Implementing and optimizing a Sparse Matrix-Vector Multiplication with UPC

Jérémie Lagravière[1], Martina Prugger[3], Lukas Einkemmer, Johannes Langguth[1], Phuong H. Ha[2], and Xing Cai[1]

[1] Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway
jeremie@simula.no, langguth@simula.no, xingca@simula.no
[2] The Arctic University of Norway, NO-9037 Tromsø, Norway
phuong.hoai.ha@uit.no
[3] University of Innsbruck, Technikerstraße 13 A-6020 Innsbruck, Austria
martina.prugger@uibk.ac.at Lukas.Einkemmer@uibk.ac.at

Abstract. Programmability and performance-per-watt are the major challenges of the race to Exascale. In this study we focus on Partitioned Global Address Space (PGAS) languages, using UPC as a particular example. This category of parallel languages provides ease of programming as a strong advantage over the *classic* Message Passing Interface(MPI). PGAS has also advantages compared to classic shared memory programming (OpenMP), as by nature a PGAS program is meant to work on a single-node and multi-node machine without changing the code. Our goal in this technical report, is to use UPC in order to implement a memory bound problem, which involves irregular inter-thread communication. To represent this problem we perform a SParse Matrix-Vector multiplication (SpMV) over unstructured data. We implemented different versions of the UPC-SpMV for different levels in the code complexity. In this technical report, we give a description of this various versions of the UPC-SpMV and a set of results using single-node and multi-node machine hardware scenarios.

1 Introduction & Motivations

In this technical report, we describe an experiment about a SParse Matrix-Vector multiplication (SpMV) implementation involving irregular and unstructured data accesses. Such irregular-memory-access applications are found in many different research fields [7].

Our goal is to propose a set of solutions for SpMV with unstructured communication patterns using the Partitioned Global Address Space (PGAS) paradigm and more particularly Unified Parallel C (UPC) [1, 6].

We propose three different implementations of the selected problem in UPC. These three implementations represent three levels of optimization in the use of UPC. In our experiments, we ran these three versions on both single-node and multi-node hardware configurations.

The report contains the following sections: in Section 2, we present the PGAS programming model and UPC. In Section 3, we describe the experimental setup for

our study. In Section 5, we show performance measurements of our implementation of UPC-SPMV and we comment those results.

2 PGAS Paradigm and UPC

In this section we describe briefly the Partitioned Global Address Space (PGAS) paradigm and the Unified Parallel C (UPC) language. PGAS is a parallel programming model. It assumes a global memory address space that is logically partitioned and a portion of it is local to each process or thread. The novelty of PGAS is that the portions of the shared memory space may have an affinity for a particular process, thereby exploiting locality of reference [2] [9].

Figure 1 shows a view of the communication model of the PGAS paradigm [6]. In this model each node (C_0 or C_1) has access to a private memory and a shared memory. Accessing to the shared memory to either read or write data can imply inter-node communication. The blue arrow in Figure 1 represents distant access to shared memory. This kind of distant accesses are of type RDMA (Remote Direct Memory Access) and are handled by one-sided communication functions.

Because of this communication model, PGAS languages are often built over a low-level communication layer. This communication layer has to be implemented efficiently to provide good performance.

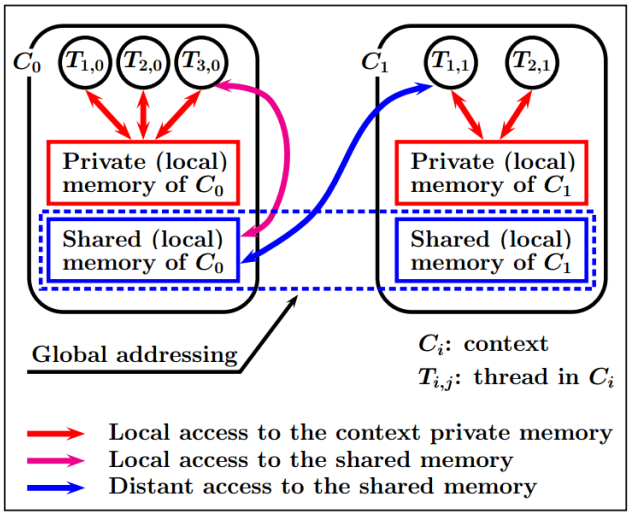

Fig. 1. PGAS Communication Model [6] - Figure used with the courtesy of Marc Tajchman

PGAS is also a family of languages, in which we have chosen UPC for this study. UPC is an extension of the C language. It is one of the first languages that

use PGAS model, and also one of the most stable ones[6]. The key characteristics of UPC are:

- A parallel execution model of Single Program Multiple Data (SPMD) type;
- Distributed data structures with a global addressing scheme, and static or dynamic allocation;
- Operators on these structures, with affinity control;
- Copy operators between private, local shared, and distant shared memories;
- Two levels of memory coherence checking (strict for computation safety and relaxed for performance).

Additionally, multiple open-source implementations of the UPC compiler and runtime environment are available, in particular Berkeley UPC [1] and GCC/UPC [3].

3 Experimental Setup

To carry out our experiments we have selected a set of hardware platforms and software that we describe in this section.

