OpenCL and UPC

Anastasia Kruchinina* Karl Ljungkvist†

March 21, 2014

Part I
OpenCL

1 Introduction

Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms, such as CPU, GPU, DSP and others. In this report we analyze performance of OpenCL on the GPU, which is a throughput oriented many core processor that offer very high peak computational throughput. All tests was performed using NVIDIA Tesla M2050 (see Table 1) for both double and float precision. The minimum time of 3 runs was taken as a result.

Max. clock frequency:	1147
Global memory size:	2817982464
Local memory size:	49152
Max. compute units:	14
Max. work item sizes:	(1024, 1024, 64)

Table 1: NVIDIA Tesla M2050

Note that in order to be able to use double precision one should enable corresponding OpenCL extension:

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
```

The code for every presented algorithm can be found on the following github repository: `https://github.com/Tourmaline/PPSC`.

2 Vector operations

2.1 Dot product

The dot product of two vectors is pretty straightforward operation in OpenCL and can be implemented using reduction. The size of the vector in the kernel is

*anastasia.kruchinina@it.uu.se
†karl.ljungkvist@it.uu.se

bounded by the global memory. For larger dimensions vector should be divided on parts. The kernel consists of two steps. First is elementwise product of two vectors and second is reduction inside each block. Kernel returns to the host sum for each block and final result is computed on the host. The number of blocks is not big, therefore it is fast to compute final sum directly on the CPU.

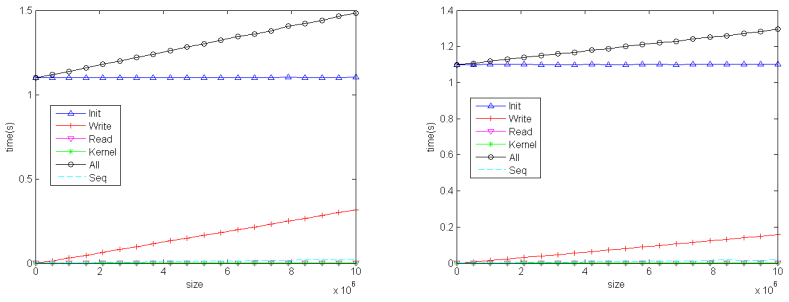
There were implemented two kernels for the dot product. The first kernel vector's elements are `double/float` and in the second kernel elements are packed into `double4/float4` in order to utilize SIMD functionality. Both version we refer as non-vectorized and vectorized, respectively. Also the SIMD function `dot` was used to compute the dot product between two `double4/float4` elements (see Listing 1). Second optimization was `#pragma unroll` on reduction loop, but it did not add any speed up.

```
//1 verson
while( i < N )
{
local_res[itemIDlocal] += x[i]*y[i];
i += globalSize;
}

// 2 version
while( i < N )
{
local_res[itemIDlocal] += dot(x[i], y[i]);
i += globalSize;
}
```

Listing 1: Dot product: main part of kernels for both versions

In the Figure 1 is plotted time for different parts of the OpenCL program for the dot product without SIMD operations for both double and single precision.



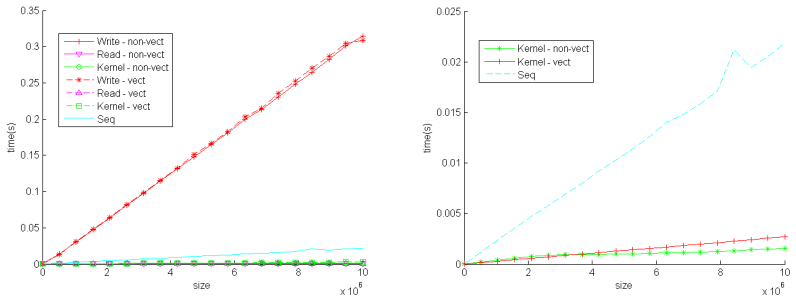
(a) Double precision (b) Single precision

Figure 1: Time for dot product function: all - total time, init - initialization of OpenCL, kernel - kernel time, write - transfer from CPU to GPU, read - transfer from GPU to CPU.

Plot for the second version is similar. Initialization time (*Init*) is the time which is spending for creating context, queue, OpenCL program and compiling kernel. One can see that it takes a little bit more than 1 second. It is possible to decrease initialization time using pre-built binaries instead of source code of kernel. *Write* and *read* measure transfer of the data from CPU to GPU and from GPU to CPU, respectively. Dot product of two vectors has low computational intensity, therefore it is bandwidth bounded, so one cannot expect to achieve

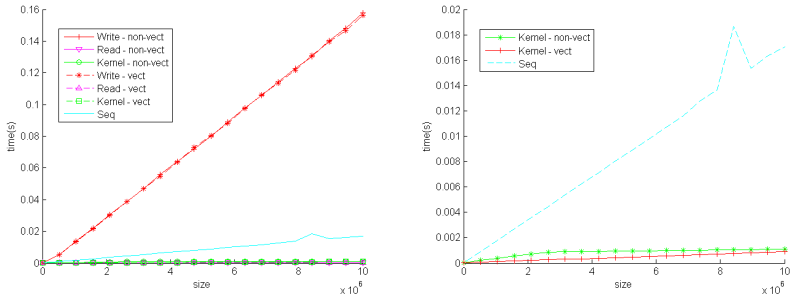
peak performance. Therefore initialization and writing are the most consuming parts of the program.

In the Figure 2 and 3 is presented comparison between two versions of kernel for double and single precisions.



(a) Read/write and kernel times (b) Kernel times and sequational time

Figure 2: Comparison between two versions of dot product with double precision: vect - using SIMD instructions, non-vect - without SIMD instructions.



(a) Read/write and kernel times (b) Kernel times and sequational time

Figure 3: Comparison between two versions of dot product with single precision: vect - using SIMD instructions, non-vect - without SIMD instructions.

For second versions we sent data from the CPU as array of `double/float`, but on the GPU it is interpreted as array of `double4/float4`. In this Figures one can see that there is no overhead of such reinterpretation of elements. The second plot is comparison between kernel times and sequential version. For the sequential version happens a jump in time when vector size reaches 8.5 million of elements, which is probably related to the cache size. In the figure one can see that in this case the vectorized version is a little bit faster.

2.2 Vector update

Here we again implemented two versions of the kernel. One is operate with vectors with `double/float` elements, and another with their SIMD versions: `double4/float4` (see Listing 2). Also we tried to use `fma` (Fused Multiply-Add). As described in the specification, `fma` returns the correctly rounded

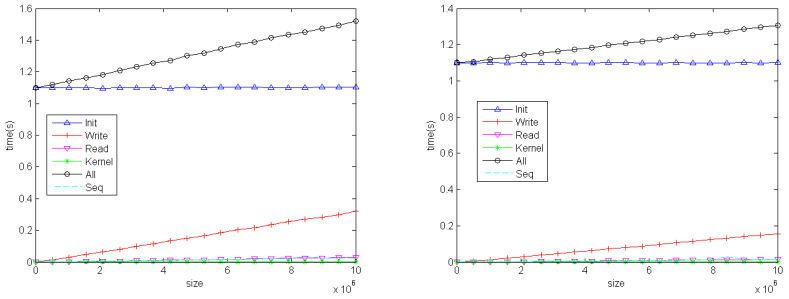
floating-point representation of the sum of c with the infinitely precise product of a and b.

