Using both Open-MP and MPI to parallelize BioHeat Transfer code

Akshay Kumar Varanasi
(av32826)

May 12, 2018

Abstract

The Bio-heat transfer problem is a complex problem which involves multiple phenomenons such as heat generation due to metabolism and heat transfer due to perfusion and ambience. Experimentally studying this problem by taking readings throughout the body is very challenging and thus Computational methods are used to model them. This project deals with increase in body(head) temperature due to cell phones. Since the usage of cell phones has increased tremendously, it is important to know the health risks due to the radiation they emit. And trying to measure temperature changes due to cell-phone radiation is even more difficult as the temperature changes are really small. This problem is modeled as Bioheat transfer problem and solved numerically using Explicit Finite Difference scheme. The existing solver solves this equation using MPI for parallelization. This project would involve studying the bottlenecks of the code and optimize it by using hybrid methods i.e including both Open-MP and MPI for parallelizing. Project would also discuss about computational efficiencies and scaling.

Introduction and Objective

Bioheat transfer is the study of thermal energy in living systems. Since most of the processes in our body are biochemical processes which are temperature dependent, heat transfer plays a major role in living systems. Studying Bio-heat transfer methods is important for diagnostic and therapeutic applications involving either mass or heat transfer. These studies are done mostly using computational methods as conducting experiments in these field is really difficult both in terms of appropriate instruments and government regulations.

Since the word "Bioheat transfer" is related to body and one of the most important part of the body is head. This project is related to that crucial part of the body and thus is in turn very crucial for reader to pay careful attention to this project.(:-)) Usage of Cell phones has become a daily business in almost everyones life and thus this comes with rising question about health hazards from these electromagnetic devices. My research is about studying the effect of cell phone radiation on temperature of the head(modeled as 4-layered sphere of tissues) using numerical methods. This involves solving the Pennes Equation, which has been used to model this phenomenon for more than half a century, with appropriate boundary conditions and thermal properties.

Theory

As explained before, the basic idea of our research is to solve the famous Pennes Equation with appropriate conditions. The Pennes equation is given below

$$\rho c \frac{\partial T}{\partial t} = \nabla \cdot (k \nabla) + \dot{q}_{gen} + \dot{q}_m + \dot{q}_p \tag{1}$$

Where ρ is the density kg/m^3 and c is the specific heat($W/kg/^\circ C($ of that material, k is the thermal conductivity$W/(m^\circ C)$, $\dot{q}_m$ is the heat transfer due to metabolism (W/m^3) and $\dot{q}_p$ is the heat transfer due to perfusion (W/m^3). The Pennes expression assumes that the blood flowing through capillaries in a small volume of tissue enters with a temperature, $T_a(^\circ C)$ and exits at the local tissue temperature, $T(^\circ C)$. Then the heat lost by the blood, which is gained by the tissue, is written per unit volume as

$$\dot{q}_p = \omega \rho_b c_b (T_a - T) \tag{2}$$

where ω is the blood perfusion rate (s^{-1}) in the tissues, ρ_b is the density of blood, 1050 kg/m^3 , and c_b is the specific heat of blood $3617 W/kg/^\circ C$. The Pennes model also assumes that the metabolism heat sink is a constant value for each type of tissue. The $\dot{q}_{gen}$ is the volumetric heat generation term(W/m^3). In our case, it is heat absorbed due to electromagnetic radiation due to cell phone which is nothing but SAR value.

