TBB: parallel_reduce

OBJECTIVES

- Learn the basics of Threading Building Blocks (TBB) Library in C++: *parallel_reduce*
- Execute parallel applications using TBB.

USEFUL INFORMATION

- Refer to the Tutorial: Embedded Intel for the source files used in this Tutorial.
- Refer to the board website or the Tutorial: Embedded Intel for User Manuals and Guides for the Terasic DE2i-150 Board.
- Board Setup: Connect the monitor (VGA or HDMI) as well as the keyboard and mouse.
  - Refer to the *DE2i-150 Quick Start Guide* (page 2) for a useful illustration.

ACTIVITIES

FIRST ACTIVITY: ACCUMULATE AN ARRAY

- For an n-element vector, we want to accumulate all $a(i)$ elements in the vector. Here, we will use *parallel_reduce*.
- Example: `a = [-1.2 2.3 3.6 6.7 0.3 0.35 2.1 0.7 5.1 -1.1]`
- Result: `sum = -1.2 + 2.3 + 3.6 + 6.7 + 0.3 + 0.35 + 2.1 + 0.7 + 5.1 - 1.1`

Serial implementation

- This is a straightforward implementation as a function:

```
float SerialSumFun( float a[], size_t n ) {
   float sum = 0;
   for (size_t i=0; i!=n; ++i )
       sum += a[i];
   return sum;
}
```

Parallel implementation

- The sum of the vector can be computed by dividing the vector in several parts, computing the sum for each part (in parallel) and then add the results for each part to get the whole sum.
- Reduction operation: Applying an operation on all members of group (e.g.: sum, max, min) and return a result.
  - ✓ The *parallel_reduce* template indicates the iteration space, as well as the object:

```
float ParallelSumFun( float a[], size_t n ) {
    SumFum sf(a);
    parallel_reduce(blocked_range<size_t>(0,n), sf );
    return sf.my_sum; }
```

  - ✓ The class `SumFun` specifies the details of the reduction (e.g.: how to accumulate subsums and combine them):

```
class SumFun {
    float * my_a; // 'private' access (default access level)
public:
    float my_sum;

    void operator()( const blocked_range<size_t> &r ) {
        float *a = my_a;
        float sum = my_sum; // to not discard earlier accumulations

        for ( size_t i=r.begin(); i!=r.end(); ++i )
           sum += a[i];
        my_sum = sum;
    }
    SumFun (SumFun &x, split): my_a(x.my_a), my_sum(0) {}
    void join (const SumFun &y) { my_sum += y.my_sum; }
    SumFun (float a[]): my_a(a), my_sum(0) {} // member initialization in constructor
};
```

- Fig. 1 provides details about some of the syntax. An object `sf` is first created with argument `a` (a pointer). The `parallel_reduce` function performs the reduction operation as specified in the class `void operator ()` function. But instead of doing it sequentially, it recursively divides the iteration space (range) and tries to compute the sub-reductions in parallel. Syntax-wise, this is specified in the splitting constructor and the merging method.

