MAE 267 – Project 5
Parallel, Multi-Block, Finite-Volume Methods For Solving 2D Heat Conduction

Logan Halstrom
PhD Graduate Student Researcher
Center for Human/Robot/Vehicle Integration and Performance
Department of Mechanical and Aerospace Engineering
University of California, Davis
Davis, California 95616
Email: ldhalstrom@ucdavis.edu

1 Statement of Problem

This analysis demonstrates the fundamentals of parallel computing through the numerical solution of the steady-state, two-dimensional temperature distribution of a 1m x 1m steel block with properties listed in Table 1.

Table 1: Steel Block Properties

Dimensions	1m x 1m
Thermal Conductivity	$k = 18.8 \frac{W}{m \cdot K}$
Density	$\rho = 8000 \frac{kg}{m^3}$
Specific Heat Ratio	$c_p = 500$

The demonstration of parallel computing techniques was accomplished in stages, starting with a serial (single-processor) solution of a single grid of dimensions 101x101 and 501x501, which serves as a solver basis and performance benchmark for later parallel codes.

The next stage was to divide the grid into NxM subdomains (blocks), on each of which the solution for a given iteration was calculated independently. 5x4 and 10x10 block decompositions of both previous grid dimensions were solved to demonstrate compartmentalization of solver processes, which is a necessary step for distributing processes in parallel computing.

Finally, the code was adapted to solve multi-block decompositions on multiple processors for the 501x501 grid decomposed into 10x10 blocks running on 1 to 8 processors. For this solution, the domain is decomposed and blocks are distributed on to processors. Decompositions are saved to restart files for each processor to be loaded by each processor in the parallel solver.

2 Methods and Equations

The core of this demonstration code is the heat transfer solver developed in the first project, but a number of domain decomposition functions have since been included, as will be detailed in this section.

2.1 Grid Initialization

The numerical solution is initialized with the Dirichlet boundary conditions (Eqn 1) using a single processor.

$$T_{BCs} = \begin{cases} 5.0\left[\sin\left(\pi x_p\right) + 1.0\right] & \text{for } \; j = j_{max} \\ \left|\cos\left(\pi x_p\right)\right| + 1.0 & \text{for } \; j = 0 \\ 3.0 y_p + 2.0 & \text{for } \; i = 0,\, i_{max} \end{cases} \tag{1}$$

$$\begin{aligned} rot &= 30.0 \frac{\pi}{180.0} \\ x_p(i) &= \cos\left[0.5\pi \frac{i_{max} - i}{i_{max} - 1}\right] \\ y_p(j) &= \cos\left[0.5\pi \frac{j_{max} - j}{j_{max} - 1}\right] \\ x &= x_p \cos(rot) + (1.0 - y_p)\sin(rot) \\ y &= y_p \cos(rot) + x_p \sin(rot) \end{aligned} \tag{2}$$

Square grids are generated according to Eqn 2 to create non-uniform spacing in both the x and y directions (with finer spacing at the larger indices). The "prime" system is then rotated by angle *rot* to create the final grid.

2.2 Numerical Solver

The solver developed for this analysis utilizes a finite-volume numerical solution method to solve the transient heat conduction equation (Eqn 3).

$$\rho c_p \frac{\partial T}{\partial t} = k\left[\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right] \tag{3}$$

To solve Eqn 3 numerically, the equation is discretized according to a node-centered, finite-volume scheme, where first-derivatives at the nodes are found using Green's theorem integrating around the secondary control volumes. Trapezoidal, counter-clockwise integration for the first-derivative in the x-direction is achieved with Eqn 4.

$$\begin{aligned}\frac{\partial T}{\partial x} = \frac{1}{2Vol_{i+\frac{1}{2},j+\frac{1}{2}}}\,[&(T_{i+1,j}+T_{i+1,j+1})Ayi_{i+1,j}\\ &-(T_{i,j}+T_{i,j+1})Ayi_{i,j}\\ &-(T_{i,j+1}+T_{i+1,j+1})Ayi_{i,j+1}\\ &-(T_{i,j}+T_{i+1,j})Ayi_{i,j}]\end{aligned} \tag{4}$$

A similar scheme is used to find the first-derivative in the y-direction.