```
// 1 version
while( i < N )
  {
    x[i] += y[i]*alpha;
    i += globalSize;
  }

  // 2 version
    while( i < N )
  {
    x[i] = fma(y[i], alpha, x[i]);
    i += globalSize;
  }
```

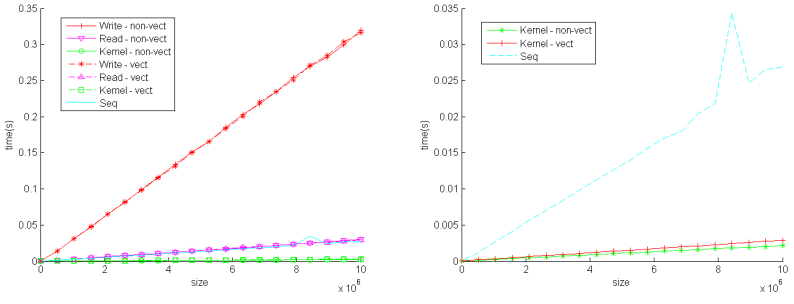
Listing 2: Vector update: main part of kernels for both versions

In the Figure 4 are presented times for every part of the program. The meaning of every part is described in the dot product section. Again, here one can see then vector update has low computational intensity, so the most time is spending for the transfer of data and initialization.



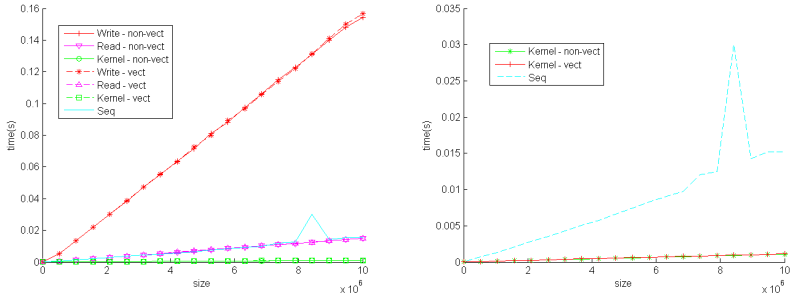
(a) Double precision (b) Single precision

Figure 4: Time for vector update function: all - total time, init - initialization of OpenCL, kernel - kernel time, write - transfer from CPU to GPU, read - transfer from GPU to CPU.



(a) Read/write and kernel times (b) Kernel times and sequational time

Figure 5: Comparison between two versions of vector update with double precision: vect - using SIMD instructions, non-vect - without SIMD instructions.



(a) Read/write and kernel times (b) Kernel times and sequational time

Figure 6: Comparison between two versions of vector update with single precision: vect - using SIMD instructions, non-vect - without SIMD instructions.

In the Figure 5 and 6 is compared both versions of kernel. In contrast to the dot product non-vectorized version of kernel perform better than vectorized.

2.3 Vector operations on the GPU

For both vector operations kernel times are less then sequential. But because of the large initialization time and slower data transfer between CPU and GPU, it is not worth to compute just one vector operation on the GPU. Such computations can be useful as part of some bigger program, which operation with larger amount of data and when vectors are already stored in the GPU memory.

3 Matrix operations

3.1 Dense matrix matrix multiplication

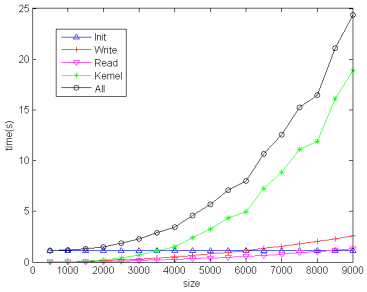
For the matrix multiplication was used *tiled* algorithm, where matrices are divided to the blocks and blocks are multiplied by the corresponding work groups using the shared memory. In the Figure 7 is plotted time which program spent on different parts of the program.

In comparison with previous vector operations, matrix multiplication performance is compute-bound, and one can see that the kernel execution is the largest part of the execution time of the whole program. The kernel time is presented in Figure 8.

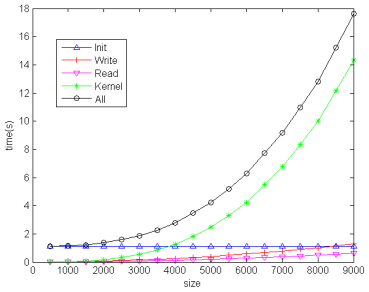
3.2 Sparse matrix vector multiplication

Dense operations are quite regular and they are limited by the floating-point throughput. Sparse operations instead are much less regular and generally limited by bandwidth.

In our implementation we used CSR (Compressed Sparse Row) format. Each warp (wavefront) gets one row of the matrix and each work item computes part of the result value for the corresponding element of the result vector. The access to the memory for the matrix is coalesced. Instead for the vector there is an irregular memory access. Threads in warp can read values from the vector in

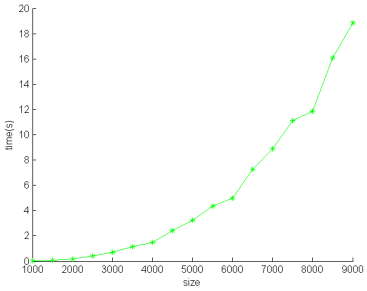


(a) Double precision

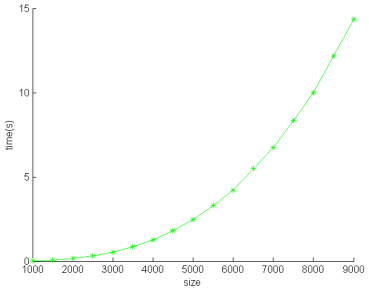


(b) Single precision

Figure 7: Time for dense matrix matrix multiplication function: all - total time, init - initialization of OpenCL, kernel - kernel time, write - transfer from CPU to GPU, read - transfer from GPU to CPU.



(a) Double precision



(b) Single precision

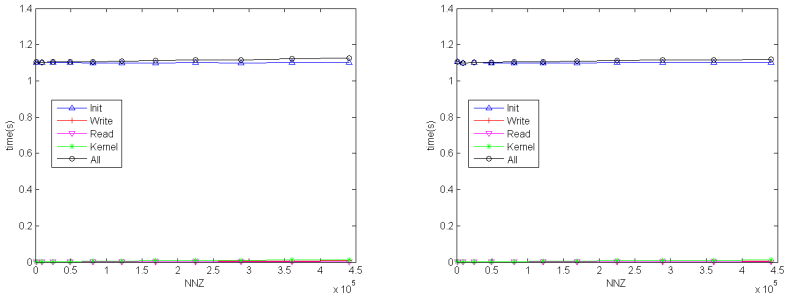
Figure 8: Kernel time for dense matrix matrix multiplication.

any position. After each work item computes local sum, it stores it in the shared memory. After all warps in work group are finished, the final value of the result vector is computed using reduction. Warps are implicitly synchronized, so one does not need explicit synchronization.

If number of nonzero elements per row vary a lot, then the kernel will not be fully utilized. Some of threads will finish their work faster. Therefore, CSR format is good for matrices with equally distributed elements per row. In the Figure 9 is presented time for different parts of the program and in the Figure 10 the kernel time.

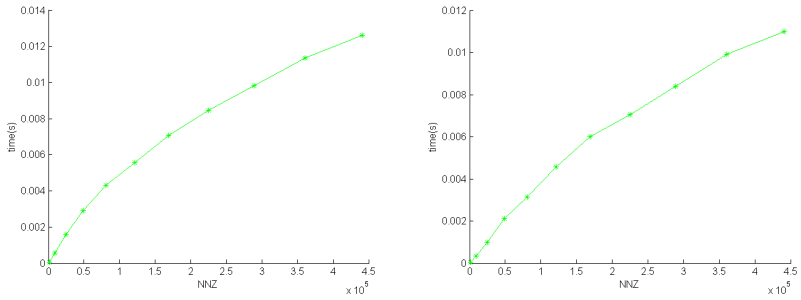
4 General remarks about OpenCL

- OpenCL is C99 based.
- Friendly user interface, easy to learn if user familiar with CUDA.
- The source file is long but straightforward and can be reused.
- Platform and OS independent.
- User can specify dimensions of the NDRange in the number of work items (instead for example in work groups as it is done in CUDA), what gives



(a) Double precision (b) Single precision

Figure 9: Time for sparse matrix vector multiplication function: all - total time, init - initialization of OpenCL, kernel - kernel time, write - transfer from CPU to GPU, read - transfer from GPU to CPU.



(a) Double precision (b) Single precision

Figure 10: Kernel time for sparse matrix vector multiplication.

more flexibility in specifying boundary conditions.

- Hardware vendors are responsible for the implementation of OpenCL.
- Kernel is a string and must be compiled at runtime (although here is a possibility to use binaries from previous run), so the kernel code cannot be integrated into the host program. It gives additional overhead, but ensures portability.

Part II
UPC

Developed at University of California at Berkeley, *Unified Parallel C*, or UPC, is language extension of standard C for parallel computations on large-scale distributed computer system[1]. Being a *Partitioned Global Address Space*, or PGAS, language its aim is to provide a shared-memory model for distributed-memory computer systems.

UPC is similar to MPI[2] in that applications are *single-program multiple-data*, or SPMD, meaning that each computational process will run the same code, but operating on different data. What UPC adds to this is distributed arrays. Such arrays are declared as *shared*, and are distributed over the processing units, i.e., each thread owns a piece of the array, but each thread can directly read the parts owned by others. In Listing 3, a shared array `A` of 1000 `double`s is declared.

Listing 3: Shared array in UPC

