from *SIAM News*, Volume 31, Number 2

Parallelization of a Relaxation Method

By C.Y. Lee, S.M. Lee, J.S. Oh, and B.H. Koo

In the solution of large linear systems, as with the large-scale Laplace equation that arises in the boundary-element formulation described here, the available memory is often insufficient for use of a direct elimination method. Iterative methods, which require less memory, can be applied in such cases, although the cost can be longer computation times. In this article we describe a fast, parallelized under-relaxation iterative algorithm used in the modeling of an injection molding process.

APPLICATIONS ON ADVANCED ARCHITECTURE COMPUTERS
Greg Astfalk, Editor

We used the Computer Aided Plastic Application (CAPA) code, which was written originally to run serially on an HP735 workstation. The limited memory (128 Mbytes) in this system makes the original, direct solver algorithm in CAPA inefficient. With a slight modification that makes it possible to use more of the available memory, however, CAPA can run significantly faster on computer systems with sufficient memory. When this strategy is used with the parallelization of the relaxation algorithm in CAPA, the result is an almost ideal speedup.

The parallelization, by taking advantage of greater amounts of the aggregate memory and concurrent execution on multiple processors, makes it possible to run larger problems. Our focus in this article is twofold: the parallelization of the relaxation algorithm in CAPA and the strategy for memory usage.

Three problems (i.e., data sets or models), shown in Table 1, are considered in this article. The table lists the numbers of boundary elements that determine the size of two large matrices, *K* and *G*, for each model, as explained in later sections.

Cooling in Injection Molding

CAPA was developed to model three-dimensional heat transfer during the cooling stage of an injection molding process. In the model, the heat transfer occurs as a cyclic–steady, three-dimensional conduction process; heat is exchanged with the ambient air and the exterior surfaces of the mold.

For the numerical implementation, we use a hybrid scheme consisting of a modified three-dimensional boundary-element method (BEM) for the mold region and a finite-difference method for the melt region. These two analyses are iteratively coupled so that the temperature and heat flux match at the mold–melt interface.

During the cooling stage, the polymer in a mold cools and solidifies owing to conductive heat transfer. Since plastic parts are usually thin, a local one-dimensional transient analysis is adequate for a three-dimensional plastic part.

The governing equation for conductive heat transfer is given as:

$$\rho C_p \frac{\partial \vec{T}}{\partial t} = \frac{\partial}{\partial z}\left(k \frac{\partial \vec{T}}{\partial t} \right) \tag{1}$$

where $\vec{T}$ denotes the melt temperature; ρ, C_p, and k are the density, thermal conductivity, and specific heat, respectively; and z is the local coordinate along the part thickness direction.

We use a cycle-averaged three-dimensional approach for the mold-filling analysis and a cycle-averaged temperature field as it affects the plastic part and the cooling system for the melt analysis. To couple the mold analysis with the melt analysis, we use compatibility conditions at the mold–melt interface; see [2] for details. For mold surfaces that are in contact with the coolant, heat transfer coefficients are based on the Dittus–Boetler correlation [1].

With the above assumptions, the governing equation for the melt cycle-averaged temperature field is given by the Laplace equation,

$$\Delta \vec{T} = 0$$

We use an implicit finite-difference method to solve equation (1). To solve for the temperature field in the mold, we use the BEM, modifying the standard BEM for any two closely spaced surfaces of the mold [3].

To numerically solve the equations resulting from the BEM, we discretize the boundary surfaces of the mold into triangular and line elements. The appropriate boundary formula is then applied to each element.

Model	Plastic part	Cooling channel
BOX4	1563	100
JAR-BODY	3698	527
GRILL-GATE	8801	160

Table 1. *Number of boundary elements in the three models used in this study.*

Once integrals have been calculated, the discretized boundary-element formula can be manipulated into the following form:

$$K_{ij}\vec{T}_i = G_{ij}\left(\vec{T}_{,n}\right)_i \tag{2}$$

where K_{ij} and G_{ij} are functions of the geometry of the boundary surfaces. Once the boundary and compatibility conditions have been incorporated, equation (2) takes the form

$$K\vec{T} = \vec{R}$$

where $\vec{T}$ contains all the unknowns. Depending on the boundary conditions, $\vec{T}$ is either the temperature or the temperature gradient; $\vec{R}_i$ depends on the coefficient G_{ij} and the boundary and compatibility conditions.

