Department of

Computer Science

NAS parallel benchmark integer sort (IS) performance on MINT

S. Sur and W. Bohm

Technical Report CS-93-107

May 20, 1993

Colorado State University

NAS parallel benchmark integer sort (IS) performance on MINT

S. Sur and W. Böhm

Department of Computer Science
Colorado State University
Ft. Collins, CO 80523

May 17, 1993

Abstract

We study several sorting routines written in Id and compare their relative performance in terms of number of instructions (S_1), length of the critical path (S_∞) and average parallelism. The sorting routines considered here are of the types (1) Exchange sort (2) Insertion sort (3) Merge sort and (4) Sorting Networks. We implement them using I-structures (e.g. merge sort) or M-structures (e.g bubble sort). We optimize the routines with respect to efficiency, minimize the number of barriers, and eliminate redundant copying. We compare our results with expected theoretical performance. It turns out that M-structures improve the performance and sometimes the elegance of our algorithms, and that the classical sort algorithms outperform the explicit parallel ones on monsoon. Mergesort outperforms all other algorithms, even with explicit deallocation.

Address for Correspondence:

A. P. W. Böhm
Department of Computer Science
Colorado State University
Ft. Collins, CO 80523
Tel: (303) 491-7595
Fax: (303) 491-6639
Email: bohm@CS.ColoState.Edu

1 Introduction

The NAS benchmark *IS* deals with sorting N integer keys in parallel. The performance of the sorting routines depends greatly on the key distribution and the process of key generation described in the benchmark is carefully followed. To generate uniformly distributed pseudo-random numbers we made use of the following recurrence:

$$x_{k+1} = ax_k(mod 2^{46})$$

where a is set to be 5^{13} and $x_0 = s$ is the seed which we set to be 314159265 as specified. We create the r array (see the code below) by dividing x_i by 2^{46}. This r-array has values between 0 and 1 and is very nearly uniformly distributed on the unit interval. The keys are generated by averaging four of such consecutive pseudo-random numbers and scaling it by a factor B_{max}. In our case we set B_{max} to be 1000.

```
def randvec_is m = {
% Generates m keys
n=4*m;
typeof r = I_vector(F); r=I_vector(1,n);
typeof rand = I_vector(I); rand=I_vector(1,m);
v=314159265;
a=round (5.0^13);
range = round (2.0^46);
{for i<-1 to n do
    temp = a*v;
    next v = mod temp range;
    r[i] = (float next v)/(float range) };
bmax=1000.0;
{for i<-1 to m do
    j  = 4*i;
    rand[i] = floor (bmax*(r[j-3]+r[j-2]+r[j-1]+r[j])/4.0) }
in rand
};
```

To obtain an M-structure filled with these integers, we simply create the above I-structure and then store it in an M-structure. To isolate the performance of the sorting routine from that of key generation we make use of a barrier as shown in the code segment given below, such that random vector generation happens without overlap with the sorting part. In the parallelism profiles the dark portion in the beginning signifies the key generation part and the lighter shade characterizes the performance of the sorting routine. In table 1, when we present individual performances of the sorting routines, we subtract the effect of key generation from the instruction count and the critical path length.

```
def runsort n ={
v = randvec_ms n;
---
w= sort v 1 n;
in w
};
```

2 Exchange of keys and Bubble Sort

In this section we present two simple routines for exchanging two elements in an M-array and a code for bubble sort that is based solely on these exchanges. The function *exchange* swaps two elements in an M-structure at the respective indices. Notice the barrier after the reading operations which prevents out of order writing into the M-structure. This exchange routine is risky in the sense that if two exchanges are carried out in parallel, e.g an i-j exchange and an i-k exchange, the result will depend on the order in which the elements are read. Moreover, this kind of exchange has the potential for deadlock if exchange of i-j, j-k and i-k elements happen simultaneously. In the sorting routines we developed either the algorithms prevent such problematic exchanges or, more interestingly, we made use of default k-bound of 1 on loops to avoid this problem. We will mention when such default k-bounding of loops is needed for the correctness of our algorithms.

```
def exchange vec i j = {
typeof vec = M_vector(I);
    a = vec![i];
    b = vec![j];
    ---
    vec![i], vec![j] = b, a;
in vec
};
```