```
double shared A[1000];
```

For an introduction of UPC, see the tutorial from Super Computing 2009[3].

5 Assignments

For this first application, the implementation process will be described in detail, to give a feeling for how to program in UPC. Many of the techniques used here can be considered standard, and will be reused in the other applications, but then without detailed explanation. The full source code can be found in an archive at `http://user.it.uu.se/~karlj502/files/upc.zip`.

5.1 Vector update

The first application was a vector update,

$$y := y + \alpha x\,,$$

for which the reference code in Listing 4 was used as a starting point.

Listing 4: Reference code for the vector update

```
double x[SIZE], y[SIZE];
for(int i=0; i<SIZE; i++)
   y[i] += alpha*x[i];
```

Because of the fact that this is a perfectly parallel application without dependencies, it was parallelized in a straight-forward way by making the arrays `x` and `y` shared, and having each thread compute the update on its part of the arrays, by replacing the `for` loop by a `upc_parfor` equivalent (see Listing 5). The additional fourth clause gives the locality of any iteration. In this case, `i`,

[1] `http://upc.gwu.edu/`
[2] `http://www.open-mpi.org/`
[3] `http://www2.hpcl.gwu.edu/pgas09/tutorials/upc_tut.pdf`

means that iteration `i` will be performed by thread number `i`. Another possibility is to specify an address, such as `&y[i]`, in which case the iteration will be performed by the thread owning the data at that address. In our case, these two are equivalent.

Listing 5: The most simple UPC parallelization of a vector update

```
shared double x[SIZE];
shared double y[SIZE];
upc_parfor(int i=0; i<SIZE; i++; i)
   y[i] += alpha*x[i];
```

However, this yielded very bad performance. The reason for this is that `x` and `y` are shared and their address needs to be resolved at each access. However, since we know that a process will only access its own data, they can simply be replaced by a *private* pointer, i.e. a regular C pointer.

For this, a contiguous memory layout within the processes is necessary, whereas in the default memory layout of UPC, elements are cyclically distributed one by one. This can be changed by explicitly specifying the block size (shown in Listing 6).

Listing 6: Setting the blocksize in UPC

```
#define BKSIZE (SIZE/THREADS)
shared [BKSIZE] double x[SIZE];
shared [BKSIZE] double y[SIZE];
```

Now each thread will only read from a single contiguous piece of memory. Then, local pointers can be introduced by casting the correct memory address to a private pointer (see Listing 7).

Listing 7: Casting shared pointers to private ones in UPC

```
double *xp = (double *)&x[MYTHREAD*BKSIZE];
double *yp = (double *)&y[MYTHREAD*BKSIZE];
```

Combining all of this, the implementation shown in Listing 8 is obtained.

Listing 8: UPC parallelization of the vector update

```
#define BKSIZE (SIZE/THREADS)
shared [BKSIZE] double x[SIZE];
shared [BKSIZE] double y[SIZE];
... // Setup etc.
double *xp = (double *)&x[MYTHREAD*BKSIZE];
double *yp = (double *)&y[MYTHREAD*BKSIZE];
for(int i=0; i<BKSIZE; i++)
   yp[i] += alpha*xp[i];
```

This implementation performed roughly similar to a corresponding OpenMP parallelization (see results under Sec. 6).

5.2 Scalar product

Due to the similarity to the vector update, a similar approach was used here. Also here, the vectors are distributed in blocks to the threads, and private pointers are used for `x` and `y`. For the output `alpha`, care must be taken to avoid race conditions since each thread has to add its contribution to a single global result. Three different approaches were used to handle this:

Global lock: In the first method, a global lock is used to achieve mutually exclusive access to the shared result variable `alpha`. Each thread computes its contribution by updating a private variable `alpha_priv`, and when it is ready it acquires the lock, adds its result to the global result, and releases the lock. Listing 9 shows the code for this method.

Listing 9: Scalar product using a global lock