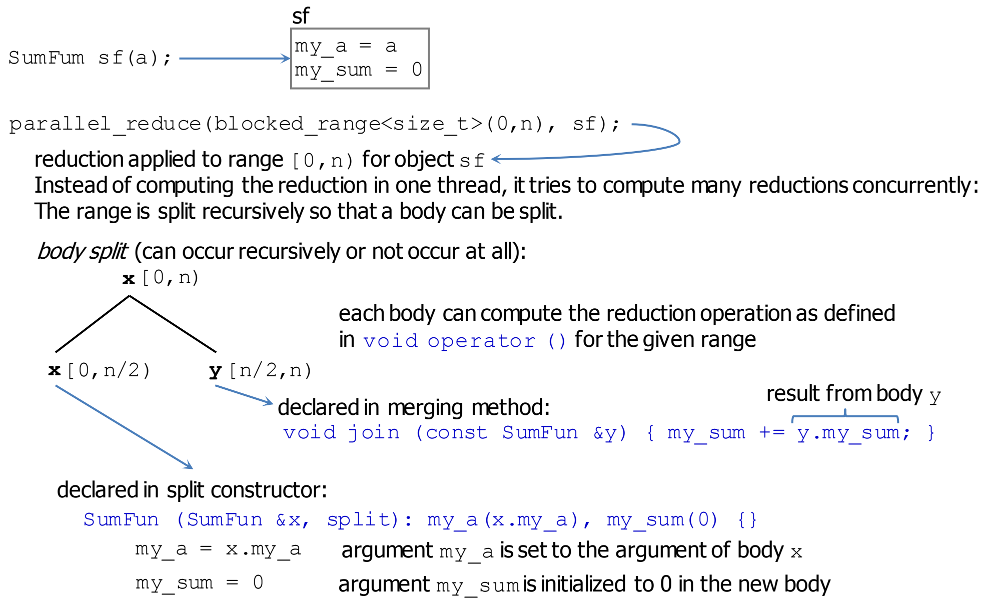

sf
SumFun sf(a);
my_a = a
my_sum = 0
parallel_reduce(blocked_range<size_t>(0,n), sf);
reduction applied to range [0,n) for object sf
Instead of computing the reduction in one thread, it tries to compute many reductions concurrently:
The range is split recursively so that a body can be split.
body split (can occur recursively or not occur at all):
x [0,n)
x [0,n/2)
y [n/2,n)
each body can compute the reduction operation as defined in void operator () for the given range
result from body y
declared in merging method:
void join (const SumFun &y) { my_sum += y.my_sum; }
declared in split constructor:
SumFun (SumFun &x, split): my_a(x.my_a), my_sum(0) {}
my_a = x.my_a
argument my_a is set to the argument of body x
my_sum = 0
argument my_sum is initialized to 0 in the new body

Figure 1. Object created and steps inside the reduction.

- Splitting constructor: `SumFun (SumFun &x, split): my_a(x.my_a), my_sum(0) {}`
  - ✓ When worker threads available, the splitting constructor for the body is invoked. It forks a *body* (function object) into two bodies that can run concurrently.
  - ✓ It splits `x` into `x` and the newly constructed object. After the constructor runs, `x` and the newly constructed object represent the two pieces of the original `x`.
  - ✓ Arguments: reference to original object (`SumFun &x`), and a dummy argument of type `split` (this distinguishes the splitting constructor from a copy constructor).
  - ✓ Member initialization: `my_a = x.my_a, my_sum = 0`.

- Merging method: `void join (const SumFun &y) { my_sum += y.my_sum; }`
  - ✓ It must be present alongside the splitting constructor for *parallel_reduce* to work.
  - ✓ For each split of the body, it causes method join to merge result from the bodies.
  - ✓ It is invoked when a task finishes its work and needs to merge the result back with the main body of work. The parameter passed to the method (`y`) is the result of the work.

- Parameterized constructor: `SumFun (float a[]): my_a(a), my_sum(0) {}`
  - ✓ It defines the initialization of the constructor members when the object is declared: `my_a = x.my_a, my_sum = 0`.

- Operator: `void operator()( const blocked_range<size_t> &r ) { … }`
  - ✓ The operator cannot be `const` copied (it needs to store intermediate results) and it needs to be able to merge.
  - ✓ The operation `sum += a[i] = F(sum, f(a[i]))` is called the reduction function.
  - ✓ In general, we can apply a function to `a[i]`. In this case, we have $f(a[i]) = a[i]$.
  - ✓ *parallel_reduce* performs a parallel reduction (e.g.: accumulate result for subrange) by applying the function defined in the `operator()` to subranges. It returns the result of the reduction.
  - ✓ When a worker thread is available (as decided by the task scheduler) *parallel_reduce* invokes the splitting constructor to create a subtask for the worker. When the subtask completes, *parallel_reduce* uses method `join` to accumulate the result of the subtask. For each such split of the body, it invokes method join to merge the results from the bodies. Fig. 2 shows a splitting case for *n = 6*. Note that there are multiple possible executions and we only show one.

