www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Towards task-parallel reductions in OpenMP

J. Ciesko, S. Mateo, X. Teruel, X. Martorell, E. Ayguadé,
J. Labarta, A. Duran, B. De Supinski, S. Olivier,
K. Li, A. Eichenberger

IWOMP - Aachen, Germany, October 1-2, 2015

Outline

1. Introduction
2. Motivation
3. Related work
4. Contribution
5. Evaluation
6. Conclusions & Future Work

1. Introduction

var = op(var, expression)

- **Defined** as **recurrent update** over a variable by applying an **associative** and **commutative** operator
- Dot product

```
float res = 0.0f;
float v1[N], v2[N];
…
for (int i = 0; i < N; ++i)
   res += v1[i] * v2[i];
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

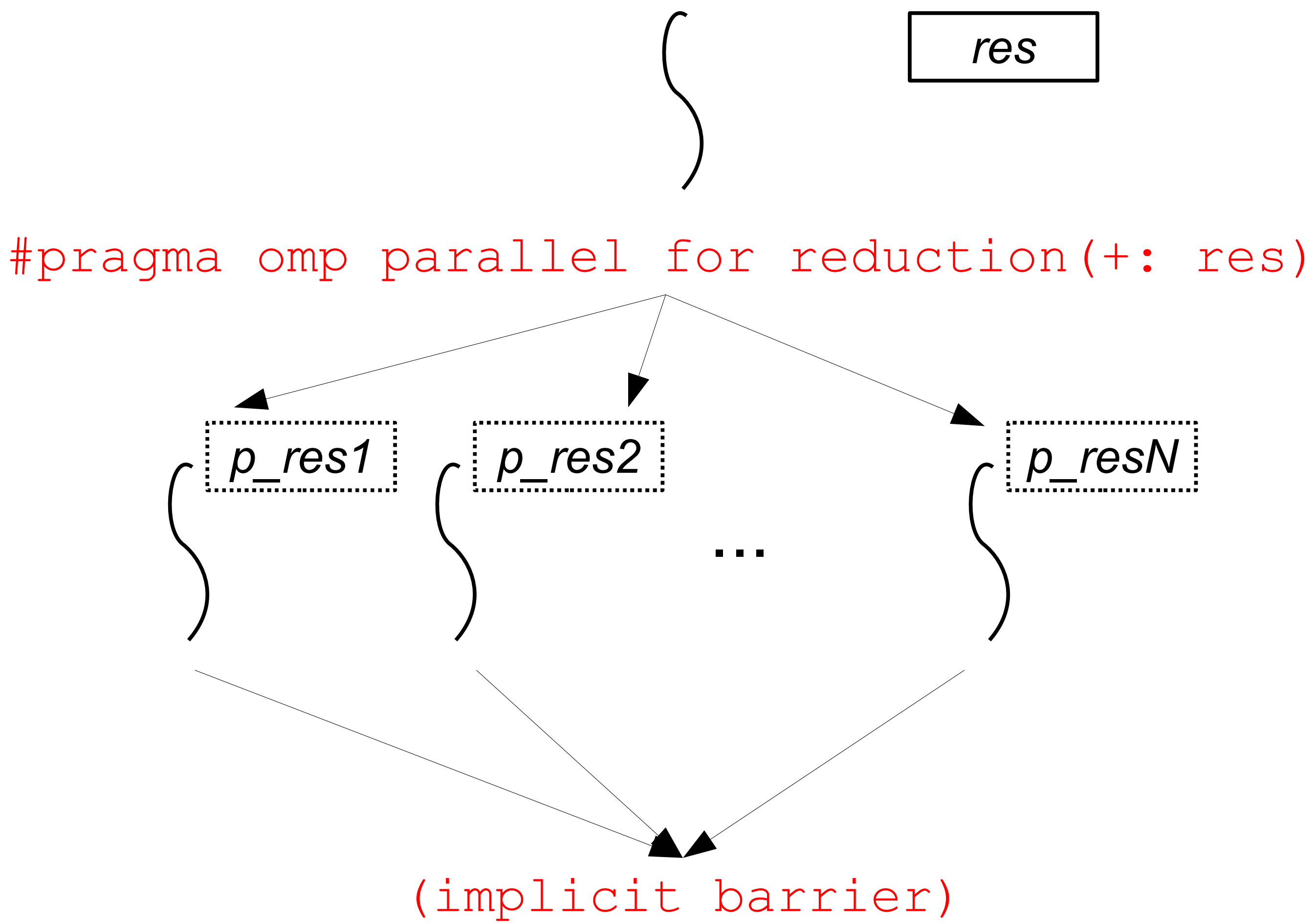
1. Introduction

var = op(var, expression)

- **Defined** as **recurrent update** over a variable by applying an **associative** and **commutative** operator
- Dot product