The function *check_and_exchange* is very similar to the previous one, except here the exchange is conditional. Note, the absence of a barrier in this case. The condition statement requires both i-th and j-th element to be read before the exchange occurs. This is an example of implicit synchronization based on data dependance where we do not need an explicit barrier.

```
def check_and_exchange vec i j = {
typeof vec = M_vector(I);
    a = vec![i];
    b = vec![j];
    vec![i], vec![j] =if ( b < a)
                          then b, a
                          else a, b;
in vec
};
```

The following routine for bubble sorting is a good example where the above exchange routine is used and default k-bound of 1 is used for synchronization. It consists of a doubly nested loop where the loop body contains one call to the check_and_exchange function for exchanging two consecutive elements of the array. The k-bound of 1 guarantees that no more than one loop body is executed at one time. Fig. 1 shows the parallelism profile for bubble sort for 256 element and as expected, this method performs very poorly having an average parallelism of only 2.77 (see Table 1). The instruction count of over 2.5 million is also excessively high.

```
% BUBBLE SORT

def bubblesort d ={
```

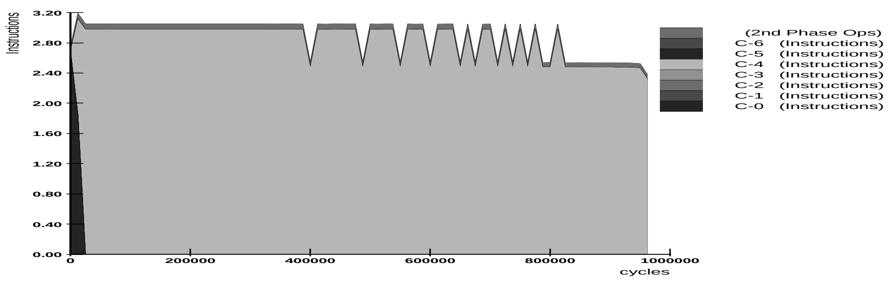

Figure 1: Sample idealized profile of bubble sort of 256 integers

```
typeof d = M_vector(I);
(_,n) = bounds d;
{for i<- n-1 downto 1 do
    {for j<- 1 to i do
      _ = check_and_exchange d j (j+1) }
};
in d
};
```

3 Heap sort

In this section we consider two implementations of heap sort, the first one using I-structures and the second one using M-structures. The purpose here is to show how costly programming with I-structures can be compared to programming with M-structures, because of unnecessary copying of I-arrays. The function *heapsort* uses I-structures. It consists of three functions (a) heapify, which builds a heap within the binary sub-tree of a specified node, passed as an argument in the form of an index to the array, (b) build_heap, which uses heapify in a loop to build a complete heap over the array, and (c) heapsort, which builds a complete heap first, strips the top element and calls heapify to build another heap for the rest of the elements, and continues the process untill all the elements have come to the top of the heap. Note, the unavoidable copying step in heapsort, which exchanges two array elements and which cannot be performed without copying. This turns the heap sort routine has turned into an $O(n^2)$ algorithm (see Table 2). The instruction count for 256 elements is about 5 million with critical path length over half a million (see Table 1). Fig. 2. shows that the average parallelism is about 9.5.

```
% HEAP SORT with I-structures

def heapsort A ={
typeof A = I_vector(I);
(_,n) = bounds A;
typeof dsort = I_vector(I); dsort = I_vector(1,n);
theap = build_heap A;
heaped= {for j<- n downto 2 do
            dsort[j] = theap[1];
            typeof temp = I_vector(I);
            temp = I_vector(1,j-1);
%unavoidable copying of the array elements to perform exchange
            {for k<- 2 to j-1 do
                 temp[k] = theap[k]};
            temp[1] = theap[j];
            next theap = heapify temp 1;
            finally theap
        };
dsort[1] = heaped[1];
in dsort
};

def build_heap A ={
typeof A = I_vector(I);
(_,n) = bounds A;
m = div n 2;
theap = A;
heaped= {for j<- m downto 1 do
            next theap = heapify theap j;
    finally theap
        };
in heaped
};

def heapify A i ={
typeof A = I_vector(I);
(_,n) = bounds A;
typeof tempd = I_vector(I); tempd = I_vector(1,n);
l= 2*i;
r= 2*i+1;
larger = if(l > n) then i
         else if(A[l] > A[i])
              then l else i;
largest = if (r > n) then larger
          else if(A[r] > A[larger])
               then r else larger;
dheap = if(largest <> i )
%awkward copying of array elements in segments to perform exchange A[i] and A[largest]
        then{{for j<- 1 to i-1 do
                 tempd[j] = A[j]};
             tempd [i] = A[largest];
             {for j<- i+1 to largest-1 do
                 tempd[j] = A[j]};
             tempd [largest] = A[i];
             {for j<- largest+1 to n do
                 tempd[j] = A[j]};
             d2 = heapify tempd largest;
             in d2 }
        else A;
```

