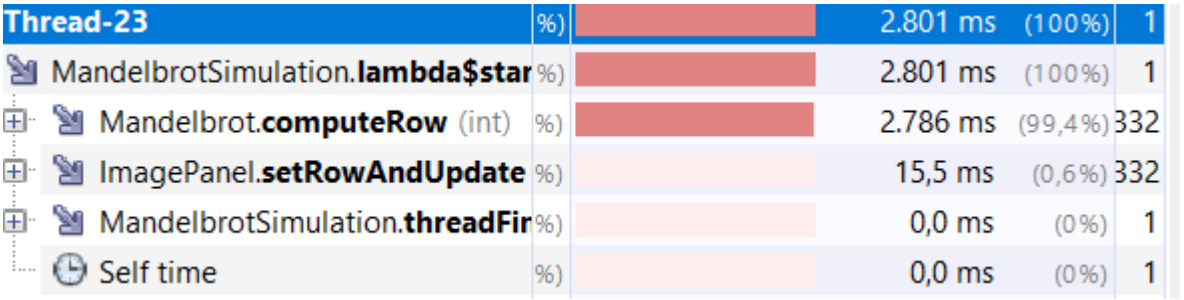
Nel programma, vengono avviati n thread per calcolare l'immagine. Ogni thread elabora in autonomia una porzione verticale dell'immagine, pari a: altezza/numThread, cioè una suddivisione equa delle righe dell'immagine tra tutti i thread.

Grazie all'uso del Sampler , si osserva che ogni thread impiega quasi il 100% del suo tempo di CPU nella funzione computeRow().



Questo è coerente con il comportamento atteso, visto che ciascun thread viene lanciato per eseguire esclusivamente il calcole delle righe assegnate e lo fa in un ciclo che processa riga per riga:

```
workers[i] = new Thread(() -> {
    try {
        // Compute one row of pixels.
        for (int row = startRow; row <= endRow; row++) {
            final int[] rgbRow = fractal.computeRow(row);
            // Check for the signal to abort the computati
            if (!running) {
                return;
            }
            imagePanel.setRowAndUpdate(rgbRow, row);
        }
    } finally {
```

L'unica operazione aggiuntiva svolta da ciascun thread, oltre al calcolo puro, è l'invocazione di setRowAndUpdate(rgbRow,row) . Questa funzione aggiorna il BufferedImage condiviso con i dati calcolati per la riga corrente e richiede un lock (imageLock) per garantire un accesso sicuro all'immagine. Inoltre, richiama repaint() per aggiornare visivamente solo quella riga.

```
1 usage
public void setRowAndUpdate(final int[] rowData, final int row) {
    final int width = getWidth();

    // Image is a shared resource!
    imageLock.lock();
    try {
        image.setRGB( startX: 0, row, width,  h: 1, rowData,  offset: 0, width);
    } finally {
        imageLock.unlock();
    }
    // Repaint just the newly computed row.
    imagePanel.repaint( x: 0, row, width,  height: 1);
}
```

Sebbene questa operazione comporti un minimo overhead legato alla sincronizzazione(lock) e all'I/O grafico, il suo impatto sul tempo CPU è trascurabile rispetto al lavoro computazionale svolto in computeRow(), che rimane il vero carico predominante del thread.