```
float res = 0.0f;
float v1[N], v2[N];
…
#pragma omp parallel for reduction(+: res)
for (int i = 0; i < N; ++i)
    res += v1[i] * v2[i];
```

1. Introduction



1. Introduction

- Reduction over a linked list

```
int res = 0;
node_t* node = NULL;
…
while (node) {
    res += node->value;
    node = node->next;
}
```

- NQueens

```
int nqueens(int row, ...){
  if(row == lastRow)
   return 1;

  int res = 0;
  for (int i=0; i<lastRow; ++i)
    if(check_attack(...))
        res += nqueens(...);

  return res;
}
```

1. Introduction

- Reduction over a linked list

```
int res = 0;
node_t* node = NULL;
…
while (node) {
    res += node->value;
    node = node->next;
}
```

We cannot solve this problem directly with the actual OpenMP reduction support!

- NQueens

```
int nqueens(int row, ...){
  if(row == lastRow)
   return 1;

  int res = 0;
  #pragma omp parallel for \
              reduction(+: res)
  for (int i=0; i<lastRow; ++i)
    if(check_attack(...))
        res += nqueens(...);

  return res;
}
```

It works but It has some disadvantatges
- OMP_NESTED=1
- Cut-off

2. Motivation

- Tasks are useful for irregular algorithms...
  - but they don't support reductions (yet) :(

```
int res = 0;
node_t* node = NULL;
...
while (node) {
    res += node->value;
    node = node->next;
}
```

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    while (node)
    {
      #pragma omp task ???
        res += node->value;

      node = node->next;
    }
  }
}
```

2. Motivation

- Reduction over a linked list using task dependences

```
int res = 0;
node_t* node = NULL;
...
while (node) {
  res += node->value;
  node = node->next;
}
```

[1] all tasks have been executed

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    while (node)
    {
      #pragma omp task \
        firstprivate(node) shared(res) \
        depend(inout: res)

        res += node->value;

      node = node->next;
    }
  } // [1]
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

2. Motivation

- Reduction over a linked list using atomics

```
int res = 0;
node_t* node = NULL;
...
while (node) {
  res += node->value;
  node = node->next;
}
```

[1] all tasks have been executed

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    while (node)
    {
      #pragma omp task \
        firstprivate(node) shared(res)
      {
        #pragma omp atomic
        res += node->value;
      }
      node = node->next;
    }
  } // [1]
}
```

2. Motivation: avoiding boilerplate codes

- Reduction over a linked list using threadprivate directive

```
int res = 0;
node_t* node = NULL;
...
while (node) {
  res += node->value;
  node = node->next;
}
```

[1] all tasks have been executed. part_res thread private variables contain the partial results of the reduction

```
int res = 0;
int part_res = 0;
#pragma omp threadprivate(part_res)
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    while (node) {
      #pragma omp task \
          firstprivate(node)
        part_res += node->value;

      node = node->next;
    }
  } // [1]

}
```

2. Motivation: avoiding boilerplate codes

- Reduction over a linked list using threadprivate directive

