Introduction to High Performance Computing

SDS406 – Fall semester, 2024 - 2025

L12: Further Optimization Topics, 16th December 2024

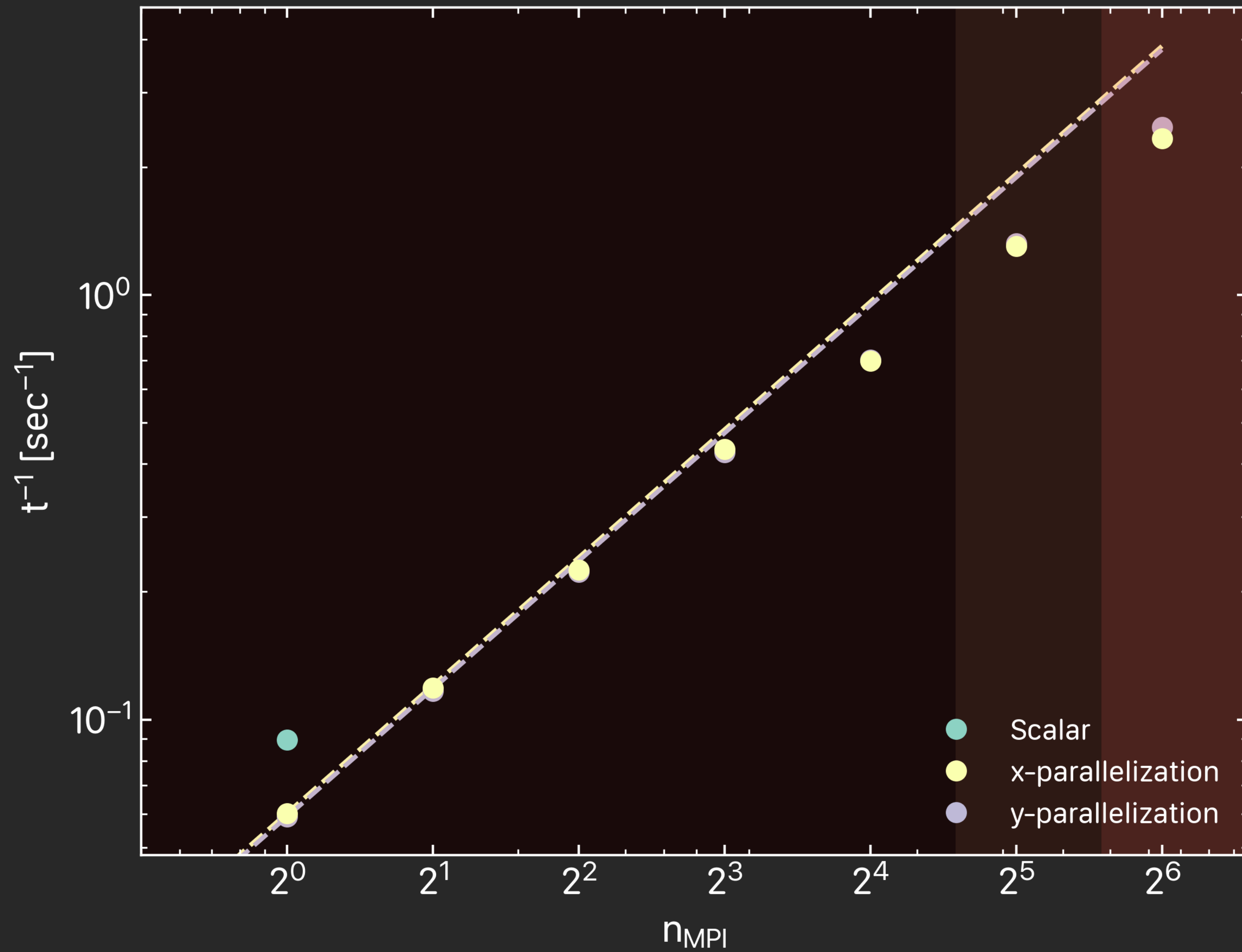
Strong scaling of the MPI-parallelized heat equation solution

$\Rightarrow$ Use at most two nodes, 32 processes per node

$\Rightarrow$ Plot against inverse time-to-solution rather than speed-up

$\Rightarrow$ Scale both parallelizations over x and y separately

$\Rightarrow$ Include in the plot the scalar version of at $n_{\mathrm{proc}} = 1$



n_{MPI}	**par-x [s^{-1}]**	**par-x [s]**	**par-y [s^{-1}]**	**par-y [s]**
1	0.060	16.622	0.059	16.940
2	0.119	8.432	0.117	8.566
4	0.225	4.446	0.222	4.497
8	0.434	2.306	0.426	2.349
16	0.699	1.430	0.702	1.424
32	1.301	0.769	1.321	0.757
64	2.331	0.429	2.482	0.403

- Scalar, $t = 11.2$ s $\Rightarrow t^{-1}$ = 0.09 s^{-1}

Outline

A closer look at the heat-equation

- Heat equation code
    - Scalar code
    - Add OpenMP
    - Implement a set of further optimization strategies
- Parallel program using MPI
    - Start from optimized CPU code and implement MPI
    - Mix OpenMP and MPI
    - Evaluate impact of MPI

