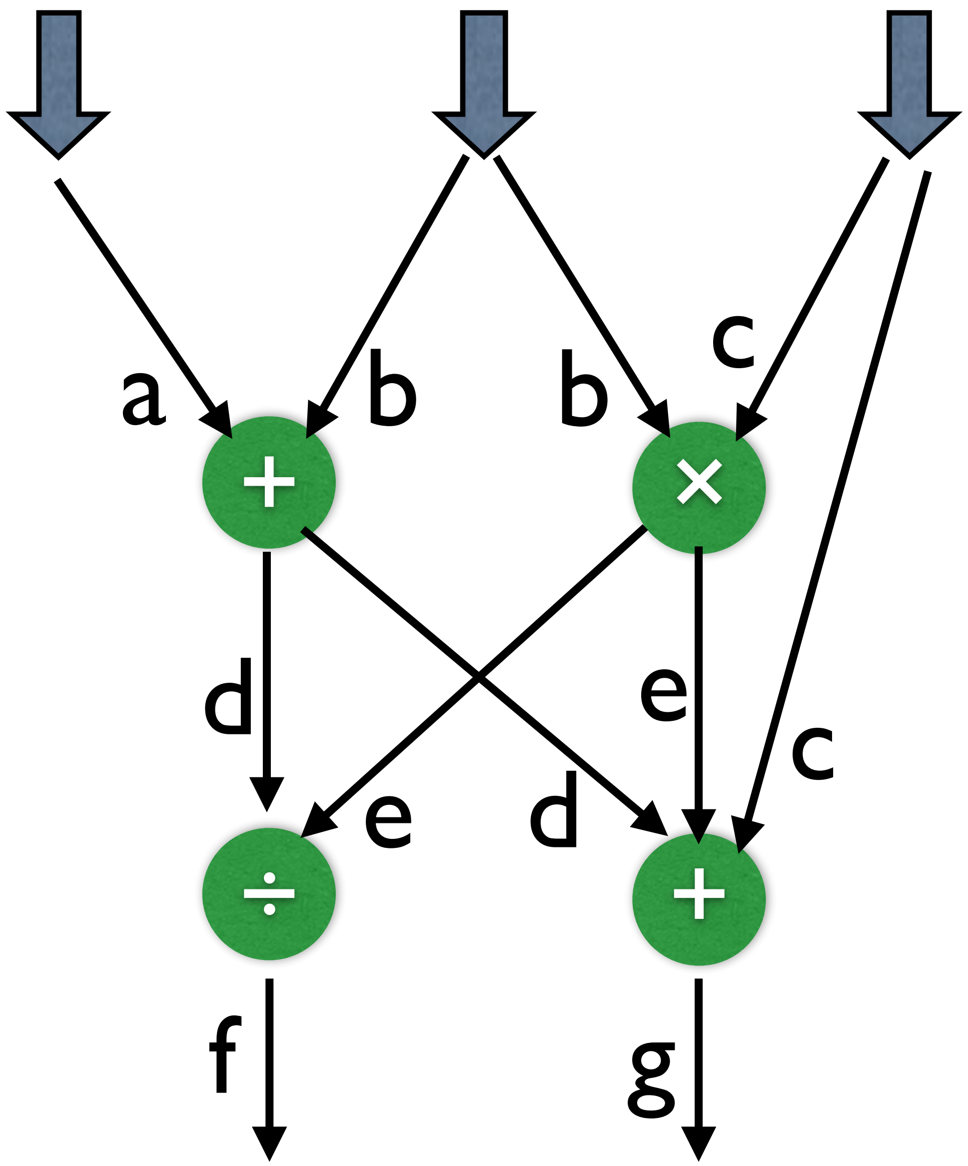


Data-flow computing …



Already exists in hardware

Data-flow computing …



Already exists in hardware

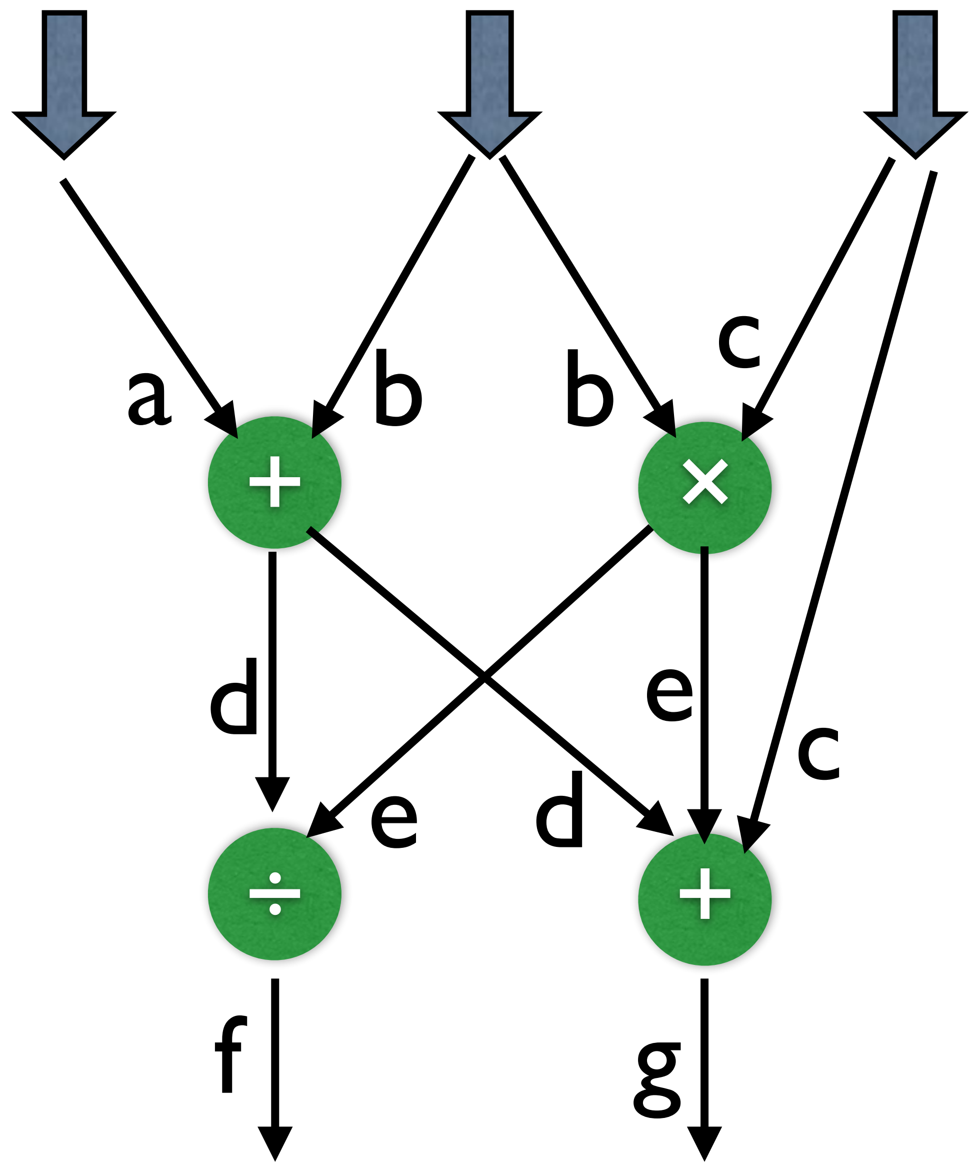
Data-flow computing ...