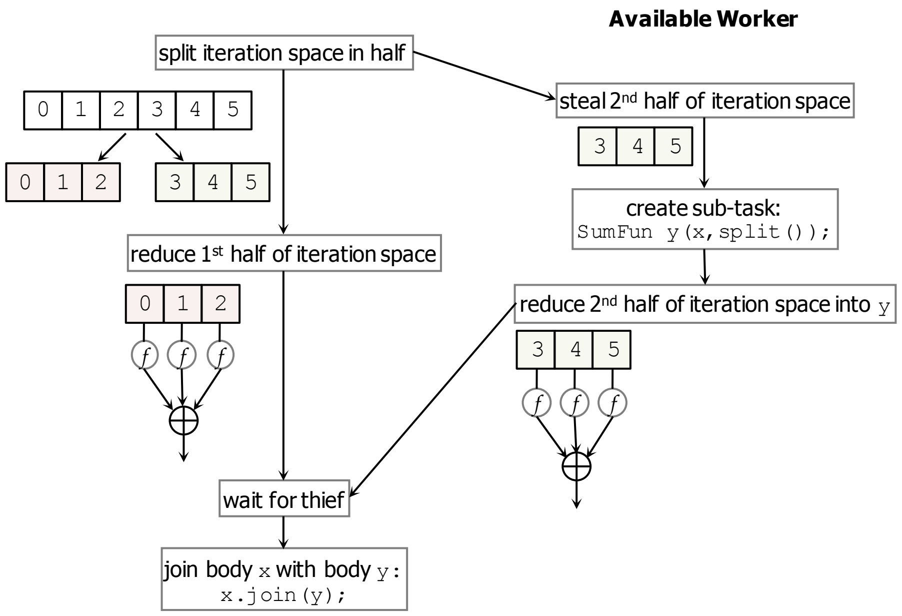

Available Worker
split iteration space in half
steal 2nd half of iteration space
create sub-task: SumFun y(x,split());
reduce 1st half of iteration space
reduce 2nd half of iteration space into y
wait for thief
join body x with body y: x.join(y);
0 1 2 3 4 5
f

Figure 2. A split-join sequence for n=6. Iteration space: [0,6). The range was only split into two subranges.

- Application files: `accum.cpp`
  - ✓ In this example, `a` is a constant vector with `n = 10` elements.
  - ✓ The operation $sum \leftarrow \sum_{i=0}^{n-1} a(i)$ is implemented both in serial and parallel form.
  - ✓ Note that we use `using namespace tbb`. This avoids having to include the prefix `tbb` before each identifier used by the `tbb` library (e.g.: *parallel_reduce, parallel_for, blocked_range*).

- Compile this application:

```
g++ accum.cpp -ltbb -o accum ↵
```

- Execute this application:

```
./accum ↵
```

SECOND ACTIVITY: ACCUMULATE SQUARED-ELEMENTS OF AN ARRAY

- For an n-element vector, we want to accumulate all squared $a(i)$ elements in the vector. Here, we will use *parallel_reduce*.
- Example: `a = [0 1 2 3 4 5 6 7 8 9]`
- Result: sum = $0^2 + 1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 + 9^2$

Parallel implementation

- The sum of the squared elements of a vector can be computed by dividing the vector in several parts, computing the sum for each part (in parallel) and then add the results for each part to get the whole sum.
- Reduction operation: Applying an operation on all members of group (e.g.: sum, max, min) and return a result.
  - ✓ This is very similar to the example in the First Activity. We show an alternative coding style.
  - ✓ The *parallel_reduce* template indicates the iteration space, as well as the object:

```
int main () {
   int tab[10];
   int i;

   for (i = 0; i < 10; i++) tab[i] = i; // Data initialization

   MyOp op(tab); // the object 'op' is created with 'tab' as the argument
   parallel_reduce (blocked_range<size_t> (0,10), op);

   cout << op.a << "\n";
   return 0;
}
```