To solve this equation numerically, we need to convert this PDE in to some form which can be solved using numerical techniques. For this we need to discretize the PDE accordingly. When we use central difference in space and forward difference in time (explicit finite difference method) to discretize this equation we get following equation in 1D.

$$T^{n+1}_{i,j,k} = T^n_{i,j,k} + \frac{\Delta t}{\rho_{i,j,k} c_{i,j,k}} \left[\frac{(k_{i+1,j,k} + k_{i,j,k}) T^n_{i+1,j,k}}{2\Delta x^2} - \frac{(k_{i+1,j,k} + 2k_{i,j,k} + k_{i-1,j,k}) T^n_{i,j,k}}{2\Delta x^2} + \frac{(k_{i,j,k} + k_{i-1,j,k}) T^n_{i-1,j,k}}{2\Delta x^2} - \omega_{i,j,k} \rho_b c_b T^n_{i,j,k} + (\omega_{i,j,k} \rho_b c_b T_a + \dot{q}_{geni,j,k} + \dot{q}_{mi,j,k}) \right] \tag{3}$$

and the stability requirement is given by

$$\Delta t \leq \frac{2 \rho_{min} c_{min} \Delta x^2}{12 k_{max} + \omega_{max} \rho_b c_b \Delta x^2} \tag{4}$$

where ρ_{min} is the minimum density of all materials in the domain, c_{pmin} is the minimum specific heat capacity of all materials in the domain, k_{max} is the maximum thermal conductivity of all materials in the domain, and ω_{max} is the maximum perfusion of all materials in the domain. In order to calculate the temperature distribution using this formulation, first an initial condition sets the temperature at t = 0. In our case it is body temperature which is around $37^\circ C$. Then time is marched through by increasing t by Δt for each time step. And this is calculated from stability requirement for given Δx. At each time step, each point of in the domain is iterated through and the temperature is

calculated using values from the previous time step. This process repeats until a desired time is reached. Time could also be marched until a steady state is reached. Whether a steady state has been reached can be determined by calculating the error between two time steps using a specific error norm and comparing the error to a preset threshold.

The boundary condition in our case is Robin boundary condition(which is nothing but heat lost to ambience) which is given as follows

$$k\frac{\partial T}{\partial t} = h(T - T_{amb}) \tag{5}$$

Where h is the heat convection co-efficient$W/m^2/^\circ C$ and the T_{amb} is the Temperature of the ambience.

Bio-heat Transfer solver

Using the theory mentioned before we can get Temperature value for next time step using Temperature at previous time step. The code used to calculate temperature distribution was written in parallel using Message Passing Interface (MPI) in order to calculate solutions more quickly. In order to parallelize an explicit finite difference method, the domain is divided into sections based on the number of processors and each processor is responsible for calculating the temperature distribution in its section of the domain. However, completely dividing the domain will not produce an accurate solution, because the temperature at the nodes on the edge of one processor's domain are dependent on the temperature at the nodes on the neighboring processor. Therefore, ghosting regions are created. Ghosting is when one processor communicates the function or property values of some of its nodes to a neighboring processor. The numerical scheme determines how much of the domain must be ghosted. For this explicit case, only the cells immediately next to interface between processors must be ghosted.

To further improve the computational efficiency or speed, Open-MP was used with MPI as Master. This project would discuss mostly about this part as this is my contribution to the existing MPI code. The advantage of using this is to reduce MPI tasks which saves memory/ reduce calls and send bigger message rather than sending smaller messages and also to use Hyper threading technology.

When you use this solver on 3D 4-layered sphere of radius 108 mm as the shape is similar to head and the properties of these materials are taken of that of skin,skull,brain and Cerebrospinal fluid we get following result. You can see that Temperature difference (w.r.t to body temp) is more towards one side. This is due to the Electromagnetic source(taken as electric dipole) which is mobile in our case at 50 mm distance from head.