Optimizing the heat equation code -- no MPI

`ex01` implements the scalar heat equation

```
[ikoutsou@front02 SDS406]$ mkdir l12
[ikoutsou@front02 SDS406]$ cd l12
[ikoutsou@front02 SDS406]$ cp -r /onyx/data/sds406f24/l12/ex01 .
[ikoutsou@front02 SDS406]$ cd ex01/
```

- Use `module load gompi/2023a`
- Compile with `-O3`
- Run with `srun` on the `p100` partition. Use `L=256`, `T=6000`, `alpha=0.2`
- Backup `v.bin` as a reference. We will be comparing outputs to it as we go along, checking our code as we optimize it.

Optimizing the heat equation code -- no MPI

`ex01` implements the scalar heat equation

```
[ikoutsou@front02 SDS406]$ mkdir l12
[ikoutsou@front02 SDS406]$ cd l12
[ikoutsou@front02 SDS406]$ cp -r /onyx/data/sds406f24/l12/ex01 .
[ikoutsou@front02 SDS406]$ cd ex01/
```

- Use `module load gompi/2023a`
- Compile with `-O3`
- Run with `srun` on the `p100` partition. Use `L=256`, `T=6000`, `alpha=0.2`
- Backup `v.bin` as a reference. We will be comparing outputs to it as we go along, checking our code as we optimize it.

```
[ikoutsou@front02 ex01]$ module load gompi/2023a
[ikoutsou@front02 ex01]$ cc -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 -p p100 ./heat 256 6000 0.2
 L = 256 T = 6000 t0 = 8.849914 sec
[ikoutsou@front02 ex01]$ mv v.bin v-reference.bin
```

Optimizing the heat equation code -- no MPI

`ex01` implements the scalar heat equation

```
[ikoutsou@front02 SDS406]$ mkdir l12
[ikoutsou@front02 SDS406]$ cd l12
[ikoutsou@front02 SDS406]$ cp -r /onyx/data/sds406f24/l12/ex01 .
[ikoutsou@front02 SDS406]$ cd ex01/
```

- Use `module load gompi/2023a`
- Compile with `-O3`
- Run with `srun` on the `p100` partition. Use `L=256`, `T=6000`, `alpha=0.2`
- Backup `v.bin` as a reference. We will be comparing outputs to it as we go along, checking our code as we optimize it.

```
[ikoutsou@front02 ex01]$ module load gompi/2023a
[ikoutsou@front02 ex01]$ cc -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 -p p100 ./heat 256 6000 0.2
 L = 256 T = 6000 t0 = 8.849914 sec
[ikoutsou@front02 ex01]$ mv v.bin v-reference.bin
```

Note that this original version takes ~9 seconds

Optimizing the heat equation code -- no MPI

Task A of `ex01`

In this part, we will enable OpenMP and report the sustained floating-point performance and bandwidth to memory

Optimizing the heat equation code -- no MPI

Task A of `ex01`

In this part, we will enable OpenMP and report the sustained floating-point performance and bandwidth to memory

- Include the `omp.h` header file

Optimizing the heat equation code -- no MPI

Task A of `ex01`

In this part, we will enable OpenMP and report the sustained floating-point performance and bandwidth to memory

- Include the `omp.h` header file
- Add `pragma omp parallel for` above the for-loops of the `update()` and `boundary_condition()` functions

Optimizing the heat equation code -- no MPI

Task A of `ex01`

In this part, we will enable OpenMP and report the sustained floating-point performance and bandwidth to memory

- Include the `omp.h` header file
- Add `pragma omp parallel for` above the for-loops of the `update()` and `boundary_condition()` functions
- Report the number of threads, floating-point performance, and sustained bandwidth

Optimizing the heat equation code -- no MPI

Task A of `ex01`

In this part, we will enable OpenMP and report the sustained floating-point performance and bandwidth to memory

- Include the `omp.h` header file
- Add `pragma omp parallel for` above the for-loops of the `update()` and `boundary_condition()` functions
- Report the number of threads, floating-point performance, and sustained bandwidth
- Add `-fopenmp` when compiling

Optimizing the heat equation code -- no MPI

Task A of `ex01`

Compile and run with a single OpenMP thread:

Optimizing the heat equation code -- no MPI

Task A of `ex01`

Compile and run with a single OpenMP thread:

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ OMP_NUM_THREADS=1 srun -n 1 -p p100 ./heat 256 6000 0.2
 nthr =   1 | L = 256 | T = 6000 | t0 = 8.633438 sec | p = 2.733e-01 Gflop/s | b = 7.330e-01 GB/s
```

Optimizing the heat equation code -- no MPI

Task A of `ex01`

Compile and run with a single OpenMP thread:

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ OMP_NUM_THREADS=1 srun -n 1 -p p100 ./heat 256 6000 0.2
 nthr =   1 | L = 256 | T = 6000 | t0 = 8.633438 sec | p = 2.733e-01 Gflop/s | b = 7.330e-01 GB/s
```

Bandwidth is less than 1 GBytes/s, where peak is $\sim$80 GBytes/s, i.e. we are obtaining performance at less than 2% of peak.