$d \leftarrow a + b$

$e \leftarrow b \times c$

$f \leftarrow d \div e$

$g \leftarrow d + e + c$



Already exists in hardware

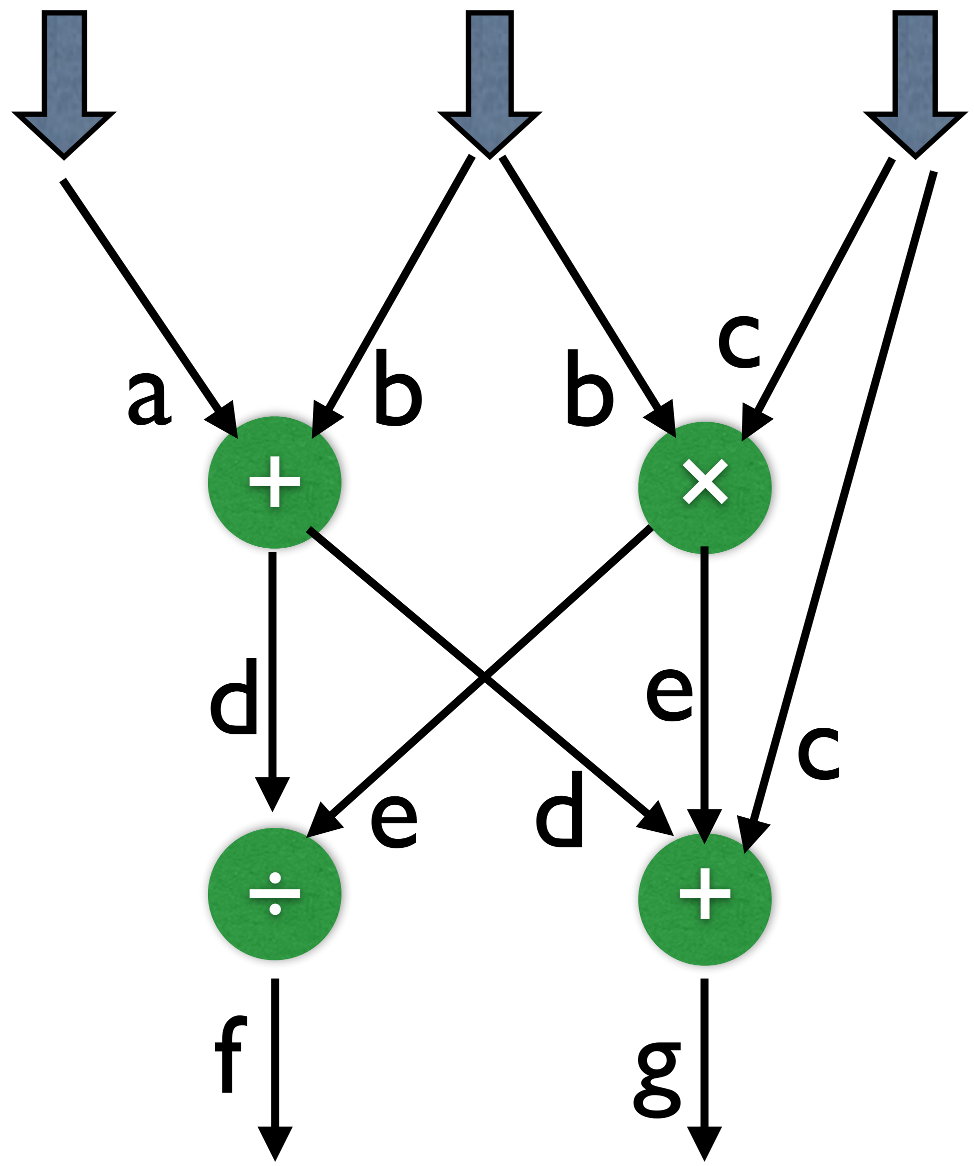
Data-flow computing …

d ← a + b

e ← b × c

f ← d ÷ e

g ← d + e + c



Already exists in hardware

Can be described procedurally

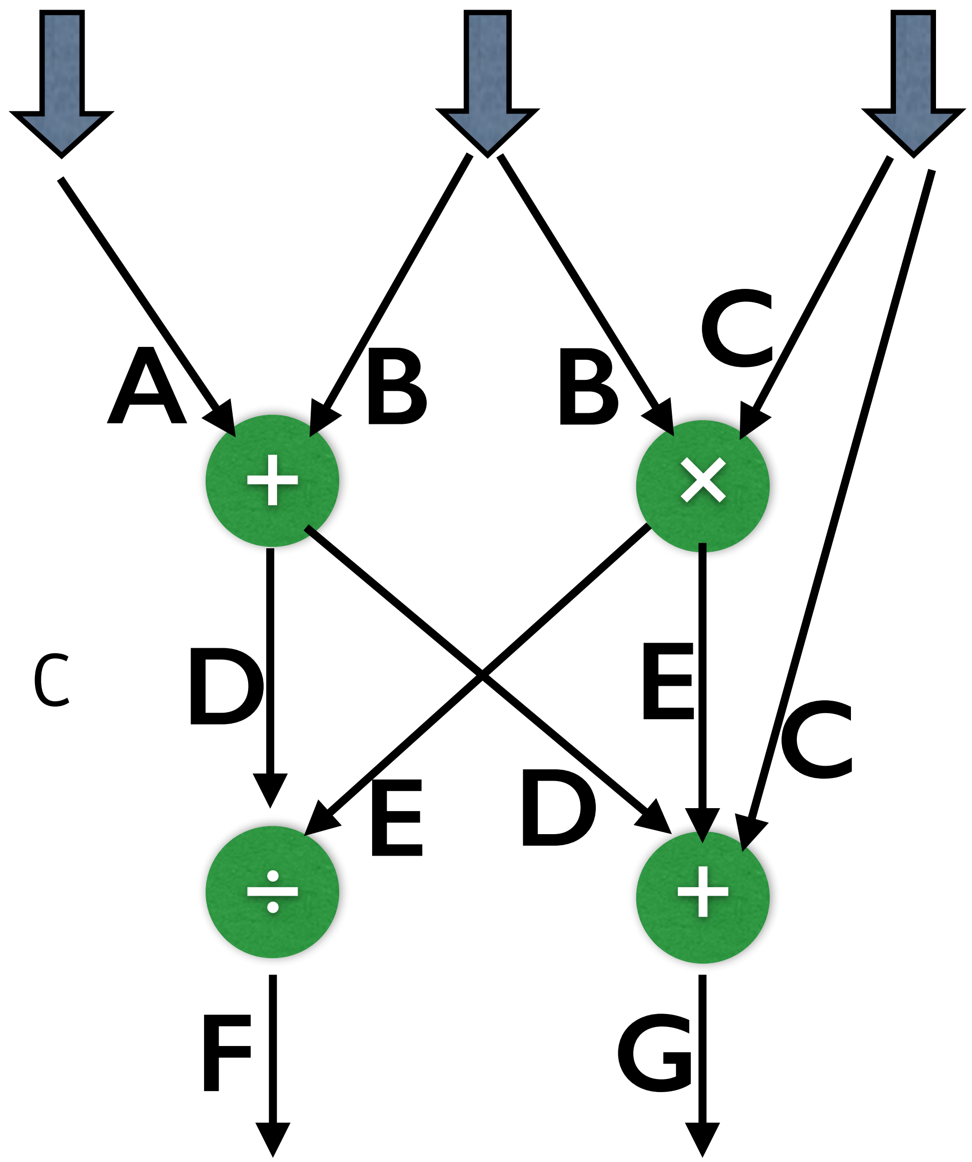
… at the Right Granularity
Macro Data-flow Computing

A ← A + B

E ← B × C

F ← D ÷ E

G ← D + E + C



- Each operation a *task* in a task-parallel library (Intel TBB)
- Low amortized creation and deletion cost
- The operation can be data-parallel (multi-threaded)
- The operation could be an optimized and parallelized library function

Data-flow Execution for MATLAB Programs

- Programmers do not need to think about it
- Great for legacy code
- Allows us to utilize the existing and already implemented modes of parallelism
- Makes use of the specialized libraries, incorporating specialized expert knowledge
- Has the potential to utilize all levels of parallelism afforded by modern hardware

Data-flow Execution for MATLAB Programs

- Programmers do not need to think about it
- Great for legacy code
- Allows us to utilize the existing and already implemented modes of parallelism
- Makes use of the specialized libraries, incorporating specialized expert knowledge
- Has the potential to utilize all levels of parallelism afforded by modern hardware

All we need is automatic extraction!

Challenges

- Right granularity for "operations"
- Memory
  - Keep the footprint in check
  - Minimize memory copies
- Programming
  - Automatically generate data-flow-style execution from procedural description
  - Respect all data- and control-dependencies
- Run-time
  - Schedule operations smartly

Approach

Approach

- Granularity
  - Treat each array statement as an atomic data-flow operation, replicate scalar operations liberally
  - Merge to coarsen the granularity without decreasing parallelism
- Memory
  - Scalars are free, arrays are mutable (hybrid data-flow / procedural)
- Programming
  - Compiler analysis to determine data and control dependencies
  - Tasks can call libraries or be implemented as explicit loops
- Run-time
  - Custom run-time around Intel Threading Building Blocks (TBB)

Utilizing Parallelism at Multiple Levels

- Across operations
  - Task parallelism (or statement-level parallelism)
- Within operations
  - Use multi-threaded library operations
  - Parallelize loops implied by array operations
- More parallelism …
  - We handle one user function at a time

Example: Array Statements

MATLAB Code

S_0 `z = rand(n,n);`

S_1 `a = v + f;`

S_2 `b = x + y;`

```
while (c)
```

S_3 `b(:,i) = a ./ pi;`

S_4 `z = b + z;`

```
end
```

S_5 `V = z';`

Example: Array Statements

MATLAB Code

S_0 `z = rand(n,n);`

S_1 `a = v + f;`

S_2 `b = x + y;`

`while (c)`

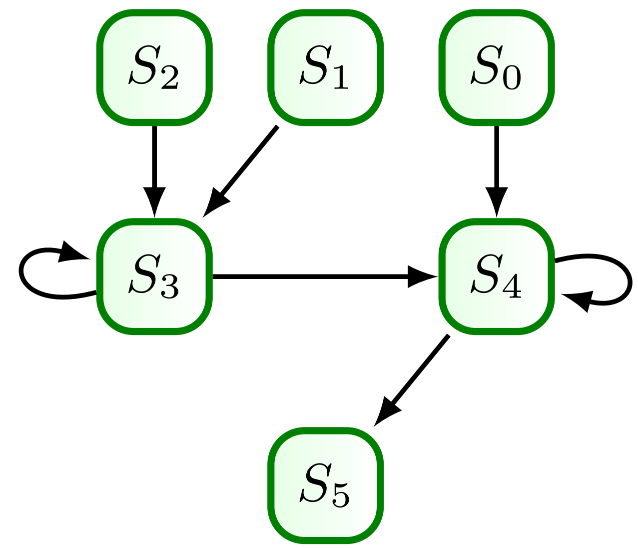
    S_3 `b(:,i) = a ./ pi;`

    S_4 `z = b + z;`

`end`

S_5 `V = z';`

Data dependencies

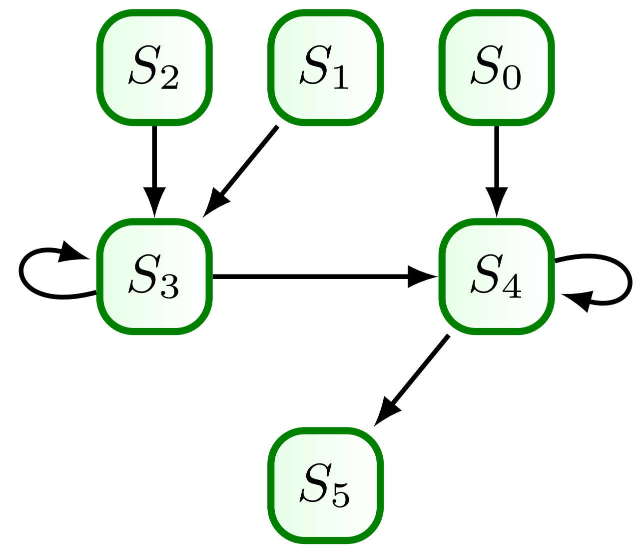


Example: Array Statements

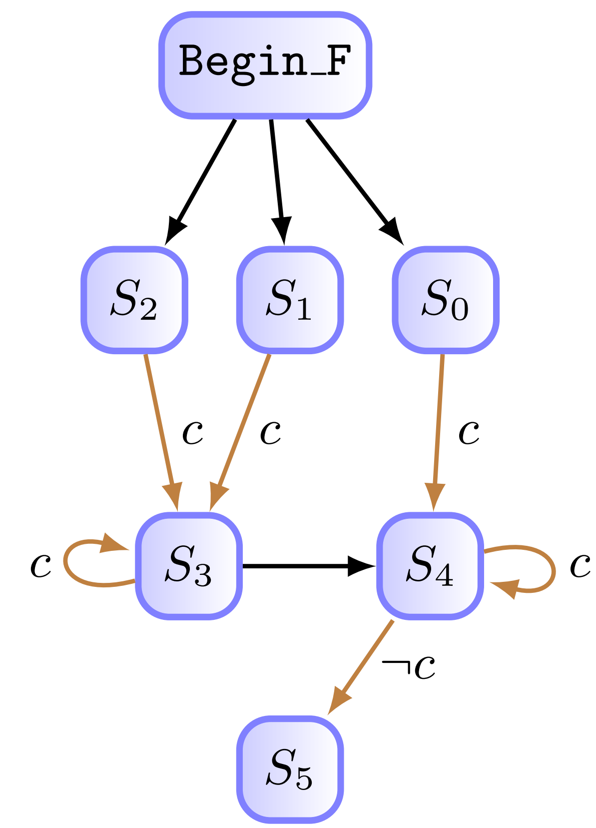
MATLAB Code

```
S0 z = rand(n,n);
S1 a = v + f;
S2 b = x + y;
while (c)
    S3 b(:,i) = a ./ pi;
    S4 z = b + z;
end
S5 V = z';
```

Data dependencies

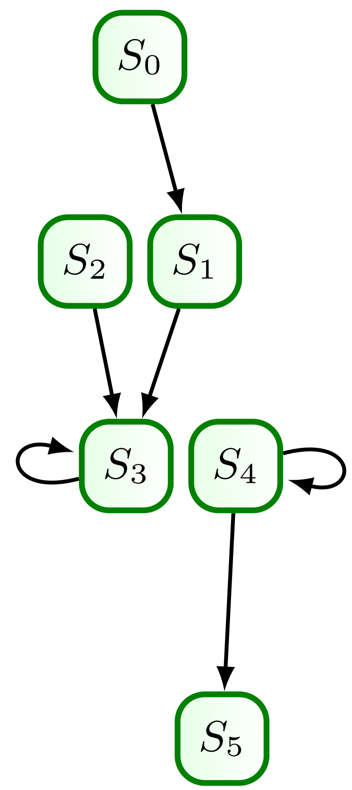


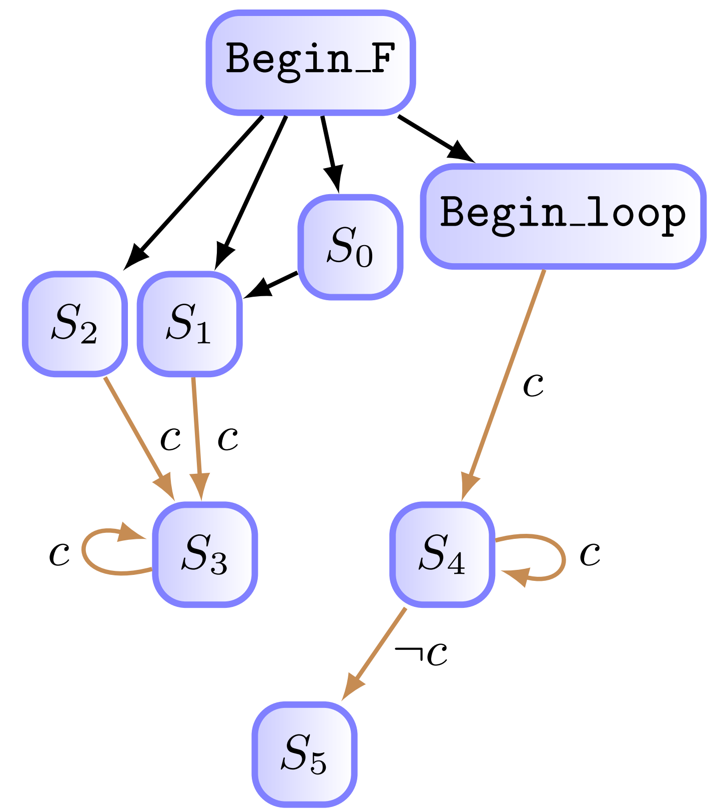
Static Data-flow graph



Example: Array Statements (Modified)

```
S0 f = rand(n,n);
S1 a = v + f;
S2 b = x + y;
while (c)
    S3 b(:,i) = a ./ pi;
    S4 z = d + z;
end
S5 V = z';
```