This system is solved by an under-relaxation iterative method. Because an iterative method requires less memory than a direct solver, we can solve larger problems with a given amount of memory. In addition, we often have a very good initial guess for the unknowns. With a good initial guess, an iterative method can converge rapidly.

Strategy for Memory Usage in CAPA

Storage of the *K* and *G* matrices in (2) accounts for most of the memory requirement in CAPA. The limited amounts of memory on the workstations where we originally ran CAPA prevented storage of these matrices in memory. Our original alternative was to store only parts of the matrices, utilizing all of the existing memory, and constantly recalculate the parts not stored. Experiments have shown that recalculating *K* and *G* can be faster than relying on the virtual memory (i.e., paging) system to handle the storage of matrices that exceed the physical memory of the system. Moreover, some systems, like Cray computers, do not support paging.

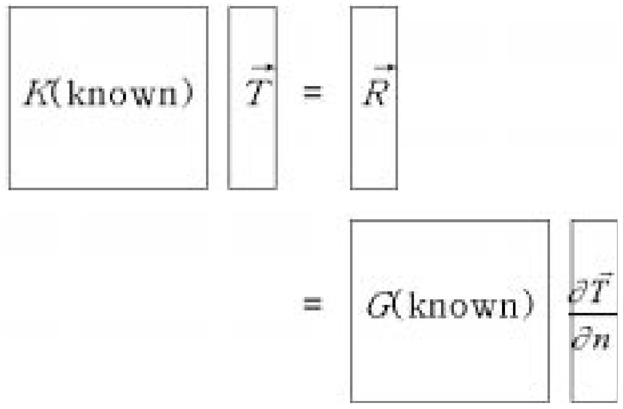

Figure 1. *The system of equations, $K\vec{T} = \vec{R}$, from the BEM portion of the CAPA code.*

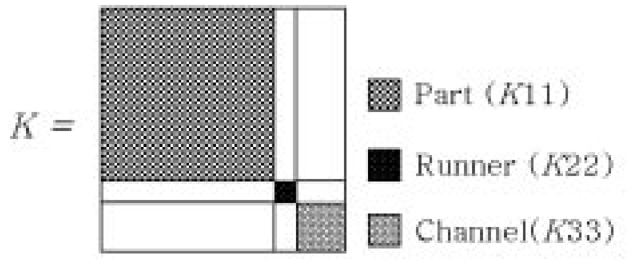

Figure 2. *Three of the nine submatrices of the K matrix.*

Figure 1 shows the *K* and *G* matrices that arise in the BEM phase of the CAPA algorithm. Each of these matrices can occupy a significant amount of memory. The *K* matrix is used more often than *G*. The *K* matrix is further divided into nine submatrices, as shown in Figure 2.

From Table 1 and Figure 2, it is obvious that K_{11} is the largest of the submatrices, accounting for 80–90% of the *K* matrix. The original version of CAPA stored all of K_{22} and K_{33} and a part of K_{11}. Values for the remaining six parts are recalculated at every iteration. The portion of K_{11} that is stored depends on the amount of available memory. With enough memory, the full K_{11} is stored.

K and *G*, which are used in the solution of the equations $K\vec{T} = \vec{R}$ and $G\frac{\partial \vec{T}}{\partial n} = \vec{R}$, are known and do not change during the execution. If *K* and *G* are in memory, they do not need to be recomputed at every iteration and the computation time will be reduced. When the model is large, the memory of one processor is generally sufficient to store only a part of *K*. On a parallel system, *K* is partitioned and distributed among the memory in different processors. If all of *K* is memory-resident, then as much of *G* as can be accommodated is also stored in memory.

As in the original CAPA, we partition *K* into nine different parts. The only part not guaranteed to be in memory is K_{11}. Since the total amount of memory in a parallel system is proportional to the number of processors, then given enough processors, extremely large *K* matrices can be stored. Parallel programs can gain advantage from more aggregate memory as well as concurrent computation. CAPA either stores the entire *K*, if there is sufficient memory among the multiple processors, or recalculates the unstored portion of K_{11}, if the number of processors (i.e., memory) is insufficient.