Optimizing the heat equation code -- no MPI

Task A of `ex01`

Compile and run with a single OpenMP thread:

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ OMP_NUM_THREADS=1 srun -n 1 -p p100 ./heat 256 6000 0.2
 nthr =   1 | L = 256 | T = 6000 | t0 = 8.633438 sec | p = 2.733e-01 Gflop/s | b = 7.330e-01 GB/s
```

Bandwidth is less than 1 GBytes/s, where peak is $\sim$80 GBytes/s, i.e. we are obtaining performance at less than 2% of peak.

Run with 16 OpenMP threads on a single socket:

Optimizing the heat equation code -- no MPI

Task A of `ex01`

Compile and run with a single OpenMP thread:

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ OMP_NUM_THREADS=1 srun -n 1 -p p100 ./heat 256 6000 0.2
 nthr =   1 | L = 256 | T = 6000 | t0 = 8.633438 sec | p = 2.733e-01 Gflop/s | b = 7.330e-01 GB/s
```

Bandwidth is less than 1 GBytes/s, where peak is ~80 GBytes/s, i.e. we are obtaining performance at less than 2% of peak.

Run with 16 OpenMP threads on a single socket:

```
[ikoutsou@front02 ex01]$ export OMP_PROC_BIND=true
[ikoutsou@front02 ex01]$ export OMP_PLACES="{0}:16:1"
[ikoutsou@front02 ex01]$ export OMP_NUM_THREADS=16
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64 -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 1.152561 sec | p = 2.047e+00 Gflop/s | b = 5.491e+00 GB/s
```

Optimizing the heat equation code -- no MPI

Task A of `ex01`

Compile and run with a single OpenMP thread:

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ OMP_NUM_THREADS=1 srun -n 1 -p p100 ./heat 256 6000 0.2
 nthr =   1 | L = 256 | T = 6000 | t0 = 8.633438 sec | p = 2.733e-01 Gflop/s | b = 7.330e-01 GB/s
```

Bandwidth is less than 1 GBytes/s, where peak is ~80 GBytes/s, i.e. we are obtaining performance at less than 2% of peak.

Run with 16 OpenMP threads on a single socket:

```
[ikoutsou@front02 ex01]$ export OMP_PROC_BIND=true
[ikoutsou@front02 ex01]$ export OMP_PLACES="{0}:16:1"
[ikoutsou@front02 ex01]$ export OMP_NUM_THREADS=16
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64 -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 1.152561 sec | p = 2.047e+00 Gflop/s | b = 5.491e+00 GB/s
```

This is ~6% of single-socket peak bandwidth. We will first optimize this CPU version before adding MPI

Optimizing the heat equation code -- no MPI

Task B of `ex01`

In `update()`, we are allocating and freeing a temporary array in each call

Optimizing the heat equation code -- no MPI

Task B of `ex01`

In `update()`, we are allocating and freeing a temporary array in each call

```
void
update(double *v, double alpha)
{
  /* Allocate a temporary field */
  double *v0 = ualloc(sizeof(double)*L*L);

  /* Copy current field to temporary field */
  memcpy(v0, v, L*L*sizeof(double));

   ...

  /* Don't need temporary field any more */
  free(v0);
  return;
}
```

Optimizing the heat equation code -- no MPI

Task B of `ex01`

In `update()`, we are allocating and freeing a temporary array in each call

```
void
update(double *v, double alpha)
{
  /* Allocate a temporary field */
  double *v0 = ualloc(sizeof(double)*L*L);

  /* Copy current field to temporary field */
  memcpy(v0, v, L*L*sizeof(double));

  ...

  /* Don't need temporary field any more */
  free(v0);
  return;
}
```

This may be suboptimal. Change `update()` to accept an input and output array `v1` and `v0`:

Optimizing the heat equation code -- no MPI

Task B of `ex01`

In `update()`, we are allocating and freeing a temporary array in each call

```
void
update(double *v, double alpha)
{
  /* Allocate a temporary field */
  double *v0 = ualloc(sizeof(double)*L*L);

  /* Copy current field to temporary field */
  memcpy(v0, v, L*L*sizeof(double));

  ...