Example: Array and Scalar Statements

```
n = length(v);
k = 500;
H = zeros(k,k);
V = zeros(n,k);
...
...
j = 2;
tmp4  = j <= k;
while(tmp4),
 ...
 ...
 V(:,j) = v;
 H(1:j,j) = h;
 j = j + 1;
 tmp4 = j <= k;
end
```

Example: Array and Scalar Statements

```
n = length(v);
k = 500;
H = zeros(k,k);
V = zeros(n,k);
...
...
j = 2;
tmp4  = j <= k;
while(tmp4),
 ...
 ...
 V(:,j) = v;
 H(1:j,j) = h;
 j = j + 1;
 tmp4 = j <= k;
end
```

Example: Array and Scalar Statements

```
n = length(v);
k = 500;
H = zeros(k,k);
V = zeros(n,k);
...
...
j = 2;
tmp4  = j <= k;
while(tmp4),
 ...
 ...
 V(:,j) = v;
 H(1:j,j) = h;
 j = j + 1;
 tmp4 = j <= k;
end
```

```
k$1 = 500;
H$1=zeros(k$1,k$1);
j$1 = 2;
tmp4$1=j$1<=k$1;
```

```
n$1 = length(v$0);
k$1 = 500;
V$1 = zeros(n$1,k$1);
j$1 = 2;
tmp4$1=j$1<=k$1;
```

```
V$1(:,j$2) = v$1;
j$3 = j$2+1;
tmp4$3=j$3<=k$1;
```

```
H$1(1:j$2,j$2)=h$1;
j$3 = j$2+1;
tmp4$3=j$3<=k$1;
```

Example: Array and Scalar Statements

```
n = length(v);
k = 500;
H = zeros(k,k);
V = zeros(n,k);
...
...
j = 2;
tmp4  = j <= k;
while(tmp4),
 ...
 ...
 V(:,j) = v;
 H(1:j,j) = h;
 j = j + 1;
 tmp4 = j <= k;
end
```

```
k$1 = 500;
H$1=zeros(k$1,k$1);
j$1 = 2;
tmp4$1=j$1<=k$1;
```

```
n$1 = length(v$0);
k$1 = 500;
V$1 = zeros(n$1,k$1);
j$1 = 2;
tmp4$1=j$1<=k$1;
```

```
V$1(:,j$2) = v$1;
j$3 = j$2+1;
tmp4$3=j$3<=k$1;
```

```
H$1(1:j$2,j$2)=h$1;
j$3 = j$2+1;
tmp4$3=j$3<=k$1;
```

Example: Array and Scalar Statements

```
n = length(v);
k = 500;
H = zeros(k,k);
V = zeros(n,k);
...
...
j = 2;
tmp4  = j <= k;
while(tmp4),
 ...
 ...
 V(:,j) = v;
 H(1:j,j) = h;
 j = j + 1;
 tmp4 = j <= k;
end
```

1

```
k$1 = 500;
H$1=zeros(k$1,k$1);
j$1 = 2;
tmp4$1=j$1<=k$1;
```

2

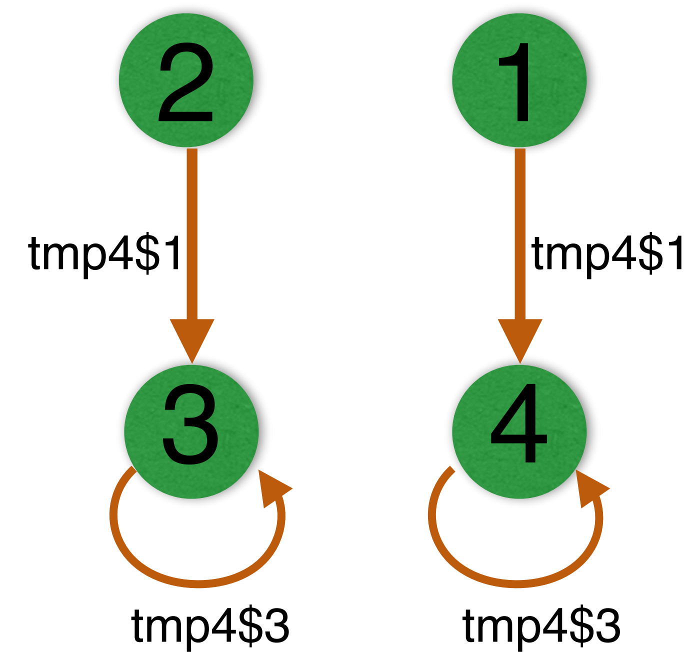
```
n$1 = length(v$0);
k$1 = 500;
V$1 = zeros(n$1,k$1);
j$1 = 2;
tmp4$1=j$1<=k$1;
```

3

```
V$1(:,j$2) = v$1;
j$3 = j$2+1;
tmp4$3=j$3<=k$1;
```

4

```
H$1(1:j$2,j$2)=h$1;
j$3 = j$2+1;
tmp4$3=j$3<=k$1;
```

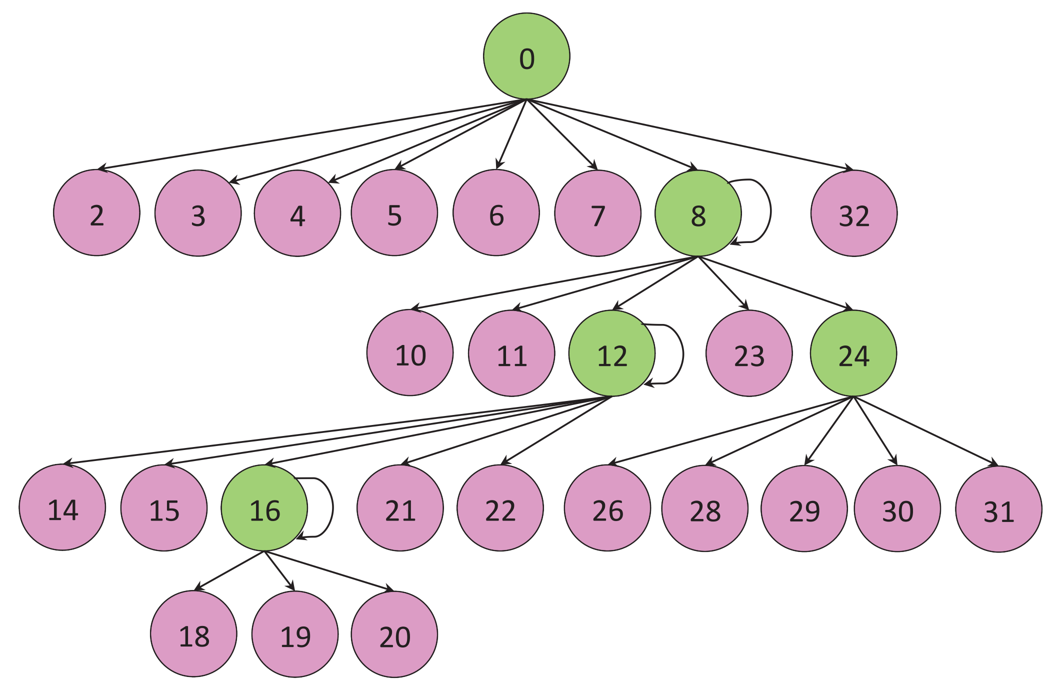


Static Data-flow Graph

Example: Accounting for Control Flow

Without any extra controller tasks

```
2  Fx$1 = zeros(n$0, a$0);
3  drx$1 = zeros(n$0, n$0);
4  x$1 = Fx$1(:, n$0);
5  G$1 = 1e-11;
6  t$1 = 1;
7  tmp1$1 = t$1<=T$0;
8  while(tmp1$2)
10   k$2 = 1;
11   tmp2$2 = k$2 <= n$0;
12   while(tmp2$3)
14     j$3 = 1;
15     tmp3$3 = j$3 <= n$0;
16     while(tmp3$4)
18       Fx$5(:,k$3) = G$1;
19       j$5 = j$4 + 1;
20       tmp3$5 = j$5<=n$0;
         end
21     k$4 = k$3 + 1;
22     tmp2$4 = k$4 <= n$0;
     end
23   tmp4$2 = t$2 == 2;
24   if(tmp4$2);
26     continue;
     end
28   Fx$6(:, t) = G$1 * drx$1;
29   f$1 = Fx$6(:, k$3);
30   t$3 = t$2 + dT$0;
31   tmp1$3 = t$3 <= T$0;
   end
```



Control dependence graph

Computing the Edge Conditions

Algorithm: ComputeDepConditions

Input: CDG G, Source src, Destination dst, CFG cfg

Output: Predicate Expression L

$S \leftarrow \{c_1, ..., c_k, s_1, ...s_k\}$ `/* seq. of all cond. exprs enclosing` *src* `*/`

$D \leftarrow \{c_1, ..., c_k, d_1, ...d_k\}$ `/* seq. of all cond. exprs enclosing` *dst* `*/`

$L \leftarrow \neg(s_1 \wedge, ..., \wedge s_k) \wedge (c_1 \wedge, ..., \wedge c_k)$