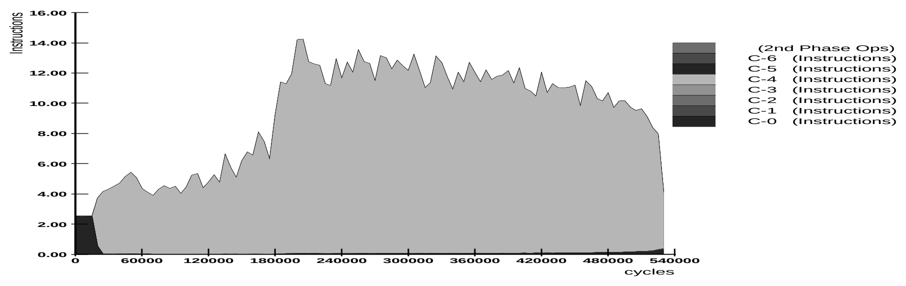

Figure 2: Sample idealized profile of heap sort with I-structures of 256 integers

```
in dheap
};
```

The function *heapsortm* shows the implementation of the same algorithm using M-structures. Here the functions with the same name as above perform the same task. Note that the exchanges of elements within an array are not as cumbersome or expensive anymore. The overhead here is that of synchronization which is implemented in the form of barriers. We need one barrier in heapsortm after building a complete heap to make sure further operations do not start before the heap is built. The barriers after the exchange operations make sure that that heapification occurs only after the exchange is complete. The k-bound of 1 is also used implicitly used in the loop for synchronization purposes so that exchanges may not conflict. The use of M-structures here paid off immensely, as the total instruction count has reduced by a factor of 10 to about half a million (see Table 1) for 256 elements. The critical path length is still very high (159,700) making average parallelism as low as 3.1 as seen in Fig. 3. This low parallelism is mainly due to use of loop bound of 1, but a higher loop bound does not perform any better either, since it increases the instruction count significantly.

```
% HEAP SORT with M-structures

def heapsortm A ={
typeof A = M_vector(I);
(_,n) = bounds A;
dsort = build_heap A;
---
sorted={for j<- n downto 2 do
           temp = exchange dsort 1 j;
           ---
           next dsort = heapify dsort 1 (j-1);
           finally dsort }
in sorted
};

def build_heap A ={
typeof A = M_vector(I);
```

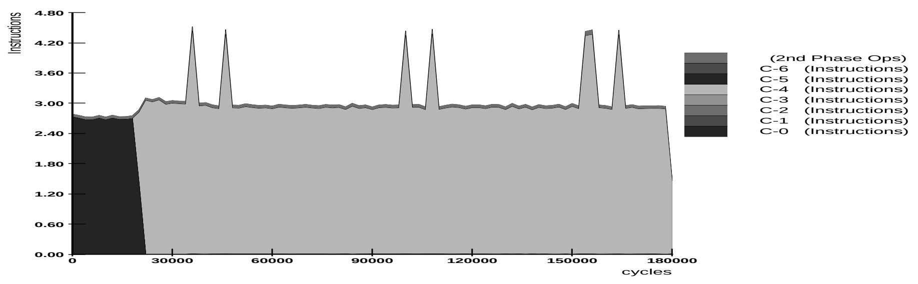

Figure 3: Sample idealized profile of heap sort with M-structures of 256 integers

```
(_,n) = bounds A;
m = div n 2;
{for j<- m downto 1 do
    next A = heapify A j n }
in A
};

def heapify A i n={
typeof A = M_vector(I);
l= 2*i;
r= 2*i+1;
larger = if(l > n) then i
         else if(A!![l] > A!![i])
              then l else i;
largest = if (r > n) then larger
          else if(A!![r] > A!![larger])
               then r else larger;
dheap = if(largest <> i )
        then{temp = exchange A largest i;
             ---
            d2 = heapify A largest n;
            in d2 }
        else A;
in dheap
};
```