  /* Don't need temporary field any more */
  free(v0);
  return;
}
```

This may be suboptimal. Change `update()` to accept an input and output array `v1` and `v0`:

```
void
update(double *v1, double *v0, double alpha)
{
#pragma omp parallel for
  for(int y=0; y<L; y++)
    for(int x=0; x<L; x++) {
      ...
      v1[r00] = (1-4*alpha)*v0[r00] + ... ;
```

Optimizing the heat equation code -- no MPI

Task B of `ex01`

We now need to allocate **two arrays** in the main program, and call `update()` alternating between input and output

Optimizing the heat equation code -- no MPI

Task B of `ex01`

We now need to allocate **two arrays** in the main program, and call `update()` alternating between input and output

One way to do this:

1. Allocate a two-component array of arrays:

```
double *v[2] = {
  ualloc(sizeof(double)*L*L),
  ualloc(sizeof(double)*L*L)
};
```

Optimizing the heat equation code -- no MPI

Task B of `ex01`

We now need to allocate **two arrays** in the main program, and call `update()` alternating between input and output

One way to do this:

1. Allocate a two-component array of arrays:

```
double *v[2] = {
  ualloc(sizeof(double)*L*L),
  ualloc(sizeof(double)*L*L)
};
```

2. Initialize the first one:

```
initial_condition(v[0]);
```

Optimizing the heat equation code -- no MPI

Task B of `ex01`

We now need to allocate **two arrays** in the main program, and call `update()` alternating between input and output

One way to do this:

1. Allocate a two-component array of arrays:

```
double *v[2] = {
  ualloc(sizeof(double)*L*L),
  ualloc(sizeof(double)*L*L)
};
```

2. Initialize the first one:

```
initial_condition(v[0]);
```

3. Alternate between them in the loop:

```
for(int i=0; i<T; i++) {
   update(v[(i + 1) % 2], v[i % 2], alpha);
   boundary_condition(v[(i + 1) % 2]);
}
```

Optimizing the heat equation code -- no MPI

Task B of `ex01`

We now need to allocate **two arrays** in the main program, and call `update()` alternating between input and output

One way to do this:

1. Allocate a two-component array of arrays:

```
double *v[2] = {
  ualloc(sizeof(double)*L*L),
  ualloc(sizeof(double)*L*L)
};
```

2. Initialize the first one:

```
initial_condition(v[0]);
```

3. Alternate between them in the loop:

```
for(int i=0; i<T; i++) {
   update(v[(i + 1) % 2], v[i % 2], alpha);
   boundary_condition(v[(i + 1) % 2]);
}
```

4. The last iteration is `i=T-1` and therefore the last update stores the result in `v[(T-1+1) % 2] = v[T % 2]`, so:

```
write_bin("v.bin", v[T % 2]);
```

Optimizing the heat equation code -- no MPI

Task B of `ex01`

Remember to free both at the end:

```
free(v[0]);
free(v[1]);
```

Optimizing the heat equation code -- no MPI

Task B of `ex01`

Remember to free both at the end:

```
free(v[0]);
free(v[1]);
```

Compile and run using 16 OpenMP threads like the last time:

Optimizing the heat equation code -- no MPI

Task B of `ex01`

Remember to free both at the end:

```
free(v[0]);
free(v[1]);
```

Compile and run using 16 OpenMP threads like the last time:

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64 -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.739561 sec | p = 3.190e+00 Gflop/s | b = 8.557e+00 GB/s
```

Optimizing the heat equation code -- no MPI

Task B of `ex01`

Remember to free both at the end:

```
  free(v[0]);
  free(v[1]);
```

Compile and run using 16 OpenMP threads like the last time:

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64 -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.739561 sec | p = 3.190e+00 Gflop/s | b = 8.557e+00 GB/s
```

We're closer to ~10% of peak. Better, but still some improvement to be gained

Optimizing the heat equation code -- no MPI

Task B of `ex01`

Remember to free both at the end:

```
  free(v[0]);
  free(v[1]);
```

Compile and run using 16 OpenMP threads like the last time:

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64 -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.739561 sec | p = 3.190e+00 Gflop/s | b = 8.557e+00 GB/s
```

We're closer to ~10% of peak. Better, but still some improvement to be gained

Now may be a good time to check our result:

```
[ikoutsou@front02 ex01]$ cmp v.bin v-reference.bin
[ikoutsou@front02 ex01]$
```

Optimizing the heat equation code -- no MPI

Task B of `ex01`

Remember to free both at the end:

```
  free(v[0]);
  free(v[1]);
```

Compile and run using 16 OpenMP threads like the last time:

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64 -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.739561 sec | p = 3.190e+00 Gflop/s | b = 8.557e+00 GB/s
```

We're closer to ~10% of peak. Better, but still some improvement to be gained

Now may be a good time to check our result:

```
[ikoutsou@front02 ex01]$ cmp v.bin v-reference.bin
[ikoutsou@front02 ex01]$
```

No output means the two files are *bit-identical*

Optimizing the heat equation code -- no MPI

Task C of `ex01`

Now let's reconsider the use of `IDX()`

Optimizing the heat equation code -- no MPI

Task C of `ex01`

Now let's reconsider the use of `IDX()`

- While convenient in improving code readability, there are two issues that affect performance when using this macro:

Optimizing the heat equation code -- no MPI

Task C of `ex01`

Now let's reconsider the use of `IDX()`

- While convenient in improving code readability, there are two issues that affect performance when using this macro:
  - The modulo operations may break the CPUs prefetcher

Optimizing the heat equation code -- no MPI

Task C of `ex01`

Now let's reconsider the use of `IDX()`

- While convenient in improving code readability, there are two issues that affect performance when using this macro:
  - The modulo operations may break the CPUs prefetcher
  - The number of modulo operations is quite large; larger than the number of floating point operations per iteration

Optimizing the heat equation code -- no MPI

Task C of `ex01`

Now let's reconsider the use of `IDX()`

- While convenient in improving code readability, there are two issues that affect performance when using this macro:
  - The modulo operations may break the CPUs prefetcher
  - The number of modulo operations is quite large; larger than the number of floating point operations per iteration

1. We will break the `x`-loop; a main loop for the inner elements and then separate `x=0` and `x=L-1` cases
2. We will pre-compute repeated factors, like `y*L`, `(y+1)*L`, and `(y-1)*L`

Optimizing the heat equation code -- no MPI

Task C of `ex01`

Now let's reconsider the use of `IDX()`

- While convenient in improving code readability, there are two issues that affect performance when using this macro:
  - The modulo operations may break the CPUs prefetcher
  - The number of modulo operations is quite large; larger than the number of floating point operations per iteration

1. We will break the `x`-loop; a main loop for the inner elements and then separate `x=0` and `x=L-1` cases
2. We will pre-compute repeated factors, like `y*L`, `(y+1)*L`, and `(y-1)*L`

Within `boundary_condition()`

```
#pragma omp parallel for
  for(int y=0; y<L; y++) {
    v[y*L] = 1.0;
    v[y*L + L-1] = 1.0;
  }

#pragma omp parallel for
  for(int x=0; x<L; x++) {
    v[L*(L/2) + x] = 1.0;
  }
```

Optimizing the heat equation code -- no MPI

Task C of `ex01`

- Loop for inner elements of `x` in `update()`:

```
#pragma omp parallel for
  for(int y=0; y<L; y++) {
    int y0L = y*L;
    int ypL = ((y+1)%L)*L;
    int ymL = ((L+y-1)%L)*L;
    for(int x=1; x<L-1; x++) {
      int r00 = y0L + x;
      int r0p = y0L + x+1;
      int r0m = y0L + x-1;
      int rp0 = ypL + x;
      int rm0 = ymL + x;

      v1[r00] = (1-4*alpha)*v0[r00] + \
                alpha*(v0[rp0] + v0[rm0] + v0[r0p] + v0[r0m]);
    }

    /* Loop over `y' continued on the right ⟶ */
```

- Parts for `x=0` and `x=L-1` done separately

```
    /* x = 0 */
    {
      int r00 = y0L;
      int r0p = y0L+1;
      int r0m = y0L+(L-1);
      int rp0 = ypL;
      int rm0 = ymL;

      v1[r00] = (1-4*alpha)*v0[r00] + \
                alpha*(v0[rp0] + v0[rm0] + v0[r0p] + v0[r0m]);
    }
    /* x = L-1 */
    {
      int r00 = y0L+(L-1);
      int r0p = y0L;
      int r0m = y0L+(L-2);
      int rp0 = ypL+(L-1);
      int rm0 = ymL+(L-1);

      v1[r00] = (1-4*alpha)*v0[r00] + \
                alpha*(v0[rp0] + v0[rm0] + v0[r0p] + v0[r0m]);
    }
  } /* ⟵ this bracket closes the `y'-loop */
```

Optimizing the heat equation code -- no MPI

Task C of `ex01`

Compile and run using 16 OpenMP threads like before

Optimizing the heat equation code -- no MPI

Task C of `ex01`

Compile and run using 16 OpenMP threads like before

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64  -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.090990 sec | p = 2.593e+01 Gflop/s | b = 6.955e+01 GB/s
```

Optimizing the heat equation code -- no MPI

Task C of `ex01`

Compile and run using 16 OpenMP threads like before

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64  -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.090990 sec | p = 2.593e+01 Gflop/s | b = 6.955e+01 GB/s
```

- Check the result with `cmp` and the reference output file!

Optimizing the heat equation code -- no MPI

Task C of `ex01`

Compile and run using 16 OpenMP threads like before

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64  -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.090990 sec | p = 2.593e+01 Gflop/s | b = 6.955e+01 GB/s
```

- Check the result with `cmp` and the reference output file!
- Compare ~70 GB/s to theoretical peak BW of ~80 GB/s

Optimizing the heat equation code -- no MPI

Task D of `ex01`

The final optimization will be to avoid forking and joining the OpenMP threads in each iteration.

Optimizing the heat equation code -- no MPI

Task D of `ex01`

The final optimization will be to avoid forking and joining the OpenMP threads in each iteration.

- Use `#pragma omp parallel` before the `T`-loop in the main program

Optimizing the heat equation code -- no MPI

Task D of `ex01`

The final optimization will be to avoid forking and joining the OpenMP threads in each iteration.

- Use `#pragma omp parallel` before the `T`-loop in the main program
- Use `#praga omp for` above the for-loops

Optimizing the heat equation code -- no MPI

Task D of `ex01`

The final optimization will be to avoid forking and joining the OpenMP threads in each iteration.