for *each* n *in* $\{c_1, ...c_k\}$ **do**

  if $(c \leftarrow ClearPath(src, n, dst, cfg))$ **then**

    $L \leftarrow L \wedge c$

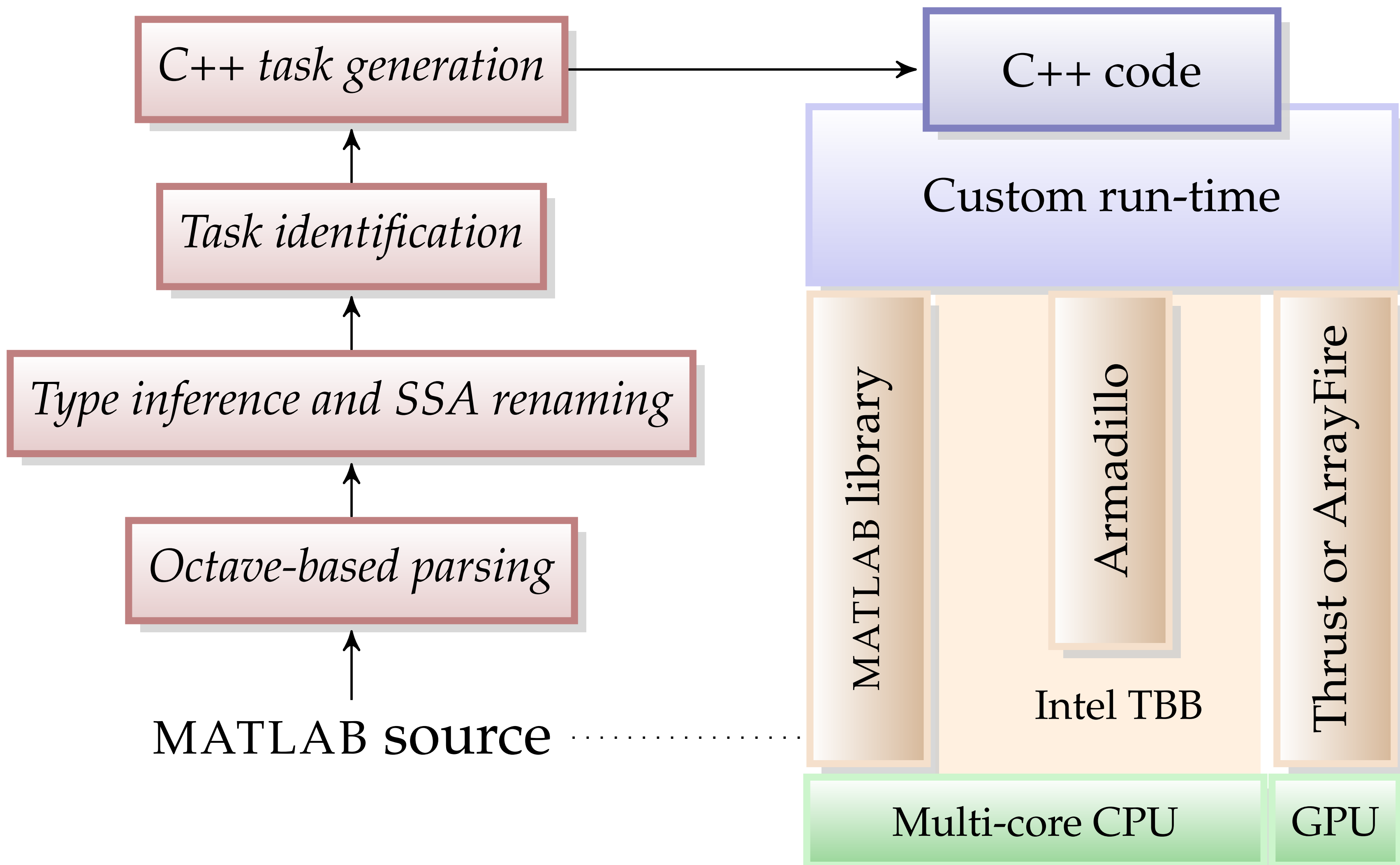
  else

    break;

Some Implementation Details

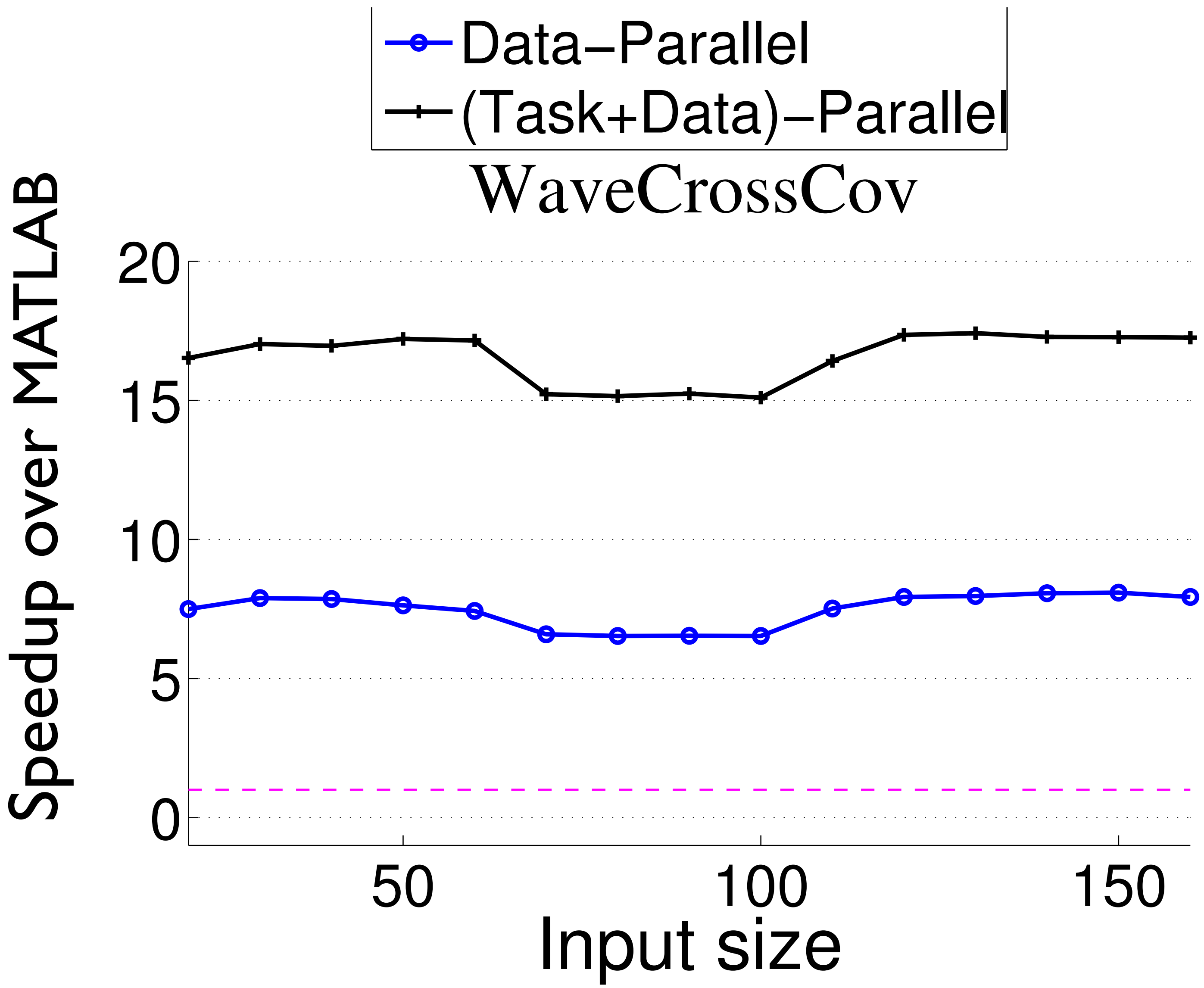
- Intel Threading Building Blocks (TBB) for tasks
- Task types
  - Subclass tbb::task
  - A type for each operation
- Concurrent hash-map for waiting tasks
  - Created, but waiting for input
  - Removed as soon as start running
- Atomic counters to track ready inputs
- Armadillo matrix library
  - Readable C++ syntax
  - Efficient implementation with expression templates

Overall System



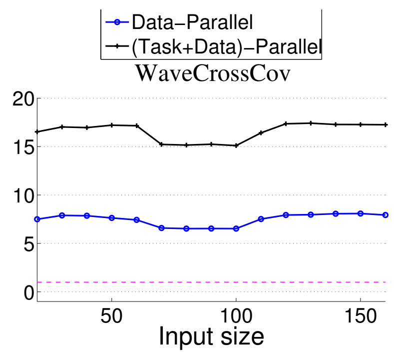
Empirical Evaluation

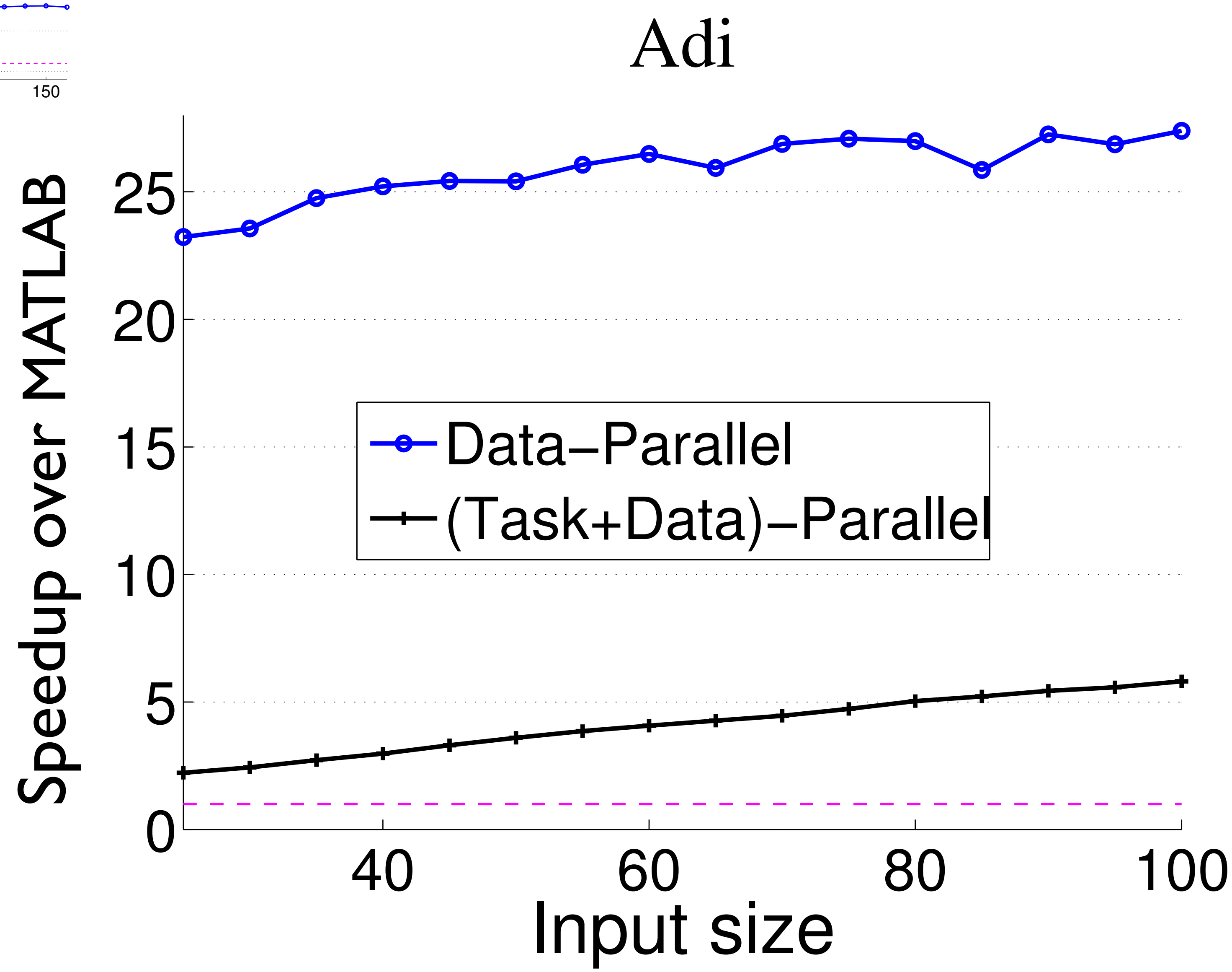
Speedups



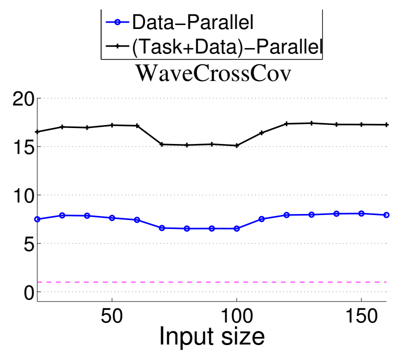
Data−Parallel
(Task+Data)−Parallel
WaveCrossCov
Speedup over MATLAB
20
15
10
5
0
50
100
150
Input size

