Introduction to MPI Profiling with Allinea MAP

Welcome! These notes are a companion to a hands-on course run by Allinea staff, which comes with full example code and MAP trace files. If you haven't attended this course, take a look at http://www.allinea.com/help-and-resources/training/ and contact Allinea to book one. They're really good!

Session 1: Use the Source

So we have a simple example program that is attempting to generate some random numbers, sqrt them all then find the maximum of those square roots. It looks like this:

```
for(j = 0; j < ITERATIONS; j++){
    /* split the array into sections and send these to workers */
    if(myid == 0){
        subsize = (int)floor(ARRAY_SIZE/numprocs);
        sIndex = subsize + ARRAY_SIZE%numprocs;

        /* fill the input array with random numbers */
        srand(time(NULL));
        for(i = 0; i < ARRAY_SIZE; i++)
            numbers[i] = rand();

        /* send the size of each section to the workers */
        for(i = 1; i < numprocs; i++){
            MPI_Send(&subsize, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
            MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
        }
        for(i = 1; i < numprocs; i++){
            MPI_Recv(&result, 1, MPI_DOUBLE,  i, 0, MPI_COMM_WORLD, &status);
            maxes[i] = result;
        }
        max = maxes[1];
        for(i = 2; i < numprocs; i++){
            if(max < maxes[i]){
                max = maxes[i];
            }
        }
        results[j] = max;
    }
    else{ /* receive a subsection of the array and find the max in it */
        MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
        for(i = 0; i<subsize; i++)
            numbers[i] = sqrt(numbers[i]);
        result = numbers[0];
        for(i = 1; i<subsize; i++){
            if(result < numbers[i])
                result = numbers[i];
        }
        MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
}

//Calculates the average, removes the largest and smallest from all the runs
if(myid == 0){
    max = results[0];
    for(i = 0; i<ITERATIONS; i++){
        if(max < results[i])
            max = results[i];
    }
    printf("Max after %d iterations:%g\n", ITERATIONS, max);
}
```

*Note: if you look at the code more closely, you'll see this is an obviously flawed algorithm - a better solution would be to find the max of the random numbers and **then** take its square root, saving a lot of computation! This is a prime example why step-by-step optimization should **always** be combined with thinking about what the algorithm is trying to achieve. For the sake of this example, though, we're going to assume it's important to perform the square root on each number before finding the largest of them, which helps us mark the distinction between improving the **algorithm** and improving its **implementation**.*

We think it should run pretty well on multiple cores as there isn't much communication necessary during the actual computation. To find out how it performs we compile and run the program like this:

```
mark@Namaste-II ~/1-use-the-source/problem $ mpicc -g -O3 sqrtmax.c -o sqrtmax

mark@Namaste-II ~/1-use-the-source/problem $ time mpirun -n 2 ./sqrtmax
Max after 500 iterations:46340.9

real    0m8.076s
user    0m13.501s
sys     0m0.036s

mark@Namaste-II ~/1-use-the-source/problem $ time mpirun -n 4 ./sqrtmax
Max after 500 iterations:46340.9
```

```
real    0m7.393s
user    0m25.410s
sys     0m0.040s
```

