Chapter 3

Multicore and Multiprocessor Systems: Part IV

Tree Reduction

The OpenMP reduction minimal example revisited: Data Sharing

Example (OpenMP reduction minimal example)

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
int   i, n;
float a[100], b[100], sum;

/* Some initializations */
n = 100;
for (i=0; i < n; i++)
  a[i] = b[i] = i * 1.0;
sum = 0.0;

#pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);
printf("   Sum = %f\n",sum);
}
```

Tree Reduction

The OpenMP reduction minimal example revisited

The main properties of the reduction are

- accumulation of data via a binary operator (here $+$)
- intrinsically sequential operation causing a race condition in multi-thread based implementations (since every iteration step depend on the result of its predecessor.)

Tree Reduction

Basic idea of tree reduction

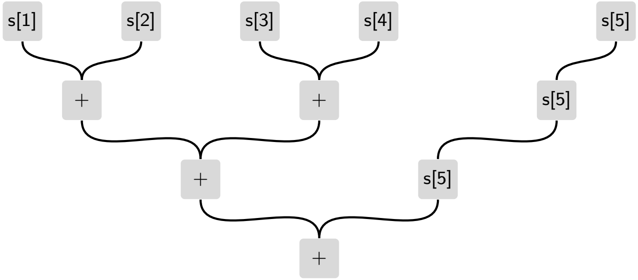


Figure: Tree reduction basic idea.

Tree Reduction

Basic idea of tree reduction

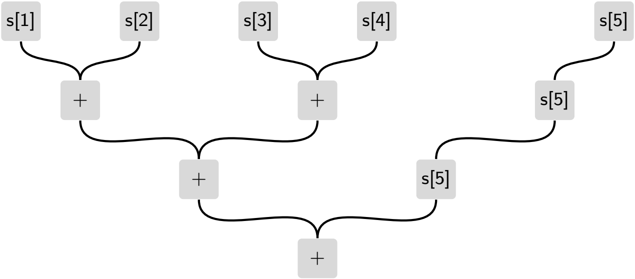


Figure: Tree reduction basic idea.

- ideally the number of elements is a power of 2
- best splitting of the actual data depends on the hardware used

Tree Reduction

Practical tree reduction on multiple cores

Example (Another approach for the `dot` example)

Consider the setting as before $a, b \in \mathbb{R}^{100}$. Further we have four equal cores. How do we compute the accumulation in parallel?

Tree Reduction

Practical tree reduction on multiple cores

Example (Another approach for the `dot` example)

Consider the setting as before $a, b \in \mathbb{R}^{100}$. Further we have four equal cores. How do we compute the accumulation in parallel? Basically 2 choices

Tree Reduction

Practical tree reduction on multiple cores

Example (Another approach for the `dot` example)

Consider the setting as before $a, b \in \mathbb{R}^{100}$. Further we have four equal cores. How do we compute the accumulation in parallel? Basically 2 choices

1. **Task pool approach:** define a task pool and feed it with $n/2 = 50$ work packages accumulating 2 elements in 1. When these are done, schedule the next 25 and so on by further binary accumulation of 2 intermediate results per work package.

Tree Reduction

Practical tree reduction on multiple cores

Example (Another approach for the `dot` example)

Consider the setting as before $a, b \in \mathbb{R}^{100}$. Further we have four equal cores. How do we compute the accumulation in parallel? Basically 2 choices

1. **Task pool approach:** define a task pool and feed it with $n/2 = 50$ work packages accumulating 2 elements in 1. When these are done, schedule the next 25 and so on by further binary accumulation of 2 intermediate results per work package.
2. **#Processors=#Threads approach:** Divide the work by the number of threads, i.e. on our 4 cores each gets 25 subsequent indices to sum up. The reduction is then performed on the results of the threads.

Dense Linear Systems of Equations

Repetition blocked algorithms

Algorithm 1: Gaussian elimination – row-by-row-version

Input: $A \in \mathbb{R}^{n\times n}$ allowing LU decomposition
Output: A overwritten by L, U

```
for k = 1 : n − 1 do
    ;
    ;
    A(k + 1 : n, k) = A(k + 1 : n, b)/A(k, k);
    for i = k + 1 : n do
        for j = k + 1 : n do
            A(i, j) = A(i, j) − A(i, k)A(k, j);
```

Dense Linear Systems of Equations

Repetition blocked algorithms

Algorithm 1: Gaussian elimination – row-by-row-version

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition
Output: A overwritten by L, U