3.1 Hardware

We used the Abel supercomputer [8] to carry out both single-node and multi-node measurements.1

The nodes on Abel are equipped with two Intel Xeon CPUs E5-2670 working at 2.60GHz. Each nodes has 64GB of RAM. The interconnect between nodes is ensured by Infiband FDR (56 Gbits/s eq 6.78 Gbytes/s).

3.2 Software

We used UPC in its 2.22.3 version for all the experiments. We built the UPC compiler and runtime using Intel Compiler version 15.0.1 20141023. When running our experiments we always used thread-binding. Thus, threads do not migrate from one core to another and the data related to those thread is placed in memory in a fixed way.

On single-node, when running on more than 1 thread, we always use two sockets. We ran our experiments on 1 thread up to 32 threads using HyperThreading.

On multi-node, we always use the 16 physical cores of the node, thus avoiding using HyperThreading.

3.3 Datasets

All experimental instances are derived from a 3D mesh of a healthy male human cardiac geometry acquired by MRI, as described in [4]. The maximum number of nonzeroes per row is bounded by 17, making the matrices extremely sparse. In our experiments we use three different datasets of increasing size:

- Instance 1: File size 1.3GB, 6810586 rows
- Instance 2: File size 2.7GB, 13009527 rows
- Instance 3: File size 5GB, 25587400 rows

4 SpMV implementation in UPC

4.1 Sequential code

The sequential code for doing the sparse matrix vector multiplication has the following basic structure. First, we read the data file. The actual data values are stored in the matrix A as **double** (8 bytes), in which each row represents one of the tetrahedra that discretize the form of our the 3D domain of the cardiac geometry. Due to the sparse format of our matrix, A has exactly 16 columns representing the 16 possible neighbors of each data point. To maintain the information of the position of these points, a second matrix I of the same size as A is created, which stores the index of each data point of A. Thus, I contains **integer** values (4 bytes). The data points located on the main diagonal of A are stored separately in the vector D as **double** (8 bytes). The values which are to be multiplied to the matrix are stored in vector V as **double** (8 bytes). Thus, the sparse matrix vector multiplication for a row line j has the form

$$V_{new}\left[j\right] = D\left[j\right] \cdot V\left[j\right] + \sum_{i=1}^{16} A_i\left[j\right] \cdot V\left[I_i\left[j\right]\right].$$

The matrix is stored using the ELL format. In this storage format, two rectangular arrays are used to store the matrix. The arrays have the same number of rows as the original matrix, but only have as many columns as the maximum number of nonzeros on any row of the original matrix. One of the arrays holds the matrix entries, and the other array holds the column numbers from the original matrix [5].

The pseudo-code in Algorithm 1 performs this computation. At the end of each time step, V_{new} is swapped with V using a temporal vector V_{temp}.

Algorithm 1 Pseudo code of the sequential implementation.

```
read data A, I, D and V;
create Vnew, Vtemp;
time loop
        loop j over number of tetraheders
                compute Vnew;
        swap pointers V and Vnew via Vtemp;
```

Algorithm 2 Pseudo code of UPC-SpMV-1: the naive implementation.

```
read A, I, D, V and distribute over THREADS;
create Vnew, Vtemp and distribute over THREADS;
time loop
        loop over each block
                create local pointers to locA, locI, locD;
                create local pointer lVtemp pointing to Vnew;
                loop over elements in each block
                        lVtemp=locD*V(position on large scale)
                                +sum(locA*V(locI));
        swap pointers V and Vnew via Vtemp;
```

4.2 UPC-SpMV-1: the naive implementation

In our first UPC implementation, we already consider the time consuming aspect of accessing shared UPC array and thus we substituted them by pointers, pointing to data that is local to each thread. Similar to the sequential code, we read A, I, D and V from the data file. However, in this UPC implementation A, I, D, V, V_{new} and V_{temp} are distributed among threads. In accordance with the UPC paradigm, this distribution is done in a blockwise manner. Therefore, the rows of A and I as well as the corresponding entries of D and V are residing on the memory of the same thread. However, the access $V[I_i[j]]$ is highly irregular and the residing thread access can not be predicted without additional amount of coding. In our first approach of parallelization, we use local pointers for the data access that is predictable, but maintain the shared array for the irregular data access. This leads to a code structure that looks like Algorithm 2 and is implemented as in Algorithm 4 (cf page 7). All the UPC code that we present in the technical report for UPC-SpMV-1, UPC-SpMV-2 and UPC-SpMV-3 are using a shared data structure called Ls which is defined as in Algorithm 3

Algorithm 3 UPC code used in UPC-SpMV-1, UPC-SpMV-2 and UPC-SpMV-3 : shared data structure.

```
typedef struct {
        //Discrete values that we store for convenience
        shared [1] int *numberOftetra;
        shared [1] int *mySize;
        shared [1] int *n_blocks;
        shared [1] int *remainder;

        //I originalLocationOfNeighbours
        shared [RNZ*BLOCKSIZE] int* I;

        //D valuesOfTetraOnDiagonal
        shared [BLOCKSIZE] double* D;

        //A valueOfTheNeighbours
        shared [RNZ*BLOCKSIZE] double* A;

         //V voltageMultiplicationVector
        shared [BLOCKSIZE] double* V;
} ELLmatrixVector_shared;
```