2.3 Subdomain Decomposition

After grid initialization, the grid is divided into N blocks in the x/I direction and M blocks in the y/J direction, creating a total number of blocks $NBLK = N \cdot M$. All blocks are constrained to have the same number of nodes, so the dimensions of every block *IBLKMAX* and *JBLKMAX* are calculated in Eqn 6 as a fraction of the total number of nodes in each direction, including one point overlap at each interblock boundary (Ghost nodes are excluded for the moment). In the I-direction, the total number of nodes including overlap (Eqn 5) is:

$$IMAX_{tot} = IMAX + (N-1) \tag{5}$$

and the total number of nodes per block in the I-direction (Eqn 6) is:

$$IMAXBLK = \frac{IMAX_{tot}}{N} = \frac{IMAX+(N-1)}{N} = 1 + \frac{IMAX-1}{N} \tag{6}$$

Note: For points in J-direction, replace I with J and N with M

Blocks are distributed from 1 to NBLK starting in the lower-left corner of the grid and zipping left to right (the x/I/N direction), then up one (the y/J/M direction) starting again at the left. This is accomplished by two DO loops, the outer loop stepping through J from 1 to M and the inner loop stepping through I from 1 to N. Block locations are stored by assigning global starting indices to each block according to Eqn 7.

$$IMIN_{block} = IMIN_{global} + (IMAXBLK-1)(I-1) \tag{7}$$

where I counts blocks in the direction of N and $IMIN_{global} = 1$. The first block in the N-direction has a global starting index of 0, and IMAXBLK must be reduced by one to account for the single-point overlap at block boundaries.

Information for each block is stored as an element in an array of BLKTYPE derived data types. BLKTYPE stores local mesh, temperature, and solver information as well as the block ID, global indices, iteration bounds to prevent overwriting boundary conditions (discussed in Section 2.5), and neighbor identification information.

2.4 Processor Distribution

For the parallel code, blocks are distributed among *NPROCS* processors (determined in 'miprun call), with the goal of equal load balancing for all processors (Eqn 8). Load balance is the ratio of a processor's workload to the "Perfect Load Balance" (*PLB*), the total load of all blocks divided by *NPROCS*. In this code, a block's load is refered to as its *SIZE*, so a processor's work load is equal to the sum of the *SIZEs* of its blocks.

$$P_{LoadBalance} = \frac{SUM(SIZEs)}{PLB} \tag{8}$$

The workload of each block (*SIZE*) is calculated as a weighted sum (Eqn 11) of its geometric cost *GEOM* due to grid size (Eqn 9) and communication cost *COMM* due to boundary size (10). Geometric cost is essentially the node area of the block iteration bounds:

$$GEOM = (IMAXLOC - IMINLOC)\cdot(JMAXLOC - JMINLOC) \tag{9}$$

Geometric cost will be greater for cells that are not on physical boundaries as they require more ghosts nodes for their inter-block boundaries. Communication cost is calculated as the total length of all faces and corners at interblock boundaries:

$$\begin{aligned}COMM(i) &= \begin{cases}0, & \text{if BC}\\ IMAXBLK - IMINBLK, & \text{if N or S Face Neighbor}\\ JMAXBLK - JMINBLK, & \text{if E or W Face Neighbor}\\ 1, & \text{if Corner Neighbor}\end{cases}\\ COMM &= SUM(COMM(i))\end{aligned} \tag{10}$$

where Eqn 10 must be evaluated for all faces and corners of a given block and the results must be summed.

Weights of each type of cost are currently set to make the maximum possible geometric cost equal to the maximum possible communication cost, as accomplished by Eqn 11.

$$\begin{aligned} WGEOM &= 1 \\ WCOMM &= FACTOR \cdot \frac{(IMAXBLK+2)(JMAXBLK+2)}{(2\cdot IMAXBLK)+(2\cdot IMAXBLK)+4} \\ SIZE &= (WGEOM \cdot GEOM)+(WCOMM \cdot COMM) \end{aligned} \tag{11}$$

where $FACTOR$ is a number that can be varied to tune cost weighting, but is currently set to 1.

