

HW0 Solutions with Instructive Comments

exchange.s.c

SLOWER

```
int I;
int X[n];
for(i=0; i<n; i++) {
    X[i] = B[i];
    B[i] = A[i];
    A[i] = X[i];
}
```

FASTER

```
int x,tmp;
for(x=0; x<n; x++){
    tmp = A[x];
    A[x] = B[x];
    B[x] = tmp;
}
```

why? In the slower version an auxiliary array is used to store the temporary value in the copy. This means that the code will use assembly instructions that fetch memory more in the first than the second version. Using local variables in general is a good idea since a limited number of them can be stored in registers directly on the TCU or processor that is using them and they are fast.

exchange1.p.c

SLOWER

```
int X[n];
spawn (0,n-1) {
    X[$] = A[$];
    A[$] = B[$];
    B[$] = X[$];
}
```

FASTER

```
spawn (0,n-1) {
    int x;
    x = A[$];
    A[$] = B[$];
    B[$] = x;
}
```

why? Because reads and writes to X[n] go to shared memory which takes more time to access. Local variables declared at the beginning of a spawn block are stored in registers directly on the TCUs/cores used to do the spawn. These registers have a much faster access time. Local variables in the spawn block are independent for every TCU.

Exchange2.p.c

2 steps, 2n processors, correct according to project description

```
int x[n], y[n];
spawn(0, 2*n-1) {
    if ($%2 == 0) {
        x[$/2] = A[$/2];
    } else {
        y[$/2] = B[$/2];
    }
}
spawn(0, 2*n-1) {
    if ($%2 == 0) {
        A[$/2] = y[$/2];
    } else {
        B[$/2] = x[$/2];
    }
}
return 0;
```

4 steps, n processors, not the algorithm sought after

```
int x[n], y[n];
spawn(0, n-1) {
    x[$] = A[$];
    y[$] = B[$];
}
spawn(0, n-1) {
    B[$] = x[$];
    A[$] = y[$];
}
```

why? The first example is correct because with infinite processors each spawn can be completed in one step (counting only read/write instructions). This way the writes to the auxiliary arrays happen in one parallel step and the writes back to the original arrays happens in one step.

Even though the second program takes two steps per spawn (4 steps total), it runs much faster. This is because there is a startup cost in cycles for a spawn block and its threads. There are half as many threads in the second program so this cost is reduced compared to the slower/correct implementation.

It takes more than just a few assembly instructions inside of a spawn block before the savings of running the 2n parallel threads is better than running n threads that do twice as much serial work. Future programs will require you to write more code within a spawn block and the cost of the spawn block will not matter as much.

On the other hand, you may find that for several problems, the fastest solution is the one that breaks the problem into smaller chunks, "solves" those chunks in a serial way, and then merges the results at the end. For the "Exchange" problem, there is no additional work needed to merge the result, making this approach even more attractive. This is in part because starting and ending a thread comes with a fixed overhead, and one way to amortize these overheads is to spawn fewer threads which do more work, while still keeping all the parallel units utilized as much as possible. You

may guess that the best number of threads to spawn is 64 because of the number of processors *currently* in the hardware, however, the hardware also has a load balancing mechanism that works with more than 64 threads to manage, so some testing would have to be done to determine the best run time. 'In a future version, the XMTC compiler will automatically perform such optimizations, leaving the programmer to focus on the algorithm instead. I just wanted to make you all aware of exactly what is happening.