Compiling and running the code

Compiling Open-MP/MPI hybrid code is slightly different from MPI or Open-MP code. In this we need to compile using the following commands

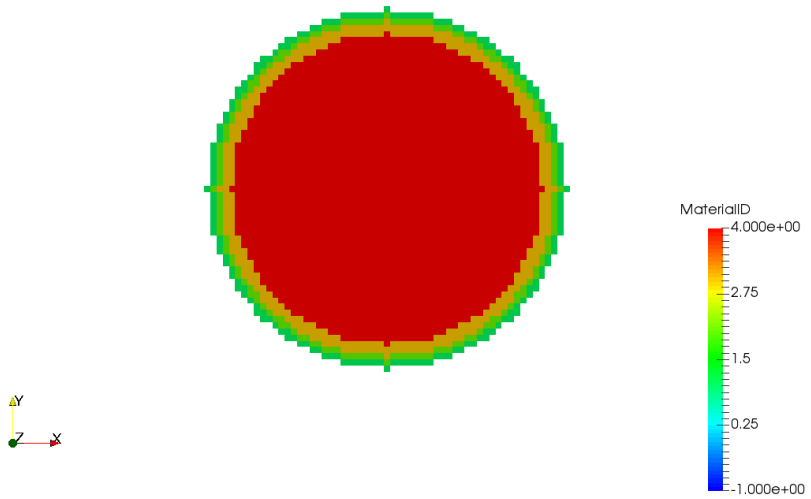

Figure 1: Layered Sphere with 4 different materials.

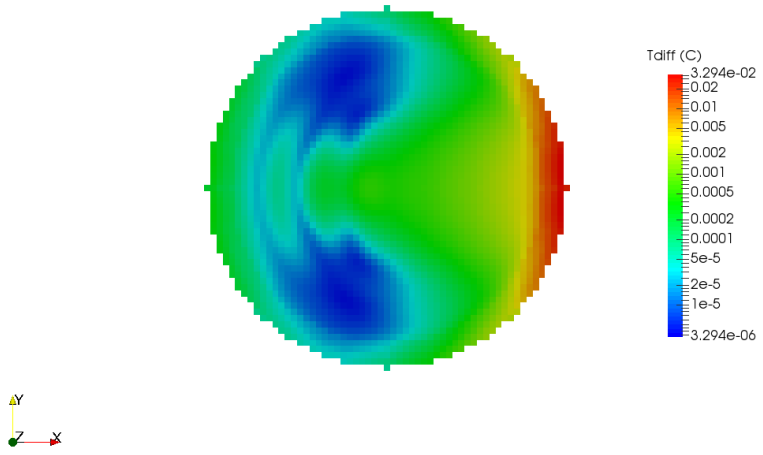

Figure 2: Distribution of Temperature Difference

```
mpiifort -qopenmp  -O3  -c bioheat.f90
```

As you can see we are using Intel compilers for compiling and since this code has MPI we use **mpiifort** instead of **ifort**. We need to use this **-qopenmp** option as well because we are also using Open-MP in our code. The **-O3** option is used for level 3 optimization of the code by the compiler.

To set up the environment and run the code we need to give following commands

```
#SBATCH -N 1
#SBATCH -n 1

export OMP_NUM_THREADS=16
ibrun ./a.out
```

Strategy used for parallelizing

Since the code has already been parallelized by MPI, the goal is to optimize it by making each processor to do its task faster using Open-MP. Basically I started to look for loops which can be parallelized because conditional statements or print statements are not possible to parallelize. For a loop to be able to parallelized using Open-MP, it needs to be independent of other threads and iterations. If the loops are dependent we need to use some techniques such loop fission for making it independent. After finding such loops we need to think about variables used in the loop like whether they should be shared or private. For example consider this loop which is used for calculating norm. If you see this loop for example, it needs loop indices to be private variable so that each thread has its value and thus prevent race condition. The variable "bndCell" is given firstprivate because its value was already initialized before. The variables `"temp_de"` and `"temp_nu"` are added to themselves after each iteration so reduction is used with plus symbol.

```
!$omp parallel do private(kk,jj,ii) firstprivate(bndCell) &
reduction(+:temp_nu,temp_de)
do kk=cellRange(1,3),cellRange(2,3)
  do jj=cellRange(1,2),cellRange(2,2)
     do ii=cellRange(1,1),cellRange(2,1)
       if (cellType(ii,jj,kk)>bndCell) then
          temp_nu=temp_nu+abs(vecIteration(ii,jj,kk,curr)-
          vecIteration(ii,jj,kk,last))**2
          temp_de=temp_de+abs(vecIteration(ii,jj,kk,last))**2
       end if
     end do
  end do
end do
```

Apart from that I also tried to see if single parallel directive can be used instead of multiple parallel directives as it reduces the cost of forking. But I hardly found such cases because directives were far apart and using single directive with critical in between would slow down. This is because the critical part is large enough which is essentially same as large serial region in between.