4 Quick sort

In this section we consider two implementations of Quicksort, the first one using I-structures and the second one using M-structures. For the sake of efficiency we use two slightly different interpretations of quicksort. In the implementation with M-structures we use exchanges of elements in an array since exchanging is cheap when M-structures are used. On the other hand, as we have seen in the case of heap sort, implementing such exchanges with I-structures is extremely expensive and we should avoid it when possible. In the I-structure implementation of quick sort, instead of partitioning an array into two by a series of exchanges (as is usually done), we create three sub-arrays: (1) with elements that are smaller than the pivotal element (2) with elements that are bigger than the pivotal element and (3) with elements that are equal to the pivotal element. The routine is then called recursively for the lesser and bigger sub-arrays. We use one sweep over the array to compute the number of elements that are smaller, bigger or equal to the pivotal element. This information is used for subsequent allocation of the sub-arrays. This implementation of quicksort performed quite well compared to other algorithms. The instruction count for 256 elements in this case is only about 300,000 and critical path length being 30,000 (see Table 1). Fig. 4 shows a sort of gradual increase in the parallelism profile with an average parallelism of 10.2. An important point to note here is that since this routine is recursive, the loss in parallelism caused by a default k-bound of 1 does not exist and different instances of the recursion can run in parallel.

```
% QUICK SORT with I-structures

def qcksort d ={
typeof d = I_vector(I);
(_,n) = bounds d;
in
if(n == 1) then d
else
  { less =0; more = 0; eql =1; t1 =0; t2 =0;
    typeof d1 = I_vector(I); typeof d2 = I_vector(I);
    typeof outvec = I_vector(I); outvec = I_vector(1,n);
    less_ct, more_ct, eql_ct={for i <- 2 to n do
    next less = if (d[i] <d[1]) then less + 1  else less;
    next more = if (d[i] >d[1]) then more + 1  else more;
    next eql = if (d[i] == d[1]) then eql + 1  else eql;
    finally less, more,eql
  };
  d1 = I_vector(1,less_ct); d2 = I_vector(1,more_ct);
  {for i <- 2 to n do
      next t1, next t2 = if (d[i] <d[1])
                         then { d1[t1 +1] = d[i];
                               in t1 + 1,t2 }
                         else if (d[i] >d[1])
                              then { d2[t2 +1] = d[i];
                                    in t1,t2 + 1 }
                              else t1,t2
  };

  outvec2 = if((less_ct>0) and (more_ct >0))
    then {sortd1 = qcksort d1;
               sortd2 = qcksort d2;
               {for i <- 1 to less_ct do outvec[i] = sortd1[i]};
               {for i <- 1 to eql_ct do outvec[less_ct+i] = d[1]};
               {for i <- 1 to more_ct do
       outvec[less_ct+eql_ct+i] = sortd2[i]};
```

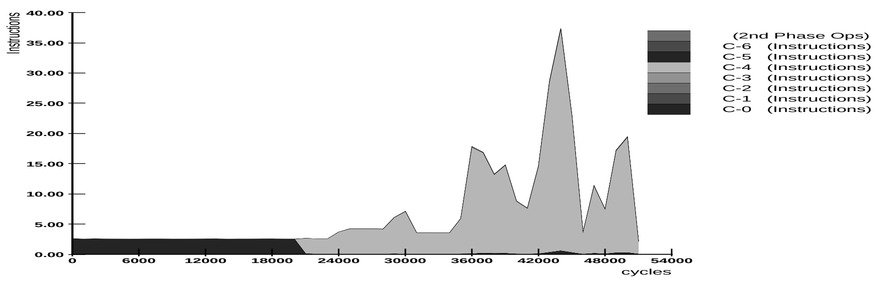

Figure 4: Sample idealized profile of quick sort with I-structures of 256 integers

```
                in outvec }
          else if((less_ct>0) and (more_ct == 0))
       then {sortd1 = qcksort d1;
                    {for i <- 1 to less_ct do outvec[i] = sortd1[i]};
                    {for i <- 1 to eql_ct do outvec[less_ct+i] = d[1]};
                    in outvec }
               else if((less_ct ==0) and (more_ct > 0))
    then {sortd2 = qcksort d2;
                         {for i <- 1 to eql_ct do outvec[i] = d[1]};
                         {for i <- 1 to more_ct do
      outvec[eql_ct+i] = sortd2[i]};
                          in outvec }
                    else { {for i <- 1 to eql_ct do outvec[i] = d[1]};
                          in outvec }
  in outvec2 }
};
```