With the BOX4 model (see Table 1) run on one processor of an SGI Challenge, the entire *K* matrix can be stored in memory. The total execution time is 30% less with the parallel code than with the original version of the algorithm. The primary difference between the original and parallel versions is in the relative portions of the *K* and *G* matrices that are stored and recalculated. The original approach, as discussed earlier, was to recalculate most parts of *K*, whereas with the parallel code we are able to store those parts and no longer need to recalculate them. Therefore, the parallelized CAPA code runs approximately 30% faster just on the basis of its ability to store the entire *K* matrix in memory. This performance gain is independent of any gain from the parallelization of the under-relaxation method.

Figure 3 illustrates the distribution of the *K* matrix over the available memory in a parallel system. Let *num_nodes* be the number of processors and *K* an $N \times N$ matrix. *K* is partitioned and interleaved row-wise among the *num_nodes* processors; $num\_nodes \le N$, and we assume here that *num_nodes* mod $N = 0$. The first *num_nodes* rows of *K* are distributed, one to each processor, in order. The second *num_nodes* rows of *K* are then distributed in the same way, and the process continues until all rows of *K* have been distributed. Each processor *p* has a subset of the rows of *K*.

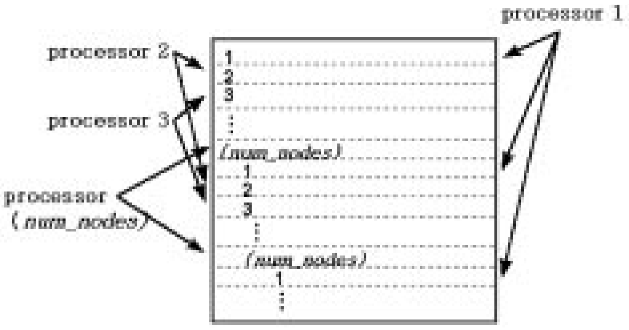

Figure 3. *The row layout of K.*

Parallelization of the Relaxation Method

The under-relaxation method (URM) is applied to solve equation (3). The general form of the URM is expressed as follows:

$$t_i^M = (1-w)t_i^{(M-1)} + \frac{w}{k_{ii}}\left(r_i - \sum_{j=i+1}^{N} k_{ij} t_j^{(M-1)} - \sum_{j=1}^{i-1} k_{ij} t_j^M \right) \tag{3}$$

In this equation *w* is the relaxation parameter ($w < 1$), *i* and *j* are the row and column indices, *M* is the iteration number, and *N* is the order of the system.

Equation (3) shows that t_i^M depends on the *i*th row of *K*, the *i*th element of the right-hand side, $\vec{R}$, and $\vec{T}$. *K* does not change during the execution, and although $\vec{R}$ is updated occasionally after an iteration completes, it remains constant within an iteration. In addition, $\vec{T}$ consists of two parts: $t_j^{(M-1)}$, where $i \le j$, and t_j^M, where $j < i$.

The first set of $t_j^{(M-1)}$ is from the previous (($M-1$)th) iteration. The second set is from the current (*M*th) iteration. In a sequential program, elements of $\vec{T}$ are updated from 1 to *N*, in order, so that all the data needed to update t_i^M are available.

The parallel code does not guarantee the availability of all the requisite data, because each processor updates only a subset of t_i^M. Most of the t_j^M values needed to update t_i^M reside on other processors. Only by message passing among the processors can all of t_j^M be made available to solve this problem. For *i* from 1 to *N*, the updated t_i^M values are communicated to the other processors. Obviously, the communication overhead results in decreased efficiency, either because processors are idle or because processors that would otherwise be doing meaningful computation are needed to perform the required communication. Therefore, it is the characteristics of the communication, more than anything else, that determine the efficiency of the parallel code. The remainder of this section shows how the communication is implemented for the URM.