```
int res = 0;
node_t* node = NULL;
...
while (node) {
  res += node->value;
  node = node->next;
}
```

[1] all tasks have been executed. part_res thread private variables contain the partial results of the reduction

[2] final reduction

```
int res = 0;
int part_res = 0;
#pragma omp threadprivate(part_res)
node_t* node = NULL;
...
#pragma omp parallel reduction(+:res)
{
  #pragma omp single
  {
    while (node) {
      #pragma omp task \
          firstprivate(node)
        part_res += node->value;

      node = node->next;
    }
  } // [1]
  res += part_res;
} // [2]
```

2. Motivation: avoiding boilerplate codes

- Reduction over a linked list using additional storage

```
int res = 0;
node_t* node = NULL;
...
while (node) {
  res += node->value;
  node = node->next;
}
```

[1] all tasks have been executed. part_res contains the partial results of the reduction

[2] final reduction

```
int res = 0;
int part_res[omp_get_max_threads()]={0};
node_t* node = NULL;
...
#pragma omp parallel reduction(+:res)
{
  #pragma omp single
  {
    while (node) {
      #pragma omp task \
                    firstprivate(node)
      {
       int id = omp_get_thread_num();
       part_res[id] += node->value;
      }
      node = node->next;
    }
  } // [1]
  res+=part_res[omp_get_thread_num()];
} // [2]
```

3. Related Work

- Previous attempts (OpenMP Lang)
  - Grant Haab & Federico Massaioli's proposal
  - Alex Duran's proposal

- IWOMP 2014
  - Ciesko, J., Mateo, S., Teruel, X., Beltran, V., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: *Task-Parallel Reductions in OpenMP and OmpSs*
  - General approach
    - Reducing on task dependences, taskwait, barrier and at the end of a taskgroup
  - Prototype implementation
  - Evaluation comparing our implementation with manual atomic approach

3. Related Work: feedback

- The taskgroup construct defines the scope of the reduction
- The specification should **allow several implementations**
  - Number of private copies
  - Calls to the combiner
- Related issues
  - Supporting untied tasks
  - Nested taskgroup reductions

4. Contribution

- Extending taskgroup clauses to support reduction clause
  - Syntax

```
#pragma omp taskgroup reduction(red-id: list_items)
structured-block
```

  - Semantics: defines the **reduction scope**

- Extending task construct to support in_reduction clause
  - Syntax

```
#pragma omp task in_reduction(red-id: list_items)
structured-block
```

  - Semantics: defines a task **as a participant** of a previously registered task reduction

4. Contribution: example

- Our proposal

```
int res = 0;
node_t* node = NULL;
...
while (node) {
  res += node->value;
  node = node->next;
}
```

[1] registering a new reduction

[2] working with a private copy

[3] final reduction

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp taskgroup reduction(+:res)
    { // [1]
      while (node)
      {
        #pragma omp task \
             firstprivate(node) \
             in_reduction(+:res)
          res += node->value; // [2]
        node = node->next;
      }
    }// [3]
  }
}
```

4. Contribution: our implementation decisions

- We register a **private copy for each implicit task** (thread) at the beginning of the taskgroup construct
- Tasks that participate in a previously registered reduction **just ask for the private storage** of the thread that execute them
- Untied reduction tasks **are implemented as tied tasks**
- Nested taskgroups performing a reduction over the same variable **do not reuse** the same private storage

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Performance results

5. Evaluation

- We compare the performance of **our prototype implementation** against **a manual reduction implementation** using threadprivate storage

- We tested our prototype in two environments:
  - Intel Xeon processors
    - GCC 4.7.2 as native compiler
  - Intel Xeon Phi coprocessors
    - Intel 15.0.2 as native compiler

- In both scenarios we used Mercurium source-to-source compiler v1.99.8 and Nanos++ RTL v0.9a

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

5. Evaluation: benchmark descriptions

Array Sum: it computes the sum of *N* elements. We create a task for each *TS* elements

Dot Product: it computes the sum of the products of the components of two vectors of *N* elements. As before, we create a task for each *TS* elements

NQueens: it computes the number of configurations of placing *N* Queens in a *N* x *N* chessboard such that none of them is able to attack to any other. We use the final clause as a cut-off.