Few other things which I could have done but did not do it because of time constraints is to see which type of scheduling is better. The reason is as follows, since there are so many loops which have been parallelized and each loop can be scheduled differently with different amount as chunk size. Thus there can so many possibilities which is difficult to study. Since I used only one node for MPI, the possibility of optimizing using numactl is ruled out.

Problems experienced

While parallelizing I have faced quite a few problems. Since my code requires reading input from files and writing output to files, parallelizing this I/O would speed up the overall runtime of the code. So I tried parallelizing reading and writing files using Open-MP but this led to race condition and thus gave an error. This is already optimized with respect to MPI as each task writes onto different files.

The solver constructs the geometry and decomposes it before it actually solves the Pennes equation for temperature distribution. Apart from this it also gives initial conditions to our problem. All these things are done in loops but parallelizing those loops actually slowed down the code. The reason could be that the cost of forking is more than the speedup obtained by them. So in the results only time taken to solve the equation has been reported.

Since this code uses MPI, it has to send and receive buffer across tasks after each iteration. Sending and receiving is done using loops as below

```
! Fill send buffer
index=0
do kk=cellRange(1,3),cellRange(2,3)
   do ii=cellRange(1,1),cellRange(2,1)
     index=index+1
     send_buf_y(index)=vecIteration(ii,cellRange(1,2),kk,curr)
   end do
end do
```

As we can clearly see that these types of loops can't be parallelize just by using Open-MP directly. One way to parallelize is define explicitly which processor needs to do which job. But that would make this loop complicated especially because of the index variable and therefore I left it like that without parallelizing.

Results and discussion

After parallelizing the MPI code using Open-MP and running the code with different settings we get following results. Ideally, doubling the number of pro-

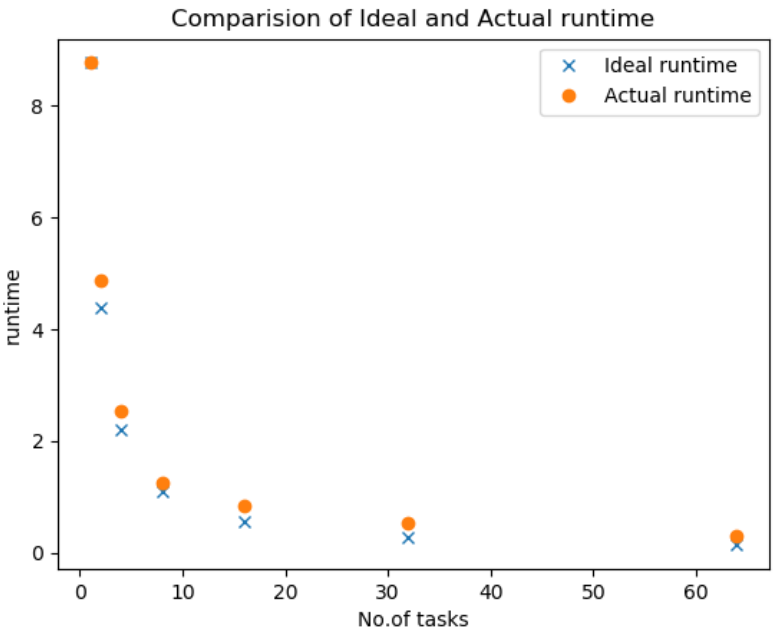

Figure 3: Runtime vs No. of tasks.