- Use `#pragma omp parallel` before the `T`-loop in the main program
- Use `#praga omp for` above the for-loops

- In `main()`:

```
double t0 = stop_watch(0);
#pragma omp parallel
for(int i=0; i<T; i++) {
```

- In `update()`:

```
#pragma omp for
for(int y=0; y<L; y++) {
int y0L = L*y;
 ...
```

Optimizing the heat equation code -- no MPI

Task D of `ex01`

The final optimization will be to avoid forking and joining the OpenMP threads in each iteration.

- Use `#pragma omp parallel` before the `T`-loop in the main program
- Use `#praga omp for` above the for-loops

- In `main()`:

```
double t0 = stop_watch(0);
#pragma omp parallel
for(int i=0; i<T; i++) {
```

- In `update()`:

```
#pragma omp for
for(int y=0; y<L; y++) {
int y0L = L*y;
...
```

- In `boundary_condition()`:

```
#pragma omp for
  for(int y=0; y<L; y++) {
  ...
```

```
#pragma omp for
  for(int x=0; x<L; x++) {
  ...
```

Optimizing the heat equation code -- no MPI

Task D of `ex01`

Compile and run with 16 OpenMP threads like before

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64  -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.046399 sec | p = 5.085e+01 Gflop/s | b = 1.364e+02 GB/s
```

Optimizing the heat equation code -- no MPI

Task D of `ex01`

Compile and run with 16 OpenMP threads like before

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64  -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.046399 sec | p = 5.085e+01 Gflop/s | b = 1.364e+02 GB/s
```

This is larger than our estimated peak, which means we are partially in cache

Optimizing the heat equation code -- no MPI

Task D of `ex01`

Compile and run with 16 OpenMP threads like before

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64  -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.046399 sec | p = 5.085e+01 Gflop/s | b = 1.364e+02 GB/s
```

This is larger than our estimated peak, which means we are partially in cache

- We will increase `L` to see when the bandwidth plateaus, indicating we are out of cache

Optimizing the heat equation code -- no MPI

Task D of `ex01`

Compile and run with 16 OpenMP threads like before

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64  -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.046399 sec | p = 5.085e+01 Gflop/s | b = 1.364e+02 GB/s
```

This is larger than our estimated peak, which means we are partially in cache

- We will increase `L` to see when the bandwidth plateaus, indicating we are out of cache
- For this benchmark, it helps to reduce `T` as we increase `L` so that the runtime doesn't explode

Optimizing the heat equation code -- no MPI

Task D of `ex01`

Compile and run with 16 OpenMP threads like before

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64  -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.046399 sec | p = 5.085e+01 Gflop/s | b = 1.364e+02 GB/s
```

This is larger than our estimated peak, which means we are partially in cache

- We will increase `L` to see when the bandwidth plateaus, indicating we are out of cache
- For this benchmark, it helps to reduce `T` as we increase `L` so that the runtime doesn't explode

```
[ikoutsou@front02 ex01]$ for((i=1; i<=128; i*=2))
> do srun -n 1 -p p100 --cpus-per-task=64 ./heat $((i*128)) $((64*16384/i/i)) 0.2
> done
 nthr =  16 | L =   128 | T = 1048576 | t0 = 7.855353 sec | p = 1.312e+01 Gflop/s | b = 3.540e+01 GB/s
 nthr =  16 | L =   256 | T =  262144 | t0 = 2.018945 sec | p = 5.106e+01 Gflop/s | b = 1.369e+02 GB/s
 nthr =  16 | L =   512 | T =   65536 | t0 = 1.296302 sec | p = 7.952e+01 Gflop/s | b = 2.127e+02 GB/s
 nthr =  16 | L =  1024 | T =   16384 | t0 = 1.107996 sec | p = 9.303e+01 Gflop/s | b = 2.484e+02 GB/s
 nthr =  16 | L =  2048 | T =    4096 | t0 = 5.515507 sec | p = 1.869e+01 Gflop/s | b = 4.987e+01 GB/s
 nthr =  16 | L =  4096 | T =    1024 | t0 = 6.591526 sec | p = 1.564e+01 Gflop/s | b = 4.172e+01 GB/s
 nthr =  16 | L =  8192 | T =     256 | t0 = 6.398171 sec | p = 1.611e+01 Gflop/s | b = 4.297e+01 GB/s
 nthr =  16 | L = 16384 | T =      64 | t0 = 6.227907 sec | p = 1.655e+01 Gflop/s | b = 4.414e+01 GB/s
```

Optimizing the heat equation code -- no MPI

Task D of `ex01`

Compile and run with 16 OpenMP threads like before

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64  -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.046399 sec | p = 5.085e+01 Gflop/s | b = 1.364e+02 GB/s
```