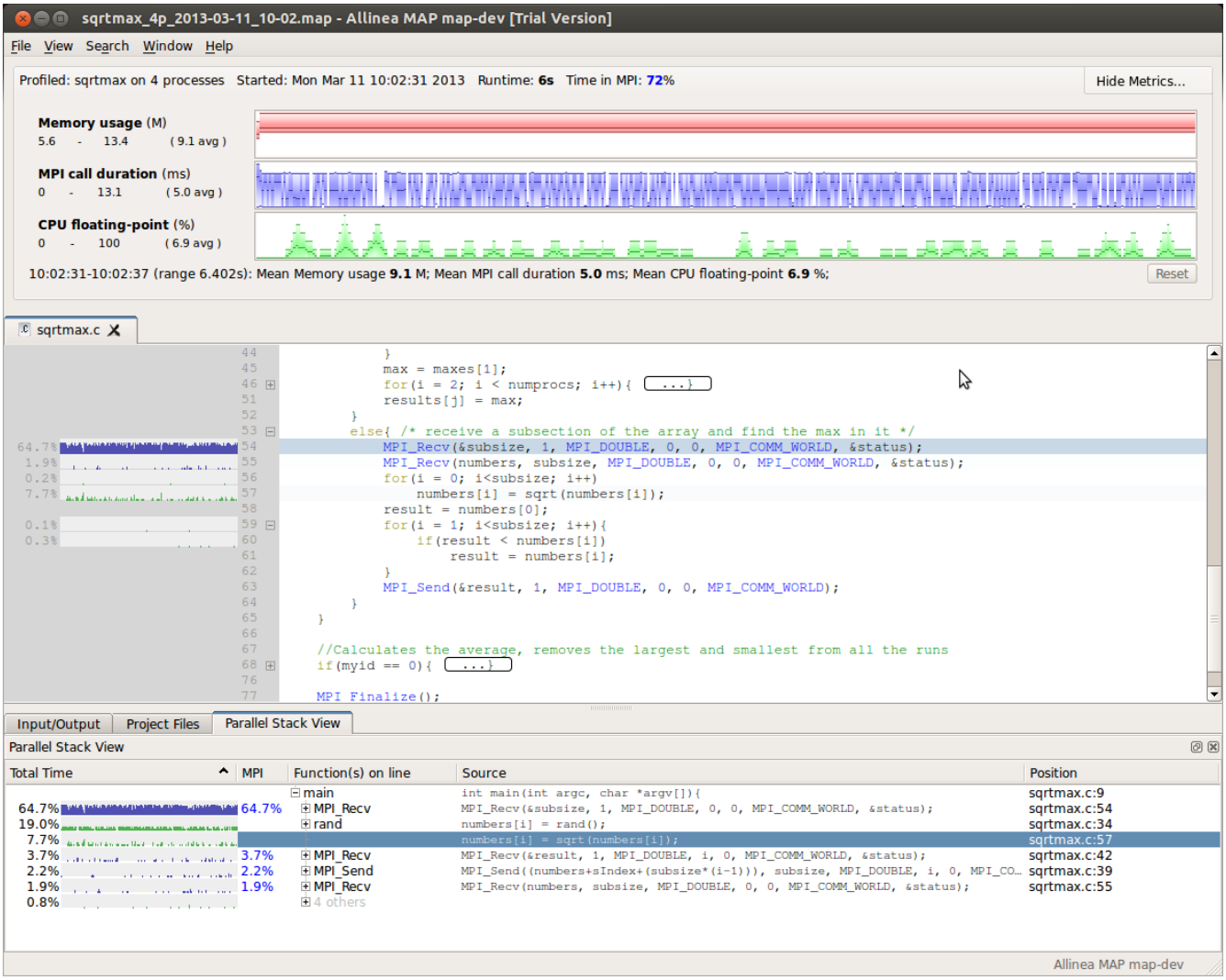
Even at this low scale there's very little parallel speedup: doubling the process count only decreases the wallclock time (real) to 91.5% of the 2-proc value:

```
mark@Namaste-II ~/1-use-the-source/problem $ bc -ql
7.393 / 8.076
.91542842991579990094
```

A linear speedup (difficult to achieve with real-world programs) would have decreased it to 50%, or 4.038 seconds. Let's see where all that extra compute power is being wasted:

```
mark@Namaste-II ~/1-use-the-source/problem $ map -n 4 ./sqrtmax
```

After the program finishes, MAP shows us this:



All MAP's percentages and charts are aggregates across all processes; the x-axis is always wall-clock time. There are a few interesting things to notice here:

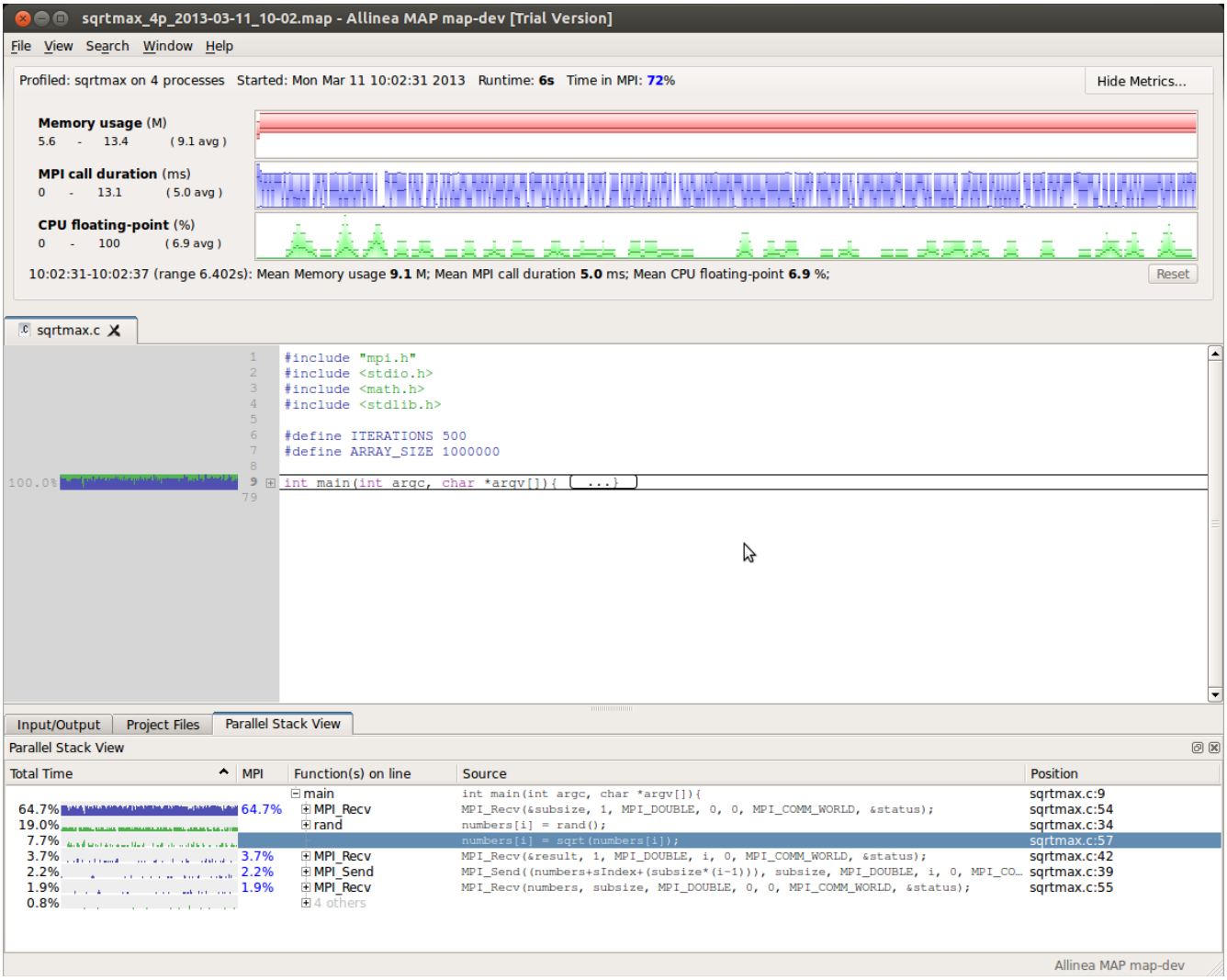
1. 72% of the total time is spent in MPI calls - that is, 3 out of the 4 processes are essentially contributing no computational work! Ideally this figure should be as low as possible, in practice the exact level achieved depends on the nature of the simulation and its communication requirements.
2. The CPU floating-point graph shows very low values, with the average being just 6.9% (shown on the left of the big green chart). This is another red flag - for codes of a computational nature, we want to see very high FPU usage.
3. The source code view is showing us a piece of code with a thick blue graph next to the MPI_Recv line. These graphs beside the source code show how much wall-clock time is spent on each line. Their x-axis is time, and their y-axis is the percentage of processes executing that line at each moment.

Note: MAP reports the runtime as 6.402s and 'time' reported it as 7.393s. Who is right? The answer is they are measuring different things. MAP only measures the time between MPI_Init and MPI_Finalize; 'time' also includes the setup time for launching the job and clearing it up afterwards. For this reason, using 'time mpirun' is not always a good idea, particularly with a non-production workload.

This snippet of source code is, then, telling us that 65% of the total job time over all processes is being spent on this line, calling MPI_Recv. That is, most processes spend most of their time here and not computing. This is our bottleneck, then - to gain speedup those processes need to be computing! So what is the cause of the bottleneck? Is there a lot of communication going on? Or is a rank being waited for here?