  - ✓ The class `MyOp` specifies the details of the reduction (e.g.: how to accumulate subsums and combine them):

```
class MyOp {
  const int *my_tab;

public:
  int a;

  void operator () (const blocked_range<size_t> &r)
  {
    const int *tab = my_tab;
    int ta = a;

    for (size_t i = r.begin(); i != r.end(); ++i)
      ta =+ tab[i]*tab[i]; //F(ta, tab[i]); --> reduction function
    a = ta;
  }

  MyOp (MyOp &x, split):  my_tab (x.my_tab), a(0) {} // mytab = x.my_tab, a = 0
  void join (const MyOp &y) { a += y.a;} // a = F(a,y.a)
  MyOp (const int tab[]): my_tab(tab), a(0) {} // my_tab = tab, a = 0.
};
```

- Here, the function applied to `tab[i]` is: $f(tab[i]) = tab[i]^2$.

- Note that *parallel_reduce* can also be used as a slightly mixed together map/reduce. We could have performed the squaring operation (on an element) in a *parallel_for*, then use *parallel_reduce* to sum all those elements. But we sort of combine the operations into just *parallel_reduce*.

- Application files: `accum_sq.cpp`
  - ✓ In this example, `a` is a constant vector with `n = 10` elements.
  - ✓ The operation $sum \leftarrow \sum_{i=0}^{n-1} a(i)^2$ is implemented both in serial and parallel form.
  - ✓ Note that we use `using namespace tbb`. This avoids having to include the prefix `tbb` before each identifier used by the `tbb` library (e.g.: *parallel_reduce, parallel_for, blocked_range*).

- Compile this application:

```
g++ accum_sq.cpp -ltbb -o accum_sq ↵
```

- Execute this application:

```
./accum_sq ↵
```

THIRD ACTIVITY: COMPUTATION OF π

- To compute π, we can use the Leibniz formula:

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

- The formula converges very slowly; however, it uses simple computations.
- Since the operation at every index is independent, we can use *parallel_reduce* to increase the computation speed.

Parallel implementation

- The equation is a sum of a function $f(k) = \frac{(-1)^k}{2k+1}$ applied to every index k. Using N=100,000 iterations can produce a precise enough value of π. So, we divide the iterations (a vector whose elements are the indices k) into many portions, computing the sum for each part (in parallel) and then add the results for each part to get the whole sum.
  - ✓ The *parallel_reduce* template indicates the iteration space, as well as the object:

```
int main(int argc, char **argv) {
   ...
   MySeries x; // the object 'x' is created with no arguments
   parallel_reduce (blocked_range<size_t> (0,N),x);
   ...
   cout << setprecision(8); cout << 4*x.my_sum << "\n";
   return 0;
}
```

  - ✓ The class `MySeries` specifies the details of the reduction (e.g.: how to accumulate partial sums and combine them):

```
class MySeries {
public:
   double my_sum;

   void operator()( const blocked_range<size_t> &r ) {
      double sum = my_sum;
      double denom;

      for(size_t i=r.begin(); i!=r.end(); i++ ) {
         denom = i*2.0 + 1.0;
         if ( i%2 == 1) { denom = -denom; }
        sum += 1.0/denom; }
      my_sum = sum;
    }

    MySeries (MySeries &x, split): my_sum(0) {} // my_sum = 0
    void join (const MySeries &y) { my_sum += y.my_sum; }
    MySeries (): my_sum (0) {} // my_sum = 0. Constructor member initialization
};
```

- Here, the function applied to k is: $f(k) = \frac{(-1)^k}{2k+1}$.

- Application files: `mypi.cpp`
  - ✓ Here, π computation is implemented both in sequential and parallel (TBB) form.
  - ✓ `x`: constant vector with `N` elements. `N`: parameter of `main()`.
  - ✓ We use `using namespace tbb` to avoid including the prefix `tbb` before each identifier used by the `tbb` library.

- Compile this application: `g++ mypi.cpp -ltbb -o mypi` ↵
- Execute this application: `./mypi <# of iterations>` ↵
  - ✓ Example: `./mypi 100000` ↵

- Table I lists the computation times for both the sequential and the parallel (TBB) implementation for different number of iterations. There is improvement in the processing time for large number of iterations.