Algorithm 4 UPC code of UPC-SpMV-1: the naive implementation.

```
temporaryVoltage = (shared [BLOCKSIZE] double*)
        upc_all_alloc(Ls->n_blocks[0],sizeof(double)*BLOCKSIZE);
temp             = (shared [BLOCKSIZE] double*)
        upc_all_alloc(Ls->n_blocks[0],sizeof(double)*BLOCKSIZE);

for(i = 1; i <= timeStep; i++)
{
        for(j=0;BLOCKSIZE*(MYTHREAD+THREADS*j)<size;j++)
        {
                int    *locI   =    (int*)(Ls->I
                        +RNZ*BLOCKSIZE*(MYTHREAD+THREADS*j));
                double *locA   = (double*)(Ls->A
                        +RNZ*BLOCKSIZE*(MYTHREAD+THREADS*j));
                double *locD   = (double*)(Ls->D
                        +BLOCKSIZE*(MYTHREAD+THREADS*j));
                double *lVtemp = (double*)
                (temporaryVoltage+BLOCKSIZE*(MYTHREAD+THREADS*j));

                int bs=BLOCKSIZE;
                if(BLOCKSIZE*(MYTHREAD+THREADS*j)
                        +BLOCKSIZE>size && rem!=0)
                {
                        bs=rem;
                }

                //loop over all elements INSIDE ONE block
                for(k=0;k<bs;k++)
                {
                        a = locA[k*16]*Ls->V[locI[k*16]];
                        //unrolled for loop...
                        d = d + locA[k*16+15]*Ls->V[locI[k*16+15]];
                        lVtemp[k] = locD[k]*Ls->V[k+BLOCKSIZE*
                                (MYTHREAD+THREADS*j)] + a+b+c+d;
                }
        }
        upc_barrier;
        //Pointer Swap
        temp = Ls->V;
        Ls->V = temporaryVoltage;
        temporaryVoltage = temp;

        upc_barrier;
}
```

4.3 UPC-SpMV-2: a more advanced UPC implementation

A strong advantage of parallelizing code using UPC is the fact that it can be done incrementally. Based on the code from UPC-SpMV-1, we therefore made further optimizations to create a better performing version. The file reading is done exactly the same as in the first version. However, this time we also consider the fact that the access $V[I]$ can be unstructured, but constant over the time integration. We therefore can predict which threads have to communicate with others during its computation and identify these as the neighbors. In this implementation, each thread gets a private vector of the size of the number of tetrahedrons in total. In a preparation step, each thread copies the content of each of its neighbors to this private thread. Consequently, each thread has now the information it needs for the computation in its own private memory space and access via shared arrays is no longer required. However, we explicitly need to communicate the updated data points at each time step via a `upc_memget()`. Using this setup, we no longer require the pointer swapping in the end of each time step. In Algorithm 5, we give an overview of the algorithm of this implementation. In Algortithm 6 (cf page 9) we give a sample code of the UPC version for the preparation step, and in Algorithm 7 (cf page 10) we give a sample code of the UPC version for the computation step of UPC-SpMV-2.

Algorithm 5 Pseudo code of UPC-SpMV-2: the advanced implementation.

```
read A, I, D, V and distribute over THREADS;
create Vnew, Vtemp and distribute over THREADS;
create local_V on each THREAD;
identify neighbors THREAD;
time loop
        upc_memget() neighboring content of V to local_V;
        loop over each block
                create local pointers to local_A, local_I,
                                local_D, local_V_temporary;
                loop over elements in each block
                        local_V_temporary=
                        local_D*V(position on large scale)
                              +sum(local_A*local_V(local_I));
```

Algorithm 6 UPC code of UPC-SpMV-2: advance version, preparation part.

```
int neighbour[THREADS];
memset(neighbour, 0, sizeof(neighbour));
int size = Ls->mySize[0];
int rem = Ls->remainder[0];

upc_barrier;

for(j=0;BLOCKSIZE*(MYTHREAD+THREADS*j)<size;j++) {
        //cast local pointers to shared memory
        //BLOCKSTART: 16*BLOCKSIZE*(MYTHREAD+THREADS*j)
        //for matrix (without 16 for vector)
        int *locI = (int*)(Ls->I
        +16*BLOCKSIZE*(MYTHREAD+THREADS*j));
        bs=BLOCKSIZE;

        if(BLOCKSIZE*(MYTHREAD+THREADS*j)+BLOCKSIZE
        >size && rem!=0) {
                        bs=rem;
        }

        //loop over all elements INSIDE ONE block
        for(k=0;k<bs;k++) {
                for(i=0;i<16;i++) {
                        if(upc_threadof(&Ls->V[locI[k*16+i]])
                                !=MYTHREAD) {
                                neighbour[upc_threadof(
                                &Ls->V[locI[k*16+i]])]=1;
                        }
                }
        }
}
upc_barrier;
//add the THREAD itself to the array
neighbour[MYTHREAD]=1;
//count, how many neighbors there are
//and create a new array that
//includes only the neighboring THREADS
int neighbourcount=0;
for(j=0;j<THREADS;j++) {
        neighbourcount=neighbourcount+neighbour[j];
}
int *neighbourarray;
neighbourarray = (int*)calloc(neighbourcount, sizeof(int));
k=0;
for(j=0;j<THREADS;j++)
{
        if(neighbour[j]!=0) {
                neighbourarray[k]=j;
                k=k+1;
        }
}
```