- *Global version*: we reduce over a global variable, so we only register one reduction for all the execution
- *Local version*: we reduce over a local variable. This means that we register a new reduction at each recursive level

UTS: this benchmark computes the number of nodes in a implicitly defined unbalanced tree

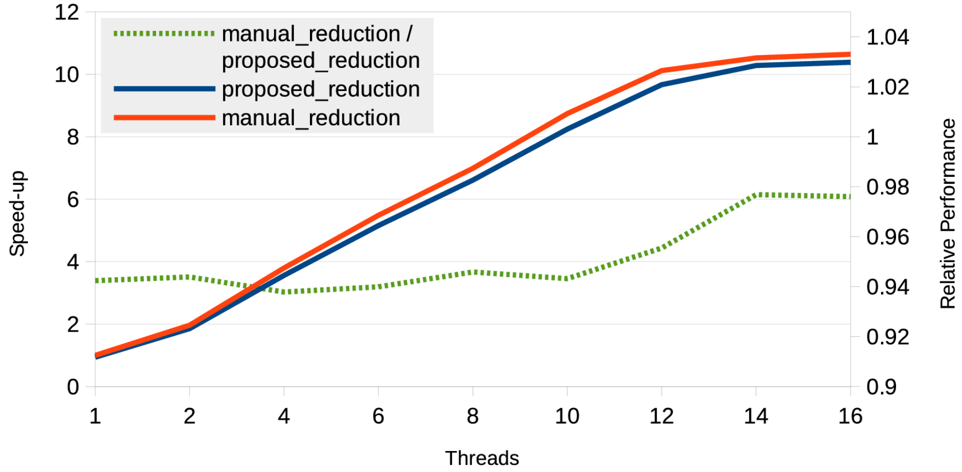
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Results on Intel Xeon processors

5. Evaluation: Array Sum and dot product

Array Sum

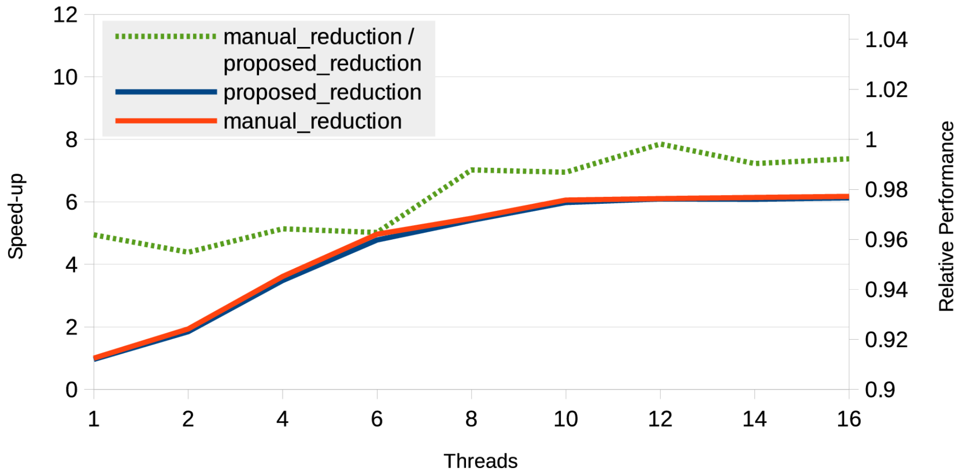
N=10^9, BS=10^6, 1000 tasks, baseline: manual_reduction with 1 thread



- Similar scalability: up to 10x using 16 threads
- The relative perfomance (manual perf. / prototype perf.) is close to 1