TABLE I. COMPUTATION TIMES (US) FOR DIFFERENT # OF ITERATIONS (N) FOR THE π COMPUTATION (DELL INSPIRON)

N	Processing Time (us)	
	Sequential	TBB
50,000	1756	1916
100,000	1915	1546
200,000	6301	2482
500,000	11704	5593

FOURTH ACTIVITY: DOT PRODUCT

- For two n-element vectors, we want to compute the dot product. Here, we will use *parallel_reduce*.
- Example: `a = [0 1 2 3 4 5 6 7 8 9], b = [10 9 8 7 6 5 4 3 2 1]`
- Result: `sum = 0*10 + 1*9 + 2*8 + 3*7 + 4*6 +5*5 + 6*4 + 7*3 + 8*2 +9*1 = 165`

Parallel implementation

- The dot product can be computed by dividing the vectors in several parts, computing the dot product for each part (in parallel) and then add the results for each part to get the whole sum.
- Reduction operation: Applying an operation on all members of group (e.g.: sum, max, min) and return a result.
  - ✓ The *parallel_reduce* template indicates the iteration space, as well as the object:

```
int main (int argc, char **argv) {
…
   // a,b: pointers to N-element vectors
   Dotp op(a,b); // the object 'op' is created with arguments ‘a’, ‘b’
   parallel_reduce (blocked_range<size_t> (0,N), op);
   cout << op.my_sum <<  “\n”;
…
```

  - ✓ The class `Dotp` specifies the details of the reduction (e.g.: how to accumulate subsums and combine them):

```
class Dotp {
  double  *my_a;
  double  *my_b;

public:
  double  my_sum;

  void operator () (const blocked_range<size_t> &r)  {
    double *a = my_a;
    double *b = my_b;
    double sum = my_sum;

    for (size_t i = r.begin(); i != r.end(); ++i)
      sum += a[i]*b[i]; //F(sum, a[i], b[i]); --> reduction function

    my_sum = sum;
  }

  Dotp (Dotp &x, split):  my_a (x.my_a), my_b (x.my_b), my_sum(0) {} // my_a = x.my_a, my_sum = 0
  void join (const Dotp &y) { my_sum += y.my_sum; }
  Dotp (double *a, double *b): my_a(a), my_b(b), my_sum(0) {} // my_a = a, my_b = b, my_sum = 0
};
```

    - ▫ When using objects with parameterized constructors, you might need to declare a group of them and then initialize each via the parameterized constructors. In the case of the functors, you can then execute the operation by calling the functor (in this example, this means calling *parallel_reduce*).
    - ▫ In some cases, it might be more convenient to declare an <u>array of objects with parameterized constructors</u>, which need to be initialized. The syntax in this case is a little bit different than expected. For example, to declare and initialize (via parameterized constructor) a group of objects of class `Dotp`, you can do:
      - `Dotp *op = (Dotp *) malloc (sizeof(Dotp)*10); // declare 10 objects of Dotp class`
      - `for (i=0; i<10; i++) op[i] = Dotp(x[i],y[i]); // x[i],y[i]: vectors. x,y: 2D arrays`
    - ▫ Then, if your object is a functor (e.g.: `Dotp`), you can execute its operation (here, it means calling *parallel_reduce*).

- Here, the function applied to `a[i]` and `b[i]` is: $f(a[i], b[i]) = a[i] \times b[i]$

- Application files: `dot_product.cpp`
  - ✓ Here, the dot product computation is implemented both in sequential and parallel (TBB) form.
  - ✓ In this example, `a` and `b` are vectors with `N` elements (initialized with random data). `N`: parameter of `main()`.
  - ✓ We use `using namespace tbb` to avoid including the prefix `tbb` before each identifier used by the `tbb` library.