With respect to the distribution of *K*, equation (3) is rewritten as:

$$\begin{aligned} t_i^M &= (1-w)t_i^{(M-1)} + \frac{w}{k_{ii}}\left(r_i - \sum_{j=i+1}^{N} k_{ij} t_j^{(M-1)} - \sum_{j=1}^{i-1} k_{ij} t_j^M \right) \\ &= \frac{w}{k_{ii}}\left[r_i + \frac{k_{ii}}{w}(1-w)t_i^{(M-1)} - \sum_{j=i+1}^{N} k_{ij} t_j^{(M-1)} - \left(\sum_{j=1}^{S} k_{ij} t_j^M + \sum_{j=S}^{i-1} k_{ij} t_j^M \right) \right] \end{aligned} \tag{4}$$

$$= D - A - B - C \tag{5}$$

where $S = \max(1, i - 1 - num\_nodes)$ with

$$A = -(1-w)t_i^{(M-1)} + \frac{w}{k_{ii}} \sum_{j=i+1}^{N} k_{ij} t_j^{(M-1)}$$

$$B = \frac{w}{k_{ii}} \sum_{j=1}^{S} k_{ij} t_j^{M}$$

$$C = \frac{w}{k_{ii}} \sum_{j=S}^{i-1} k_{ij} t_j^{M}$$

$$D = \frac{w}{k_{ii}} r_i$$

We assume that processor *p* updates t_i^M and that all processors have the same initial data for $\vec{T}^M$. Since t_j^M, for $j < i$, changes as *i* progresses from 1 to *N*, processor *p* needs the changed t_j^M values in order to update t_i^M. Processor *p* receives all the necessary t_j^M values from processor $p - 1$, updates t_i^M, and then sends those t_j^M values and t_i^M to processor $p + 1$. Processor $p + 1$ can then update t_{i+1}^M. This process repeats until all the t_i^M values have been updated. Since the last processor does not have a next processor, it sends the data to the first processor. Similarly, the first processor receives the necessary t_j^M from the last processor. With each processor updating the t_i^M assigned to it, the updating of t_i^M proceeds in a round-robin fashion.

If *p* is a processor ID from 1 to *num_nodes*, an arbitrary processor *p* would recalculate the t_i^M values for $i = p$, $p + num\_nodes$, $p + 2 \times num\_nodes$, . . . , and $i \leq N$. A careful look at this algorithm reveals that processor *p* already has updates for most t_j^M values. The processor that updates t_i^M ($i = p + num\_nodes$) received t_j^M for $1 < j < p$ when it was updating t_i^M ($i = p$). To calculate t_i^M in equation (3), the processor needs only to get t_j^M for $0 < i - num\_nodes < j < i$. This corresponds to *C* in equation (5). Terms *A*, *B*, and *D* are already in the processor's memory. As a result, *C* represents the only portion of t_j^M that needs to be obtained from the previous node.

Equation (5) shows that processors can start subtracting *A* and *B* before *C* arrives; a processor does most calculations, in fact, while waiting for *C*. Overlapping the computation and communication reduces the detrimental effect of the communication over- head. Having the processors subtract *A* and *B* while waiting for *C* almost entirely prevents the processors from being idle due to the communication. Algorithm 1, which is very efficient and yields close to ideal speedups, is shown in pseudocode in the box below. Figure 4 pictorially illustrates the algorithm when four processors are used. The arrows indicate the direction of communication and the amounts of data involved.

```
do $i = p$ to $N$ increment by $num\_nodes$
     Subtract $A$ from $D$
     Subtract $B$ from $D$
     Receive $C$ in equation (5) from the previous processor
     Subtract $C$ from $D$
     Assign $D$ to $t_i^M$
     Send $t_j^M$ ($0 < i - num\_nodes < j \leq i$) to the next processor
enddo
The last processor distributes $t_j^M$ ($0 < N - num\_nodes < j \leq N$)
to all other processors.
```

Results

We implemented the parallel version of CAPA on a 256-node Intel Paragon XP/S at the Samsung Advanced Institute of Technology (SAIT) in Suwon City, Korea. The performance measurements were obtained with the three models listed in Table 1. The memory requirements for the *K* matrix alone for BOX4, JAR-BODY, and GRILL-GATE are 47, 320, and 1300 Mbytes, respectively. Each node of the Paragon at SAIT has only 32 Mbytes of memory; even with a paging space of 128 Mbytes, then, JAR-BODY and GRILL-GATE cannot run on one node. Even BOX4 requires virtual memory (paging) when run on one node.