Algorithm 7 UPC code of UPC-SpMV-2: advance version, computation part.

```
//create local copy of V on each THREAD
int nblocks = Ls->n_blocks[0];
int sizeV   = nblocks/**sizeof(double)*/*BLOCKSIZE;
double *locV;
locV=(double *)malloc(sizeV*sizeof(double));
//Time step loop
for(i = 1; i <= timeStep; i++) {
        upc_barrier;

        for(j=0;BLOCKSIZE*j<size;j++) {
                for(l=0;l<neighbourcount;l++) {
                        if(upc_threadof
                        (&Ls->V[BLOCKSIZE*j])==
                                neighbourarray[l])
                        {
                                bs=BLOCKSIZE;
                                if(BLOCKSIZE*j+BLOCKSIZE>
                                size && rem!=0)
                                        bs=rem;

                                upc_memget(&locV[BLOCKSIZE*j],
                                &Ls->V[BLOCKSIZE*j],
                                bs*sizeof(double));
                        }
                }
        }

        //loop over the first element of EACH block
        for(j=0;BLOCKSIZE*(MYTHREAD+THREADS*j)<size;j++) {
                //cast local pointers to shared memory
                int    *locI   =    (int*)(Ls->I
                +16*BLOCKSIZE*(MYTHREAD+THREADS*j));
                double *locA   = (double*)(Ls->A
                +16*BLOCKSIZE*(MYTHREAD+THREADS*j));
                double *locD   = (double*)(Ls->D
                +BLOCKSIZE*(MYTHREAD+THREADS*j));
                double *lVtemp = (double*)(Ls->V
                +BLOCKSIZE*(MYTHREAD+THREADS*j));
                bs=BLOCKSIZE;
                if(BLOCKSIZE*(MYTHREAD+THREADS*j)
                        +BLOCKSIZE>size && rem!=0)      {
                        bs=rem;
                }
                //loop over all elements INSIDE ONE block
                for(k=0;k<bs;k++) {
                        a = locA[k*16]*locV[locI[k*16]];
                        //unrolled for loop...
                        d = d + locA[k*16+15]
                        *locV[locI[k*16+15]];
                        lVtemp[k] = locD[k]*
                        Ls->V[k+BLOCKSIZE*(MYTHREAD+THREADS*j)]
                        + a+b+c+d;
                }
        }
}
upc_barrier;
```

4.4 UPC-SpMV-3: a better communication pattern

In our last implementation step of the code, we employ an optimization idea often used in parallelization with MPI. Similarly to UPC-SpMV-2, we prepare our data structure for communication before we start the computation. However, in this case we do not only identify the neighbors, but the data points needed by each thread. As before, each thread gets a local vector of the size of V. The position of the data points that are needed by a thread are stored in a separate array as well as the data points themselves. This data array is then communicated via `upc_memget()`. Therefore, compared with the previous implementation, we keep the cost of communication to the necessary minimum. Using the additional arrays for indexes and data points, we need to perform a pointer swap at the end of each time step again. This version was the object of many optmiziation which are not usual compared to the "classic" and "easy" UPC programming paradigm. In this version, we are closer to an MPI-programming style, however thanks to the mechanisms provided by UPC we were able to implement UPC-SpMV-3 without getting to an extremely fine-grain communication management. Thus, we still benefit from the UPC ease of programming in addition to the incremental aspect of developpment as UPC-SpMV-3 is based on the code from UPC-SpMV-2. An overview of UPC-SpMV-3's algorithm is available in Algorithm 8. In Algortithm 9 (cf page 12) we give a sample code of the UPC version for the preparation step, and in Algorithm 10 (cf page 13) we give a sample code of the UPC version for the computation step of UPC-SpMV-3.

Algorithm 8 Pseudo code of UPC-SpMV-3: reduced amount of communication

```
read A, I, D, V and distribute over THREADS;
create locV and locVtemp on each THREAD;

//from here on, V is no longer used
copy V to locV on each THREAD;

 //stores needed data points for each thread
create shared array of arrays data;

//stores corresponding indizes
create shared array of arrays index;
time loop
        upc_memget() needed array of data;
        put these data back to locV using info of index;
        loop over each block
                create local pointers to locA, locI, locD;
                loop over elements in each block
                        lVtemp(position on large scale)
                        =locD*locV(position on large scale)
                        +sum(locA*locV(locI));
        pack new values into data;
        swap pointers locV and locVtemp;
```