The function *qcksortm* is an M-structure implementation of the same quicksort routine. Here the function partition divides the array into two by a series of exchanges. We start with a pointer at the left end of the array and move it to right until a key with larger value than the pivot is found. Another pointer at the right end is moved to the left until a key will value smaller than the pivot is found. We exchange these two keys and proceed likewise until the partition point is found. We use the function quicksort recursively on the partitioned sub-arrays. This method performs really well, having the minimum instruction count among all the methods we consider. For 256 elements this method has an instruction count of 244,558 and critical path length of only 17,000 (see Table 1). The parallelism profile has a similar appearance (see Fig. 5) as the I-structure implementation with average parallelism of 14.39.

```
% QUICK SORT with M-structures

def qcksortm A b u ={
typeof A = M_vector(I);
p = u-b+1;
in
```

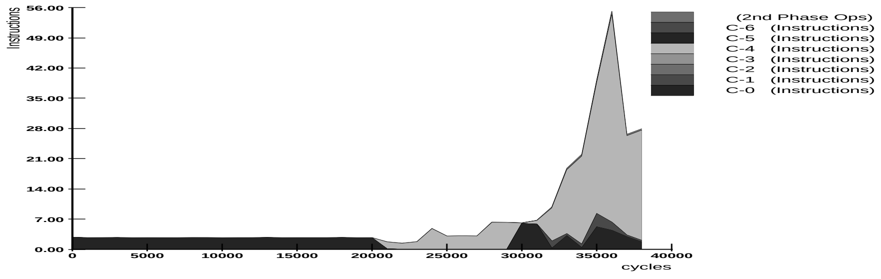

Figure 5: Sample idealized profile of quick sort with M-structures of 256 integers

```
if(p == 1) then A
    else
    {q = partition A b u;
    ---
    sort1 = qcksortm A b q;
    sort2 = qcksortm A (q+1) u;
    in sort2 }
};

def partition  A b u={
typeof A = M_vector(I);
x=A!![b];
i=b; j=u; temp =1;
partn={while (i <= j)  do
          p=j;
          tj ={while(A!![p] >x) do
                  next p=p-1;
              finally p };
          q=i;
          ti ={while(A!![q] < x) do
                  next q = q+1;
               finally q };
          next temp = if (ti < tj)
                      then {_=exchange A ti tj;
                            in tj-1}
                      else tj;
          next i, next j = (ti+1), (tj-1);
       finally temp };
in partn
};
```

5 Merge based Sorting

In this section we consider two algorithms which are based on merging of arrays. I-structures are used in implementation of both the algorithms. The function *mergesort* recursively builds a sorted array by merging two half sized sorted arrays. The actual merging operation of the sorted array is done by fixing two pointers to the arrays and moving them when an element of the pointer array is selected, while building a third array. This algorithm outperformed all the other methods, which came as a surprise! It has one of the lowest instruction counts and by far the lowest critical path length. In the case of mergesort we have deallocated intermediate arrays, as this could be a useful algorithm. However, we have not yet implemented explicit deallocation of the intermediate arrays, which may change the picture considerably. Table 1 shows that for 256 elements, this algorithm, including deallocation of I-structures, requires in the order of 200,000 instructions and has a critical path length of 8,000. Fig. 6 shows its idealized parallelism profile, and notice the phenomenal peak parallelism of about 150.

```
% MERGE SORT

def mergesort A low up ={
typeof A = I_vector(F);
p = up-low+1;
in
if(p == 1) then
        {typeof R = I_vector(F); R = I_vector(1,1); R[1] = A[up] in R}
        else
        { m = div p 2;
        typeof sorted = I_vector(F);
        typeof sort1  = I_vector(F);
        typeof sort2  = I_vector(F);
        sort1 = mergesort A low (low+m-1);
        sort2 = mergesort A (low+m) up;
        sorted = merge sort1 sort2;
        ---
         @release sort1; @release sort2;
        in
        sorted
        }
};


def merge A B ={
typeof A = I_vector(I); typeof B = I_vector(I);
(_,m) = bounds A; (_,n) = bounds B;
typeof merged = I_vector(I); merged = I_vector(1,m+n);
ctr = 1; i=1; j=1;
count, lasti, lastj =
    {while ((i <= m) and (j <= n)) do
         merged[ctr], next i, next j =
             if (A[i] < B[j]) then A[i],i+1, j
```