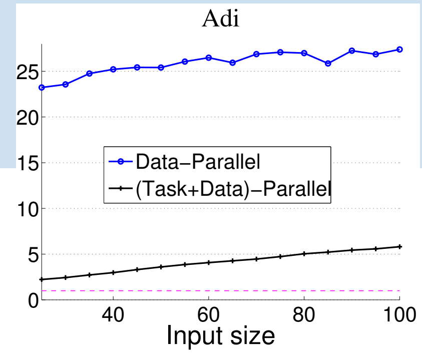
Speedups

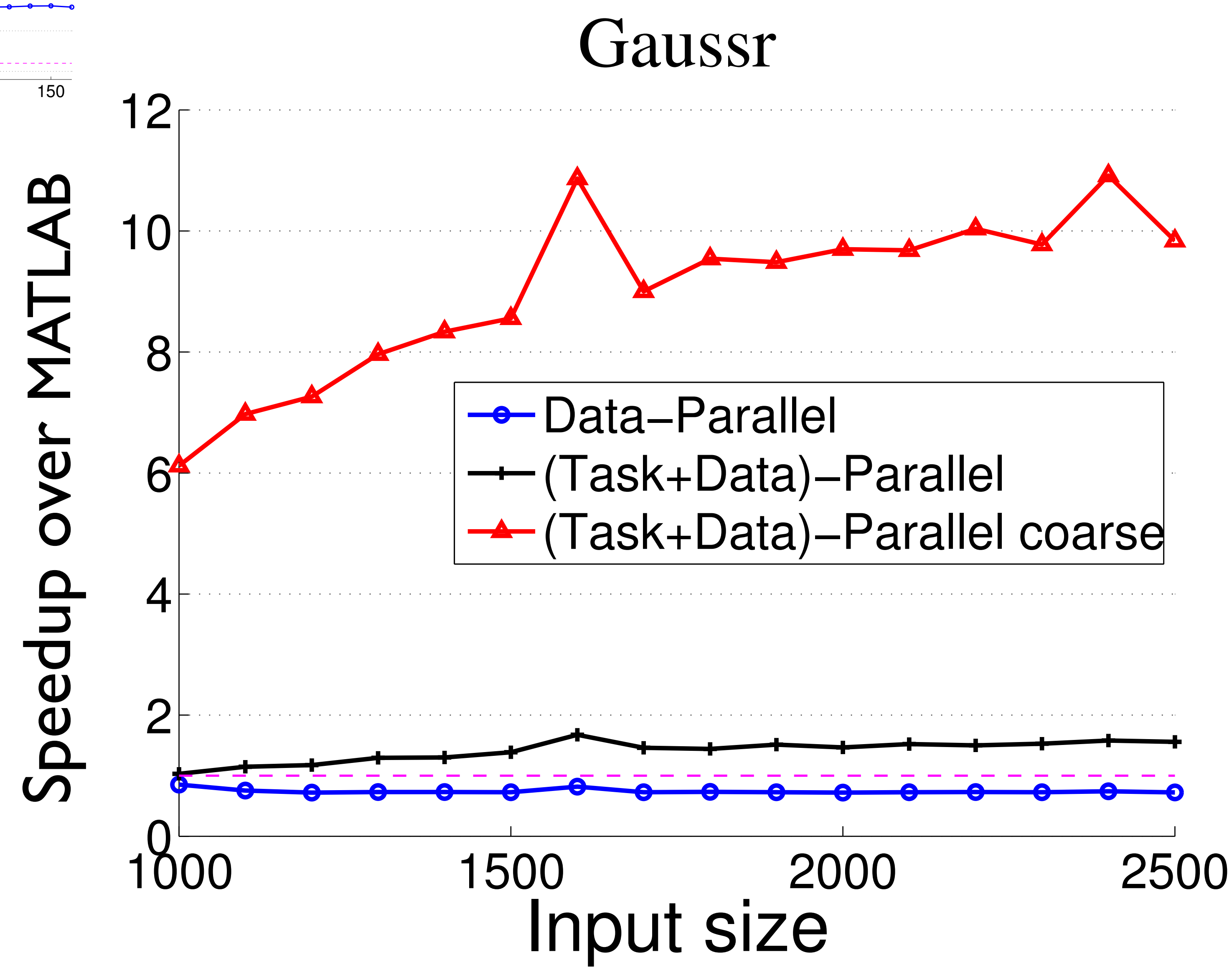

Data−Parallel
(Task+Data)−Parallel
WaveCrossCov
20
15
10
5
0
50
100
150
Input size


Adi
Speedup over MATLAB
25
20
15
10
5
0
Data−Parallel
(Task+Data)−Parallel
40
60
80
100
Input size

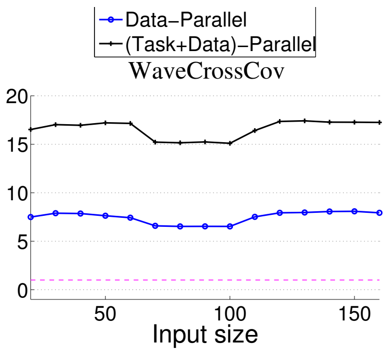
Speedups

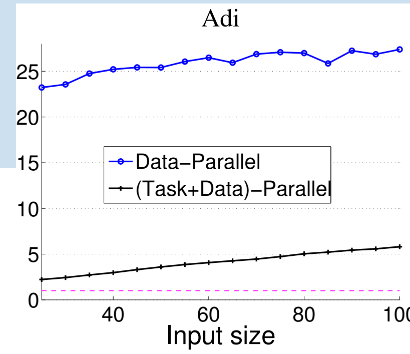

Data−Parallel
(Task+Data)−Parallel
WaveCrossCov
20
15
10
5
0
50
100
150
Input size

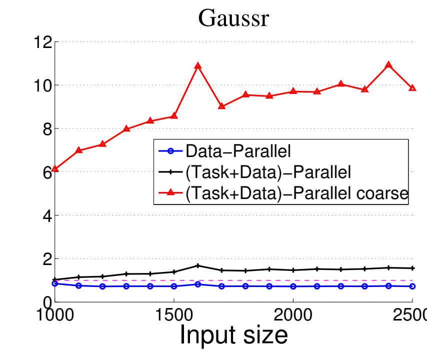

Adi
25
20
15
10
5
0
Data−Parallel
(Task+Data)−Parallel
40
60
80
100
Input size

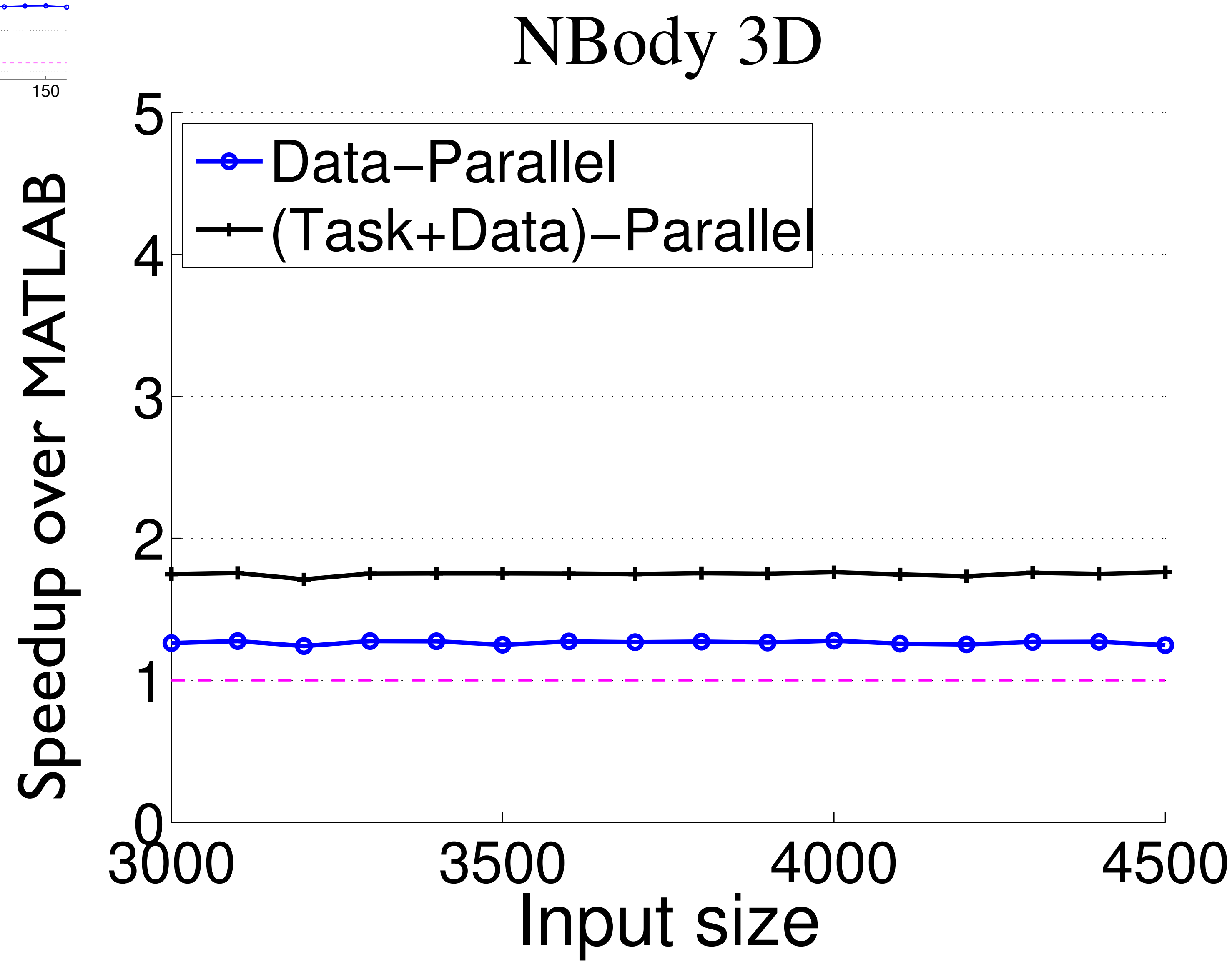

Gaussr
Speedup over MATLAB
12
10
8
6
4
2
0
Data−Parallel
(Task+Data)−Parallel
(Task+Data)−Parallel coarse
1000
1500
2000
2500
Input size