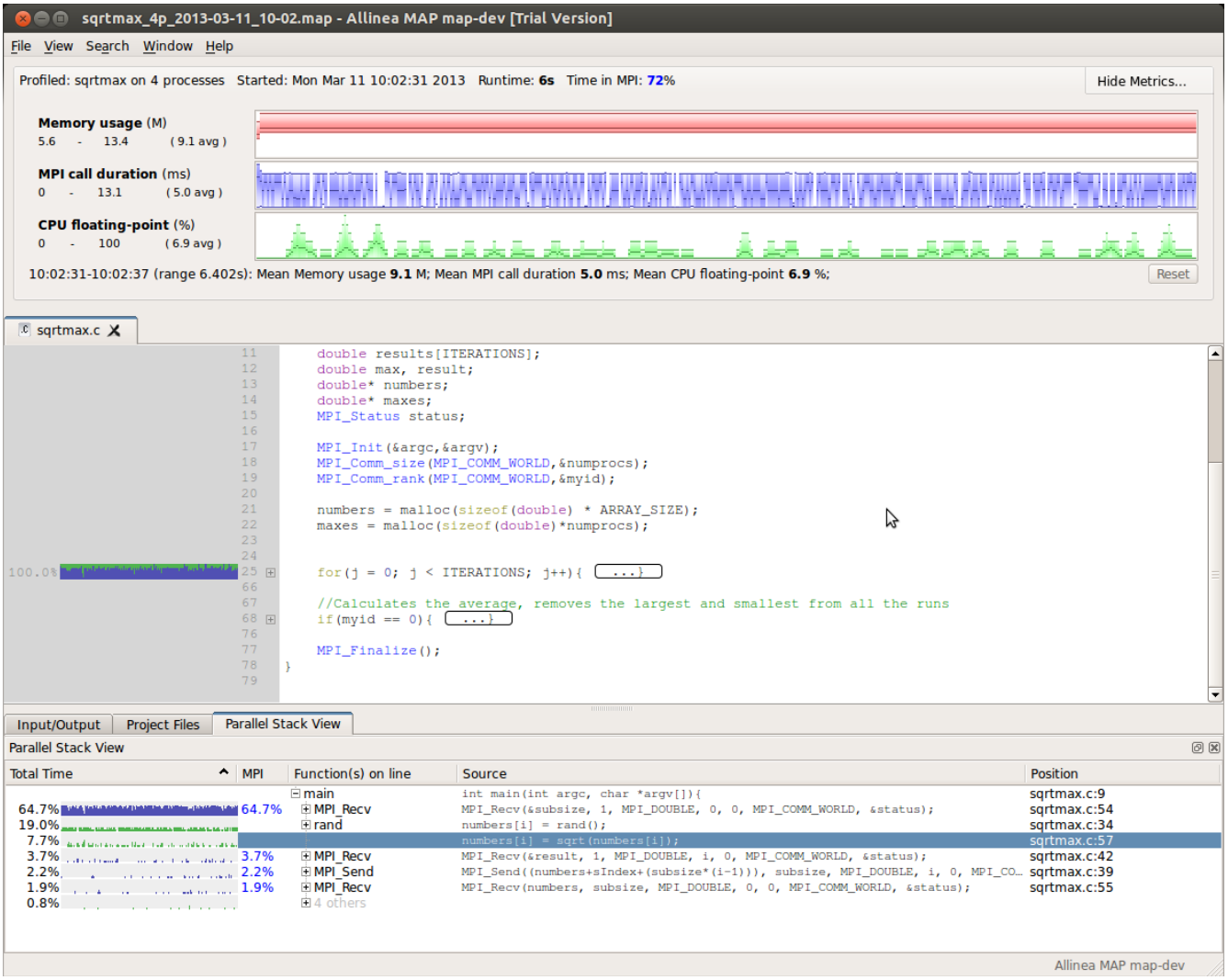
There are many ways to find track this sort of thing from the bottom-up, such as exploring MAP's MPI metrics further, or running the code in DDT and using the message queues window, or simply hitting Ctrl-F and searching for the matching call. We're not going to do any of those just yet - in unfamiliar code it almost always pays to get an overview of how the code is running as a whole first. We're going to use a couple of features of MAP's code viewer to do just that.