Algorithm 9 UPC code of UPC-SpMV-3: reduced amount of communication, preparation part.

```
shared [] double* shared [1] datatosend[THREADS][THREADS];
int arraylength =ceil(size/THREADS);
//create two local V vectors for pointer swapping later on
//therefore, we do no longer use the shared V in general
double *locV;
locV=(double *)calloc(sizeV,sizeof(double)); double *locVtemp;
locVtemp=(double *)calloc(sizeV,sizeof(double)); int *locIprep;
int sizeI=Ls->n_blocks[0]*BLOCKSIZE*RNZ; upc_barrier;
for(j=0;BLOCKSIZE*j<size;j++) { bs=BLOCKSIZE;
        if(BLOCKSIZE*j+BLOCKSIZE>size && rem!=0) {
                bs=rem;
        }       upc_memget(&locV[BLOCKSIZE*j],
        &Ls->V[BLOCKSIZE*j],bs*sizeof(double));
} upc_barrier;
int *whattocopy[THREADS]; int *whattocopy_insert[THREADS];
int *whattoget[THREADS]; int *whattoget_insert[THREADS];
for(int thrds=0;thrds<THREADS;thrds++) {        upc_barrier;
        if(thrds==MYTHREAD)             {
                locIprep=(int *)calloc(sizeI,sizeof(int));
                for(j=0;16*BLOCKSIZE*j<16*size;j++) {
                        bs=16*BLOCKSIZE;
                        if(16*(BLOCKSIZE*j+BLOCKSIZE)
                                >16*size && rem!=0)
                                bs=16*rem;

                        upc_memget(&locIprep[BLOCKSIZE*j*16],
                        &Ls->I[BLOCKSIZE*j*16],bs*sizeof(int));
                }
                for(i=0;i<THREADS;i++) {
                        whattocopy[i]=(int *)
                        malloc(arraylength*sizeof(int));
                        whattocopy_insert[i]=whattocopy[i];
                        whattoget[i]=(int *)
                        malloc(arraylength*sizeof(int));
                        whattoget_insert[i]=whattoget[i];
                }
                for(j=0;BLOCKSIZE*j<size;j++) { bs=BLOCKSIZE;
                        if(BLOCKSIZE*j+BLOCKSIZE>size && rem!=0)
                                bs=rem;
                        current_id = j%THREADS;
                        for(k=0;k<bs;k++) {     for(i=0;i<16;i++) {
                                points_to  = (locIprep[16*BLOCKSIZE*j+k*16+i]
                                /BLOCKSIZE)%THREADS;
                                if(current_id != MYTHREAD
                                && points_to == MYTHREAD)       {
                                        *(whattocopy_insert[current_id])
                                        = locIprep[16*BLOCKSIZE*j+k*16+i];
                                        whattocopy_insert[current_id]++;
                                }
                                if(current_id==MYTHREAD
                                && points_to != MYTHREAD) {
                                        *(whattoget_insert[points_to])
                                        = locIprep[16*BLOCKSIZE*j+k*16+i];
                                        whattoget_insert[points_to]++;
                                }}}}free(locIprep);}upc_barrier;}upc_barrier;
```

Algorithm 10 UPC code of UPC-SpMV-3: reduced amount of communication, computation part.

```
for(i = 1; i <= timeStep; i++)
{
        upc_barrier;
        for(k=0;k<THREADS;k++) {
                indgetarraylength=whattoget_insert[k]-whattoget[k];
                if (!indgetarraylength)
                        continue;

                upc_memget(locDataCopy[k],
                datatosend[k][MYTHREAD],
                indgetarraylength*sizeof(double));
                //rearrange the new data points to the correct positions in V
                for(j=0;j<indgetarraylength;j++)
                        locV[whattoget[k][j]]=locDataCopy[k][j];
        }

        //upc_barrier;

        //loop over the first element of EACH block
        for(j=0;BLOCKSIZE*(MYTHREAD+THREADS*j)<size;j++) {
                int    *locI   =    (int*)(Ls->I
                +16*BLOCKSIZE*(MYTHREAD+THREADS*j));
                double *locA   = (double*)(Ls->A
                +16*BLOCKSIZE*(MYTHREAD+THREADS*j));
                double *locD   = (double*)(Ls->D
                +BLOCKSIZE*(MYTHREAD+THREADS*j));

                bs=BLOCKSIZE;
                if(BLOCKSIZE*(MYTHREAD+THREADS*j)+BLOCKSIZE
                >size && rem!=0)
                        bs=rem;
                //loop over all elements INSIDE ONE block
                for(k=0;k<bs;k++) {
                        a = locA[k*16]*locV[locI[k*16]];
                        //unrolled for loop
                        d = d + locA[k*16+15]*locV[locI[k*16+15]];

                        locVtemp[k+BLOCKSIZE*(MYTHREAD+THREADS*j)]
                        = locD[k]*locV[k+BLOCKSIZE
                        *(MYTHREAD+THREADS*j)] + a+b+c+d;
                }
        }
        upc_barrier;
        //pack the new data back into the shared package
        for(k=0;k<THREADS;k++) {
                indivarraylength=whattocopy_insert[k]-whattocopy[k];
                for(j=0;j<indivarraylength;j++)
                        locSendData[k][j]=locVtemp[whattocopy[k][j]];
        }
        double *tmp=locV;
        locV=locVtemp;
        locVtemp=tmp;
}
```

5 Results & Discussion

In this section we present the results obtained by running our implementations of UPC-SpMV-1, UPC-SpMV-2 and UPC-SpMV-3 (cf section 4).

5.1 Results: single-node scenario

In Figures 2, 3 and 4 we report the performance of respectively UPC-SpMV-1, UPC-SpMV-2 and UPC-SpMV-3 running on one node from 1 thread up to 32 threads using Instance 1, 2 and 3 (described in Section 3). As mentioned earlier, when using more than 1 thread, we use both sockets on the node.