```
xp = (double *)(x+MYTHREAD*BKSIZE);     // Private pointers
yp = (double *)(y+MYTHREAD*BKSIZE);

if(MYTHREAD==0) alpha = 0.0;            // Reset the global result
upc_barrier;

double alpha_priv = 0.0;                // Create local variable
for(int i=0; i<BKSIZE; i++)
  alpha_priv += xp[i]*yp[i];            // Compute my contribution

upc_lock(lock);                         // Acquire lock
alpha += alpha_priv;                    // Add contribution
upc_unlock(lock);                       // Release lock
```

Reduction: In the second method, a shared array with one element per thread is used to store the contributions. In other words, each thread stores its contribution at its position in a the shared array. Then, when each thread has finished computing, a single step performs the reduction by adding up all the individual contributions. The code for the reduction version can be seen in Listing 10.

Listing 10: Reduction for the scalar product

```
alpha_priv = (double *) (alpha+MYTHREAD); // Initialize private
xp = (double *)(x+mybk*BKSIZE);           //   pointers
yp = (double *)(y+mybk*BKSIZE);

*alpha_priv = 0.0;                        // Reset my contribution
for(int i=0; i<BKSIZE; i++)
  *alpha_priv += xp[i]*yp[i];             // Compute my result

upc_barrier;                              // Wait for everyone to
if(MYTHREAD==0){                          //   be done
  for(int i = 1; i < THREADS; ++i)        // Sum up contributions
    alpha[0] += alpha[i];
}
```

Parallel reduction: The third method is a variation on the second, where the reduction phase is performed in parallel. Here, every other thread sums up the contributions from itself and its successor, then every other of those threads computes sums, etc (see code in Listing 11).

Listing 11: Parallel reduction for the scalar product

```
for(int s = 1; s < THREADS; s*=2) {
  upc_forall(int i=0; i<THREADS; i+=(s*2); &alpha[i])
    *alpha_priv += alpha[i+s];
  upc_barrier;
}
```

5.3 Matrix multiplication

Here, the following implementation was taken as a starting point for the parallelization:

Listing 12: Matrix-matrix multiplication reference implementation

```
for(int i=0; i<SIZE; i++) {
  for (int j=0; j<SIZE; j++)                    // Reset row
    C[i*SIZE+j] = 0.0;
  for (int k=0; k<SIZE; k++)                     // Compute row
    for (int j=0; j<SIZE; j++)
      C[i*SIZE+j]+=A[i*SIZE+k]*B[k*SIZE+j];
}
```

This loop order was found to perform the best, probably since now two matrices are indexed row-wise in the innermost loop, compared to only one for the standard `ijk` ordering. This was parallelized by splitting the `i` loop over the number of threads, thus making each thread responsible for a specific set of rows. Correspondingly, all matrices are split row-wise over the processors.

Once again private pointers were introduced, this time for matrices `A` and `C`, since each thread only access a block of rows for them. Private pointers cannot be used to access the matrix `B`, since the whole matrix is being read by each thread. A way to improve the performance in this case is to duplicate the whole `B` in the local memory of each thread. This is achieved by a number of `upc_memget` calls (multiple calls are necessary since the data being copied must belong to a single thread). This, together with the code for the actual multiplication, can be seen in Listing 13.

Listing 13: Efficient matrix-matrix multiplication in UPC

```
for(int j = 0; j < NBLOCKS; ++j)            // Copy B to local
  upc_memget(&Bp[j*(SIZE*SIZE/NBLOCKS)],    //   memory
             &B[j*(SIZE*SIZE/NBLOCKS)],
             (SIZE*SIZE/NBLOCKS)*sizeof(double));

upc_forall(int i=0; i<SIZE; i++; &C[i*SIZE+0]) {
  Ap = (double *) &A[i*SIZE];               // Setup private
  Cp = (double *) &C[i*SIZE];               //   pointers

  for (int j=0; j<SIZE; j++)                // Reset row
    Cp[j]=0.0;

  for (int k=0; k<SIZE; k++)                // Compute row
    for (int j=0; j<SIZE; j++)
      Cp[j]+=Ap[k]*Bp[k*SIZE+j];
}
```

5.4 Sparse matrix-vector product

The parallelization of the SpMV operation is in many respects similar to the one for the matrix-matrix product. Also here, the output is blocked row-wise and each thread is responsible for computing the results of a its own rows. A common storage format for a sparse matrix is the *compressed sparse row* – or CSR – format, where the non-zeros of the matrix are stored in the floating-point array `val`, the column indices of the non-zeros are stored in the integer array

`ind`, and the staring index of each row in `val` and `ind` are stored in the integer array `ptr`. However, if such a data structure is to be distributed in the same way as the dense matrices in the previous section, then the size of the chunks will not be the same for each thread, since number of non-zeros per row is not the same for each row. Having such a variable block size of a shared array is not possible in UPC – only a constant block size is permitted. For this reason, the related ELL format was adopted, where the rows in `ind` and `val` were padded to the same length. In `ind`, the padding value is a specific invalid value, which was chosen to be -1, whereas in `val` any value can be used, since they will never be used anyways. Now, since the rows of `ind` and `val` are of equal length, `ptr` is rendered superfluous and is replaced by the constant `PADDEDROWSIZE`, which is equal to the largest number of non-zeros in any row. The ELL multiplication algorithm can be seen in Listing 14.

Listing 14: ELL matrix-vector multiplication

