Parallel Solutions for Dynamic Focussing of Large Acoustic Arrays

Karen P. Watkins

CS 392C: Methods and Techniques of Parallel Programming

Dr. J. C. Browne

Spring 1999

1 Problem Statement: Parallelization of Beamformer Coefficient Computation

Computationally intense spatial filtering called beamforming is central to acoustic imaging. Typically, in order to achieve an update rate similar to slow scan television, beamforming must operate in real-time. The real-time nature of beamforming leads to massively parallel execution for state-of-the-art applications in diverse fields such as seismic exploration, aeroacoustic measurements, ultrasonic medical imaging, and underwater acoustics [1].

1.1 Interpolation Beamforming

The beamforming function is a spatial filter that processes discrete signals from a number of sensors in order to produce a directionally-sensitive response. The signals arriving from the desired look angle are reinforced by imposing time delays on each sensor so that the signal formed by the summation of all sensor signals is maximized. Multiple beams are formed simultaneously at incrementally different angles to form an image. However, the sampling rate required to adequately describe the temporal signal may be too coarse for the resolution of the time delays needed to accurately point the beam in the desired direction. The concept of interpolation beamforming is used to mitigate the spatial sampling requirement and control the amount of processing needed to form accurately pointed beams.

In interpolation beamforming, the signals from each sensor are sampled at or above the Nyquist rate dictated by the bandwidth of the signal (Figure 1). These signals are augmented with inter-sample zeroes (zero padded) to effectively increase the input sample rate. Next, a set of coefficients implementing a lowpass finite impulse response (FIR) filter is used to interpolate between the samples, thus creating an accurate representation of the input signal now sampled at the higher frequency [2]. In addition, a weighting function, W, is applied to enhance the angular discrimination of the formed beam. Following the FIR filter, the output is decimated to return to the original sampling rate. The filtered signal is then modulated to the original operating frequency [3]. The final beam is formed by summing across all modulated sensor output signals.

The filter coefficients selected for a given sensor signal are chosen to impose the relative delay needed between this sensor and the others in the array to point the beam in the intended direction and at the intended distance, thus keeping the beam “in focus”, much like focussing a camera. To achieve highest quality images, the coefficients must be computed aperiodically and dynamically as a function of distance (range) and as a function of sound speed for proper focussing. In addition, these coefficients must be calculated and output in real-time for subsequent use in the interpolation filtering. This leads to a massive number of coefficients to be computed in real-time. For instance, in the research presented here, 20 beams are formed simultaneously across 32 sensors at 128 range increments using 11 coefficient values. Thus, nearly one million coefficient values must be computed in real-time. Because of the intense computational load and the real-time nature of this problem, multiple processors working in parallel must be utilized to meet the required throughput. In addition, for the processors to work efficiently and in

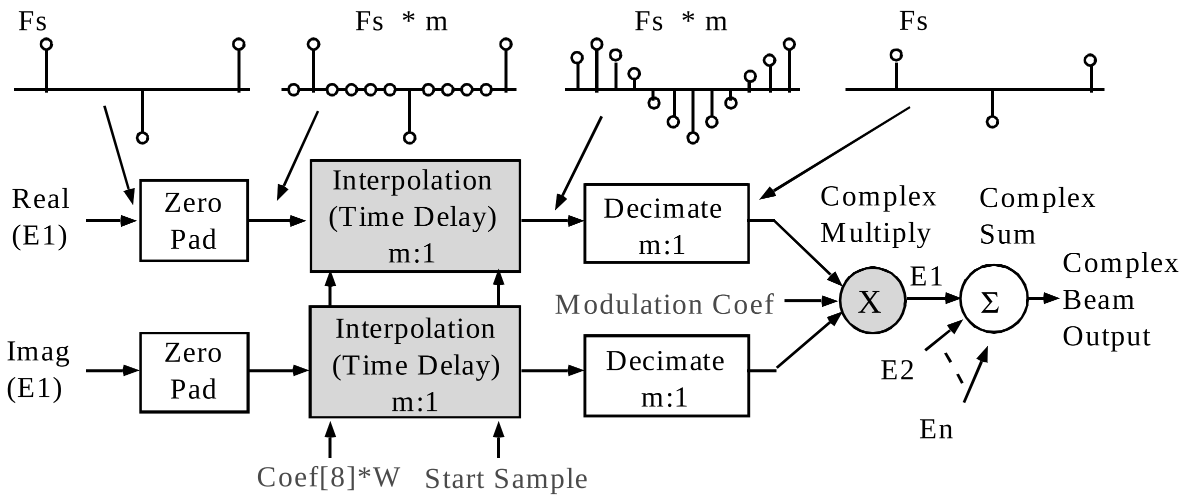


Fig. 1: Time Delay Interpolation Beamforming with Complex Samples

concert, careful design of task loading, partitioning, and synchronization of the parallel system must be considered for a successful solution.

1.2 Focussing Coefficient Calculation Algorithm

The filter coefficient calculation algorithm is shown in figure 2. This diagram depicts a single set of computations that is invoked for each beam and each sensor (element) at each focus range. In general, this algorithm calculates the time delay from a point in space to each element in the array and uses this information to output 1) the correct sample starting number, 2) the FIR filter coefficient set weighted appropriately, and 3) the modulation coefficients. This conjunct output hereafter will be referred to as a coefficient set. The inputs to this algorithm are the element number, beam number, range, and sound speed. As shown in this figure, several parameters are pre-calculated and stored for subsequent processing. They include the location of each element in the array in 3-D space (i.e. array geometry), the interpolation FIR filter coefficients, and the weighting coefficients, W.