cessors that are used to run a program would half the amount of time needed and would use the same amount of total memory. However, programs do not behave ideally. Storing ghosting data requires additional memory that would not be necessary on a single processor. Communication between processors also adds inefficiencies to the parallelization of a code. Codes must be designed to minimize these inefficiencies so that parallelization enables problems to be solved more quickly and larger problems to be solved. In order to measure the parallel inefficienies in a code, scalability studies are performed. The two major scalability studies are strong and weak scaling. Strong scaling is when the problem size is left constant and the number of processors is increased. Weak scaling is when the problem size is increased as the number of processors is increased to ensure that the each processor is responsible for the same amount of domain in each run. Since each processor is limited by memory and most supercomputing clusters have a limited run time, these scalability studies can be used to estimate the largest problem size and number of time steps that is possible with the given constraints. The scaling studies in this report were performed by measuring the wall clock time to ma to a set time used in that process.

Initially we tried running the code using MPI only and we observed that as number of tasks increase time taken is reduced. The speed up in code due to parallelization using MPI is shown in the following figure 3 and figure 4.

But the speedup deviates from the ideal case after 8 MPI tasks. The reason is increase in tasks results in increase in communication cost and thus is inefficient.

Now lets fix no .of MPI tasks as 8 and change no.of threads to the speedup. The speedup seems to be good initially in the figure 5 and 6 but after sometime

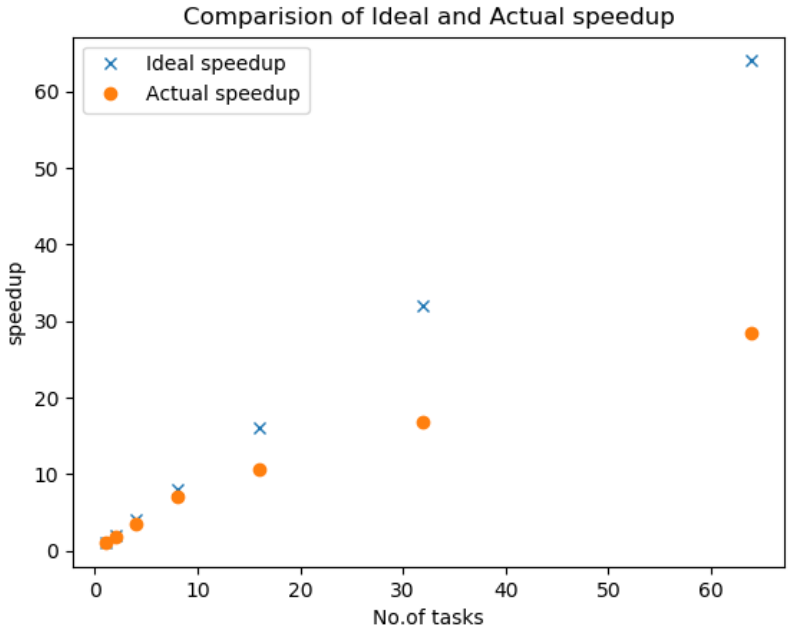

Figure 4: Speedup vs No. of tasks.

the speedup starts to deviate drastically from the ideal speedup. In this case, we can see that best speedup is at 8 threads but after that there is not much difference. Scaling is reasonable but not comparable with ideal case as we are using both MPI and Open-MP. Deviation may be because the costs of both MPI and Open-MP interact apart from getting added.

If you observe using 8 MPI tasks and 8 threads, we were able to achieve same runtime as 64 MPI tasks, which is really helpful as it saves us lot of memory.

Execution	Run-times for x	Run-times for y
8 MPI with 8 threads	0.330118179321289 s	6.17129206657410 s

Table 1: Comparision of run-times of different no.of nodes.

Due to lack of time I could only perform weak scaling for the following configuration where x is the case with 89543 nodes (4mm*4mm*4mm grid size) and y is the case with 686781 nodes (2mm*2mm*2mm grid size). We can see that scaling is not perfect because it should take around 8 times to solve it but it took around 20 times. The reason may be because of the serial part of the code or maybe due to other costs like communication and threading.