- Compile this application: `g++ dot_product.cpp -ltbb -o dot_product ↵`
- Execute this application: `./dot_product <# of elements> ↵`
  - ✓ Example: `./dot_product 200000 ↵`

- We made extensive comparisons between the sequential and TBB implementations. The conclusion is that for this application, TBB implementation does not improve the processing time. For very large number of elements (~5 million) there was negligible improvement. A main reason for this might be that the reduction function (product) is very simple.

FIFTH ACTIVITY: GET THE MAXIMUM OF EACH MATRIX ROW

- For NV n-element vectors, we want to get the maximum value of each vector. Here, we will use *parallel_reduce* combined with *parallel_for*.
  - ✓ The NV vectors can be thought of as an array `A` with NV rows and n columns.
- Example:

```
A[0][0] = 0.25;  A[0][1] = 0.5;   A[0][2] = -3.2;  A[0][3] = -4.5; A[0][4] = -2.0;
A[1][0] = 2.0;   A[1][1] = 3.25;  A[1][2] = 5.75;  A[1][3] = 6.25; A[1][4] = 7.15;
A[2][0] = 0.25;  A[2][1] = -3.5;  A[2][2] = 0.25;  A[2][3] = 0.25; A[2][4] = 0.25;
A[3][0] = 2.0;   A[3][1] = 3.25;  A[3][2] = 0.75;  A[3][3] = -6.5; A[3][4] = 1.5;
A[4][0] = 3.0;   A[4][1] = -3.25; A[4][2] = 0.75;  A[4][3] = -6.5; A[4][4] = 1.5;
A[5][0] = -2.5;  A[5][1] = -3.25; A[5][2] = -0.15; A[5][3] = -6.5; A[5][4] = -1.5;
```

- Result: `R = [0.5 7.15 0.25 3.25 3.0 -0.15]’`

Parallel implementation

- The computations of the maximum value for each vector can be processed in parallel (via *parallel_for*).
- Within each vector, the result (maximum value) can be computed by dividing the vectors in several parts, computing the maximum for each part (in parallel) and then compare the results for each part to get the maximum value.
- Reduction operation: Here, the function applied to `A[i]` (a n-element vector) is: $f(A[i]) = \max(A[i])$
  - ✓ First, matrix `A` is initialized and a function (get max. value of a vector: `getmax_tbb`) is executed in a *parallel_for* loop:

```
int main() {
  size_t i;
  double **A, *m;
  size_t NV = 6; // # of vectors
  size_t N = 5; // length of each vector

  A = (double **) calloc (NV, sizeof (double *)); // allocating memory for 'NV' vectors
  for (i = 0; i < NV; i++) A[i] = (double *) calloc (N, sizeof (double));
  m = (double *) calloc (NV, sizeof(double));
  A[0][0] = 0.25;  A[0][1] = 0.5;  A[0][2] = -3.2; A[0][3] = -4.5; A[0][4] = -2.0;
  A[1][0] = 2.0;   A[1][1] = 3.25; A[1][2] = 5.75; A[1][3] = 6.25; A[1][4] = 7.15;
  A[2][0] = 0.25;  A[2][1] = -3.5; A[2][2] = 0.25; A[2][3] = 0.25; A[2][4] = 0.25;
  A[3][0] = 2.0;   A[3][1] = 3.25; A[3][2] = 0.75; A[3][3] = -6.5; A[3][4] = 1.5;
  A[4][0] = 3.0;   A[4][1] = -3.25; A[4][2] = 0.75; A[4][3] = -6.5; A[4][4] = 1.5;
  A[5][0] = -2.5;  A[5][1] = -3.25; A[5][2] = -0.15; A[5][3] = -6.5; A[5][4] = -1.5;

  parallel_for(int(0), int(NV), [&] (int i) {
       m[i] = getmax_tbb(A[i],N);           } );

  for (i = 0; i < NV; i++) printf ("m[%ld] = %4.4f\n",i,m[i]);
  for (i = 0; i < N; i++) free (A[i]); free(A); free (m);
  return 0;
}
```