This is larger than our estimated peak, which means we are partially in cache

- We will increase `L` to see when the bandwidth plateaus, indicating we are out of cache
- For this benchmark, it helps to reduce `T` as we increase `L` so that the runtime doesn't explode

```
[ikoutsou@front02 ex01]$ for((i=1; i<=128; i*=2))
> do srun -n 1 -p p100 --cpus-per-task=64 ./heat $((i*128)) $((64*16384/i/i)) 0.2
> done
 nthr =  16 | L =   128 | T = 1048576 | t0 = 7.855353 sec | p = 1.312e+01 Gflop/s | b = 3.540e+01 GB/s
 nthr =  16 | L =   256 | T =  262144 | t0 = 2.018945 sec | p = 5.106e+01 Gflop/s | b = 1.369e+02 GB/s
 nthr =  16 | L =   512 | T =   65536 | t0 = 1.296302 sec | p = 7.952e+01 Gflop/s | b = 2.127e+02 GB/s
 nthr =  16 | L =  1024 | T =   16384 | t0 = 1.107996 sec | p = 9.303e+01 Gflop/s | b = 2.484e+02 GB/s
 nthr =  16 | L =  2048 | T =    4096 | t0 = 5.515507 sec | p = 1.869e+01 Gflop/s | b = 4.987e+01 GB/s
 nthr =  16 | L =  4096 | T =    1024 | t0 = 6.591526 sec | p = 1.564e+01 Gflop/s | b = 4.172e+01 GB/s
 nthr =  16 | L =  8192 | T =     256 | t0 = 6.398171 sec | p = 1.611e+01 Gflop/s | b = 4.297e+01 GB/s
 nthr =  16 | L = 16384 | T =      64 | t0 = 6.227907 sec | p = 1.655e+01 Gflop/s | b = 4.414e+01 GB/s
```

Plateaus at ~45 GB/s or ~60% of peak bandwidth

Optimizing the heat equation code -- no MPI

Task D of `ex01`

Compile and run with 16 OpenMP threads like before

```
[ikoutsou@front02 ex01]$ cc -fopenmp -O3 -o heat heat.c
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=64  -p p100 ./heat 256 6000 0.2
 nthr =  16 | L = 256 | T = 6000 | t0 = 0.046399 sec | p = 5.085e+01 Gflop/s | b = 1.364e+02 GB/s
```

This is larger than our estimated peak, which means we are partially in cache

- We will increase `L` to see when the bandwidth plateaus, indicating we are out of cache
- For this benchmark, it helps to reduce `T` as we increase `L` so that the runtime doesn't explode

```
[ikoutsou@front02 ex01]$ for((i=1; i<=128; i*=2))
> do srun -n 1 -p p100 --cpus-per-task=64 ./heat $((i*128)) $((64*16384/i/i)) 0.2
> done
 nthr =  16 | L =   128 | T = 1048576 | t0 = 7.855353 sec | p = 1.312e+01 Gflop/s | b = 3.540e+01 GB/s
 nthr =  16 | L =   256 | T =  262144 | t0 = 2.018945 sec | p = 5.106e+01 Gflop/s | b = 1.369e+02 GB/s
 nthr =  16 | L =   512 | T =   65536 | t0 = 1.296302 sec | p = 7.952e+01 Gflop/s | b = 2.127e+02 GB/s
 nthr =  16 | L =  1024 | T =   16384 | t0 = 1.107996 sec | p = 9.303e+01 Gflop/s | b = 2.484e+02 GB/s
 nthr =  16 | L =  2048 | T =    4096 | t0 = 5.515507 sec | p = 1.869e+01 Gflop/s | b = 4.987e+01 GB/s
 nthr =  16 | L =  4096 | T =    1024 | t0 = 6.591526 sec | p = 1.564e+01 Gflop/s | b = 4.172e+01 GB/s
 nthr =  16 | L =  8192 | T =     256 | t0 = 6.398171 sec | p = 1.611e+01 Gflop/s | b = 4.297e+01 GB/s
 nthr =  16 | L = 16384 | T =      64 | t0 = 6.227907 sec | p = 1.655e+01 Gflop/s | b = 4.414e+01 GB/s
```

Plateaus at ~45 GB/s or ~60% of peak bandwidth

Cache size is 20 MB $\Rightarrow$ can fit two arrays of 1024×1024, but not 2048×2048

Optimizing the heat equation code -- with MPI

- As a general approach, we will add MPI parallelization along the y-axis as we did in lesson 9
- We would like to keep the non-MPI version. We will use `#ifdef MPI ... #endif` around the regions with MPI code
- This allows us to switch on MPI *at compile time*, e.g.:

```
...
ly = L;

#ifdef MPI
int nproc;
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
ly = L/nproc;
#endif
...
```

Optimizing the heat equation code -- with MPI

- As a general approach, we will add MPI parallelization along the y-axis as we did in lesson 9
- We would like to keep the non-MPI version. We will use `#ifdef MPI ... #endif` around the regions with MPI code
- This allows us to switch on MPI *at compile time*, e.g.:

```
...
ly = L;

#ifdef MPI
int nproc;
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
ly = L/nproc;
#endif
...
```

- Compiling with `cc -o heat heat.c` will omit the code within `ifdef`
  $\Longrightarrow$ `MPI_Comm_size()` **will not** be called and `ly` will be equal to `L`