Speedups


Data–Parallel
(Task+Data)–Parallel
WaveCrossCov
Input size


Adi
Data–Parallel
(Task+Data)–Parallel
Input size

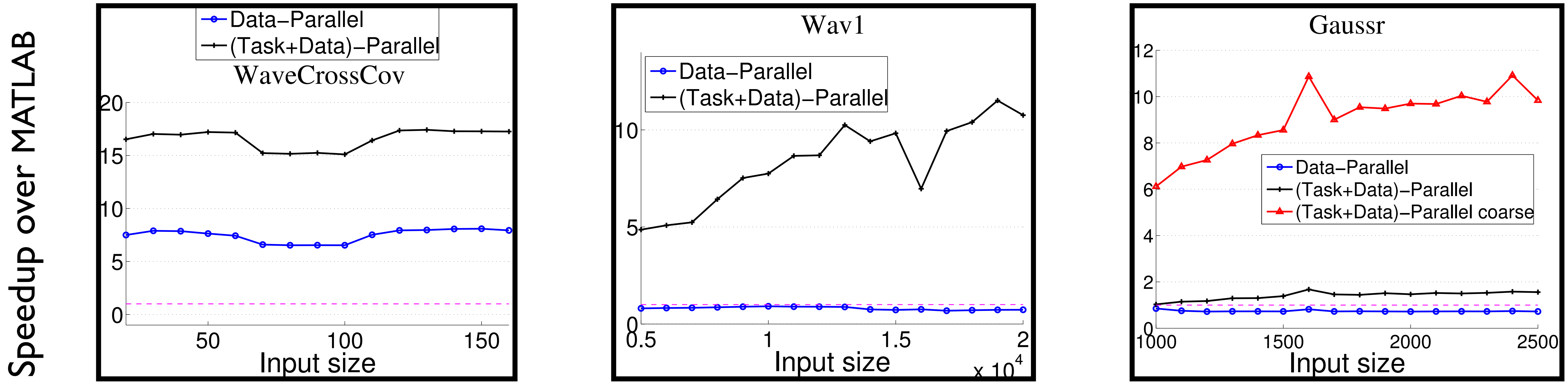

Gaussr
Data–Parallel
(Task+Data)–Parallel
(Task+Data)–Parallel coarse
Input size


NBody 3D
Data–Parallel
(Task+Data)–Parallel
Speedup over MATLAB
Input size

Experimental Setup

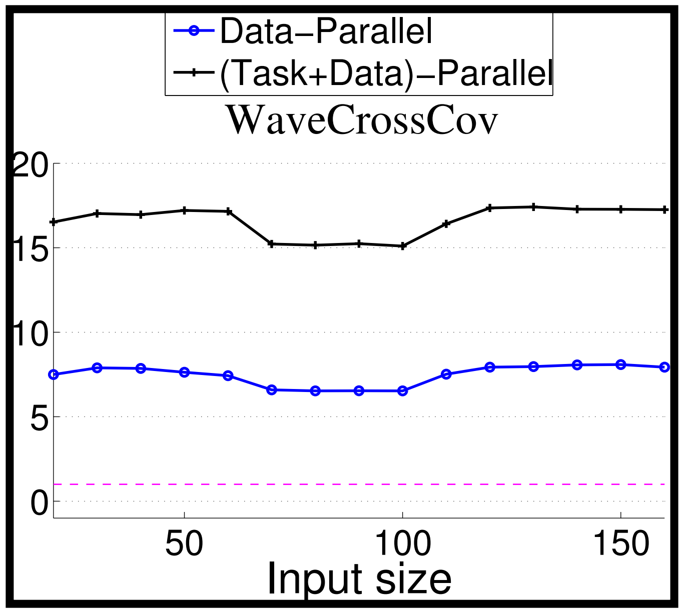
- Dual 16-core AMD Opteron 6380
  - 2.5 GHz, 64 GB DDR3 memory, 16 MB L3 cache
- Cray Linux Environment 4.1.UP01
- GCC 4.8.1
- Armadillo C++ library version 4.000
- Intel MKL 11.0
- Median of 10 runs
- Studied several benchmarks, only some reported
  - Studied code with large proportion of array operations