  - ✓ In the function, the *parallel_reduce* template indicates the iteration space, as well as the object:

```
double getmax_tbb (double *a, size_t L)
  MyOp op(a);
  parallel_reduce (blocked_range<size_t> (0,L),op);
  return op.my_max;
}
```

  - ✓ The class `MyOp` specifies the details of the reduction (e.g.: how to get the maximum of each portion and combine them):

```
class MyOp {
public:
  double *my_a;
  double my_max;

  void operator() (const blocked_range<size_t> &r) {
     double *a = my_a;
     double max = my_max;
     // Get the maximum for a portion of the vector:
     for (size_t i = r.begin(); i!= r.end(); i++) { if (a[i] >= max) max = a[i]; }
     my_max = max;
  }

  MyOp (MyOp &x, split): my_a (x.my_a), my_max(x.my_a[0]) {}
  void join (const MyOp &y) { if (y.my_max > my_max) my_max = y.my_max; } //get max of two subresults
  MyOp (double *a): my_a(a), my_max(my_a[0]) {} // my_max = my_a[0]
  MyOp () {}
};
```

- Application files: `vm.cpp`
- Compile this application: `g++ -std=c++11 vm.cpp -ltbb -o vm ↵`
- Execute this application: `./dot_product ↵`

SIXTH ACTIVITY: ADD A GROUP OF VECTORS ELEMENT-WISE

- For NV n-element vectors, we want to add them all element-wise. Here, we will use *parallel_reduce* combined with *parallel_for*.
  - ✓ The NV vectors can be thought of as an array `A` with NV rows and n columns.

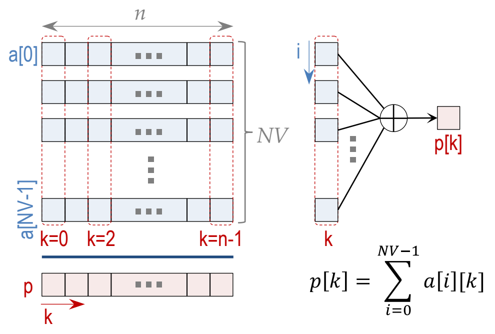
- Two approaches:
  - ✓ *parallel_for + parallel_reduce*: The n computations (from $k=0$ to $n-1$) can be performed in parallel. For every index, the computation is a sum of NV elements; this is a reduction. Fig. 3 illustrates the approach.
    - The class `SumFun` specifies the details of the reduction (how to accumulate subsums).

```
class SumFun {
   double *my_a;
public:
   double my_sum;

   void operator() (const blocked_range<int> &r) {
      double *a = my_a;
      double sum = my_sum;

      for (int i=r.begin(); i != r.end(); ++i)
          sum += a[i];
      my_sum = sum;
   }

   SumFun (double a[]): my_a(a), my_sum(0) {}
   SumFun (SumFun &x, split): my_a(x.my_a), my_sum(0) {}
   void join (const SumFun &y) {my_sum += y.my_sum; }
};
```



$$p[k] = \sum_{i=0}^{NV-1} a[i][k]$$

Figure 3. Parallel approach to add up NV vectors element-wise.

    - We use the *parallel_for* template to perform n computations concurrently. Iteration space: [0,n-1]. Each computation (sum of NV elements) is carried out via the *parallel_reduce* template (iteration space: [0,NV-1]).

```
double getsum_tbb (double **ai, int ki, int LC) {
   double t[LC]; // LC=nv
   for (int i = 0 ; i < LC; i++) t[i] = ai[i][ki];

   SumFun pf(t);
   parallel_reduce (blocked_range<int>(0,LC), pf);
   return pf.my_sum;
}

int main (int argc, char **argv) {
   const int n = 10; // number of elements per vector
   const int nv = 8; // number of vectors
   int i, j;
   double *p, **a;
   p = (double *) calloc (n,sizeof(double)); // setting data to 0's
   a = (double **) malloc (nv*sizeof(double *));
   for (i = 0; i < nv; i++)  a[i] = (double *) malloc (n*sizeof(double));

   // Assigning random data
   for (i = 0; i < nv; i++)
        for (j = 0; j < n; j++) a[i][j] = sin(i*0.1 + j *0.3);

   // Parallel Approach:
   parallel_for (blocked_range<int>(0,n), [&] (const blocked_range<int> r) {
         for (int k = r.begin(); k != r.end(); ++k)
               p[k] = getsum_tbb(a,k,nv);
   } );
   return 0;
}
```

