

## Homework # 1 Solutions

### Ex 1: Partitioning

For the rowwise partition, processor  $i$  ( $1 \leq i \leq 4$ ) should get  $B_{i,1..4}$ . The boundary conditions belong in the same processor as the adjacent element of  $B$ ; that is, the top row and the top elements of the side columns go in processor 1, the second elements of the side columns go in processor 2, the third elements in processor 3, and the bottom elements, plus the entire bottom boundary, go in processor 4.

### Ex 2: Communication

The submesh partitioning requires only 16 communications per relaxation step, compared to 24 for the row partition, so the submesh is better.

### Ex 3: 64 Processors, Strips

If it takes one unit of work to compute one matrix element, then it takes  $w = 64$  units of work to compute one row. Each row (except for the top and bottom rows) needs the values of the rows above and below it, so  $c = 64 \times 2 = 128$ . Thus,  $T = sw + lc = (10)(64) + (1)(128) = 768$ .

If we were to compute this sequentially,  $w$  would be  $64 \times 64 = 4096$  (since one processor would have to compute all elements). Except for the initial loading of the processor with the matrices,  $c$  would be 0 since all “neighbor” elements are local. Thus,  $T = (10)(4096) + (1)(0) = 40960$ . This is a speedup of  $53\times$  over the sequential running time,  $T(1)/T(P) = \frac{53}{1}$ .

### Ex 4: 16 Processors, Tiles

Each processor is responsible for computing 256 nodes, so  $w = 256$ , assuming each node is one computational unit of work per iteration. Ignoring boundary conditions, processors require 16 node values from each of their four neighbors, so  $c = 16 \times 4 = 64$ . Thus,  $T = (10)(256) + (1)(64) = 2624$ . This is a speedup of  $15.6\times$  over the sequential running time.

### Ex 5: A General Expression for Jacobi

We have the aspect ratio

$$r = \frac{a}{b} \tag{1}$$

and we know  $a$  and  $b$  are the actual dimensions of each block. Each processor is assigned

$$ab = \frac{n^2}{P} \tag{2}$$

grid points. Solving equation 1 for  $b$  and equation 2 for  $a$  and substituting, we get

$$\begin{aligned} b &= \frac{a}{r} \\ a &= \frac{n^2}{Pb} \\ &= \frac{n^2 r}{Pa} \\ &= n\sqrt{\frac{r}{P}} \\ b &= n\sqrt{\frac{1}{rP}}. \end{aligned}$$

Finally, solving for  $c$  produces

$$\begin{aligned} c &= 2n \left( \sqrt{\frac{1}{rP}} + \sqrt{\frac{r}{P}} \right) \\ c &= \frac{2n}{\sqrt{P}} \left( \sqrt{\frac{1}{r}} + \sqrt{r} \right). \end{aligned}$$

### **Ex 6: Optimal Partitioning**

The communication per processor is directly proportional to the perimeter of the area simulated. Given a particular number of simulated processors, this is minimized when  $a = b$ , or the area is a square.

More formally, let  $p = ab$  be the number of grid points per processor. The communication cost is

$$\begin{aligned} c &= 2a + 2b \\ &= 2a + \frac{2p}{a}. \end{aligned}$$

Find an extremum of this value by setting its derivative to zero.

$$\begin{aligned} c' &= 2 - 2pa^{-2} \\ 1 &= pa^{-2} \\ a &= \sqrt{p} \\ b &= \frac{p}{a} \\ &= \sqrt{p} \\ &= a \end{aligned}$$

A simple check confirms that this produces a minimum, not a maximum, of the communication function.

### **Ex 7: Ordinary Speedup**

Using the solution above:

$$\begin{aligned}
w &= \frac{N}{P} \\
c &= 2a + 2b \\
&= 2n \left( \sqrt{\frac{1}{rP}} + \sqrt{\frac{r}{P}} \right) \\
&= \frac{4n}{\sqrt{P}} \\
T &= sw + lc \\
&= s \frac{N}{P} + l \frac{4n}{\sqrt{P}} \\
&= n \left( \frac{sn}{P} + \frac{4l}{\sqrt{P}} \right) \\
&= \frac{n}{\sqrt{P}} \left( \frac{sn}{\sqrt{P}} + 4l \right) \\
\text{Speedup} &= \frac{Psw}{T} \\
&= \frac{Ns}{n \left( \frac{sn}{P} + \frac{4l}{\sqrt{P}} \right)} \\
&= \frac{sn}{\frac{sn}{P} + \frac{4l}{\sqrt{P}}} \\
&= \frac{1}{\frac{1}{P} + \frac{4l}{sn\sqrt{P}}}
\end{aligned}$$

When we plug in the numbers from exercise 4, we come up with 15.6, as expected.