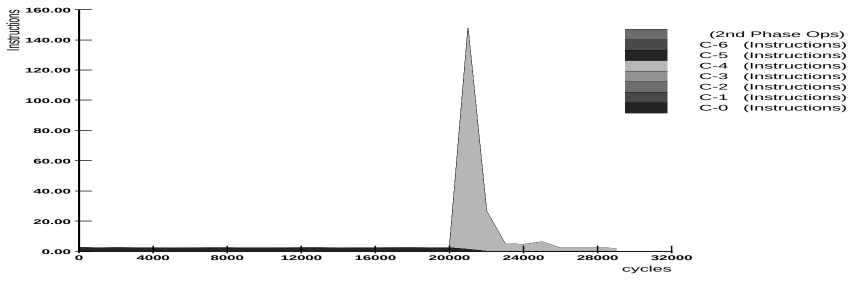

Figure 6: Sample idealized profile of merge sort of 256 integers

```
     else B[j], i, j+1;
         next ctr = ctr + 1;
      finally ctr, i, j };
ctr2 = 0;
{for k<- count to m+n do
     merged[k] = if(lastj > n)
                 then A[lasti + ctr2]
                 else B[lastj + ctr2];
     next ctr2 = ctr2 +1 }
in merged
};
```

The following code is that of Todd's sort which is a pipelined version of merge sort. Groups of size powers of two are merged and the group size is doubled at every step. Our computation here shows that this sort of algorithm that tries to extract parallelism explicitly does not pay off in the functional/dataflow environment. Our paradigm is already capable of pipelining simultaneously executable processes. Hence, trying to express it explicitly not only increases the instruction count but also curbs the implicitly available parallelism by a large extent. The parallelism is lost mainly due to the use of default bound of 1 in the loops and increasing this bound increases the instruction count excessively. Fig 7 shows the idealized parallelism profile and an average parallelism of 4.75 can be seen for an array of size 256. The number of instruction required in this method is 272,844 and critical path length is 57,400 (see Table 1).

```
% TODD SORT

def todd A ={
% Works only if dimension of input vector is a power of 2
typeof A = I_vector(I);
(_,p) = bounds A;
m = round (2.0^(log_2 p));
in
if(p == 1) then A
else { grpcnt = 1;
       dsort = A;
```

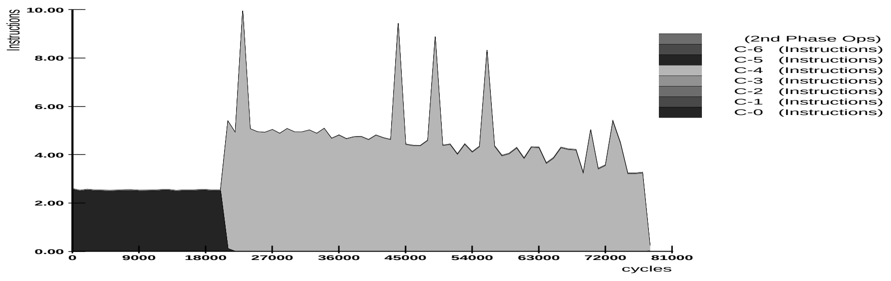

Figure 7: Sample idealized profile of todd sort of 256 integers

```
        sort={while (grpcnt < p) do
                i =1;
                typeof temp = I_vector(I); temp = I_vector(1,p);
                {while (i < p) do
                   typeof d1 = I_vector(I); d1 = I_vector(1,grpcnt);
                   typeof d2 = I_vector(I); d2 = I_vector(1,grpcnt);
                   {for k<- 1 to grpcnt do
                      d1[k] = dsort[i+k-1];
                      d2[k] = dsort[i+k-1+grpcnt] };
                   sorted = merge d1 d2;
                   {for k<- 1 to 2*grpcnt do temp[i+k-1] = sorted[k]};
                   next i = i+2*grpcnt };
                 next dsort = temp;
                 next grpcnt = grpcnt*2;
               finally dsort };
      in sort }
};


% This function returns (nearest power of 2 less than n)
def log_2 n ={
res = floor ((log10 (float n))/(log10 2.0))
in res
};
```

6 Sorting Networks

We consider two sorting algorithms based on sorting networks. Since sorting networks use element exchanges, use of M-structures seems natural. The following routine shows an implementation of odd-even sort in which only elements with an index difference (stride) of 2 are checked for exchange at every step. Alternate steps exchange only the odd or only the even indexed element, hence the name. The steps are performed n times, making the method an $O(n^2)$ one, and executing sequentially. Since for our implementation only one loop body is executed at one time this method