Dot product

N=10^9, BS=10^5, 10000 tasks, baseline: manual_reduction with 1 thread

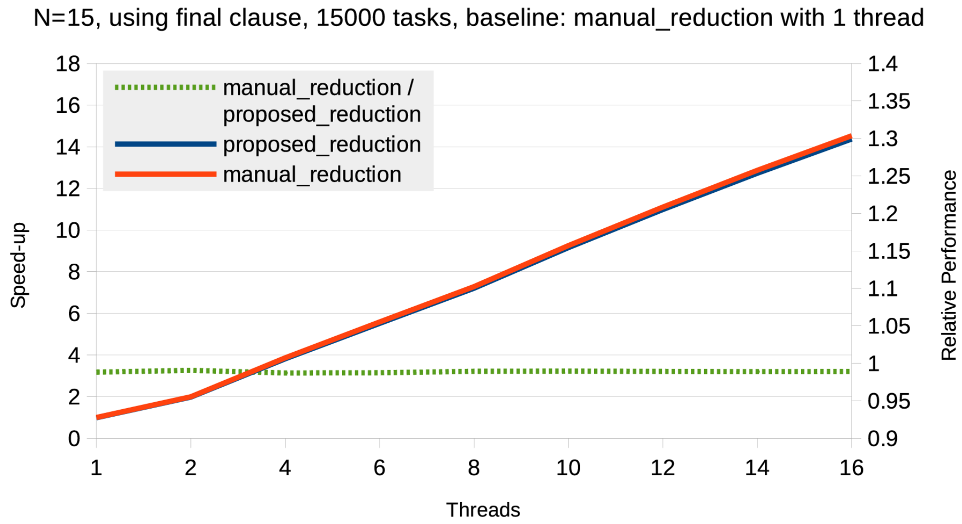


- Similar scalability: up to 6x using 16 threads
  - not enough work
  - NUMA
- The relative perfomance (manual perf. / prototype perf.) is close to 1

BSC Supercomputing Center
Centro Nacional de Supercomputación

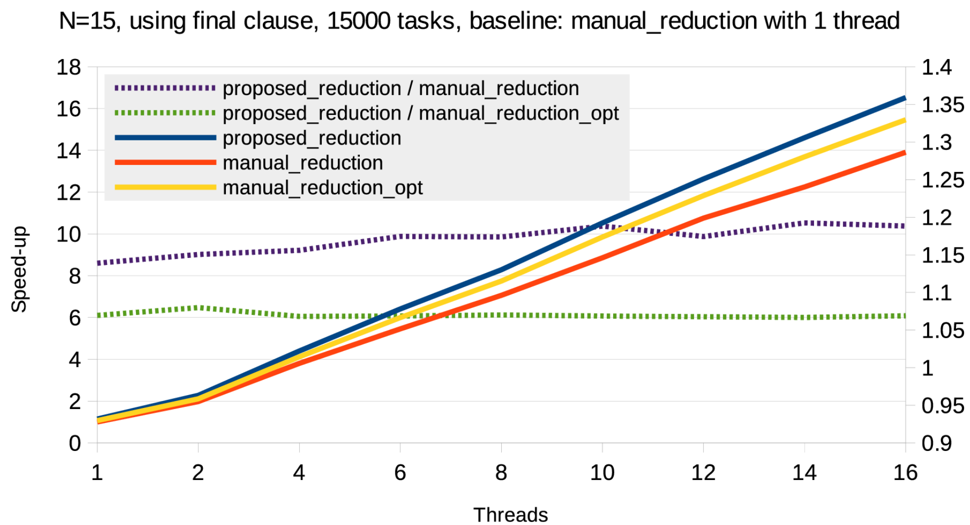
5. Evaluation: NQueens

Nqueens global



- Similar scalability: up to 14x using 16 threads
- The relative perfomance (manual perf. / prototype perf.) is close to 1