In Figure 2, UPC-SpMV-1 reaches 4 GFLOPS depending on the instance. The smallest instance (Instance 1) gives the better results. Using HyperThreading has no interest in this case, as it represents an increased amount of communication for a lower bandwidth per thread. We can also see that UPC-SpMV-1 scales when we add more threads (up to 16 threads). This is interesting as this version is based on a very straight-forward implementation using basic UPC optimization. In UPC-SpMV-1, the V vector is still accessed through a shared array. The ease of programming provided by shared arrays is extremely high and in UPC-SpMV1 it does not prevent the application to scale over an increasing threads.

In Figure 3, UPC-SpMV-2 reaches 3.5GFLOPS on both 4 and 8 threads. All the instances are processed at the same speed when using 2,4 and 8 threads. On more than 8 threads the use of the bandwidth is less efficient by UPC-SpMV-2. An increased amount of local pointers and local neighbors map added to the access to the V vector during the computing through a shared array become too costly, hence the decrease in performance.

In Figure 4, UPC-SpMV-3 reaches 4.3 GFLOPS on 8 threads and 16 threads. On 4 threads, UPC-SpMV-3 is already performing better than UPC-SpMV-1 and UPC-SpMV-2, particularly when using Instance 1. We can notice, that results are slightly dependent on the size of the Instance when using more than 2 threads. Interestingly, by reducing the amount of communication to its minimum, the performance does not drop significantly when using 32 threads and HyperThreading.

5.2 Results: multi-node scenario

In Figures 5, 6 and 7, we report the performance of respectively UPC-SpMV-1, UPC-SpMV-2 and UPC-SpMV-3 on a muti-node machine [8] from 16 threads (1 node) to 1024 threads (64 nodes). As we are dealing with a memory-bound problem, adding more nodes increases the available bandwidth between CPU's and RAM. The computing power of the CPU's is not the direct factor to increase performance in this case.

In Figure 5, we can see that UPC-SpMV-1 does not scale at all on multi-node. The explicit use of shared arrays in this version is too costly in terms of access time preventing the application from scaling. The interest of the ease of programming

of UPC in this case, is not valid as it does not allow to implement an application that scales over multiple nodes.

In Figure 6, we can see that UPC-SpMV-2 scales up to a certain amount of threads/nodes depending on the size of the Instance. The bigger the Instance is, the more UPC-SpMV-2 achieves better performance on more threads/nodes. With Instance 3, UPC-SpMV-2 reaches 17 GFLOPS on 32 nodes. We can notice that once UPC-SpMV-2 has reached its peak performance, the scaling stops and the performance flats-out and decreases slowly when adding more nodes.

In Figure 7, we can see that UPC-SpMV3 scales up to a certain amount of nodes, depending on the instance. Also, we would like to emphasize the huge difference in performance of UPC-SpMV-3 compared to UPC-SpMV-1 and UPC-SpMV-2. When using 8 nodes and more, UPC-SpMV-3 delivers more than 20 GFLOPS which is a performance that is already above the performance of the two other implementations. The best performance achieved with UPC-SpMV-3 is on 64 nodes (1024 threads) using Instance 3, where the application delivers 63 GFLOPS. We can notice that when using Intance 1 and 2 the program scales up to 16 nodes and then the performance flats-out and decreases when adding more nodes.