```
for k = 1 : n − 1 do
    ;
    ;
    A(k + 1 : n, k) = A(k + 1 : n, b)/A(k, k);
    for i = k + 1 : n do
        for j = k + 1 : n do
            A(i, j) = A(i, j) − A(i, k)A(k, j);
```

Observation:

- Innermost loop performs rank-1 update on the $A(k+1:n, k+1:n)$ submatrix in the lower right,
- i.e. a BLAS level 2 operation.

Dense Linear Systems of Equations

Repetition blocked algorithms

Algorithm 2: Gaussian elimination – Outer product formulation

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition
Output: $L, U \in \mathbb{R}^{n \times n}$ such that $A = LU$ stored in A stored in A

```
for k = 1 : n − 1 do
    rows= k + 1 : n;
    A(rows, k) = A(rows, k)/A(k, k);
    A(rows,rows) = A(rows,rows) − A(rows, k)A(k,rows);
```

Dense Linear Systems of Equations

Repetition blocked algorithms

Algorithm 2: Gaussian elimination – Outer product formulation

Input: $A \in \mathbb{R}^{n\times n}$ allowing LU decomposition
Output: $L, U \in \mathbb{R}^{n\times n}$ such that $A = LU$ stored in A stored in A

```
for k = 1 : n − 1 do
    rows= k + 1 : n;
    A(rows, k) = A(rows, k)/A(k, k);
    A(rows,rows) = A(rows,rows) − A(rows, k)A(k,rows);
```

Idea of the blocked version

- Replace the rank-1 update by a rank-r update ,
- Thus replace the $\mathcal{O}(n^2)$ / $\mathcal{O}(n^2)$ operation per data ratio the more desirable $\mathcal{O}(n^3)$ / $\mathcal{O}(n^2)$ ratio,
- Therefore exploit the fast local caches of modern CPUs more optimally.

Dense Linear Systems of Equations

Repetition blocked algorithms

Algorithm 3: Gaussian elimination – Block outer product formulation

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition, r prescribed block size

Output: $A = LU$ with L, U stored in A