Nqueens local

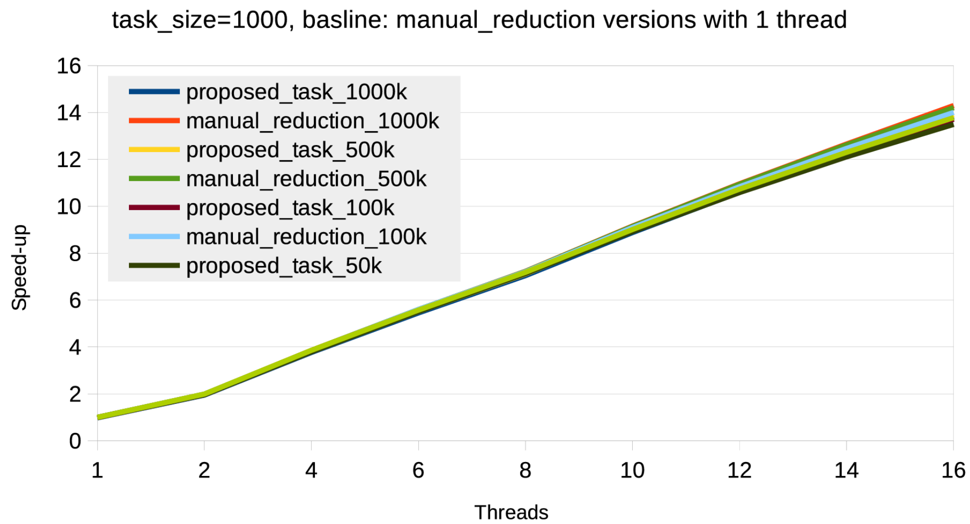


- The scalability of our approach is better than the manual versions
- The perfomance of our approach is 5% higher than the best manual version

Supercomputing Center
Centro Nacional de Supercomputación

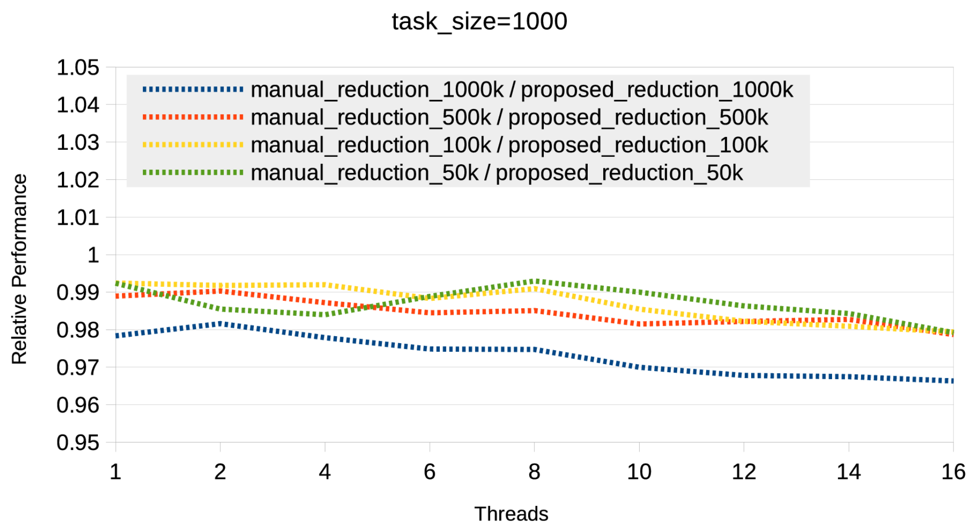
5. Evaluation: UTS

Scalability



- Several executions: 50k, 100k, 500k and 1000k
- not using final clause
- Task granularity is fixed
- The scalability of all the versions is similar