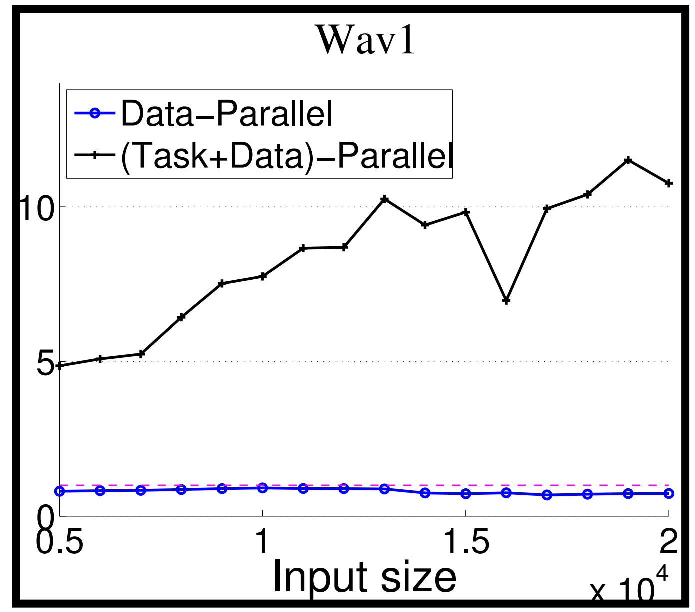
Performance and Concurrency

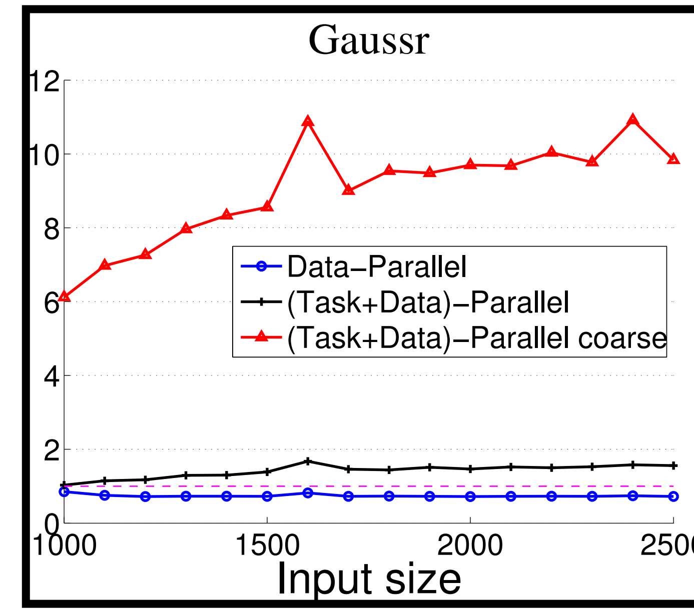


Performance and Concurrency

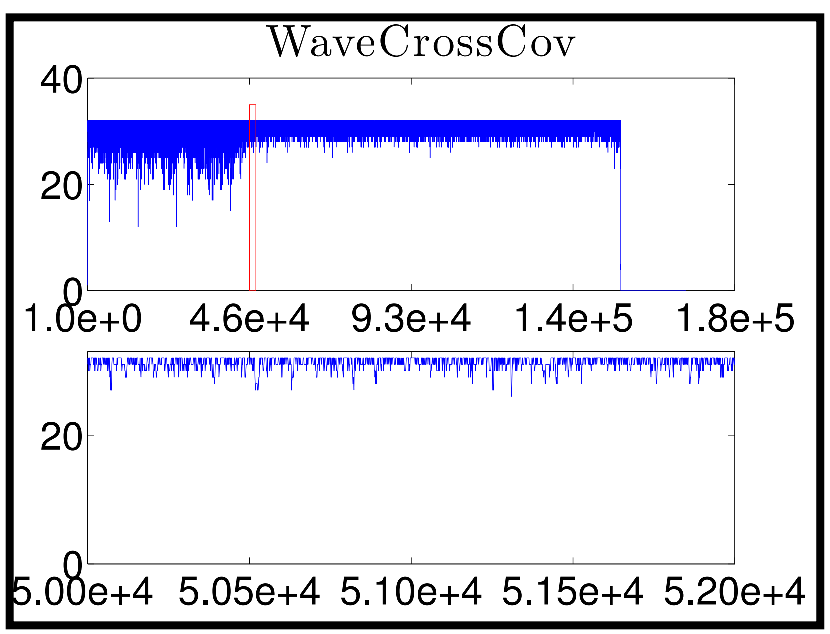
Speedup over MATLAB

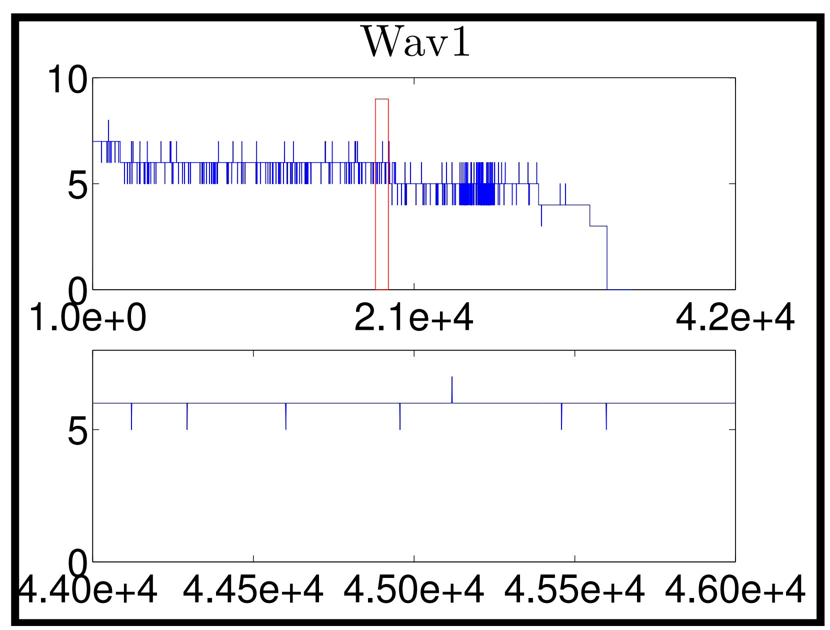


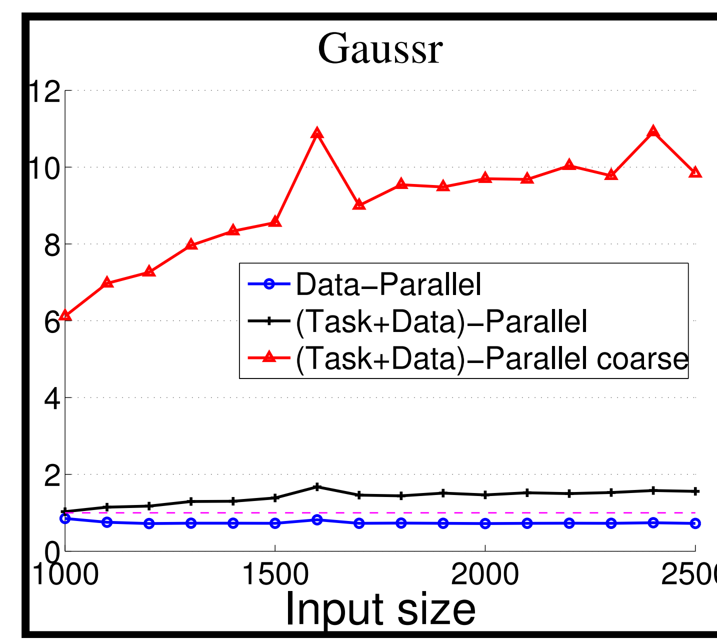




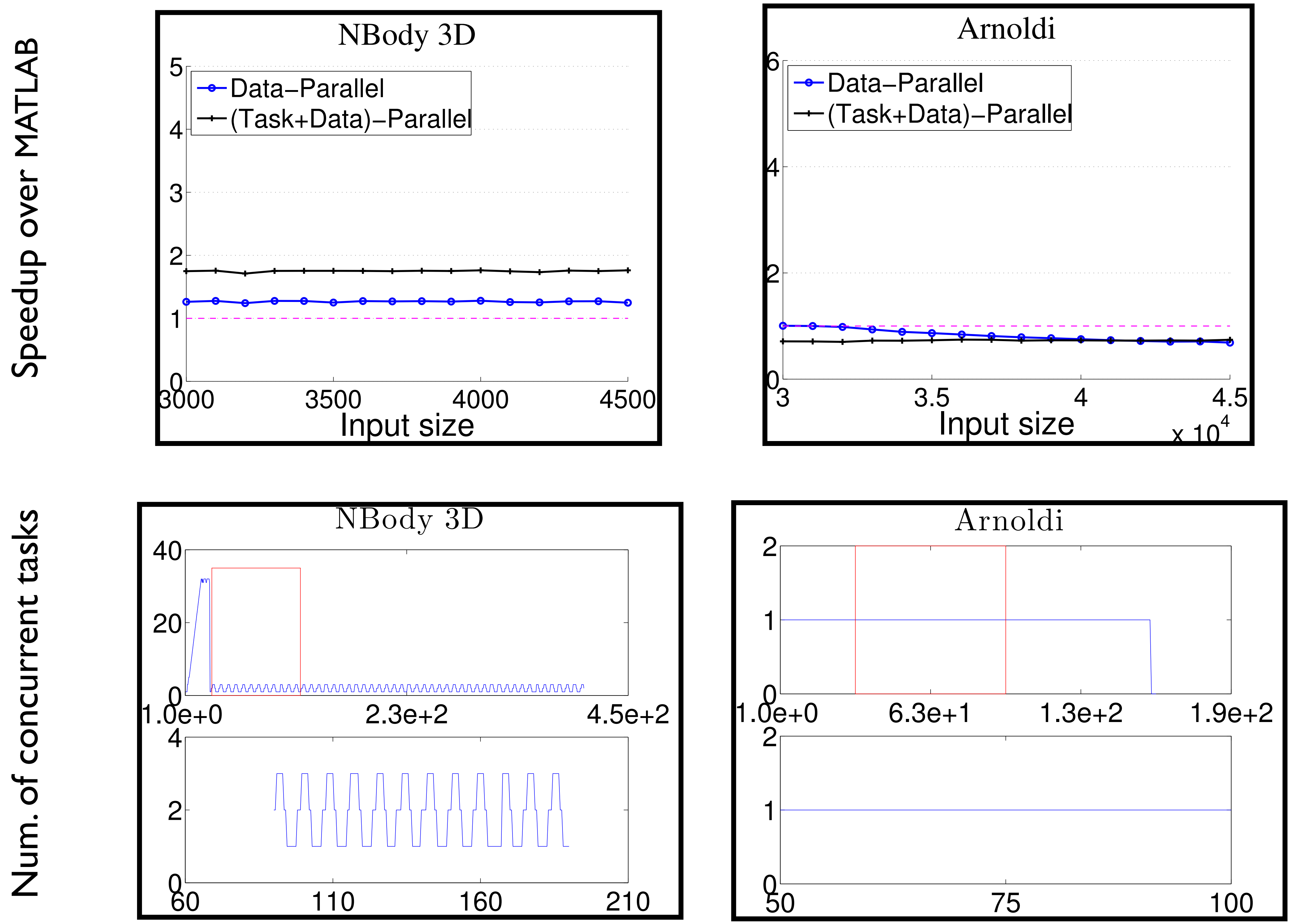
Num. of concurrent tasks



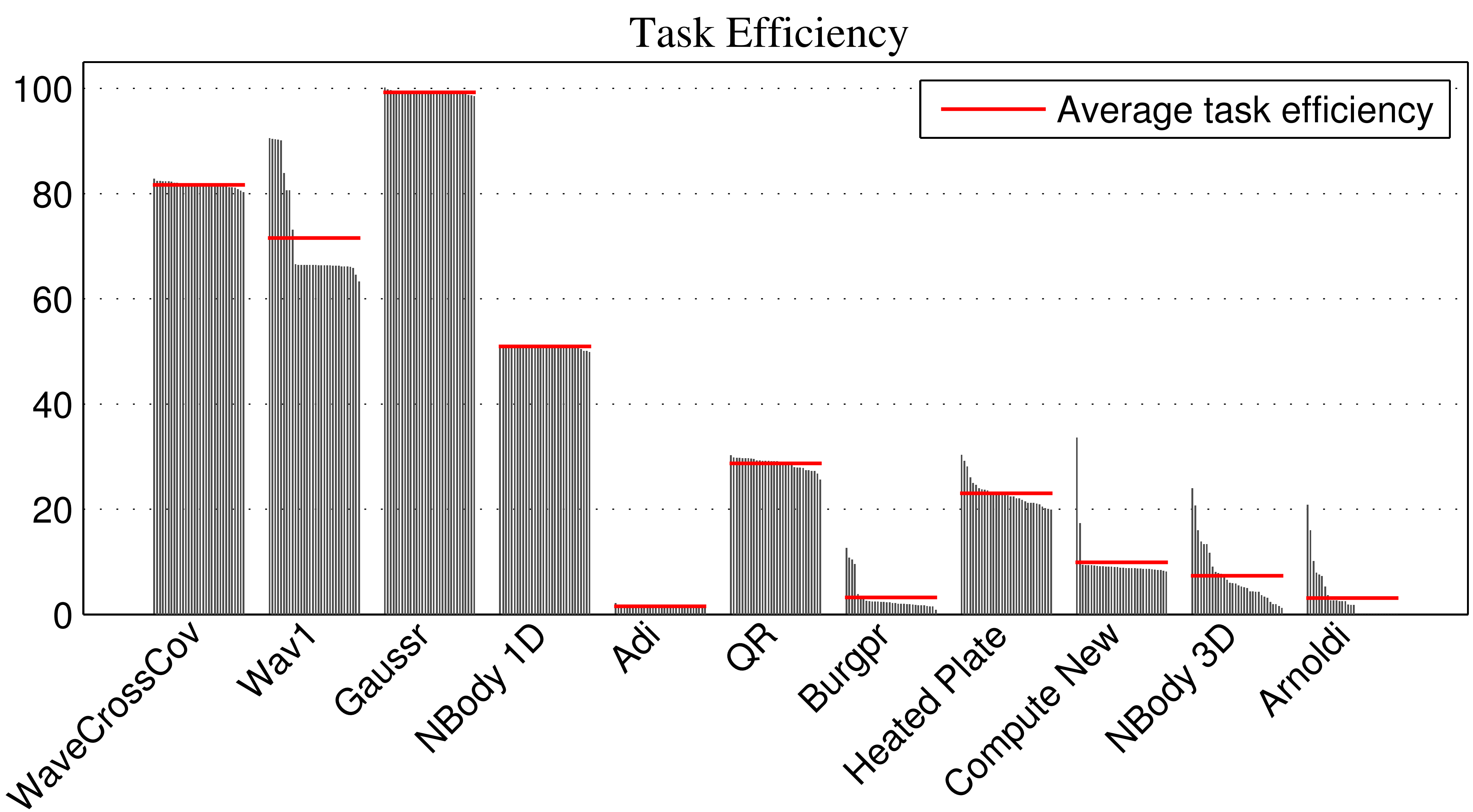




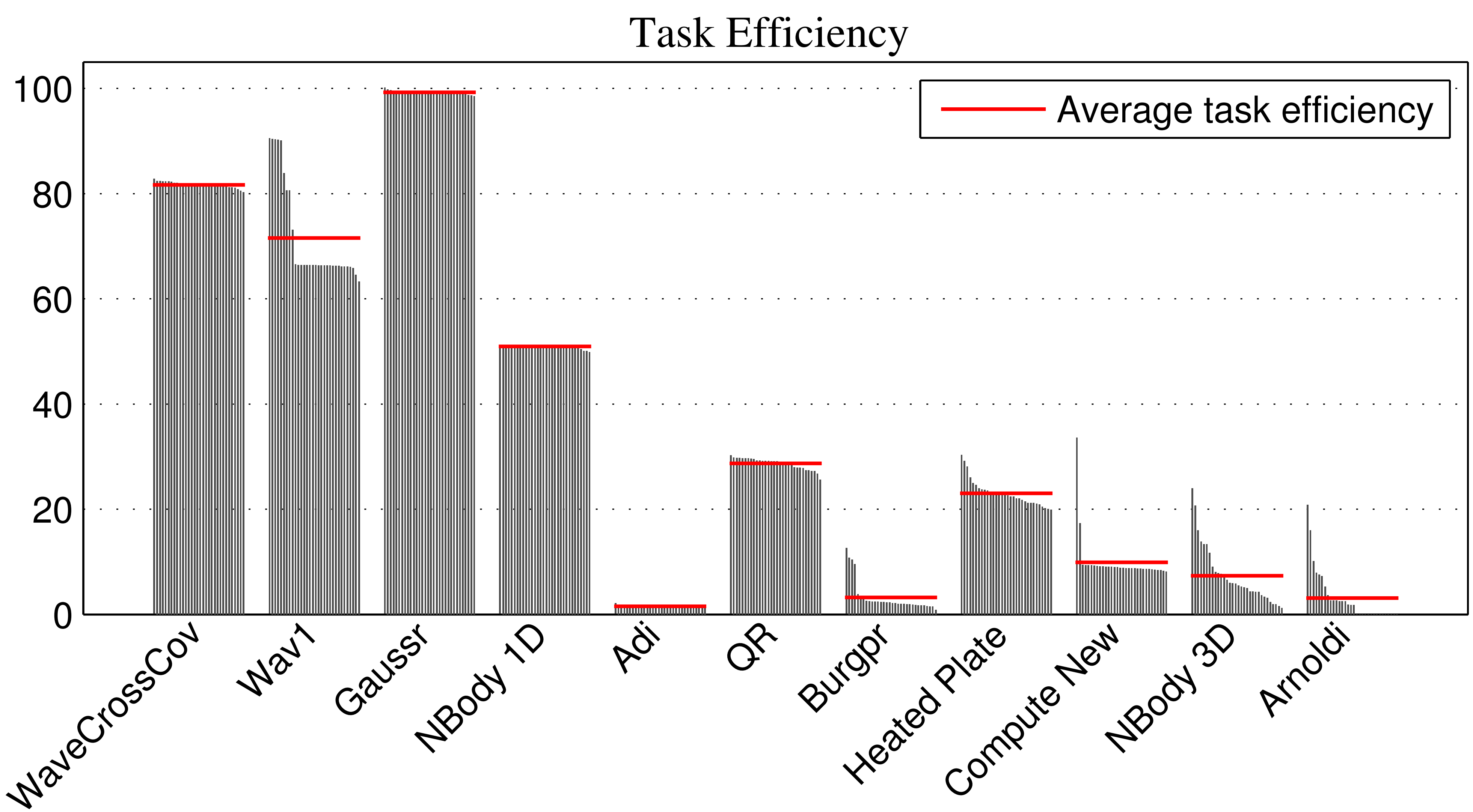
Performance and Concurrency (contd.)



Task Efficiency on 16 Cores



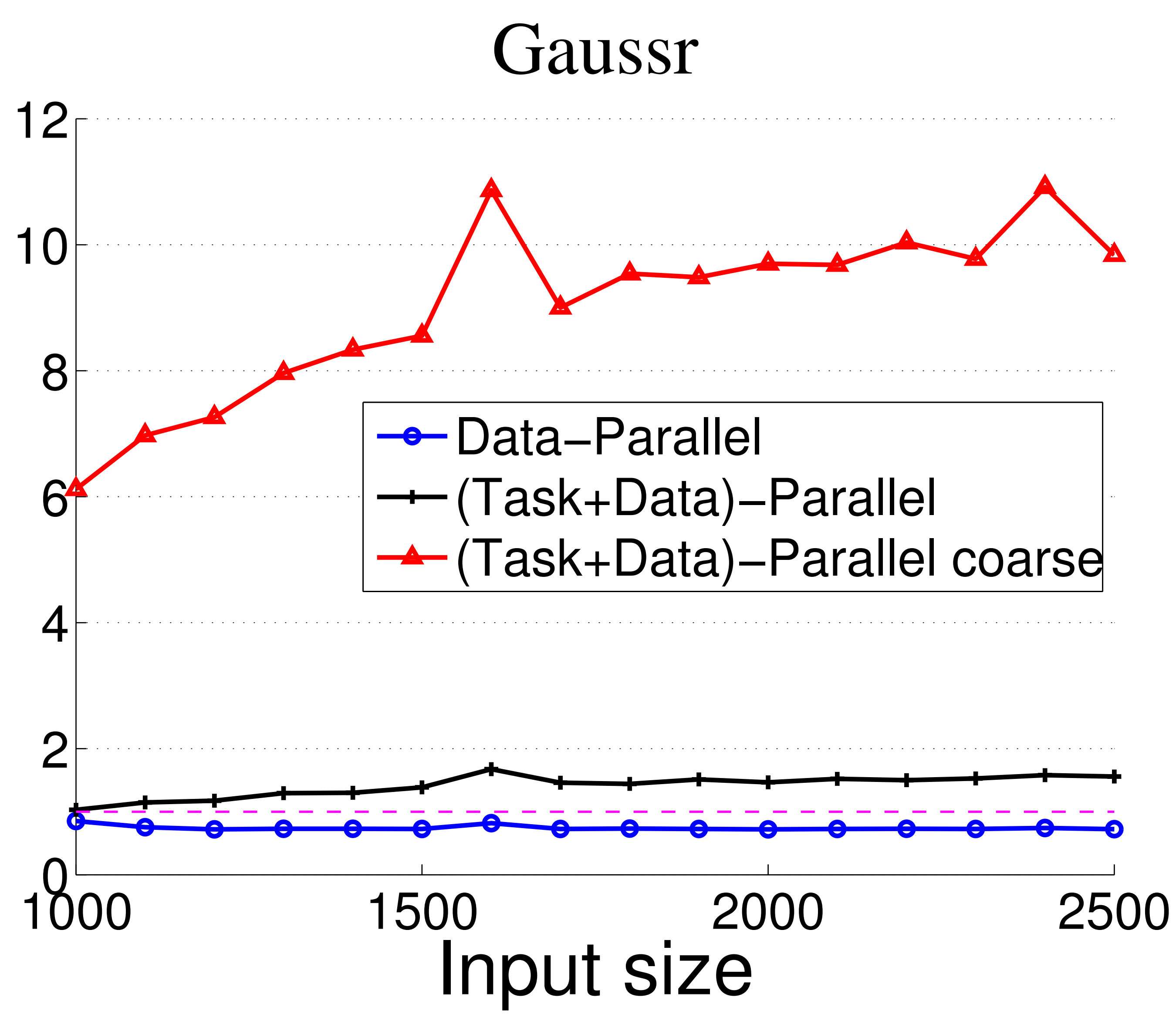
Task Efficiency on 16 Cores



Can we use this information?