### **Ex 8: Scaled Speedup**

$N(P) = kP$  for  $k = 4096$ ; let  $n(P) = \sqrt{N(P)} = \sqrt{kP} = 64\sqrt{P}$ .

$$\begin{aligned}
\text{Scaled speedup} &= \frac{T(N(P), 1)}{T(N(P), P)} \\
&= \frac{N(P)s}{n(P) \left( \frac{sn(P)}{P} + \frac{4l}{\sqrt{P}} \right)} \\
&= \frac{n(P)s}{\frac{sn(P)}{P} + \frac{4l}{\sqrt{P}}} \\
&= \frac{\sqrt{kPs}}{\frac{\sqrt{ks}}{\sqrt{P}} + \frac{4l}{\sqrt{P}}} \\
&= \frac{P}{1 + \frac{4l}{s\sqrt{k}}} \\
&= \frac{P}{1 + \frac{l}{16s}}
\end{aligned}$$

### **Ex 9: Asymptotic Speedup**

There were a number of ways of considering this problem. Perhaps the most practical is to consider that  $w$ , the work per node per iteration cannot be less than 1. This is a reasonable assumption because there has been no indication to this point as to what communication between nodes would look like for  $w$  less than 1. After making this assumption it can be said that asymptotic speedup occurs when each processor performs 1 unit of work per cycle. In other words  $P = N$ . Additional processors would sit idle during this time since all nodes are covered by existing processors.

$$\begin{aligned}\text{Asymptotic Speedup} &= \frac{1}{\frac{1}{P} + \frac{4l}{sn\sqrt{P}}} \\ &= \frac{1}{\frac{1}{N} + \frac{4l}{sN}} \\ &= \frac{5}{7}N\end{aligned}$$

### **Ex 10: Speedup**

We use ordinary speedup when we have a given problem we want to solve faster by running it in parallel. We use scaled speedup when we expect the presence of larger machines to make us want to solve larger problems. We use asymptotic speedup when we have as many processors as we need and the only issue is getting the fastest possible response.

### **Ex 11: Optimal Rectangular Partitioning**

Suppose the optimal aspect ratio of the block is  $a : b$ . Vertical communication amounts to  $2ay$  per block and horizontal to  $2bx$ . If there are  $P$  processors, each one is responsible for  $\frac{N}{P} = ab$  grid points. The computation is quite similar to that of problem 5, above:

$$\begin{aligned}a &= n\sqrt{\frac{r}{P}} \\ b &= n\sqrt{\frac{1}{rP}} \\ c &= 2ay + 2bx \\ &= \frac{2n}{\sqrt{P}} \left( y\sqrt{r} + \frac{x}{\sqrt{r}} \right).\end{aligned}$$

Set the derivative to zero to find an extremum.

$$\begin{aligned}c' &= \frac{2n}{\sqrt{P}} \left( \frac{y}{2\sqrt{r}} - \frac{x}{2r\sqrt{r}} \right) \\ r &= \frac{x}{y}\end{aligned}$$

### **(Optional) Ex 12: Optimal Rectangular Partitioning**

No, the answer does not change, because the values that need to be fetched from or communicated to adjacent processors in order to do the  $A_{i+x',j}$  calculations is a subset of those that must be communicated in for the  $A_{i+x,j}$  calculations because  $x > x'$ . This would only change the results if, because  $x$  was less than  $a$ , a particular value was needed by two processors, *neither of which* had just computed it in the previous round.

### **Ex 13: Matrix Vector Product**

(a). We would like each element of  $A$  to be on the same processor as the element of  $s$  with which it will be multiplied. Therefore, partition  $A$ 's rows the same way that  $s$  was partitioned: element  $i$  of each row is assigned to processor  $\lfloor \frac{i}{N/P} \rfloor$ . The partial products which will be summed to make  $r_i$ , and which are distributed across all processors, then need to be sent to the processor responsible for  $r_i$ .

Alternately, we can distribute  $A$  similarly to the result vector  $r$ ; put row  $i$  of  $A$  on the same processor as  $r_i$ . Then only  $s$  needs to be distributed in order to carry out the matrix multiplication.

(b). If  $A$  is tridiagonal, then it only contains  $3n$  elements. This means that we can distribute the entire matrix to each processor and still no more storage per processor than in the dense case, if  $N/P > 2$ . We can then solve the equation via LU decomposition and recursive doubling; each processor can compute only those parts of most interest to it, and there need be absolutely no communication. (There may be more efficient algorithms, but none with less communication.)

If  $A$  is known *a priori*, we can perform LU decomposition *a priori* and then solve the problem by recursive doubling; we need not distribute  $A$  at all, since we'll only be working with its decomposition.

If we don't really care that communication be minimized, we can perform the recursive doubling in parallel.

### **Ex 14: Parallel Programming Exercises**

Attached is a solution to the parallel Jacobi relaxation that uses the blocking versions of iLib messages. Because the send and receive calls are blocking, we can't have two processes attempting to send to one another at the same time. In that scenario, both processes will block and the whole application will deadlock. To break the deadlock, we arbitrarily decide to have all the processes of even rank send to those of odd rank, then have the odd processes send to the evens. This is a common idiom for avoiding deadlock situations in parallel processing.

It is also possible to implement this same code using the non-blocking versions of iLib send and receive calls. In this case, you do not need to implement the odd-even trick for avoiding deadlock as you can have two processes post non-blocking sends to one another, then proceed to their respective receive calls.

