

# An Introduction to Parallel Programming Solutions, Chapter 1

Jinyoung Choi and Peter Pacheco

February 1, 2011

1.
 

```

quotient = n / p;
remainder = n % p;
if (my_rank < remainder) {
    my_n_count = quotient + 1;
    my_first_i = my_rank * my_n_count;
} else {
    my_n_count = quotient;
    my_first_i = my_rank * my_n_count + remainder;
}
my_last_i = my_first_i + my_n_count;
      
```
2. We are assigning blocks of elements to cores in order (the first  $n/p$  elements to core 0, the next  $n/p$  elements to core 1, so on). So, for example, if  $n = 12$  and  $p = 4$ , core 0 spends 12 milliseconds in the call to `Compute_next_value` ( $i = 0, 1, 2$ ), core 1 spends 30 milliseconds ( $i = 3, 4, 5$ ), core 2 spends 48 milliseconds ( $i = 6, 7, 8$ ), and core 3 spends 66 milliseconds ( $i = 9, 10, 11$ ). So clearly this assignment will do a very poor job of load balancing.

A better approach uses a *cyclic* assignment of the work to the cores:

```

/* assign[c][j] is the jth value of i assigned to core c */
/* work[c] is the total amount of work assigned to core c */
c = j = 0;
for (i = 0; i < n; i++) {
    work[c] += 2*(i+1);
    assign[c][j] = i;
    c = (c + 1) % p;
    if (c == 0) j++;
}
      
```

Prof. Timothy Rolfe of Eastern Washington University came up with a *much* better approach. He uses a cyclic assignment of the work to the cores, but he starts with the largest amount of work ( $2n$ ) and works backward through the work ( $2n - 2, 2n - 4, \dots, 4, 2$ ). However, he alternates between going forward ( $0, 1, \dots, p - 1$ ) and backward ( $p - 1, p - 2, \dots, 1, 0$ ) through the cores. For example, suppose  $p = 5$  and  $n = 23$ . Then the cyclic assignment outlined above will assign work as follows:

Core	Value of $i$					Total Work
0	0	5	10	15	20	110
1	1	6	11	16	21	120
2	2	7	12	17	22	130
3	3	8	13	18		92
4	4	9	14	19		100

On the other hand, Prof. Rolfe's solution assigns the work as follows:

Core	Value of $i$					Total Work
0	22	13	12	3	2	114
1	21	14	11	4	1	112
2	20	15	10	5	0	110
3	19	16	9	6		108
4	18	17	8	7		108

His algorithm can be described as follows:

```

j = 0; i = n-1;
while (i >= 0) {
    /* Go forward through cores */
    for (c = 0; c < p && i >= 0; c++) {
        work[c] += 2*(i+1);
        assign[c][j] = i;
        i--;
    }
    j++;

    /* Go backward through cores */
    for (c = p-1; c >= 0 && i >= 0; c-- ) {
        work[c] += 2*(i+1);
        assign[c][j] = i;
    }
    j++;
    i--;
}

```

```

        i--;
    }
    j++;
}

```

```

3. divisor = 2;
   core_difference = 1;
   sum = my_value;
   while ( divisor <= number of cores ) {
       if ( my_rank % divisor == 0 ) {
           partner = my_rank + core_difference;
           receive value from partner core;
           sum += received value;

       } else {
           partner = my_rank - core_difference;
           send my sum to partner core;
       }
       divisor *= 2;
       core_difference *=2;
   }

```

```

4. bitmask = 1;
   divisor = 2;
   sum = my_value;
   while ( bitmask < number of cores ) {
       partner = my_rank ^ bitmask;
       if ( my_rank % divisor == 0 ) {
           receive value from partner core;
           sum += received value;
       } else {
           send my_sum to partner core;
       }
       bitmask <<= 1;
       divisor *= 2;
   }

```

5. It could happen that some cores wait for non-existent cores to send values, and this would probably cause the code to hang or crash. We can simply add a condition,

```

    if (partner < number of cores) {

```

```

    receive value
    sum += received value
}

```

when a cores tries to receive a value from its partner to make sure the program will handle the case in which the number of cores isn't a power of 2.

6. (a) The number of receives is  $p - 1$ , and the number of additions is  $p - 1$ .
- (b) The number of receives is  $\log_2(p)$ , and the number of additions is  $\log_2(p)$ .

(c)

p	Original	Tree-Structured
2	1	1
4	3	2
8	7	3
16	15	4
32	31	5
64	63	6
128	127	7
256	255	8
512	511	9
1024	1023	10

7. The example is a combination of task- and data- parallelism. In each phase of the tree-structured global sum, the cores are computing partial sums. This can be seen as data-parallelism. Also, in each phase, there are two types of tasks. Some cores are sending their sums and some are receiving another cores partial sum. This can be seen as task-parallelism.
8. (a) Cleaning the place for the party, bringing food, scheduling the setup, making party posters, etc.
- (b) There are several locations to clean. We can partition them among the faculty.
- (c) For instance, we can assign the task of preparing the food and drinks to some of the faculty. Then, this group can be partitioned according to the types of food: some individuals can be responsible for hors d'oeuvres, some for sandwiches, some for the punch, etc.
9. (ESSAY)