From the View menu, choose "Fold all". This uses a powerful feature of MAP's code editor to collapse blocks and loops, making complex code easier to navigate and understand:

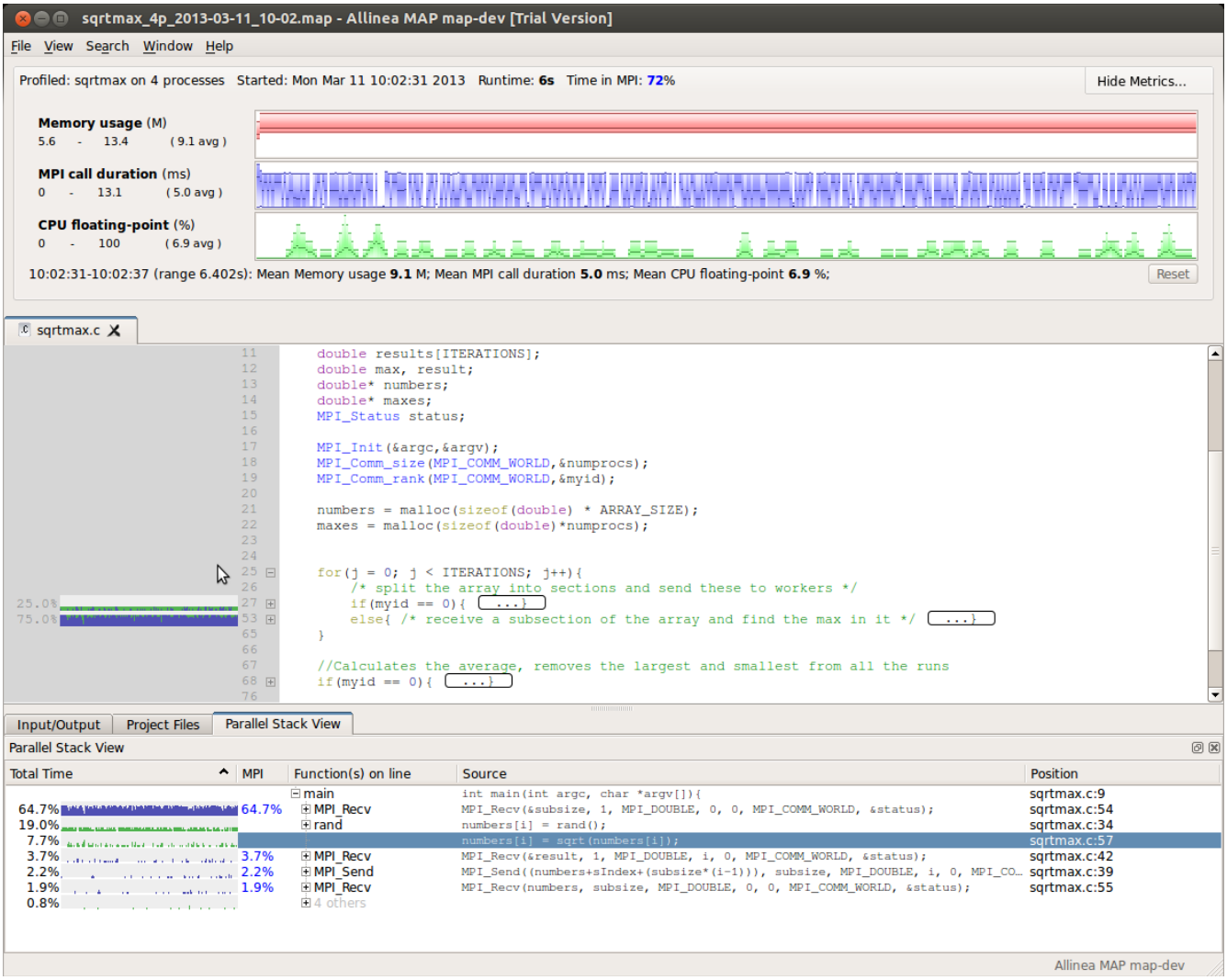


Here we can see that 100% of the time is spent in the main function (or functions called by it) - hardly a surprise - and if we hover over the green/blue coloured line a tooltip tells us 72% of that time is spent in MPI calls. In these little charts green represents time spent in user code or libraries and blue represents time spent in MPI calls. We can see that the proportion of time over all processes in MPI calls is roughly constant during this job.