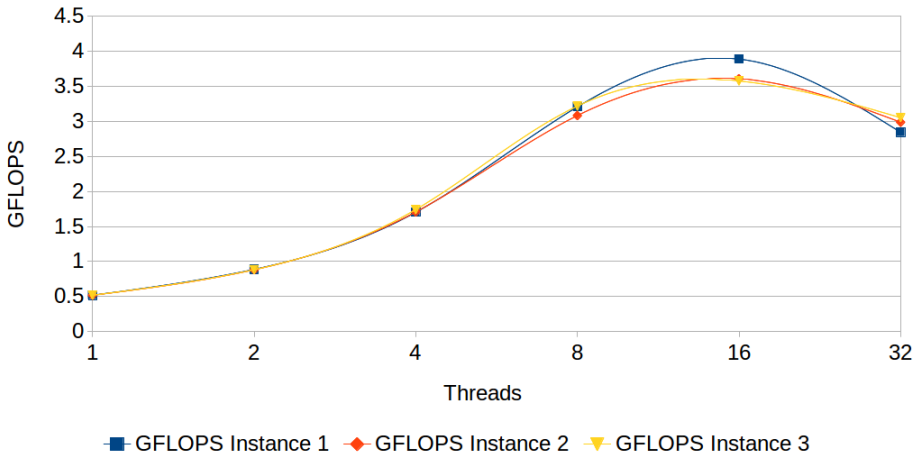

Fig. 2. UPC SpMV Version 1 - Single Node Performance

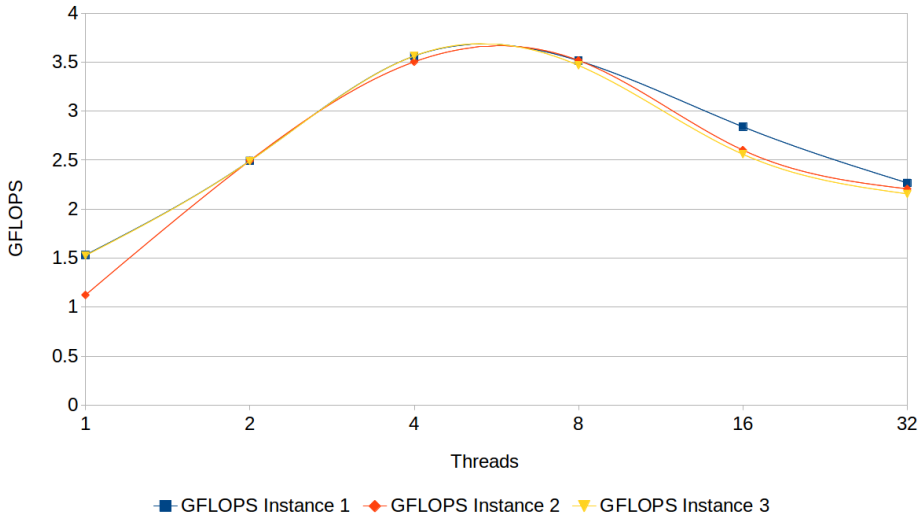

Fig. 3. UPC SpMV Version 2 - Single Node Performance

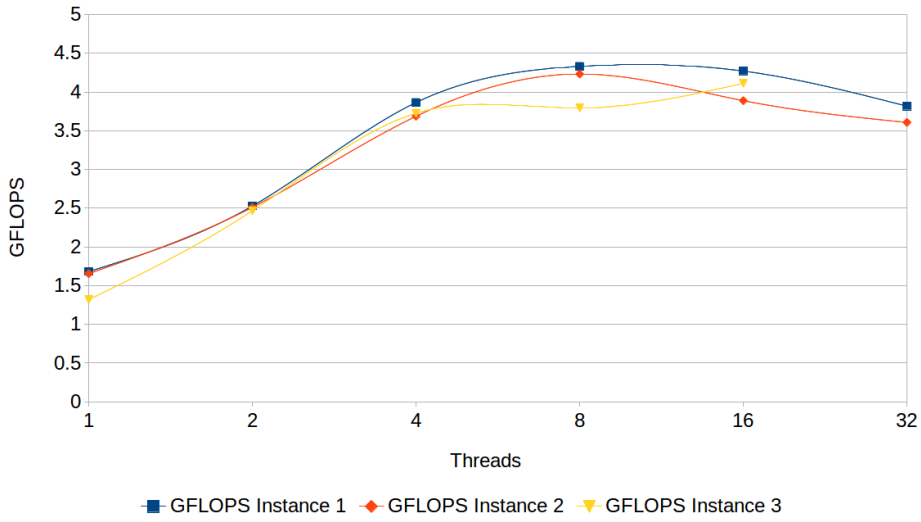

Fig. 4. UPC SpMV Version 3 - Single Node Performance

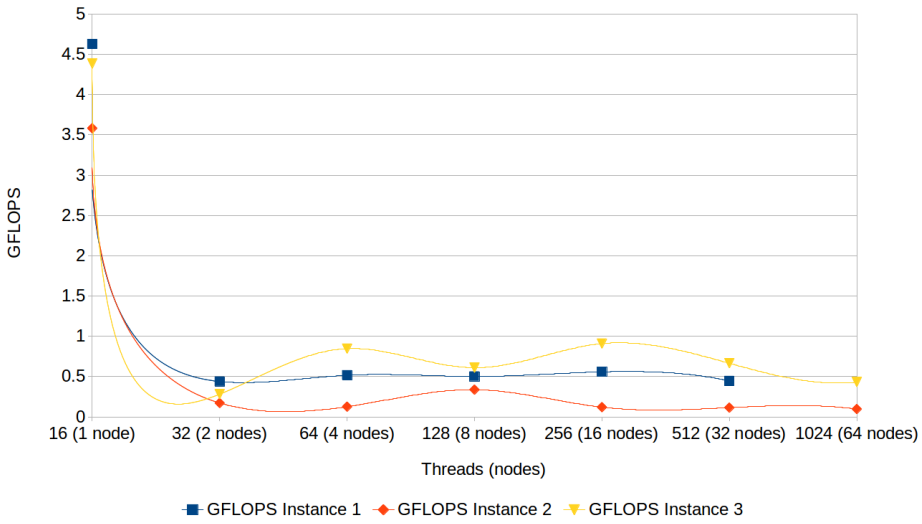

Fig. 5. UPC SpMV Version 1 - Multi Node Performance

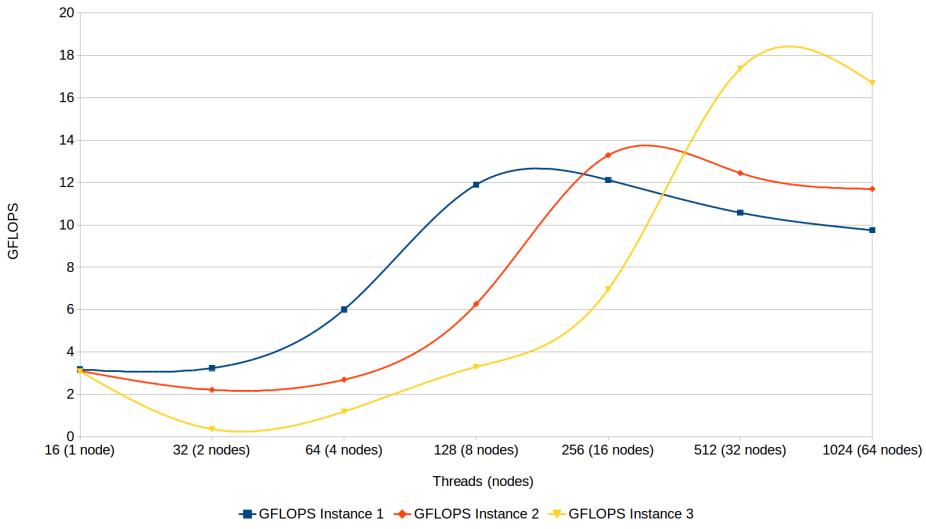

Fig. 6. UPC SpMV Version 2 - Multi Node Performance

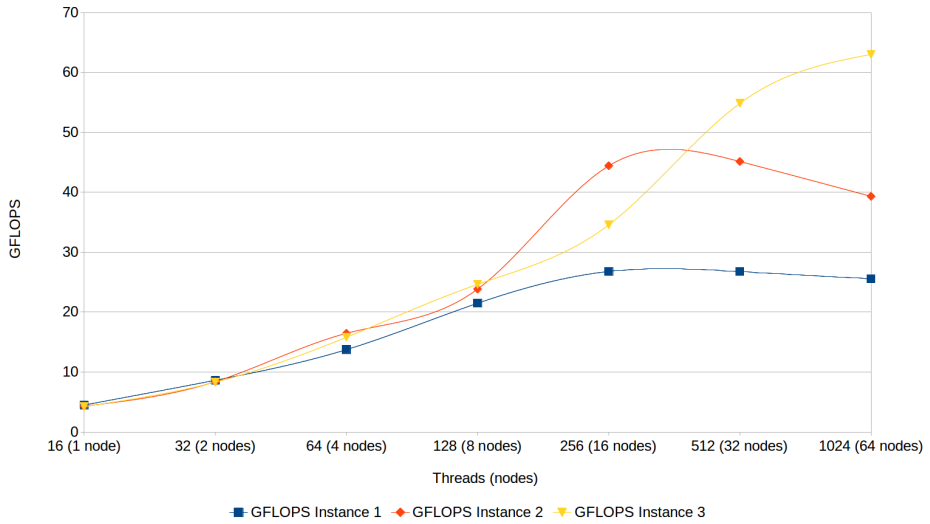

Fig. 7. UPC SpMV Version 3 - Multi Node Performance

6 Conclusion & Future Work

In this technical report we have shown that UPC can be a valid selection to implement sparse matrix-vector multiplication. In a more general way, memory bound problems can be addressed by using UPC. However, we also discovered that there are at least two major costs to consider when choosing UPC for memory bound problems with unstructured communication:

- The extremely poor performance of accessing UPC's shared arrays;
- The necessity to optimize the code to reduce the impact of communication to their minimum.

In this study, we provide UPC implementations of SpMV with unstructured communication. These versions scale in different ways on both single-node and multi-node configurations depending on their level of optimizations and the instance size. For future work, we consider running the application on bigger instances and on higher amount of nodes and we also consider using Many Integrated Core machines such as Intel Xeon Phi Knight's Corner and Knight's Landing.

7 Acknowledgements