```
for(int i=0; i<SIZE; i++) {
  y[i] = 0.0;
  for(int j = i*PADDEDROWSIZE; j < (i+1)*PADDEDROWSIZE ; ++j) {
    int col_idx=ind[j];
    if( col_idx >= 0)
    y[i] += val[j]*x[col_idx];
  }
}
```

Now the row-wise blocking and parallelization can be performed straightforwardly. Similarly to the other applications, private pointers are used for `y`, `val` and `ind` since these are only read locally. The whole `x` is (possibly) read by all threads, so it is copied into the local memory.

5.5 Gram-Schmidt

For UPC, the extra assignment was the parallelization of the Gram-Schmidt procedure, which is an algorithm for finding an orthogonal basis $\{q_i\}$ of a set of m linearly independent n-dimensional vectors $\{v_i\}$. The original algorithm – Classical Gram-Schmidt – can be seen in Alg. 1.

Algorithm 1 Classical Gram-Schmidt

1: **for** $i = 1 \dots n$ **do**
2: $\quad w_i = v_i$
3: $\quad$ **for** $j = 1 \dots (i-1)$ **do**
4: $\quad\quad w_i = w_i - (v_i \cdot q_j) q_j$
5: $\quad$ **end for**
6: $\quad q_i = \frac{w_i}{||w_i||}$
7: **end for**

However, this is numerically unstable since the projection in line 4 introduces errors. This is solved in the Modified Gram-Schmidt procedure (Alg. 2) where each successive result is used when computing the next projection.

Two ways of parallelizing Alg. 2 have been studied. The first and most straight-forward one is to parallelize each of the vector operations separately – the scalar product, the vector update, and the norm computation. Here we

Algorithm 2 Modified Gram-Schmidt, version 1

1: **for** $i = 1 \dots n$ **do**
2: **for** $j = 1 \dots (i-1)$ **do**
3: $v_i = v_i - (v_i \cdot q_j)q_j$
4: **end for**
5: $q_i = \frac{v_i}{||v_i||}$
6: **end for**

simply reuse the efficient parallelizations developed in Sec. 5.1 and 5.2. This parallelization uses a lot of synchronization, both from locks in the dot product and norm computations, and from the barriers which separate different parallelization batches.

The other option is to try a more coarse parallelization. It would appear that all the operations in Alg. 2 are dependent on each other, but if the operations are reordered slightly, it is clear that this is not the case. As soon as the basis vector q_i has been computed, the projection on q_i can be subtracted from all the following vector v_j. In other words, the i'th step of all the inner loops can all be performed independent of each other. Using this to rewrite Alg. 2, one obtains Alg. 3.

Algorithm 3 Modified Gram-Schmidt, version 2

1: **for** $i = 1 \dots n$ **do**
2: $q_i = \frac{v_i}{||v_i||}$
3: **for** $j = (i+1) \dots n$ **do**
4: $v_j = v_j - (v_j \cdot q_i)q_i$
5: **end for**
6: **end for**

As noted above, for a given i iteration, all the iterations of the j loop can be performed in parallel. This leads to the second parallelization, where whole vectors are distributed cyclically between the processors. Then, loop step number j will be performed by the thread to which vector v_j belongs. Two things must be taken care of. Firstly, the normalization on line 2 concerns a single vector which will reside in a single thread. This is solved by letting the owning thread perform the calculation. Secondly, all threads will need read the basis vector q_i from the current i loop iteration. To speed up this, the whole q_i is fetched into the local memory. Finally, private pointers are utilized to speed up accesses to local parts of shared data. The resulting code can be seen in Listing 15.

Listing 15: Coarse parallelization of MGS

```
for(int i=0; i<M; i++){
  upc_barrier;                              // wait for previous
                                            // iteration to be done
  if(MYTHREAD == i%THREADS) {
    vp = (double *)&V[i*N];                 // setup local pointers
    qp = (double *)&Q[i*N];

    double temp_norm = vecNorm(vp,N);       // compute norm
    for (int k=0; k<N; k++)                 // normalize q_i
      qp[k] = vp[k]/temp_norm;
```

```
  }

  upc_barrier;                              // wait for q_i to be done
  upc_memget(qcpy,&Q[i*N],                  // then fetch it
             N*sizeof(double));

  upc_forall(int j=i+1;j<M;j++;j) {         // subtract projection
    vp = (double *)&V[j*N];                 //  of v_j on q_i

    double sigma = 0;
    for(int k=0; k<N; k++)
      sigma += qcpy[k]*vp[k];               // compute scalar product