Click on the small [+] to expand the main block, then scroll down to the end:

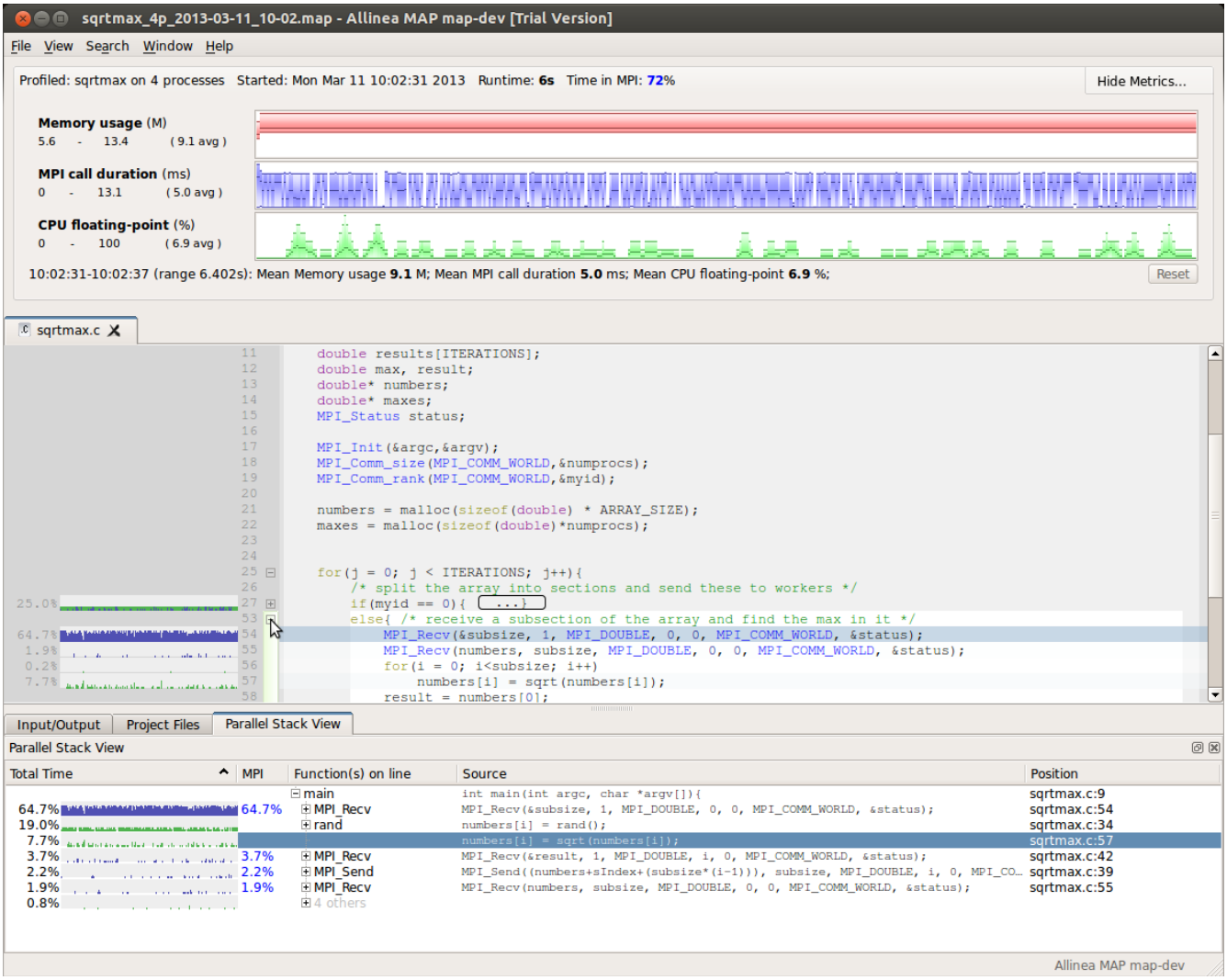


We see that the main function does a little initialization then enters a loop ITERATIONS times. All the time is spent inside this loop. Click the [+] to expand it:

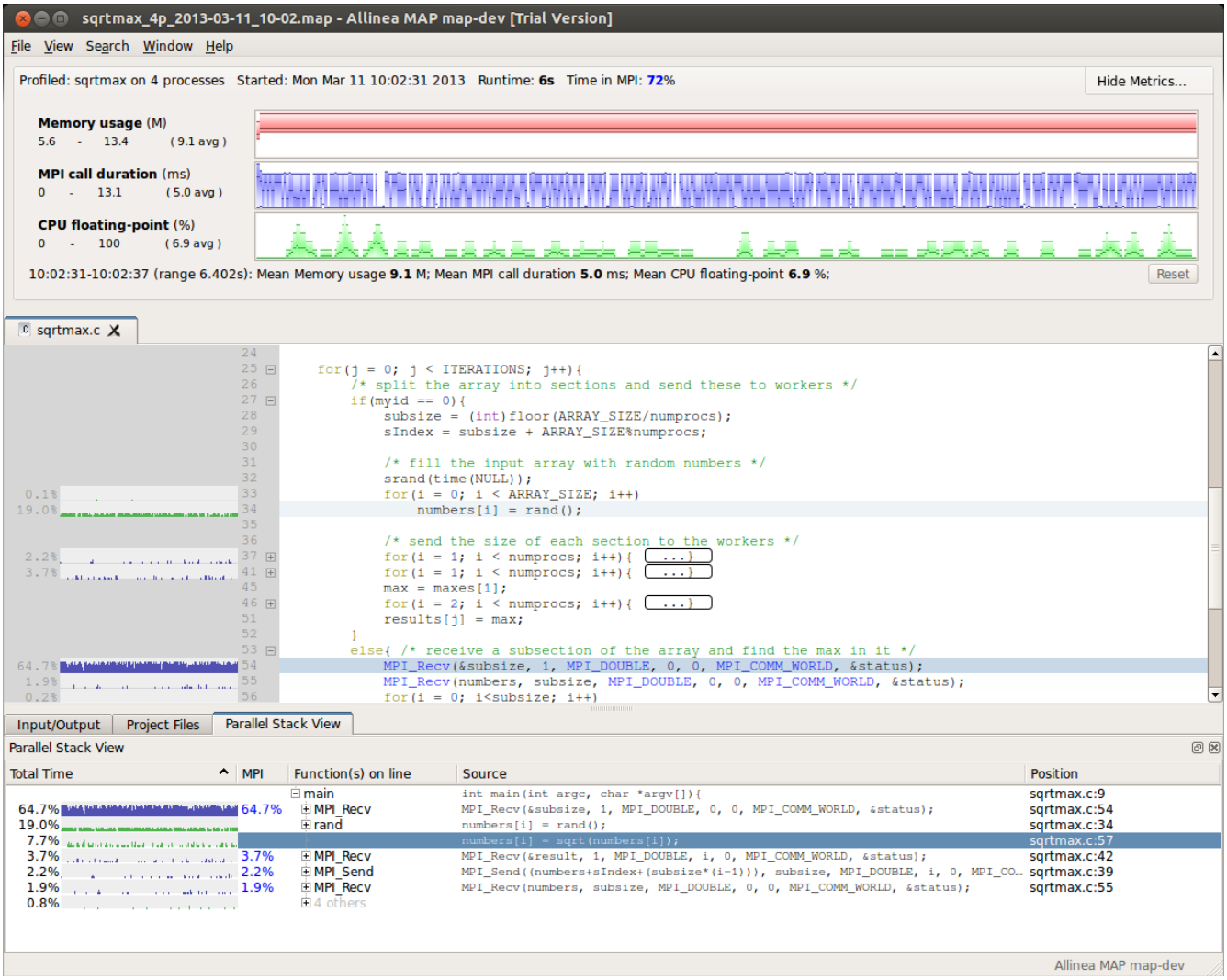


Now we see the first interesting information about the program. Inside the loop is an “if” statement, giving rank 0 a different path to the rest of the ranks. We can also see this in the graphs - exactly 25% of the time is spent in the first branch (rank 0 is 1 of 4 processes) and 75% is spent in the rest. We can also see that although rank 0 mixes communication with computation (blue and green), ranks 1-3 spend almost all their time in MPI calls (blue). This is not making good use of their CPUs!