If you use the non-blocking calls, you must pass an `ilibRequest` object to the call. This object is used to store the status of the request. Be sure to call `ilib_wait()` on the request object to clear its status. Otherwise, the call to `ilib_finish()` will hang.

```

#include <stdio.h>
#include <sys/archlib.h>
#include <ilib.h>

#define NUM_CPUS 16
#define PROBLEM_SIZE 64
#define ARRAYSIZE (PROBLEM_SIZE+2)
#define ITERS 12

#define DATA int

void print_array(DATA *array)
{
    int i;
    for (i = 1; i <= PROBLEM_SIZE; ++i)
        printf("%d ", array[i]);
    printf("\n");
}

void update_array(DATA *oldarr, DATA *newarr,
    int rank, int group_size, int local_writes)
{
    int from, to, i;
    int last_local_write = local_writes;

    from = 1;
    to = local_writes;

    for (i = from; i <= to; ++i)
        newarr[i] = (oldarr[i-1] + oldarr[i+1])/2;

    if( ((rank % 2) == 0) ) {
        if(rank+1 != group_size) {

            ilib_msg_send(ILIB_GROUP_SIBLINGS,
                rank+1, 0, &newarr[last_local_write], sizeof(DATA));
        }
        if(rank != 0) {
            ilib_msg_send(ILIB_GROUP_SIBLINGS,
                rank-1, 0, &newarr[1], sizeof(DATA));
        }
    }
    else {

```

```

ilibStatus status;

if(rank != 0) {
    ilib_msg_receive(ILIB_GROUP_SIBLINGS,
        rank-1, 0, &newarr[0], sizeof(DATA), &status);
}

if(rank+1 != group_size) {
    ilib_msg_receive(ILIB_GROUP_SIBLINGS,
        rank+1, 0,
        &newarr[last_local_write+1], sizeof(DATA), &status);
}
}

if( ((rank % 2) == 1) ) {
    if(rank+1 != group_size) {
        ilib_msg_send(ILIB_GROUP_SIBLINGS,
            rank+1, 0, &newarr[last_local_write], sizeof(DATA));
    }
    if(rank != 0) {
        ilib_msg_send(ILIB_GROUP_SIBLINGS,
            rank-1, 0, &newarr[1], sizeof(DATA));
    }
}
else {
    ilibStatus status;

    if(rank != 0) {
        ilib_msg_receive(ILIB_GROUP_SIBLINGS,
            rank-1, 0, &newarr[0], sizeof(DATA), &status);
    }

    if(rank+1 != group_size) {
        ilib_msg_receive(ILIB_GROUP_SIBLINGS,
            rank+1, 0,
            &newarr[last_local_write+1], sizeof(DATA), &status);
    }

}

}

}

#define SWAP(a,b,t) (((t) = (a)), ((a) = (b)), ((b) = (t)))

int main(void)

```

```

{
    int i, t;
    DATA *oldarr, *newarr, *tmp;
    int start, stop;
    int rank, size;
    int local_array_size;
    DATA master_array[ARRAYSIZE];

    ilib_init();

    if (ilib_proc_go_parallel(NUM_CPUS, NULL) != ILIB_SUCCESS)
        ilib_die("Failed to go parallel.");

    rank = ilib_group_rank(ILIB_GROUP_SIBLINGS);
    size = ilib_group_size(ILIB_GROUP_SIBLINGS);

    local_array_size = (PROBLEM_SIZE/size)+2;

    oldarr = (DATA *) malloc(local_array_size * sizeof(DATA));
    newarr = (DATA *) malloc(local_array_size * sizeof(DATA));

    if( rank == 0)
        oldarr[0] = newarr[0] = 32768;
    else
        oldarr[0] = newarr[0] = 0;

    for (i = 1; i < local_array_size; ++i)
        oldarr[i] = newarr[i] = 0;

    ilib_msg_barrier(ILIB_GROUP_SIBLINGS);
    start = get_cycle_count();
    for (t = 0; t < ITERS; ++t) {
        update_array(oldarr, newarr, rank, size, (PROBLEM_SIZE/size));
        SWAP(oldarr, newarr, tmp);
    }

    if(rank == 0) {
        ilibStatus status;
        memcpy(master_array+1, oldarr+1, (PROBLEM_SIZE/size)*sizeof(DATA));

        for(i = 1; i < size; i++) {
            ilib_msg_receive(ILIB_GROUP_SIBLINGS,
                i, 0,
                &master_array[(PROBLEM_SIZE/size)*i+1],
                (PROBLEM_SIZE/size)*sizeof(DATA), &status);
        }
    }
}

```

```
}
else {
    ilib_msg_send(ILIB_GROUP_SIBLINGS,
0, 0, &oldarr[1], (PROBLEM_SIZE/size)*sizeof(DATA));
}
ilib_msg_barrier(ILIB_GROUP_SIBLINGS);
stop = get_cycle_count();

if(rank == 0) {
    print_array(master_array);
    printf("Program took %d cycles\n", stop-start);
}

return 0;
}
```