    for(int k=0;k<N;k++)
      vp[k] -=sigma*qcpy[k];                // subtract projection
  }
}
```

Both these parallelizations have their advantages. The coarse parallelization has much less overhead since the amount of synchronization is small, and parallel sections are comparatively large. On the other hand, if the number of vectors is small, the parallelism will be low whereas the fine-grained parallelization will perform the same independently of number of vectors. Typically, MGS is used in iterative Krylov solvers, where the vectors are usually quite few and very large (the size is equal to the size of the problem being solved, and their number is equal to the dimensionality of Krylov subspace). Also, since MGS is typically part of a larger algorithm, the vectors v_i and q_i already have some distribution, usually a blocked distribution like in the fine-grained parallelization. In these cases, performing the redistribution might not perform well, especially on a distributed system, and when the vectors are large.

6 Experiments

The test system was the Uppmax cluster Tintin, which consists of nodes with two AMD Opteron 6220 CPUs. Furthermore, each node has 64GB RAM each, and is connected by a QDR Infiniband network. The system is Scientific Linux 6.5 with Linux kernel version 2.6.32. The GNU UPC compiler was used, since this was most convenient to use, but the performance when compiling with the Berkeley UPC compiler was similar. All experiments were run on the four nodes, on 1–64 processes in powers of 2 (i.e., $1, 2, 4, \ldots$). Furthermore, to get a more stable runtime estimate, each application is run 10 times, and the minimum time is recorded.

For comparison, each of the standard applications – vector update, scalar product, matrix-matrix multiplication and sparse matrix vector product – were compared to straight-forward but efficient OpenMP parallelizations. For the Gram-Schmidt application, an efficient serial C implementation was used for comparison. Also, for vector update, scalar product and matrix-matrix multiplication, the runtime for the corresponding single-threaded Matlab built-in functions are included.

6.1 Vector update

The vector update was performed for vectors of three different sizes, 1024000, 10240000 and 102400000. The results can be seen in Figures 11 – 13.

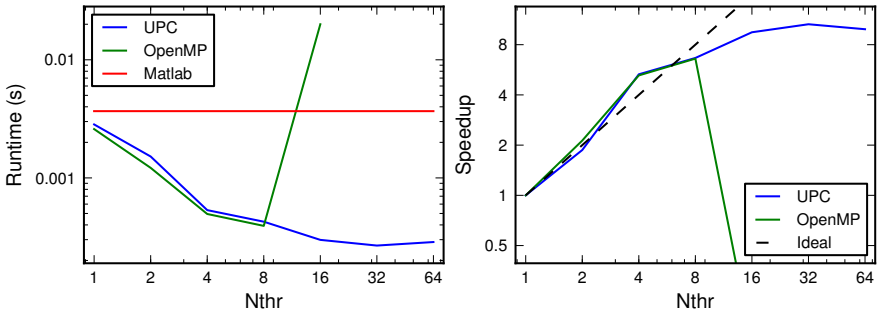


Figure 11: *Vector update with vector size* 1024000

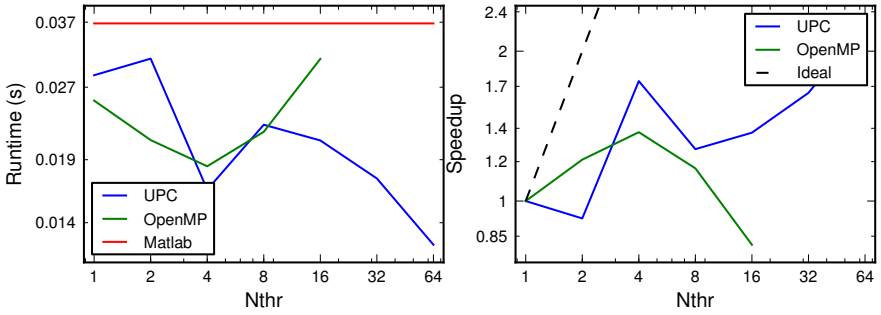


Figure 12: *Vector update with vector size* 10240000

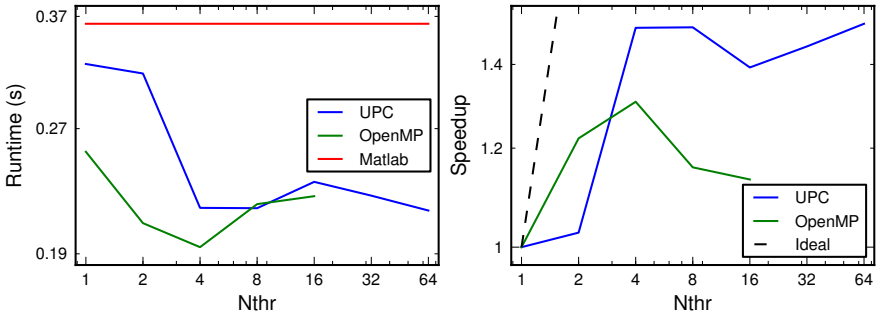


Figure 13: *Vector update with vector size* 102400000

For the smallest vector, we see that UPC performed more or less the same as the OpenMP version, except for the drastic slowdown at 16 threads for the OpenMP version (see Fig. 11). UPC on the other hand continues to improve all the way up to 32 threads.

For the larger vectors, a much more bumpy picture appears. Most importantly, both UPC and OpenMP shows almost no scaling at all. An explanation

for this can be found by looking at Table 2, where the memory footprint of the three vector sizes is listed. Here, it is clear that the two larger problems are far too big to fit in any cache, and since most of the bandwidth to the DRAM is available already for a single thread, we cannot expect any speedup.

On the other hand, for the smallest problem, the data will fit in the cache system (2*8MB L3 cache per processor). Since the cache is distributed over the cores, adding cores makes more cache available, giving a speedup which is even super linear.

Table 2: *Data footprint of the vector update application for the different problem sizes.*

Elements	Memory
1024000	15.6 MB
10240000	156 MB
102400000	1.56GB

The problem with OpenMP performing exceptionally slow at 16 threads was repeated for all applications where cache was essential and can be explained by the fact that OpenMP does not pin threads to cores. This might cause threads to migrate around and thus being moved away from their warmed up caches, leading to thrashing. Because of the twin eight-core setup, this becomes apparent only at 16 threads.