Relative Perfomance



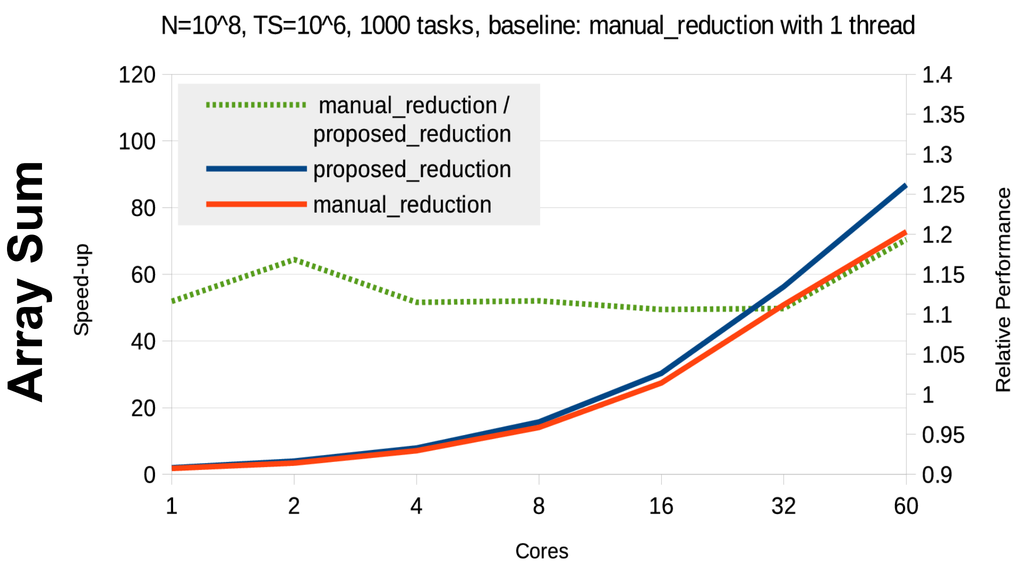
- The relative performances are between 0.96-0.99
- The scenario with the worse performance is also the one that has the higher number of tasks

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Results on Intel Xeon Phi coprocessors

5. Evaluation: Array Sum

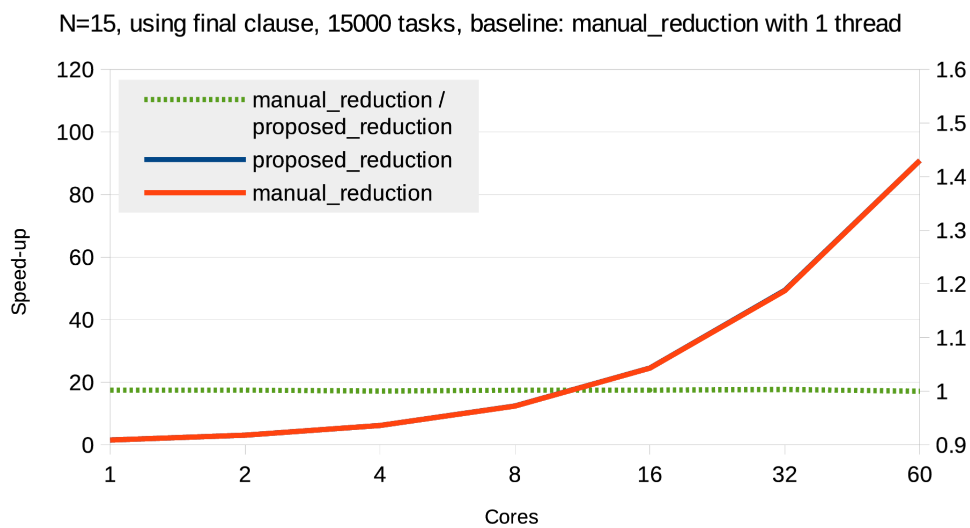
Important: we only show the results of the **best configuration** of threads per core



- The scalability of our approach is better: up to 85x using 60 cores
- The performance of our approach is a bit better than the performance of the manual version
  - But it's not significant since the exeuction times are small