Optimizing the heat equation code -- with MPI

- As a general approach, we will add MPI parallelization along the y-axis as we did in [lesson 9]
- We would like to keep the non-MPI version. We will use `#ifdef MPI ... #endif` around the regions with MPI code
- This allows us to switch on MPI *at compile time*, e.g.:

```
...
ly = L;

#ifdef MPI
int nproc;
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
ly = L/nproc;
#endif
...
```

- Compiling with `cc -o heat heat.c` will omit the code within `ifdef`
  $\Longrightarrow$ `MPI_Comm_size()` **will not** be called and `ly` will be equal to `L`
- Compiling with `mpicc -DMPI -o heat heat.c` will include the code within `ifdef`
  $\Longrightarrow$ `MPI_Comm_size()` **will** be called and `ly` will be equal to `L/nproc`

Optimizing the heat equation code -- with MPI

- One thing to take care of is calling MPI functions within `omp parallel` regions

Optimizing the heat equation code -- with MPI

- One thing to take care of is calling MPI functions within `omp parallel` regions
- Ensure only one OpenMP thread is calling MPI functions, e.g.:

```
#pragma omp single
    MPI_Sendrecv( ... );
```

Optimizing the heat equation code -- with MPI

- One thing to take care of is calling MPI functions within `omp parallel` regions
- Ensure only one OpenMP thread is calling MPI functions, e.g.:

```
#pragma omp single
    MPI_Sendrecv( ... );
```

- Another "trick" is to define `nproc` and `rank` and set them to `nproc = 1` and `rank = 0` outside the `#ifdef MPI` regions, so that the non-MPI code works when you have `if(rank == 0)` and similar

Running Hybrid MPI/OpenMP Code

- Running this hybrid version needs some care. In general you need to ensure that:
    - An OpenMP thread is bound to a core
    - When using multiple OpenMP threads per MPI process, the `mpirun` command allocates multiple cores to each MPI process
- Achieving this depends on the MPI implementation (e.g. OpenMPI, MVAPICH, IntelMPI, etc.)
- Can be found looking at the manual pages or help output of the `mpirun` or `mpiexec` command

Running Hybrid MPI/OpenMP Code

- Running this hybrid version needs some care. In general you need to ensure that:
  - An OpenMP thread is bound to a core
  - When using multiple OpenMP threads per MPI process, the `mpirun` command allocates multiple cores to each MPI process
- Achieving this depends on the MPI implementation (e.g. OpenMPI, MVAPICH, IntelMPI, etc.)
- Can be found looking at the manual pages or help output of the `mpirun` or `mpiexec` command
- Example for our specific use-case:

```
OMP_NUM_THREADS=16 mpirun -n 4 -npernode 2 --map-by slot:PE=16 ./heat 2048 100 0.2
```

  - Run using 4 MPI processes
  - `mpirun` is instructed to allocate 16 "slots" (here meaning cores) per MPI process
  - 2 MPI processes will run on each physical node
  - Each MPI process will spawn 16 OpenMP threads within `omp parallel` regions

Running Hybrid MPI/OpenMP Code

- Running this hybrid version needs some care. In general you need to ensure that:
  - An OpenMP thread is bound to a core
  - When using multiple OpenMP threads per MPI process, the `mpirun` command allocates multiple cores to each MPI process
- Achieving this depends on the MPI implementation (e.g. OpenMPI, MVAPICH, IntelMPI, etc.)
- Can be found looking at the manual pages or help output of the `mpirun` or `mpiexec` command
- Example for our specific use-case:

```
OMP_NUM_THREADS=16 mpirun -n 4 -npernode 2 --map-by slot:PE=16 ./heat 2048 100 0.2
```

  - Run using 4 MPI processes
  - `mpirun` is instructed to allocate 16 "slots" (here meaning cores) per MPI process
  - 2 MPI processes will run on each physical node
  - Each MPI process will spawn 16 OpenMP threads within `omp parallel` regions
- You can also use `--report-bindings` to see how each MPI process is bound

```
[cyc02:49468] MCW rank 1 bound to ... : [../../../../../../../../../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB]
[cyc02:49468] MCW rank 0 bound to ... : [BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../../../../../../../../../..]
[cyc03:49399] MCW rank 3 bound to ... : [../../../../../../../../../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB]
[cyc03:49399] MCW rank 2 bound to ... : [BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../../../../../../../../../..]
```

This report shows that for each MPI process, the 16 OpenMP threads are restricted to a different socket

Running Hybrid MPI/OpenMP Code

Scaling plot of two cases, `L=256` (left) and `L=2048`

Can this behavior be interpreted?

- `L=256` fits in a single-socket's L3 cache
- `L=2048` does not, but `2048 x 1024` does
- Average and error taken over 10 runs
- Setup:
  - $n_{OMP} \times n_{MPI} \leqslant 16$,
    $\Rightarrow$ single-socket is used
  - $n_{OMP} \times n_{MPI} = 32$,
    $\Rightarrow$ 1 node, 2 sockets
  - When $n_{OMP} \times n_{MPI} = 64$,
    $\Rightarrow$ 2 nodes, 2 sockets/node