We also note that Matlab performed exceptionally poor here. A possible explanation for this is the fact that relatively verbose code is necessary for this operation, and maybe Matlab has trouble performing this at the speed of native compiled code.

6.2 Scalar product

Also in the scalar product application, the same three vector sizes were used – 1024000, 10240000 and 102400000. The results can be seen in Figures 14 – 16.

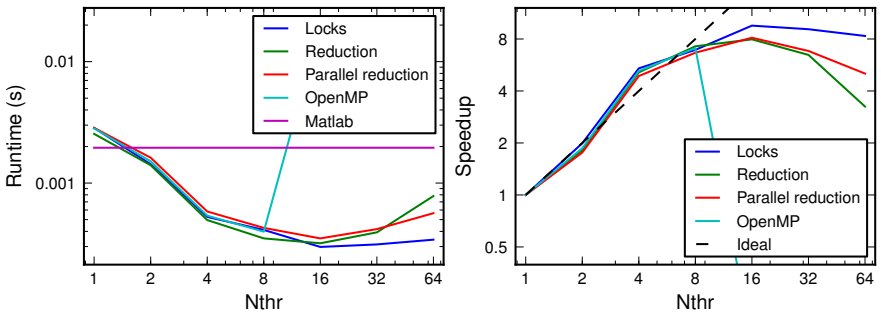


Figure 14: *Scalar product with vector size* 1024000

Here, the performance was very similar to the previous application. Also here, there is enormous speedup at the smallest vector size, whereas the larger problems are almost without any speedup. We also see that there was not much to gain by using a more clever reduction algorithm, since the lock-based

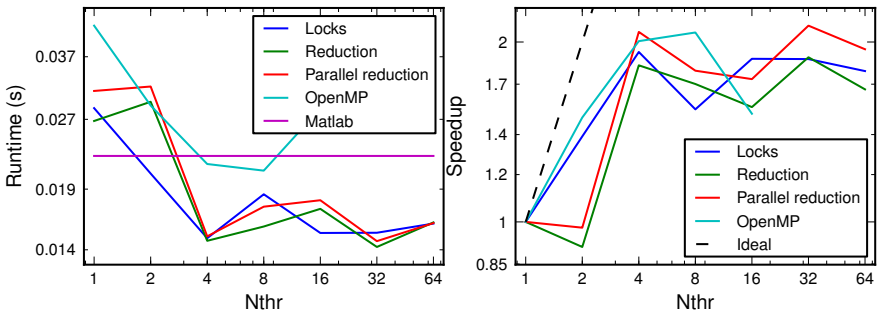


Figure 15: *Scalar product with vector size* 10240000

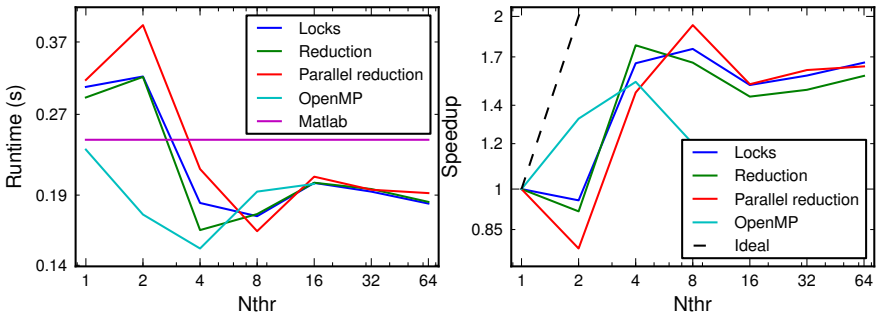


Figure 16: *Scalar product with vector size* 102400000

implementation performed equally or better than the more advanced implementations. Here, a more reasonable performance was obtained in Matlab.

6.3 Matrix-matrix product

For the matrix-matrix multiplication application, matrices of size 256×256 to 2048×2048 were used. In Figures 17, 18, and 19, we see the results for sizes 256×256, 1024×1024 and 2048×2048, respectively.

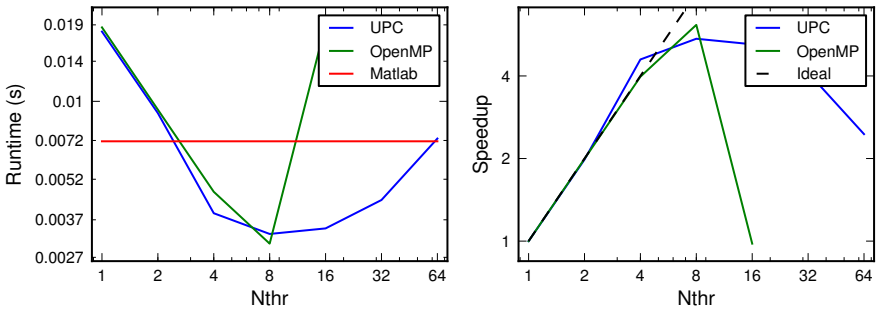


Figure 17: *Matrix-matrix multiplication with matrices of size* 256×256

For the two smaller-sized problems, just as for the previous applications, we see a somewhat super-linear scaling due to increased utilization of the cache.

However, since the computational intensity is considerably higher for the

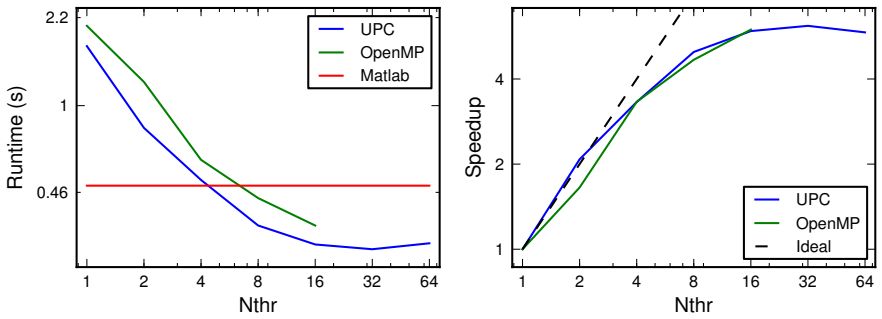


Figure 18: *Matrix-matrix multiplication with matrices of size* 1024×1024

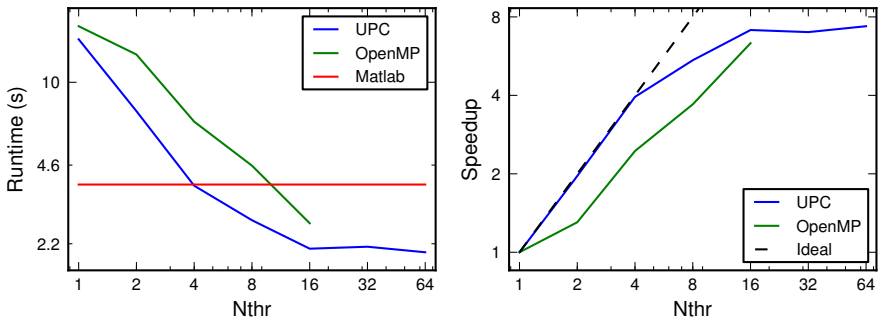


Figure 19: *Matrix-matrix multiplication with matrices of size* 2048×2048

matrix-matrix multiplication than for the previous applications, problems which do not fit in the cache will also see a benefit from adding cores, since more computations can be done.

We see that the difference between OpenMP and UPC grows with the problem size, implying that UPC has a lower overhead. However, it is not possible to rule out other contributing factors such as the fact that dynamically allocated arrays were used for the OpenMP version, whereas arrays in the global memory was used for the UPC version.

Finally, when reaching a multi-node configuration (i.e. 32 or 64 processors), the UPC performance stagnates noticeably, although there in a few cases was a small reduction in runtime.

6.4 Sparse matrix-vector product

The sparse matrix-vector multiplication application was run on a large variety of matrices with different sparsity and structure. Firstly, random unstructured $N \times N$ matrices of two types were used; a denser one with density 0.1, and a sparser one with density 0.001. For the denser matrix, values of N from 16 to 4096 in powers of 4 were used (i.e. 16, 64, 256, ...). For the sparser matrix, N was 64, 256, ..., 65536. Secondly, a structured finite-difference approximation matrix of a 2-D Laplacian operator was used, with $N = 16, 64, \ldots, 262144$. Such a matrix is a five-diagonal and is thus even more sparse than both the unstructured matrices. Figures 20, 21 and 22, show the results for the largest matrix of each kind.

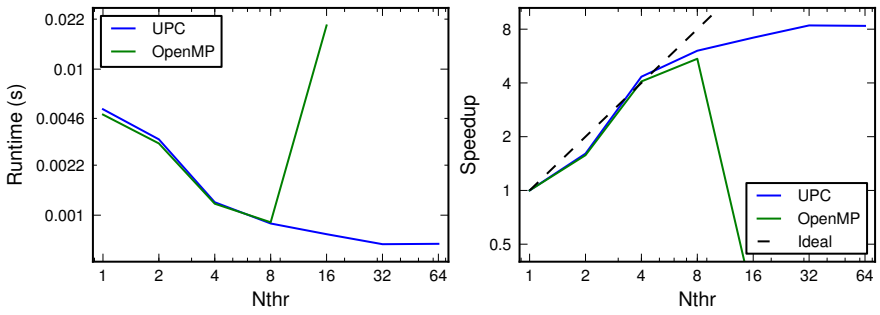


Figure 20: *Sparse matrix-vector multiplication using an unstructured, random matrix with size* 4096×4096 *and density* 0.1

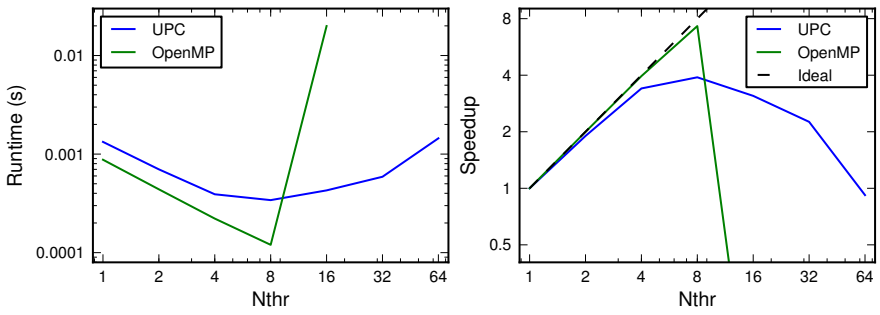


Figure 21: *Sparse matrix-vector multiplication using an unstructured, random matrix with size* 65536×65536 *and density* 0.001

Just as the first two applications, this an operation that is memory bound. However, all of the matrices have a size in the order of the cache size (10 - 32 MB), so there is still a potential for a speedup.

