# Supplementary Material #1: BSGP Primitive Algorithms

Table 1 summarizes the algorithms of BSGP's primitive functions.

## 1 Fork algorithm

We first use a collective `scan` on `fork`'s parameter to compute the new rank of each thread's first child. A list of `thread.oldrank` for each child thread is then filled in $\log_2 m$ iterations. $m$ is the maximal number of child threads for a single parent. This is achieved by filling $O(2^i)$ entries for each parent thread in iteration $i$. Listing 1 is the pseudo code of the algorithm.

## 2 Sorting algorithm

We use a binary search based merge sort in our library. An array `a` with `n` elements is said to be `m`-sorted if `a[k*m+i]<=a[k*m+j]` for all non-negative integers `i,j,k` satisfying that `i<j<m`, `k*m+j<n`. Any array is trivially 1-sorted. For a `m`-sorted array, neighboring sorted segments may be merged in parallel to yield a `2*m`-sorted array. We perform such merge iteratively until array `a` becomes at least `n`-sorted.

We implement the merge using binary search. Each element is binary searched in its neighboring sorted segment to compute its offset in the merged array. All elements are processed entirely in parallel in each iteration. See Listing 2 for a pseudo code of the merge algorithm.

In actual implementation, first few merges are done in a single superstep using CUDA's shared memory and local synchronization. These merges may be bundled with previous superstep. Such detail is omitted in Listing 2 for simplicity. Also, Listing 2 assumes elements in `a` to be distinct. Duplicated elements may be handled with minimal modification.

The algorithm has a coherent memory access pattern and is highly parallel. Therefore, it outperforms the asymptotically more optimal radix sort in our experiments. Also, our algorithm is comparison based and easier to generalize than radix sort. Table 2 compares our performance with the $O(n)$ work $O(1)$ passes radix sort in CUDA SDK.

| n | 2M | 4M | 8M |
|---|---|---|---|
| Radix | 69.6ms | 179ms | 403ms |
| Merge | 46.8ms | 97.8ms | 230ms |

**Table 2:** *Sort time for array with* `n` *elements. Each element consists of a 32-bit integer key and a 32-bit integer data.*

**Listing 1** Pseudo code of `fork` algorithm

```
inline int fork(int m){
    require{
        pchild = dnew[thread.size]int;
        nchild = dnew[thread.size]int;
    }
    p = m;
    n = scan(p);
    pchild[thread.rank] = p;
    nchild[thread.rank] = m;
    barrier(RANK_REASSIGNED); require{
        //compute oldrank for children
        rankp = dnew[n]int;
        fork_fill(rankp,pchild,nchild,thread.size);
        thread.size = n;
    }
    thread.oldrank = rankp[thread.rank];
    return thread.rank-pchild[thread.oldrank];
}

int fork_fill(int* rankp, int* pc, int* nc, int n0){
    sz = 1;
    id = dnew[n0]int;
    spawn(n0){
        i = thread.rank;
        id[i] = i;
    }
    while(n0>0){
        //fill sz oldranks for each parent thread
        spawn(n0*sz){
            i = thread.rank/sz;
            f = thread.rank%sz;
            if(f<nc[i])
                rankp[pc[i]+f]=id[i];
        }
        /*
        remove parent threads with all children's
        oldrank completely filled
        */
        spawn(n0){
            i = thread.rank;
            p = pc[i];
            m = nc[i];
            d = id[i];
            thread.kill(m-sz<0);
            i = thread.rank;
            pc[i] = p+sz;
            nc[i] = m-sz;
            id[i] = d;
            barrier; require
                n0 = thread.size;
        }
        sz*=2;
    }
}
```

## References


CHATTERJEE, S., BLELLOCH, G. E., AND ZAGHA, M. 1990. Scan primitives for vector computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, 666–675.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for gpu computing. In *Graphics Hardware*, 97–106.

| Primitive | Work | Supersteps | Algorithm |
|---|---|---|---|
| **reduce** | $O(n)$ | $O(1)$ | We follow the reduction sample in CUDA SDK. |
| **scan** | $O(n)$ | $O(1)$ | First perform one local scan as in [Sengupta et al. 2007]. [Chatterjee et al. 1990] is then used to scan per-block result in $O(1)$ supersteps. Local result are finally added with per-block result. Local scan and result adding may be bundled with surrounding supersteps. |
| **compact** | $O(n)$ | $O(1)$ | Implemented using `scan` as in [Sengupta et al. 2007]. |
| **split** | $O(n)$ | $O(1)$ | Implemented using `scan` as in [Sengupta et al. 2007]. |
| **sort_idx** | $O(n \log^2 n)$ | $O(\log n)$ | Binary search based merge sort, see Section 2 of the paper. |
| **thread.split** | $O(n)$ | $O(1)$ | Implemented by passing `thread.rank` to `split` and using its result as new rank. |
| **thread.sortby** | $O(n \log^2 n)$ | $O(\log n)$ | Implemented by adjusting rank to `sort_idx`. |
| **thread.kill** | $O(n)$ | $O(1)$ | Implemented by passing `thread.rank` to `compact` and using its result as new rank. `thread.size` is adjusted accordingly. |
| **thread.fork** | $O(n')$ | $O(\log m)$ | $n'$ is the total amount of child threads. $m$ is the maximum number of child threads of a single thread. For details, see Section 1 of the paper. |

**Table 1:** *Algorithm of primitive functions*

**Listing 2** Pseudo code of `sort_idx` and the merge algorithm.

```
inline int sort_idx(int k){
    //make key and index arrays
    require{
        n = thread.size;
        ka = dnew[n]int;
        a = dnew[n]int;
        kb = dnew[n]int;
        b = dnew[n]int;
    }
    ka[thread.rank] = k;
    a[thread.rank] = thread.rank;
    barrier require{
        //merge sort
        for(int m=1;m<n;m+=m){
            merge(b,kb,a,ka,n,m);
            swap(b,a);
            swap(kb,ka);
        }
    }
    return a[thread.rank];
}

//merge (ka,a) to (kb,b)
merge(int* b, int* kb, int* a, int* ka, int n, int m){
    spawn(n){
        id = thread.rank;
        ofs = id%m;
        k = ka[id]; d = a[id];
        //locate the neighboring segment
        l = ((id-ofs)^m); r = min(l+m,n)-1;
        //binary search
        l0 = l;
        while(l<=r){
            m = (l+r)>>1;
            if(a[m]<=k)
                l = m+1;
            else
                r = m-1;
        }
        /*
        copy element to new position:
            id-id%(2*m) is start of the merged segment.
            There're ofs and (l-l0) elements less than
        k in the two segments respectively, and it should
        be stored at offset ofs+(l-l0).
        */
        addr = (id-id%(2*m))+ofs+(l-l0);
        kb[addr] = k;
        b[addr] = d;
    }
}
```