```
k = 1;
while k ≤ n do
    ℓ = min(n, k + r − 1);
    Compute A(k : ℓ, k : ℓ) = L̃Ũ via Algorithm 7;
    Solve L̃Z = A(k : ℓ, ℓ + 1 : n) and store Z in A;
    Solve WŨ = A(ℓ + 1 : n, k : ℓ) and store W in A;
    Perform the rank-r update:
    A(ℓ + 1 : n, ℓ + 1 : n) = A(ℓ + 1 : n, ℓ + 1 : n) − WZ;
    k = ℓ + 1;
```

Dense Linear Systems of Equations

Repetition blocked algorithms

Algorithm 3: Gaussian elimination – Block outer product formulation

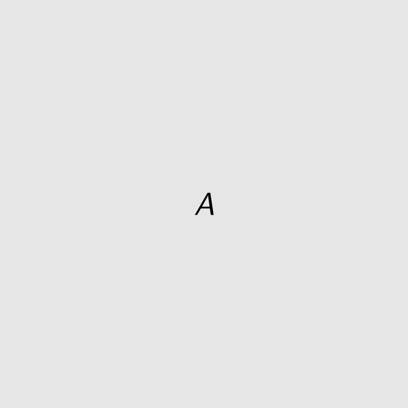
Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition, r prescribed block size
Output: $A = LU$ with L, U stored in A

$k = 1$;
while $k \leq n$ **do**
  $\ell = \min(n, k + r - 1)$;
  Compute $A(k : \ell, k : \ell) = \tilde{L}\tilde{U}$ via Algorithm 7;
  Solve $\tilde{L}Z = A(k : \ell, \ell + 1 : n)$ and store Z in A;
  Solve $W\tilde{U} = A(\ell + 1 : n, k : \ell)$ and store W in A;
  Perform the rank-r update:
  $A(\ell + 1 : n, \ell + 1 : n) = A(\ell + 1 : n, \ell + 1 : n) - WZ$;
  $k = \ell + 1$;

The block size r can be further exploited in the computation of W and Z and the rank-r update. It is used to optimize the data portions for the cache.

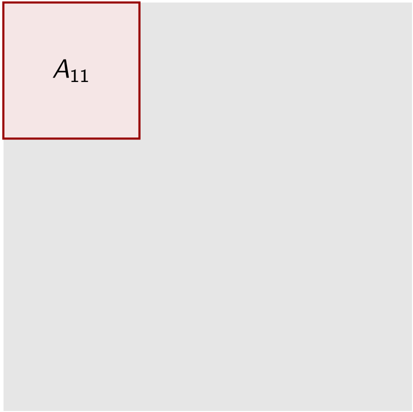
Dense Linear Systems of Equations

Repetition blocked algorithms



Dense Linear Systems of Equations

Repetition blocked algorithms

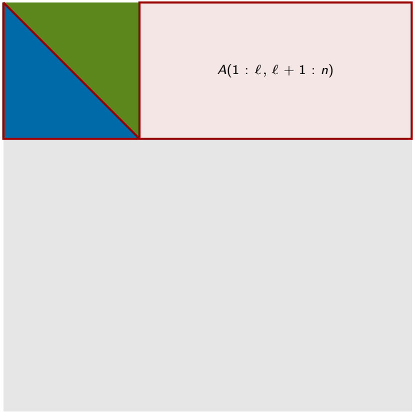


Dense Linear Systems of Equations

Repetition blocked algorithms

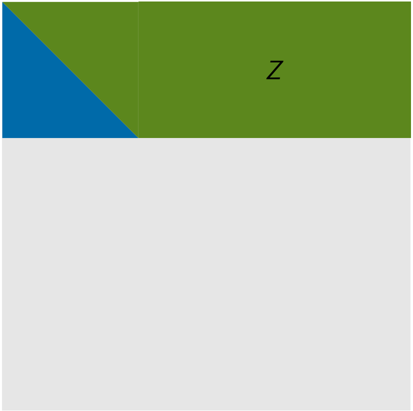
Dense Linear Systems of Equations

Repetition blocked algorithms



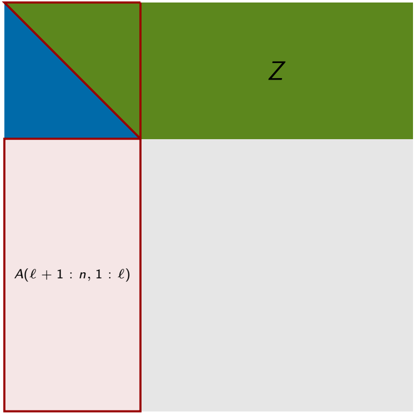
Dense Linear Systems of Equations

Repetition blocked algorithms



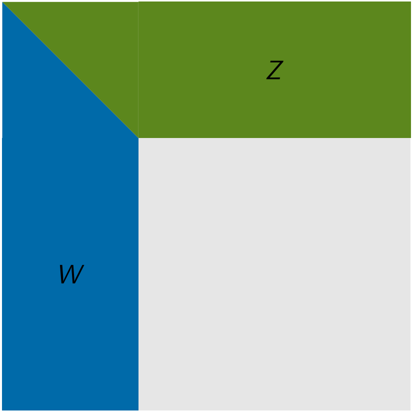
Dense Linear Systems of Equations

Repetition blocked algorithms



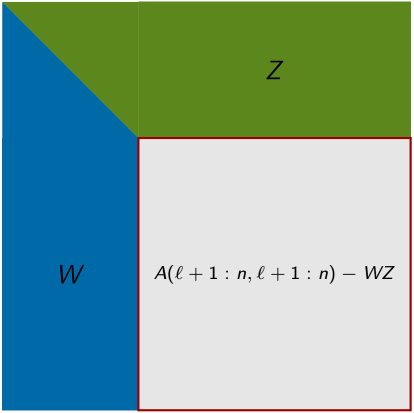
Dense Linear Systems of Equations

Repetition blocked algorithms



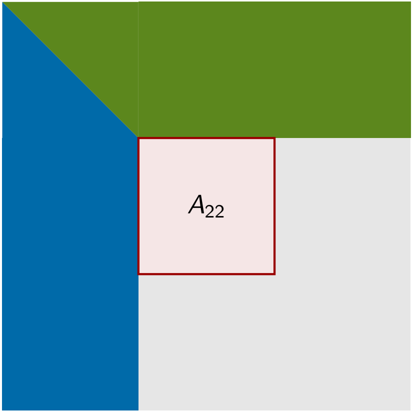
Dense Linear Systems of Equations

Repetition blocked algorithms



Dense Linear Systems of Equations

Repetition blocked algorithms



Dense Linear Systems of Equations

Repetition blocked algorithms

Dense Linear Systems of Equations

Repetition blocked algorithms

Dense Linear Systems of Equations

Repetition blocked algorithms

Dense Linear Systems of Equations

Fork-Join parallel implementation for multicore machines

We have basically two ways to implement naive parallel versions of the block outer product elimination in Algorithm 6.

Threaded BLAS available

- Compute line 4 with the sequential version of the LU
- Exploite the threaded BLAS for the block operations in lines 5–7

Dense Linear Systems of Equations

Fork-Join parallel implementation for multicore machines

We have basically two ways to implement naive parallel versions of the block outer product elimination in Algorithm 6.

Threaded BLAS available

- Compute line 4 with the sequential version of the LU
- Exploite the threaded BLAS for the block operations in lines 5–7

Netlib BLAS

- Compute line 4 with the sequential version of the LU
- Employ OpenMP/PThreads to perform the BLAS calls for the block operations in lines 5–7 in parallel.

Dense Linear Systems of Equations

Fork-Join parallel implementation for multicore machines

Both these approaches fall into the class of parallel codes described by the following definition.