Once block loads are calculated, they are sorted by size in order of greatest to least. They are then distributed to the processors in this order, where each block is assigned to the current processor with the least load. This produces the theoretical load balancing presented in Section 3. Actual load balancing performance will be determined in Project 5 and tuning will be performed to optimized load balancing.

2.5 Ghost Nodes and Neighbor Indentification

In order for each block to function independently for a given iteration of the solver, it must know information about the nodes immediately outside of its boundaries, or, in other words, the interior nodes of its neighbors. To preserve block independence, each block stores the information it needs from its neighbor at the beginning of each iteration in extra, off-block nodes called ghost nodes. These nodes change the local size of each block and necessitate the local iteration parameters $ILOCMIN$, $ILOCMAX$, etc. discussed earlier.

To update each ghost boundary, the identity of the neighbor block for each face is stored in a variable NB, which is a neighbor derived data type $NBRTYPE$, which contains IDs for the north, south, east, and west faces and the north east, south east, south west, and north west corners. If the block boundary is a physical boundary instead of an inter-block boundary, the corresponding neighbor identifier is instead set to 0 to indicate a BC boundary. For parallel computing, if a neighbor block is on a different processor (indicating a processor boundary), the neighbor block ID is negated to indicate as such while still preserving the neighbor block ID.

Neighbor information is used to populate a linked list for each boundary type with block IDs so that all similar types of boundaries may be looped through in sequence, rather than using logical sorting at the beginning of each iteration. (Linked lists were shown to produces a 25% speed-up compared to logical sorting for the serial, multi-block code).

When moving to parallel computing, the ID of the neighbor blocks processor must also be bookeeped, as it is required information for accessing the neighbor block for ghost updating. In addition to the neighbor blocks processor, the local index of the neighbor block on its processor must also be stored for this same reason. Thus, this data is stored in corresponding $NBRTYPEs$. Neighbor processor IDs are stored in the variable NP. If a block boundary is a BC, the processor ID is negated to indicate as such. Local indices of neighbor blocks on neighbor processors are stored in $NBLOC$ and are set to 0 if a boundary is a BC.

2.6 Configuration Restart Files

After all of the above mentioned initialization processes have been completed, this information is stored in individual restart files for each processor so that the solver may start up independently from these files without needing to determine boundary procedures. Neighbor information, grid, and temperature files are written **for each processor**.

2.7 Parallel Performance

Performance of the parallel solver is measured in the results section using Amdahl's Law and the concepts of parallel speedup S_P (Eqn 12) and parallel computational efficiency E_P (Eqn 13).

$$S_P = \frac{t_s}{t_p} \tag{12}$$

where t_s is the serial solution time and t_p is the parallel solution time.

Ideal speedup produces a line with a slope of 1, with the serial solution time decreasing by exactly the amount of processors used. Ideal efficiency is 1, and corresponds to the ideal speedup.

$$E_P = \frac{S_p}{N_P} \tag{13}$$

where N_P is the number of processors in the parallel solution.

Speedup can be predicted as a function of the fraction of the code that is run in parallel according to Eqn 14. Because only the parallel portion of this code is timed, optimal speedup is expected to by near-ideal.

$$S_{P,Opt} = \frac{N_P}{f+(1-f)N_P} \tag{14}$$

3 Results and Discussion

Solutions of a 501x501 grid decomposed into 10x10 blocks were solved on 2, 4, 6, and 8 processors on the hpc1 front end (Batch jobs on hpc1 took significantly longer, which seemed to be due to running more processes than allocated cores).

Another important lesson learned from this project was the importance of shared variables in MPI. Many days of debugging ultimatley resulted in discovering that the global grid size was only read in by Processor 0, and was not available to the other processors for calculation of mesh parameters. This was solved with a simple MPI_BCAST.

Another pitfall was in creating unique tag IDs for MPI sends and recieves. Each send is identified by the sending and recieving processor, as well as a tag, so the first attempt

used 8 independent tags for the 8 directions of communication. This proved to be non-unique as one processor could send to another processor more than once in the same direction for a given iteration. In the end, a unique tag was created for each send as the concatination of the direction number with the global sending block ID.