  - ✓ *parallel_reduce*: Here, the reduction is applied to the summation of vectors; the final result is one vector. Iteration space: [0,NV-1].
    - Every step of the reduction adds up vectors. Fig. 4 shows an execution example for NV=4.
    - This is an odd example for illustrative purposes: in some applications, you need to reduce into one vector rather than just into one element.

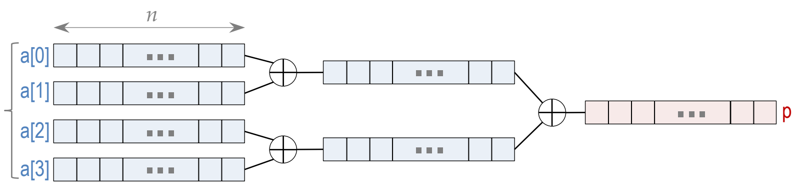


Figure 4. Parallel approach where a reduction adds up vectors resulting in one vector. Example with NV=4.

- The class `SumFun` specifies the details of the reduction (how to accumulate subsums).

```
class SumVec {
    double **my_hp;
    int nb; // size of output vector
public:
    double *my_sum;

    void operator () (const blocked_range<int> &r) {
        double *sum = my_sum;

        for (int i=r.begin(); i!=r.end(); ++i)
            for (int j=0; j < nb; j++)
                sum[j] += my_hp[i][j];

        my_sum = sum;
    }

    SumVec (double *h_i, double **hp_i, int nb_i) {
       my_sum = h_i; my_hp = hp_i; nb = nb_i;
    }

    SumVec (SumVec &x, split) {
       my_hp = x.my_hp; nb = x.nb; // all inputs should be included.

       // we must allocate its own my_sum
       my_sum = (double *) calloc (nb, sizeof(double)); // Setting data to 0's
    }

    void join (const SumVec &y) {
        for (int j=0; j < nb; j++)
           my_sum[j] = my_sum[j] + y.my_sum[j];

        free (y.my_sum); //freeing up my_sum when bodies are joined.
    }

};
```

- We use the *parallel_for* template to perform *n* computations concurrently. Iteration space: [0,n-1]. Each computation (sum of NV elements) is carried out via the *parallel_reduce* template (iteration space: [0,NV-1]).

```
double getsum_tbb (double **ai, int ki, int LC) {
   double t[LC]; // LC=nv
   for (int i = 0 ; i < LC; i++) t[i] = ai[i][ki];

   SumFun pf(t);
   parallel_reduce (blocked_range<int>(0,LC), pf);
   return pf.my_sum;
}

int main (int argc, char **argv) {
   const int n = 10; // number of elements per vector
   const int nv = 8; // number of vectors
   int i, j;
   double *p, **a;
   p = (double *) calloc (n,sizeof(double)); // setting data to 0's
   a = (double **) malloc (nv*sizeof(double *));
   for (i = 0; i < nv; i++)  a[i] = (double *) malloc (n*sizeof(double));

   // Assigning random data
   for (i = 0; i < nv; i++)
        for (j = 0; j < n; j++) a[i][j] = sin(i*0.1 + j *0.3);

   // Parallel Approach:
   SumVec sf(p,a,n);
   parallel_reduce(blocked_range<int>(0,nv), sf);
   return 0;
}
```

- Application files: `sumvecs.cpp`
- Compile this application: `g++ -std=c++11 sumvecs.cpp -ltbb -o sumvecs ↲`
- Execute this application: `./sum_vecs ↲`

- To test for memory issues, use the valgrind program.

```
g++ -Wall -std=c++11 sumvecs.cpp -ltbb -g -o sumvecs
valgrind ./sumvecs
```