The run-times are shown in Table 2 for various numbers of nodes (i.e., processors). Because of the working set size, we had to rely on paging in some of the runs. As more processors are used, more memory becomes available to store greater portions of *K* and *G*. The elements of *K* and *G* that are stored in memory are calculated only once. Elements of *K* and *G* that are not memory-resident must be recalculated once every iteration. Thus, as the number of processors grows, the amount of calculation decreases, with the expected result.

Table 2 shows that the parallelized under-relaxation algorithm is efficient. In the case of BOX4, for example, the full *K* and *G* matrices can be stored in memory if eight or more processors are used, and any reduction in time achieved with more

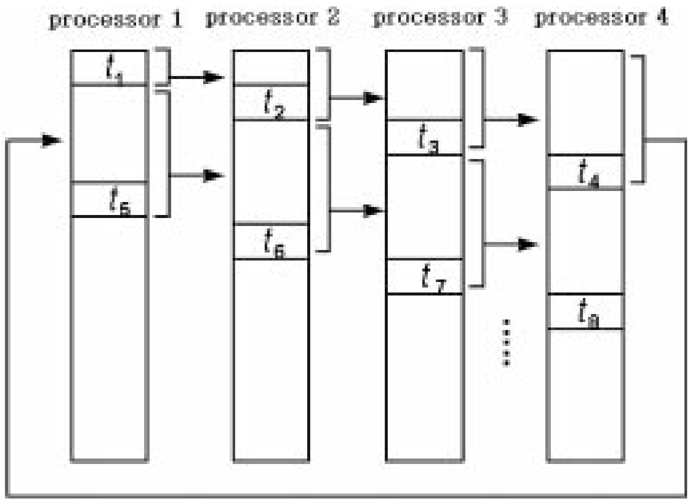

Figure 4. *Message passing of the elements of $\vec{T}$ for num_nodes = 4.*

nodes	BOX4	JAR-BODY	GRILL-GATE
1 (sequential)	3022	18820	60037
1 (parallelized)	*1813*	—	—
2	*582*	—	—
4	*151*	—	—
8	63	*1564*	—
16	38	*484*	*3337*
32	25	146	*1600*
64	24	89	*559*
128	31	76	180

Table 2. *Run-times, in seconds, on the Intel Paragon. The times shown in italics are for runs that used paging because the working set size exceeded the physical memory. The sequential times are for CAPA run on the SGI Challenge.*

than eight processors is a result of the concurrency of the computation. The relative efficiency is defined as $\frac{T_S}{T_{S+L}} \times \frac{S}{S+L}$, where S is the base number of processors and L is the number of additional processors. The relative efficiency for BOX4 run on 16 processors, as compared with eight processors, is 82.9%.

References

[1] B. Gebhart, *Heat Transfer*, 2nd edition, McGraw-Hill, 1977, New York.

[2] K. Himasekhar, K.K. Wang and J. Lottey, *Mold-cooling simulation in injection molding of three-dimensional thin plastic parts,* Heat Transfer Conference, HTD, 110 (1989), 129–136.

[3] M. Rezayat and T.E. Burton, *A boundary-integral formulation for complex three-dimensional geometries*, Int. J. Numerical Methods in Eng., 29 (1990), 263–273.

[4] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, 2nd edition, Springer-Verlag, 1987, New York.

C.Y. Lee (cylee@radon.sait.samsung.co.kr), S.M. Lee (smlee@radon.sait.samsung.co.kr), and J.S. Oh (soo@radon.sait.samsung.co.kr) are all researchers in the Samsung Advanced Institute of Technology's Supercomputer Application Lab. B.H. Koo (bhkoo@rnd.sec.samsung.co.kr) is with Samsung Electronics Co., Ltd., in Suwon City, Kyungki-Do, Korea.