The computational results presented have been achieved [in part] using the Vienna Scientific Cluster (VSC).
This work was performed, in part, on the Abel Cluster, owned by the University of Oslo and the Norwegian metacenter for High Performance Computing (NOTUR), and operated by the Department for Research Computing at USIT, the University of Oslo. http://www.hpc.uio.no/

References

1. Berkeley: UPC Implementation From Berkeley (last accessed on 27/09/2016). URL http://upc.lbl.gov
2. Coarfa, C., Dotsenko, Y., Mellor-Crummey, J., Cantonnet, F., El-Ghazawi, T., Mohanti, A., Yao, Y., Chavarría-Miranda, D.: An evaluation of global address space languages: Co-Array Fortran and unified parallel C. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 36–47. ACM (2005)
3. Inc., I.T.: UPC Implementation on GCC (last accessed on 27/09/2016). URL http://www.gccupc.org/
4. Langguth, J., Sourouri, M., Lines, G.T., Baden, S.B., Cai, X.: Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes. IEEE Micro **35**(4), 6–15 (2015)
5. lanl: ELL Format Definition (last accessed on 27/09/2016). URL https://www.lanl.gov
6. Marc Tajchman, C.: Programming paradigms using PGAS-based languages. Figure used with the courtesy of Marc Tajchman (2015). URL http://www-sop.inria.fr/manifestations/cea-edf-inria-2011/slides/tajchman.pdf
7. Scholar, G.: Irregular Memory Access Relevance (last accessed on 27/09/2016). URL https://scholar.google.fr/scholar?q="irregular+memory+access"
8. UiO: Abel SuperComputer Official Webpage (last accessed on 27/09/2016). URL http://www.uio.no/english/services/it/research/hpc/abel/
9. Wikipedia: Wikipedia Definition of PGAS - Last accessed on 27/03/2015 (2015)