Granularity Adjustment

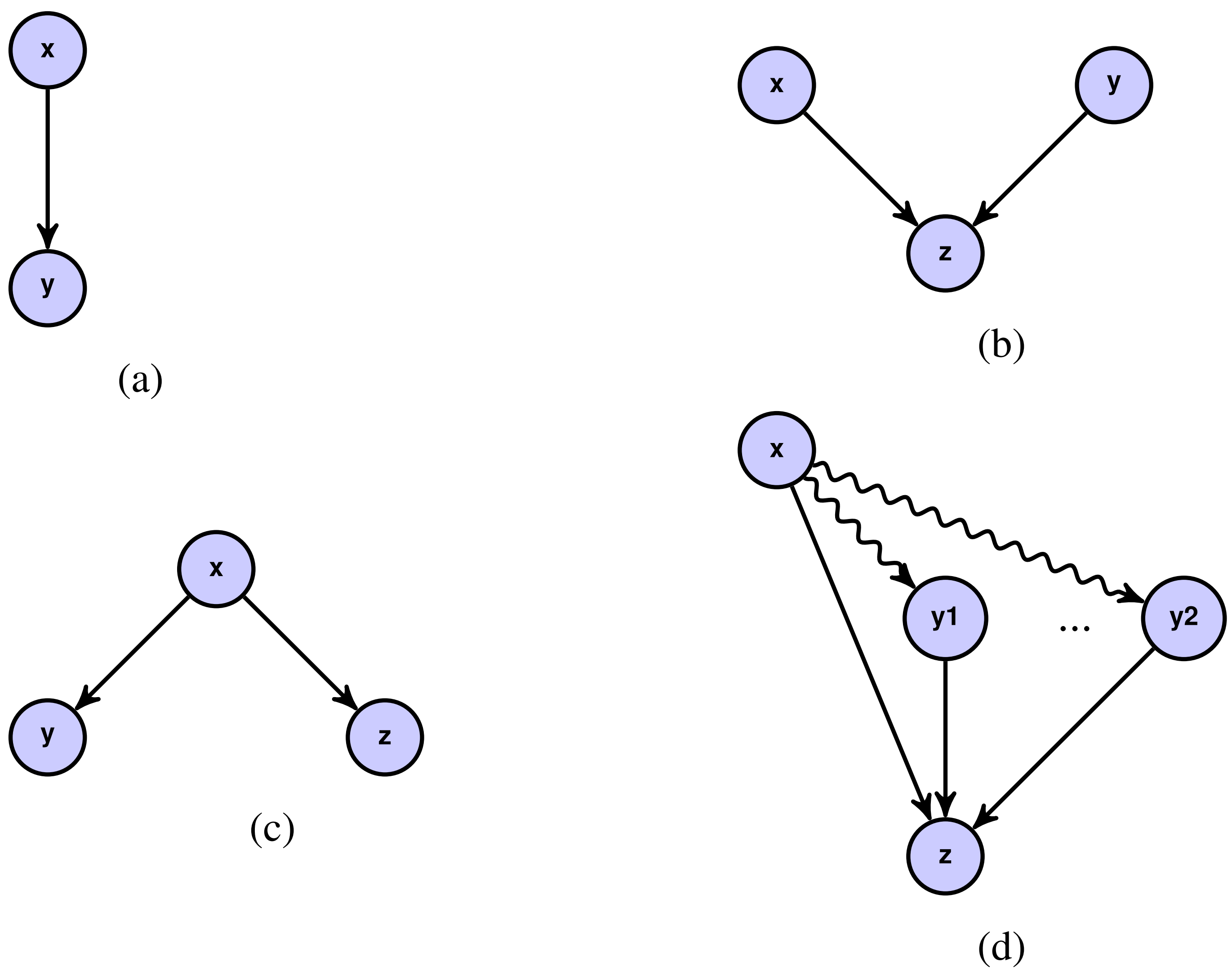
Task Granularity can have Dramatic Impact



Challenges and Opportunities

- Problems
  - Cost model for when and how much to coarsen
  - Challenging to estimate the gains
  - Should not sacrifice parallelism (not too much)
- Potential gains
  - Reduced task creation and deletion overhead
  - Improved data locality
    - Also possible to fuse loops and scalarize array temporaries

Cases to Consider

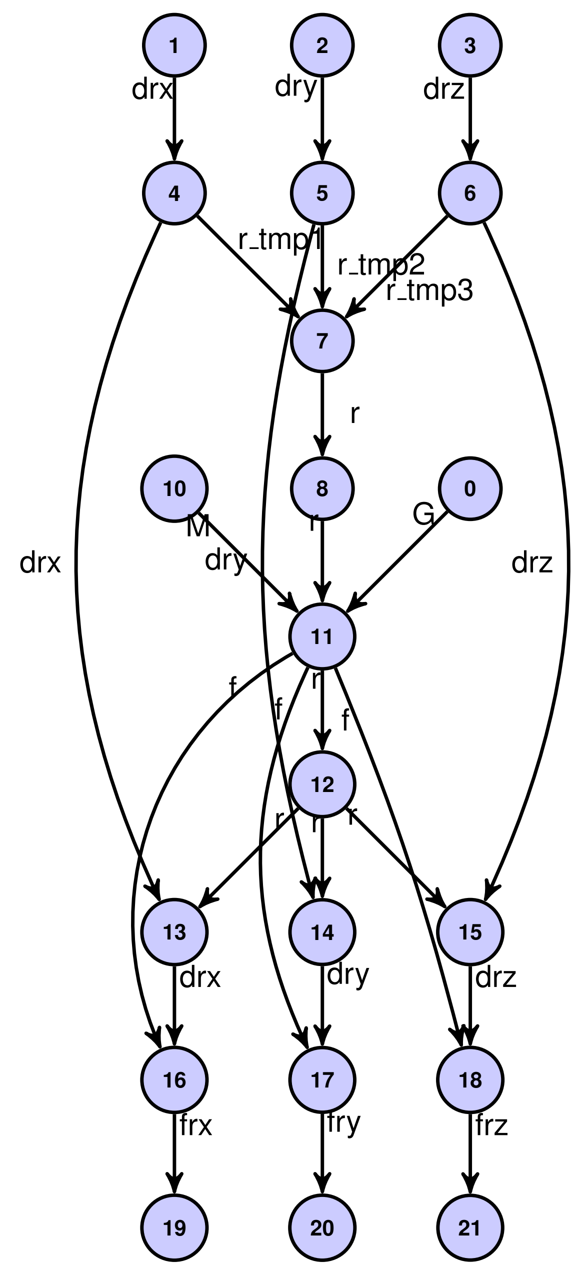


Properties

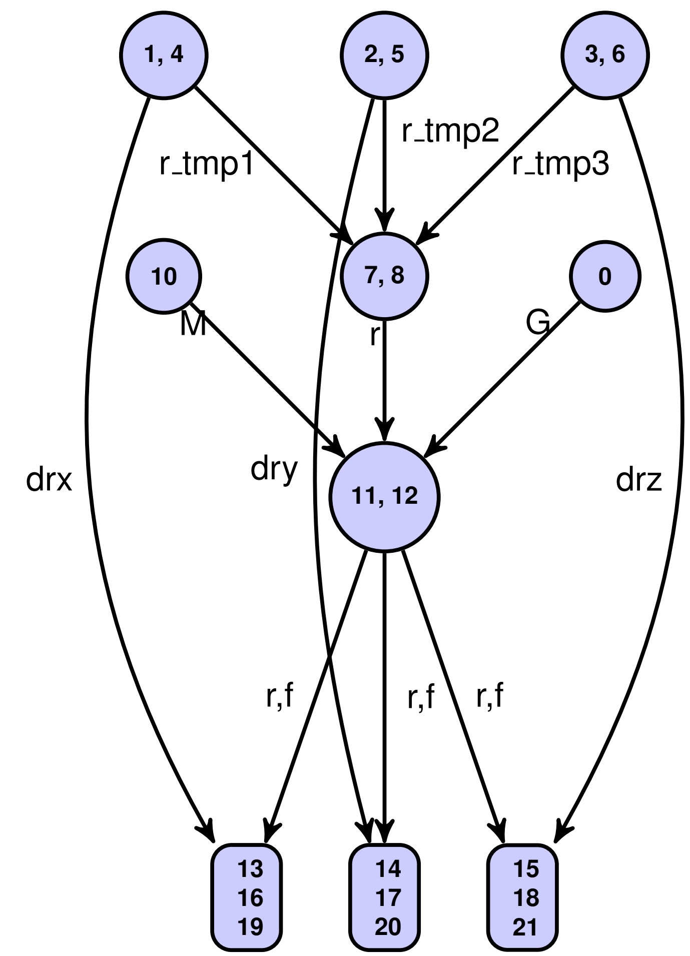
- No dependency violation
- No reduction in parallelism
- Prefer merging related tasks for improved locality

Example: GaussR

```
G = 1e-11;
drx = Rx-Rx(k);
dry = Ry-Ry(k);
drz = Rz-Rz(k);
r_tmp1  = drx.*drx;
r_tmp2  = dry.*dry;
r_tmp3  = drz.*drz;
r = r_tmp1+r_tmp2+r_tmp3;
r(k) = 1.0;
M = m*m(k);
 M(k)  = 0.0;
 f = G*(M./r);
 r = sqrt(r);
 drx = drx./r;
 dry = dry./r;
 drz = drz./r;
 frx = f.*drx;
 fry = f.*dry;
 frz = f.*drz;
 Fx(k) = mean(frx)*n;
 Fy(k)  = mean(fry)*n;
 Fz(k)  = mean(frz)*n;
```



Static data-flow graph
(21 nodes)



Coarsened graph
(10 nodes)

Concluding Remarks

Take-away Message

- We use data-flow style of parallelism to be able to extract parallelism at all levels, automatically, from MATLAB
- We can extract parallelism that the libraries cannot utilize
- We utilize and build upon the existing modes of parallelism, instead of discarding them
- We can utilize software tools to create loop-level parallelism (e.g., using OpenMP)
- We would like to do better!

http://www.cs.indiana.edu/~achauhan