We can expand the bottom part of the if statement (the 75% branch) and see the same code that MAP showed us at the start:



Lots of time spent in an MPI_Recv waiting for a single double to be sent from rank 0. So what is rank 0 doing with its time instead of sending this double over? Expand its part of the if statement to see:



The graph next time the line numbers[i] = rand() makes the cause of the bottleneck crystal clear - rank 0 is spending almost all of its time computing random numbers while the other ranks are all waiting! Actually sending the data once it has been prepared takes only a few % of the total runtime, insignificant in comparison.

To improve this code at all, we're going to have to fix that by parallelizing the random number generation. We're assuming for the sake of the example that it's important to start each iteration with fresh random numbers each time!

There are several ways to do this; one simple one is shown in the solution/ directory, in which the code to calculate random numbers has simply been moved onto the workers.

Exercise 1: Compile and run the solution - how much faster is it? What is the speedup now? How much time is spent waiting in MPI now? What is the FPU utilization at now? What should we concentrate on to improve this code further?

Answers: The program now completes in just 3.6s on my laptop (44% of the 2-process time, e.g. slightly better than linear speedup. Working out why this is possible is an extension). Now around 33% of the time is spent waiting in MPI and FPU utilization is up to 20%. Further improvements should

concentrate on increasing the FPU utilization and not on reducing the MPI overhead. We do exactly this in the next session.

Extension: Why have we apparently acheived a better-than-linear speedup?

Answer: Rank 0 doesn't participate in sqrt computations, so effectively we're comparing 1 compute core to 3 compute cores, not 2 to 4. Clearly we could boost performance by up to 25% by allocating rank 0 a workload too, although this benefit drops off quickly as the application scales up.

Exercise 2: This solution has overlooked something - the entire numbers array is still being sent from the root to the workers. Does that matter? Does it matter at this scale? Compile and run the solution code under MAP to find out.

Answer: It matters a little, but perhaps surprisingly not a lot at this scale. 3% of the time is spent sending it, 3.5% of the time is spent receiving it, so it's a 6.5% overhead. Almost as much time is spent receiving the 'subsize' variable (4.5%) - which is surprisingly high.

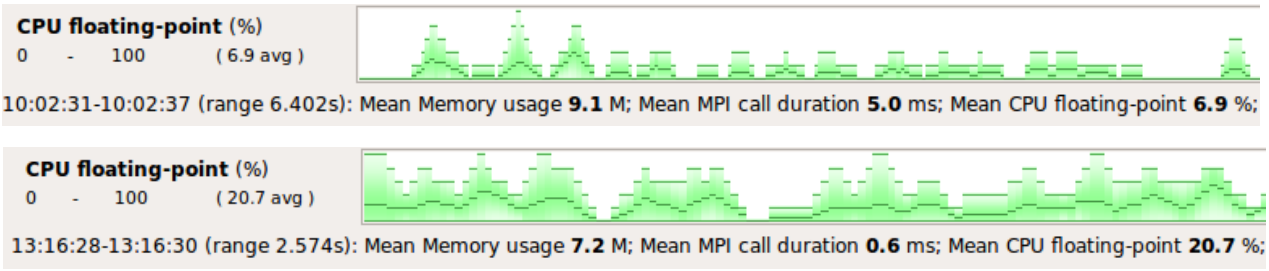
Extension: Why is 4.5% of the time being spent receiving the subsize variable - more than on receiving the entire numbers array?

Answer: Again the answer lies with the corresponding MPI_Send. Rank 0 sends the subsize variable to rank 1, then numbers to rank 1, then subsize to rank 2 etc. This means that rank 3 waits longest to receive subsize each time. MPI_Scatter would be the right way to distribute these variables. Notice that the MPI_Send for the result at the end of the loop takes very little time despite being of the same size. This is because rank 0 is always waiting to receive.

Review

In this session we were introduced to MAP and how to measure the performance of MPI programs. We used code folding in MAP's source code viewer to explore and understand an unfamiliar program and to find not just a bottleneck, but also the cause of that bottleneck.

With a relatively small change we were able to parallelize the perhaps surprising serial bottleneck in the code and reduce program runtime by well over 50%, a 2x speedup:



How much speedup can you get from one of your codes? Write to me and let me know – I answer every email, even the ones about dubious-sounding financial investments in third-world countries.

Happy profiling!
Mark O'Connor
mark@allinea.com.