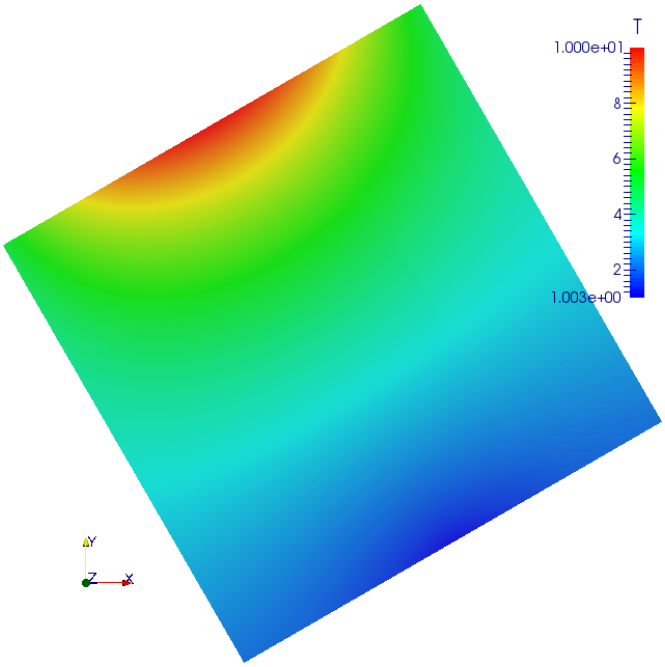

Fig. 1: Steady-state heat transfer solution for a 501x501 grid decomposed into 10x10 blocks solved on 4 processors

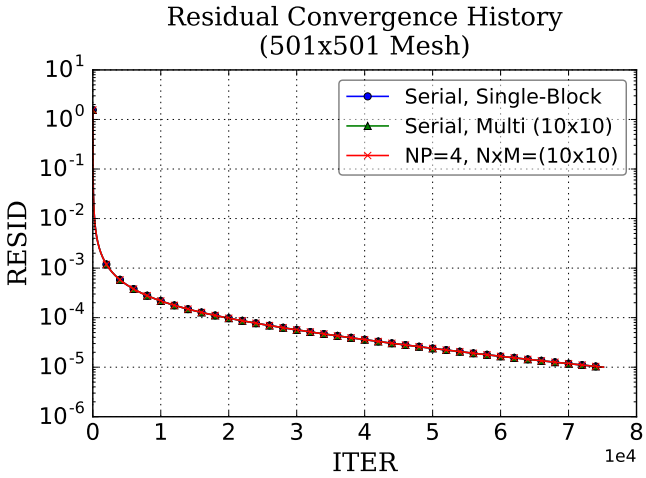

Fig. 2: Serial and parallel convergence history demonstrating similar performance between all solution methods

Convergence history for the serial, single-block and multi-block cases are compared to the parallel case in Fig 2.

It can be seen from the convergence comparison that the parallel solver performs almost identically to its serial analogs, and Fig 1 demonstrates that the parallel solution is accurate.

After much deliberation with hpc1, results demonstrating actual parallel speedup were produced and total solution wall times are presented for serial and parallel runs of the Project 5 code in Table 2.

Table 2: **Serial (Left) and Parallel (Right) Solutions Times**

N_P	(1x1)	(10x10)	2	4	6	8
t (s)	360.7	377.8	179.4	91.95	70.56	76.09

It can be seen that the parallel code does indeed reach a solution faster than its serial analogs. The speedup and efficiency of the parallel calculations are further demonstrated in Figures 3 and 4.

Because near-perfect load balancing was accomplished (shown in Table 3) and because the timed portion of the code was entirely parallel (see Eqn 14), it was expected that optimal speedups would be near the ideal case, as Fig 3 demonstrates.