I would like to conclude saying that using both MPI and Open-MP has speeded up the computation immensely. This is important as it gives us results faster and thus makes our research easier as we don't have to wait long enough to test new things. Faster research helps people to get things quicker which they should be getting few years later. Thanks to Parallel computing which is making this possible and I am happy that I learned this from two amazing

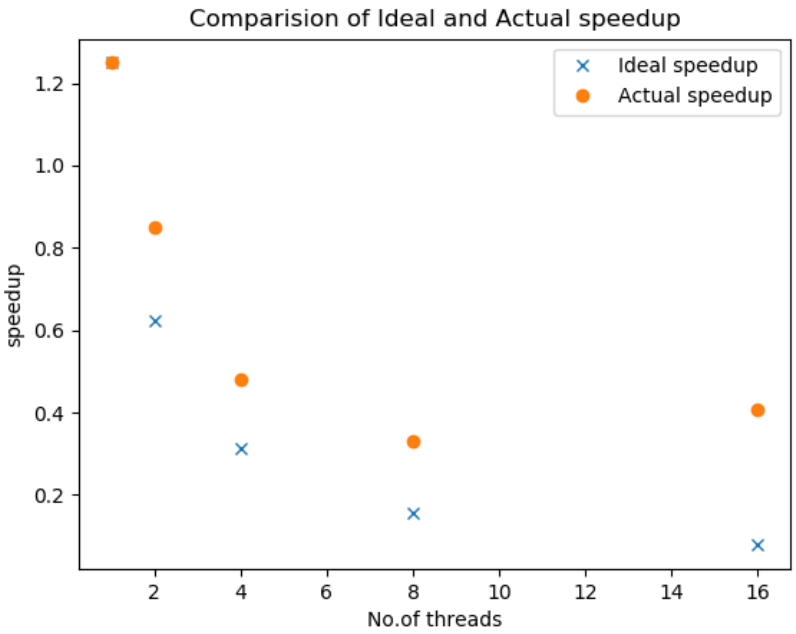

Figure 5: Runtime vs No. of threads.

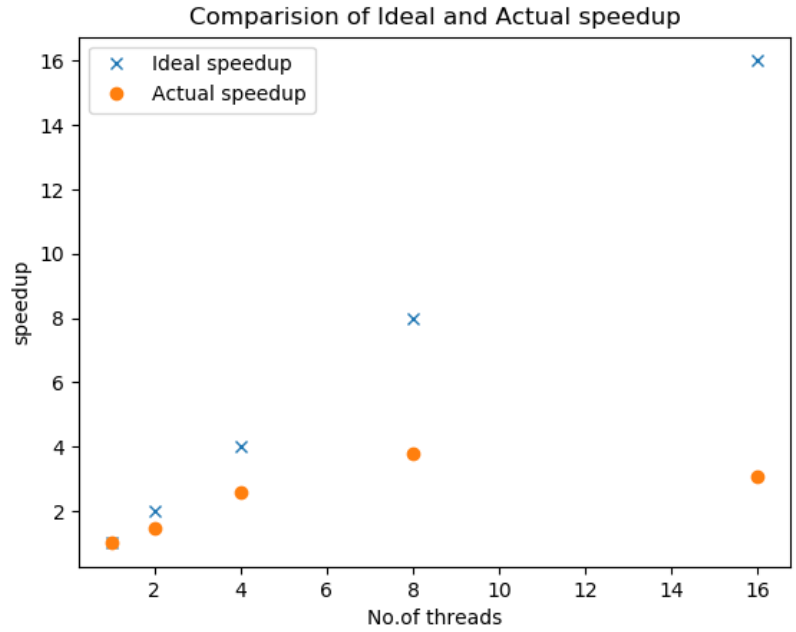

Figure 6: Speedup vs No. of threads.

teachers.:-)