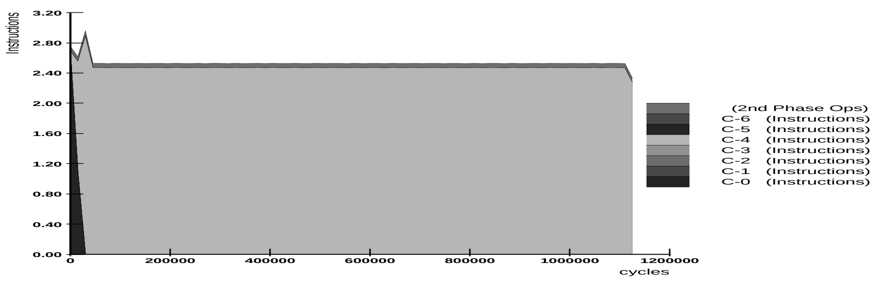

Figure 8: Sample idealized profile of oddeven sort of 256 integers

performs quite poorly. Fig. 8 shows the paralleism profile for the following implementation and an average parallelism of 2.53 for the input vector size 256. But a very high instruction count of over 2.8 million and critical path length of over 1.1 million for vector length of 256 questions the usefullnes of this algorithm.

Note, for this method default loop bound of one destroys almost all the parallelism of this method. In the following code one can see that two instances of the outer loop and $n/2$ instances of the inner loop can run simultaneously without any synchronization problem. We measured the performance with this modification and the row of oddeven sort (B) in table 1 describes it. We can observe that the ideal parallelism has increased about 8-fold, but at a significant cost of instruction count, which almost doubled in everey case. Fig 9. depicts the idealized parallelism profile of 256 integers for this case.

```
% ODDEVEN SORT

def oddeven d ={
typeof d = M_vector(I);
(_,n) = bounds d;
{for p<- 1 to n do
     i =  if ((mod p 2)==0) then 2 else 1;
     {while (i < n) do
          m = i+1;
          _ = check_and_exchange d i m;
          next i = i+2 }
}
in d
};
```

The last code in this memo is bitonic sort. This routine consists of 5 short functions: (a) half_cleaner: this function compares and exchanges (if necessary) all elements of the array that are half the array size apart; (b) premerger: works similarly to the half cleaner except the i-th element from the top and the bottom of the array are compared; (c) sort_biton: this function sorts an input of bitonic sequence (d) merge_biton: this function merges two sorted sequences and (e) bitonicsort: Sort any input array by recursion using the merge_biton function like normal merge sort. This algorithm has an O(n logn) instruction count and low critical path length. Fig 10 shows the ideal

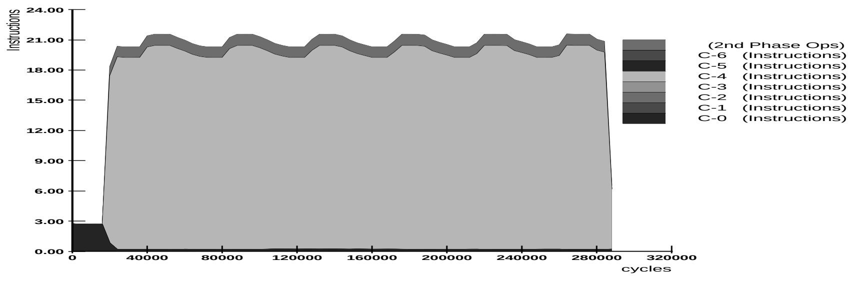

Figure 9: Idealized profile of oddeven sort with bounds of 256 integers

parallelism profile for this method for 256 elements which indicates an average parallelism of about 43. The instruction count is about 86,000 and critical path length is 19,000 for the same size.

```
% BITONIC SORT
% Number of elements needs to be a power of 2

def half_cleaner A b u={
typeof A = M_vector(I);
m = div (u-b+1) 2;
    {for j<-  b to (b+m-1) do
        next A = check_and_exchange A j (j+m) };
in A
};

def premerger A b u={
typeof A = M_vector(I);
m = div (u-b+1) 2;
    {for j<-  b to (b+m-1) do
        next A = check_and_exchange A j (u-j+b) };
in A
};

def sort_biton A b u={
typeof A = M_vector(I);
n = u-b+1;
in
if(n == 1) then A
  else
    {m = div n 2;
    D = half_cleaner A b u;
    ---
    sort1 = sort_biton D b (b+m-1);
    sort2 = sort_biton D (b+m) u;
    in sort2 }
};
```