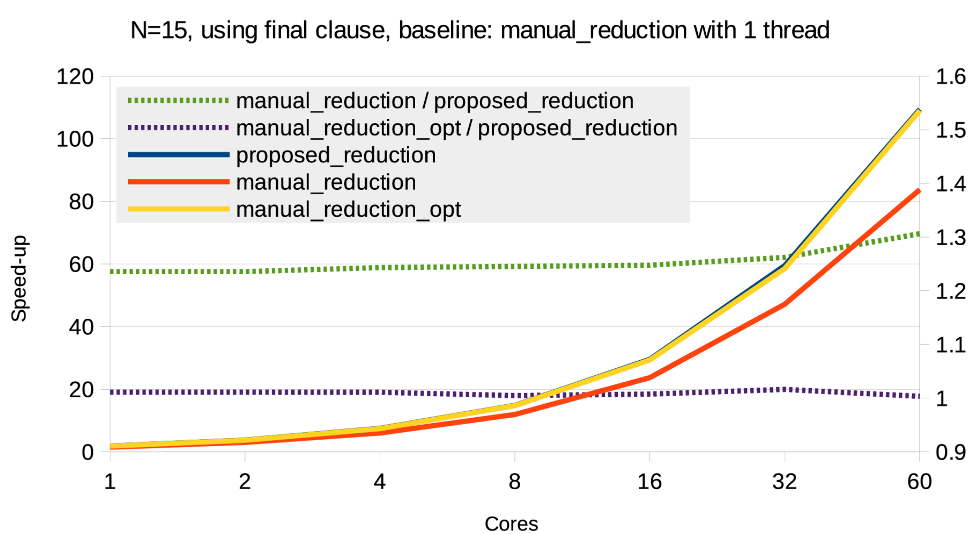
5. Evaluation: NQueens

Nqueens global



- similar scalability: up to 90x using 60 cores and two threads per core
- The relative performance is almost one in all the scenarios

Nqueens local



- similar scalability to the best manual approach: up to 110x using 60 cores and 2 threads per core
- The relative performance of our approach compared with the best manual version is close to 1

6. Conclusions & Future Work

- We **extended** the tasking model adding **support** to task reductions
- The performance of our prototype **is equivalent to** a manual implementation using thread private storage which **was our goal**...
  - but improves **code readability**!
- Future work
  - Write the formal specification of this proposal
  - Extend the taskgroup construct to support the reduction clause

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

sergi.mateo@bsc.es

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

BACKUP

Evaluation: NQueens Local (2)

```
int nqueens(int row, int N, ...) {
  if(n == row) return 1;

  int num_threads =omp_get_num_threads();

  int p_res[num_threads]= {0};
  for(int i = 0; i < N; ++i) {
    if(check_attack(...)) {
      #pragma omp task shared(p_res) \
        final(...) mergeable
      {
        int id = omp_get_thread_num();

        p_res[id] += nqueens(row+1, N);
      }
    }
  }
  #pragma omp taskwait
  int res = 0;
  for(int i = 0; i < N; ++i) {
    res += p_res[i];
  }
  return res;
}
```

Manual version

```
int nqueens(int row, int N, ...) {
  if(n == row) return 1;

  int num_threads = omp_in_final()
       ? 1: omp_get_num_threads();

  int p_res[num_threads]= {0};
  for(int i = 0; i < N; ++i) {
    if(check_attack(...)) {
      #pragma omp task shared(p_res) \
        final(...) mergeable
      {
        int id = omp_in_final()
             ? 0 : omp_get_thread_num();
        p_res[id] += nqueens(row+1, N);
      }
    }
  }
  #pragma omp taskwait
  int res = 0;
  for(int i = 0; i < N; ++i) {
    res += p_res[i];
  }
  return res;
}
```

Manual optimized version

Barcelona Supercomputing Center
Centro Nacional de Supercomputación