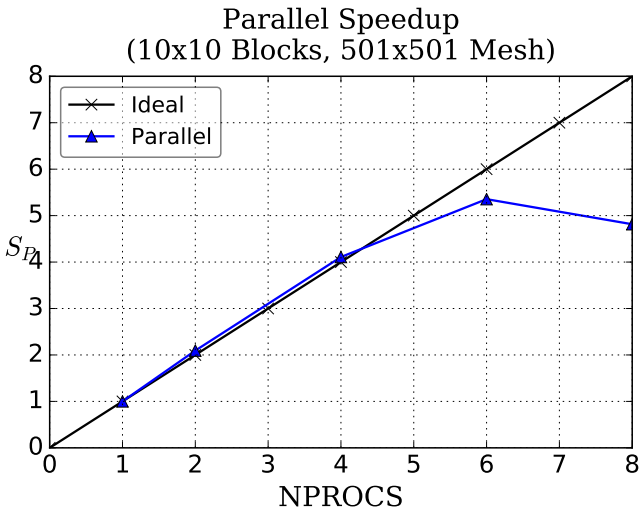

Fig. 3: Parallel speedup for a 501x501 grid decomposed into 10x10 blocks demonstrating an ideal speedup and sometimes super-linear that tapers off as the number of processors increases

Fig 3 also demonstrates that the maximum possible speedup for this case is approximately 5.25 running on 6 processors, which is a significant improvement on the performance of the serial version.

The parallel efficiency of the solver is greatest for lower processor numbers. It actualy exceedes maximum efficiency for the 2 and 4 processor cases, but this phenomenon may be due to variation in loading of the hpc1 front-end cores between runs.

A steep drop-off in efficiency is noted as the number of processors is increased. To extend the curve further out, the solver could be applied to a larger grid system decomposed into more sub-domains.

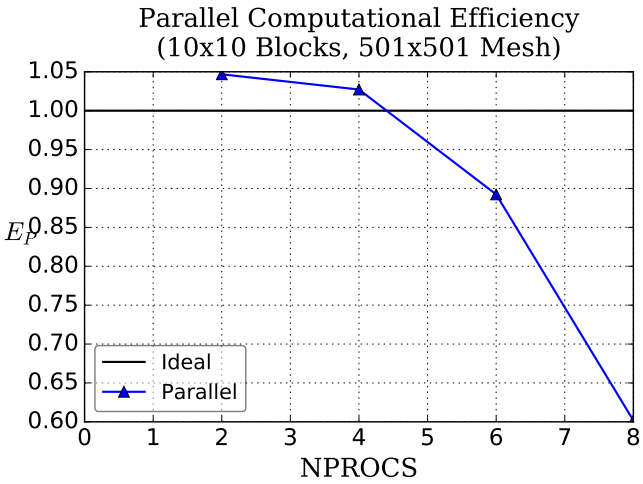

Fig. 4: Parallel efficiency for a 501x501 grid decomposed into 10x10 blocks demonstrating a greater than ideal efficiency for lower amounts of processors that may be due to variations in core loading

Table 3: **Processor Theoretical Load Balances**

NPROCS	4	8
Proc0	1.0000	1.0306
Proc1	1.0000	1.0306
Proc2	1.0000	1.0306
Proc3	1.0000	1.0306
Proc4	N/A	0.96943
Proc5	N/A	0.96943
Proc6	N/A	0.96943
Proc7	N/A	0.96943

4 Conclusion

The product of this project is a parallel, multi-block heat conduction solver demonstrating the fundamental principles of computations in parallel. It increased knowledge of parallel information passing, parallel speedup performance, load balancing optimization.

The working code is just a beginning of what can be accomplished with parallel computing. This code itself could be improved with better sub-domain decomposition optimization and optimization of calculations like send/recieve tags. Beyond that, the principles of this code could be applied to numerical solvers of different principles such as fluid dynamics or structures. Parallel computing could be adapted to GPUs instead of CPU cores. The list is endless.

These projects served as an effective introduction to parallel computing by gradually building on principles; starting with a standard numeric heat transfer solver and then incrementally incorperating parallel computing concepts like domain decomposition and solution restart files so that each concept could be solidified before the next was attempted. Aside from developing a strong parallel computing basis from which to build off of, the course also developed skills for aquiring further knowledge, leaving the student prepared to delve further into the field of parallel computing.

This course is highly recommended for students interested in numerical solutions of engineering problems, as the principles of these projects can be adapted to any application that may be of interest.