Definition (Fork-Join Parallelism)

An algorithm that performs a loop where certain parts need to be performed in a sequential way, but others can be done in parallel is *called fork-join-parallel*.

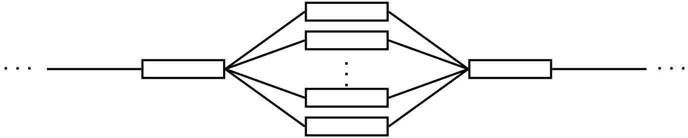


Figure: A sketch of the fork-join execution model.

Dense Linear Systems of Equations

Fork-Join parallel implementation for multicore machines

Advantages

- Easy to achieve,
- many threaded BLAS implementations available,
- Basically usable from any user code that requires linear system solves

Disadvantages

- very naive implementation,
- sequential fraction limits the speedup (Amdahl's law),
- i.e., only useful for small numbers of cores.

Dense Linear Systems of Equations

DAG scheduling of block operations aiming at manycore systems

Definition (Directed Acyclic Graph (DAG))

A *directed acyclic graph* is a graph where

- all edges have one distinct direction,
- directions are such that no cycles are possible for any path in the graph.

Where is the connection to parallel mathematical algorithms?

- Consider every node in the graph a task in the computation.

Dense Linear Systems of Equations

DAG scheduling of block operations aiming at manycore systems

Definition (Directed Acyclic Graph (DAG))

A *directed acyclic graph* is a graph where

- all edges have one distinct direction,
- directions are such that no cycles are possible for any path in the graph.

Where is the connection to parallel mathematical algorithms?

- Consider every node in the graph a task in the computation.
- Every task requires certain number of previous tasks to have finished.

Dense Linear Systems of Equations

DAG scheduling of block operations aiming at manycore systems

Definition (Directed Acyclic Graph (DAG))

A *directed acyclic graph* is a graph where

- all edges have one distinct direction,
- directions are such that no cycles are possible for any path in the graph.

Where is the connection to parallel mathematical algorithms?

- Consider every node in the graph a task in the computation.
- Every task requires certain number of previous tasks to have finished.
- Also none of the previous tasks depend on the later ones.

Dense Linear Systems of Equations

DAG scheduling of block operations aiming at manycore systems

Definition (Directed Acyclic Graph (DAG))

A *directed acyclic graph* is a graph where

- all edges have one distinct direction,
- directions are such that no cycles are possible for any path in the graph.

Where is the connection to parallel mathematical algorithms?

- Consider every node in the graph a task in the computation.
- Every task requires certain number of previous tasks to have finished.
- Also none of the previous tasks depend on the later ones.
- Thus, the dependencies give us the directions and cycles can not appear by construction.

Dense Linear Systems of Equations

DAG scheduling of block operations aiming at manycore systems

Dense Linear Systems of Equations

DAG scheduling of block operations aiming at manycore systems

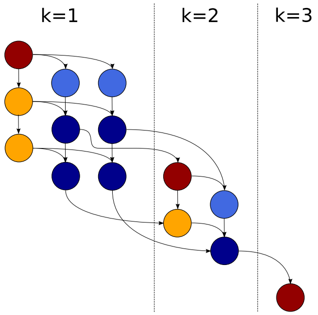


Figure: Dependency graph of Algorithm 6 for a 3×3 block subdivision.

Dense Linear Systems of Equations

DAG scheduling of block operations aiming at manycore systems

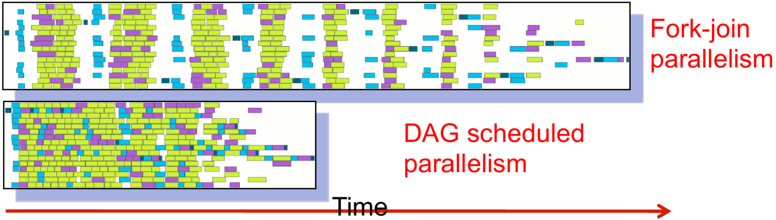


Figure: The superiority of DAG scheduling of tasks over fork-join parallelism.