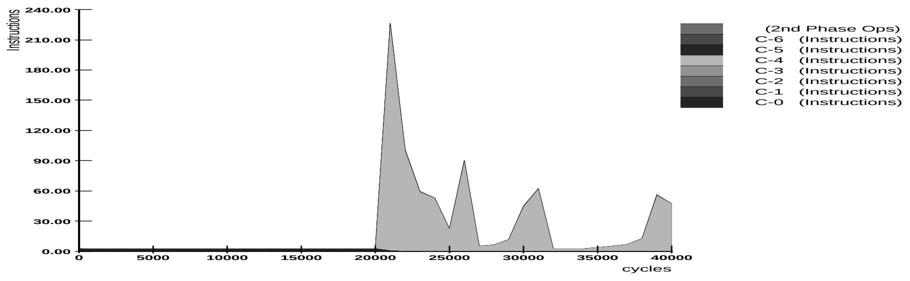

Figure 10: Sample idealized profile of bitonic sort of 256 integers

```
def merge_biton A b u={
typeof A = M_vector(I);
n = u-b+1;
in
if(n == 1) then A
  else
    {m = div n 2;
    D = premerger A b u;
    ---
    sort1 = sort_biton D b (b+m-1);
    sort2 = sort_biton D (b+m) u;
    in sort2 }
};

def bitonicsort A b u={
typeof A = M_vector(I);
n = u-b+1;
in
if(n == 1) then A
  else
    {m = div n 2;
    sort1 = bitonicsort A b (b+m-1);
    sort2 = bitonicsort A (b+m) u;
    ---
    mfinal = merge_biton sort2 b u
    in mfinal }
};
```

Vector length	128			256			512		
	S_1	S_∞	π	S_1	S_∞	π	S_1	S_∞	π
Bubble	688,802	257,500	2.67	2,688,000	972,000	2.77	10,468,594	3,690,000	2.84
Heap (I)	1,181,088	135,500	8.75	4,862,186	509,000	9.55	20,156,000	1,900,000	10.61
Heap (M)	211,898	69,000	3.07	496,735	159,700	3.11	1,121,594	354,000	3.17
Quick (I)	144,453	16,500	8.75	306,117	30,000	10.20	636,407	58,000	10.97
Quick (M)	108,950	10,500	10.38	244,558	17,000	14.39	532,992	46,000	11.58
Merge	93,873	5,000	18.77	199,937	8,000	24.98	424,497	19,000	22.34
Todd	126,841	26,000	4.88	272,844	57,400	4.75	585,343	118,000	4.96
Oddeven	708,565	280,500	2.53	2,826,186	1,118,000	2.53	11,288,347	4,455,000	2.53
Oddeven (B)	1,403,963	68,500	20.50	5,610,186	267,000	21.01	22,426,594	1,060,000	21.16
Bitonic	337,800	10,000	33.78	816,747	19,000	42.99	1,937,594	32,000	60.54

Table 1: Instruction Count, Critical Path length and avg. parallelism of the sorting routines

Vector length	32	64	128	256	512
Bubble $[n^2]$	42.2	42.1	42.0	41.0	39.9
Heap (I) $[n^2]$	82.2	74.5	72.1	74.2	76.9
Heap (M) $[nlogn]$	226.8	231.4	236.5	242.5	243.4
Quick (I) $[nlogn]$	184.6	168.8	161.2	149.5	138.1
Quick (M) $[nlogn]$	137.7	125.6	121.6	119.4	115.7
Merge $[nlogn]$	126.6	113.8	110.4	97.6	92.1
Todd $[nlogn]$	170.0	152.9	141.6	133.2	127.0
Oddeven $[n^2]$	44.1	43.1	43.2	43.1	43.1
Oddeven (B) $[n^2]$	86.6	86.0	85.7	85.6	85.6
Bitonic $[n(logn)^2]$	67.41	59.9	53.9	49.9	46.7

Table 2: Constants obtained by dividing the instruction counts by the order

7 Conclusion

We have studied nine sort algorithms and measured their performance on monsoon. Table 2 shows that, in terms of total work, the algorithms conform with their theoretical complexity. It turns out that M-structures improve the performance and sometimes the elegance of our algorithms, and that the classical sort algorithms outperform the explicit parallel ones. Mergesort outperforms all other algorithms, even with explicit deallocation.