For the most dense matrix, a somewhat decent scaling was obtained, although performance once again stagnates for the multi-node configuration. For the sparser matrices, OpenMP continues to scale almost linearly, whereas the UPC implementation stops scaling at about 8 threads. We see that in both of the sparser cases, UPC is slower than OpenMP. A factor which contributes to this is the fact that different sparse-matrix formats are used – CSR for OpenMP versus ELL for UPC. Probably, the additional overhead and memory footprint of the ELL format, cause the longer execution.

Once again, the failure of OpenMP at 16 threads is evident.

6.5 Gram-Schmidt

In the final application – the Modified Gram-Schmidt procedure – two classes of problems were solved; a) dense problems where the number of vectors is equal to the dimensionality, and b) problems where the vector size is considerably larger than the number of vectors. Figure 23 shows the result of the problem of type a) with 512 vectors of size 512; while Figures 24 and 25 show the result of the problem of type b) with 64 vectors of size 32000, and with 32 vectors of size 128000, respectively.

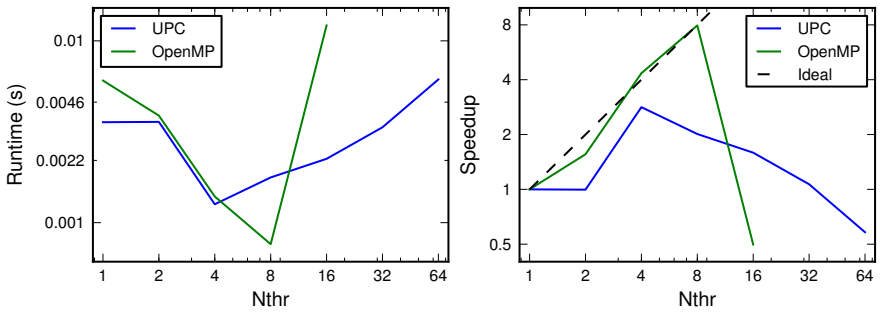


Figure 22: *Sparse matrix-vector multiplication using a 2-D Laplacian of size* 262144×262144

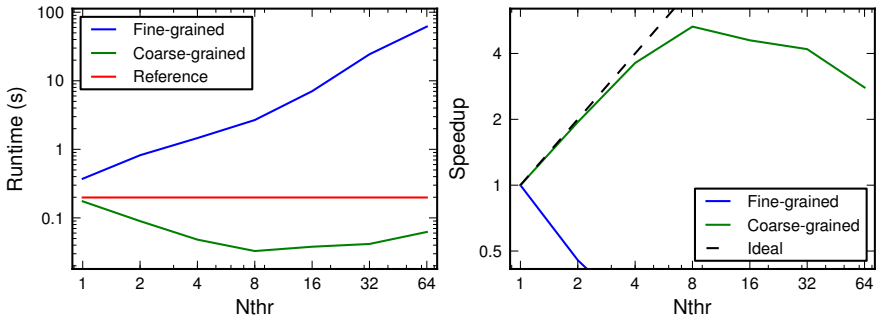


Figure 23: *Modified Gram-Schmidt procedure with* 512 *vectors of size* 512

In Fig. 23, we see that the fine grained parallelization performed extremely poor when many, relatively small vector are used. This is expected since the vectors are too small to amortize the overhead of all the synchronizations in the individual scalar products and vector updates. Equally expected was the very good scaling of the coarse grained parallelization, since the number of vectors is relatively large. For 16 threads and more, performance drops, probably due to the increasing amount of data that has to be copied between the thread-local memories. For a distributed configuration, this communication will have a very high price.

In Fig. 24, we see that for larger vectors, the fine grained parallelization starts scaling better, whereas the coarse-grained parallelization still performs well. Once again, performance deteriorates when using multiple nodes, due to the increased cost of communication.

Finally, looking at the results for the last problem configuration, we see that for large enough vectors, the fine-grained parallelization will eventually perform better (Fig. 25). As noted in Sec. 5.5, this a typical usage scenario in, e.g., a Krylov subspace solver. In a realistic setting, the vector may have millions or even billions of elements, in which case copying them around in a distributed cluster is completely unrealistic.

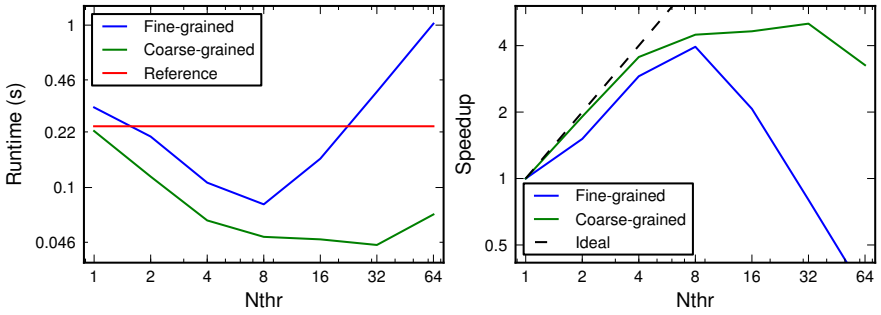


Figure 24: *Modified Gram-Schmidt procedure with* 64 *vectors of size* 32000

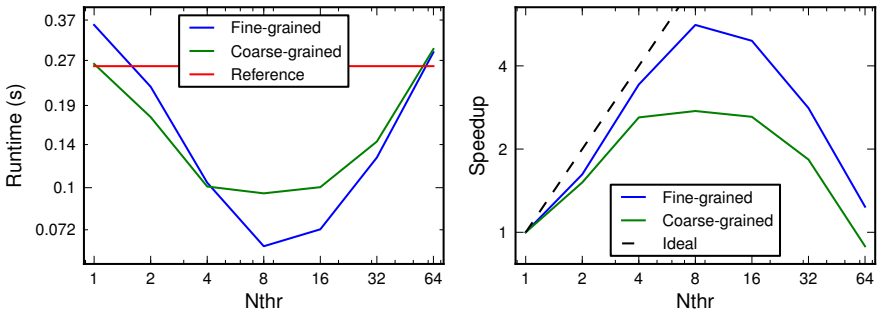


Figure 25: *Modified Gram-Schmidt procedure with* 32 *vectors of size* 128000

7 Experiences and conclusions

Overall, the performance of UPC was good. It performed on-par with an OpenMP parallelization, and in some cases even better. However, when running on multiple nodes, there was a stagnation in the performance. It is difficult to say whether this is due to UPC, the applications being studied, or the Uppmax computer system. More insight into this would have been gained from a comparison with corresponding implementations in MPI.

An obstacle was the fact that the block size is limited to 1048575. This means that already for moderately sized arrays, a simple slicing in one block per thread, i.e. `SIZE/NTHREADS`, will be too large. This was an issue for all applications except the Gram-Schmidt procedure. In these cases, a blocking factor was necessary, by which the original block size was divided.

Since the block size of a shared array is part of its type, it has to be known at compile time. However, often the choice of block size depends on the array size and the thread count. Because of this, for each of the problem configurations, the application had to be recompiled. This is very impractical for real-world codes where generality, portability and efficiency are important.

In summary, UPC is a good effort at simplifying the programming of distributed parallel programs, but with some very strict limitations which constrain its usage in practice.