The details of the computational kernel are described here. The time delay calculation computes the distance from a point along the center of the desired beam at the given focus range to each array element. These distances are translated into a time delay governed by the sound speed as:

$$\text{time\_delay}[i] = \text{distance}[i] \, / \, \text{sound\_speed}; \tag{1}$$

where i is the element number which ranges from 1 to the total number of elements in array. The relative time delay between each element is then computed by equation (2), where max_delay is the maximum time delay across all elements.

$$\text{relative\_delay}[i] = \text{max\_delay} - \text{time\_delay}[i]; \tag{2}$$

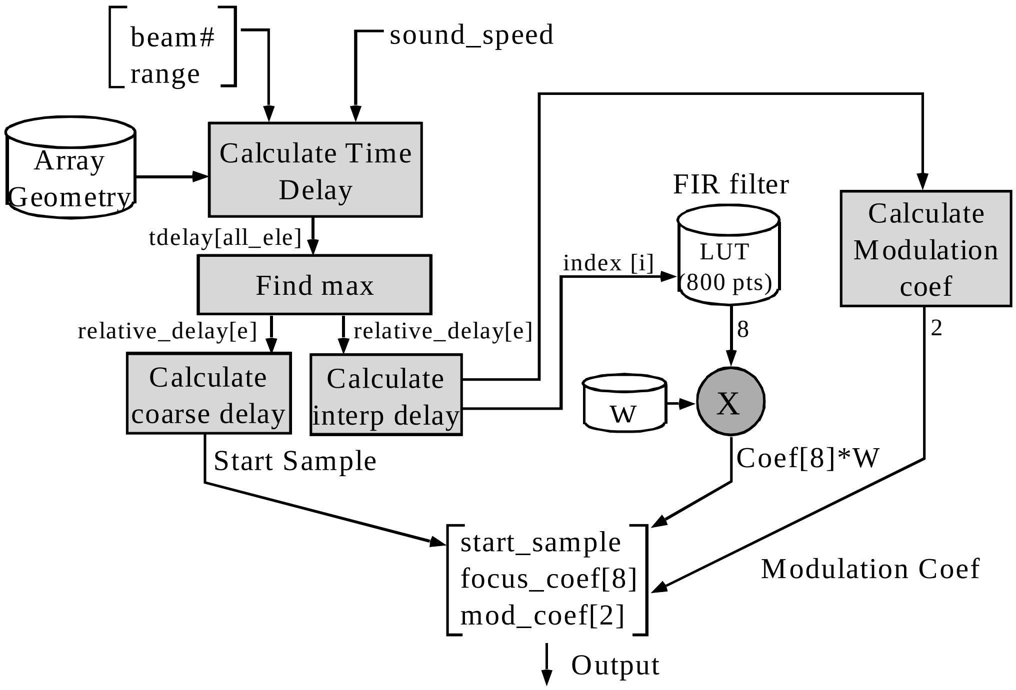


Fig. 2: Focussing Coefficients Calculation Algorithm

The set of relative time delays computed above is translated into start sample numbers by multiplying the sample rate of the system, converting each result into an integer, and subtracting the largest of these values from the set (equation 3).

$$\begin{gathered}\text{coarse\_time\_delay[i]} = \text{floor(relative\_delay[i]*sample\_rate);} \\ \text{sample\_number[i]} = \text{max\_coarse\_delay} - \text{coarse\_time\_delay[i];}\end{gathered} \tag{3}$$

where i is the element number.

The relative time delays, the system sampling rate, and the interpolation ratio of the filter are used to compute the filter coefficient set, as shown in equation 4. In particular, the interpolated time delay and coefficient indices are calculated as:

$$\begin{gathered}\text{sampled\_delay[i]} = \text{relative\_delay[i]*sample\_rate;} \\ \text{interp\_delay[i]} = \text{(sampled\_delay[i]} - \text{floor(sampled\_delay[i])*interp\_ratio;} \\ \text{coef\_index[k*i]} = \text{round(interp\_delay[i]);}\end{gathered} \tag{4}$$

where i is the element number and k is the required number of FIR filter coefficients. The index computed is the first coefficient in the set to be read from the table. The required number of coefficients are then read in order and multiplied by a weighting coefficient, W[i].

The modulation coefficients are calculated by taking the sine and cosine of an argument based on the relative time delays and the acoustic operating frequency as shown below [1].

$$\text{modulationRe[i] = cos(acoustic\_frequency*relative\_delay[i]);} \quad (5)$$
$$\text{modulationIm[i] = sin(acoustic\_frequency*relative\_delay[i]);}$$

where i is the element number.

1.3 Prototypical Example

For the purposes of analytic comparisons, a typical configuration of the above general description is desirable. The array consists of 32 elements. A total of 20 beams are to be formed using all elements. To keep the beam "in focus", coefficients must be computed for 128 different range increments. In order to compute a full set of coefficients in real-time, multiple processors must be used. Here, the number of processors is limited to five by system constraints.

This specific filter coefficient computation system is developed with two primary goals: 1) produce an effective implementation that runs in real-time with a fixed number of processors and 2) use an appropriate parallel programming system for development that naturally expresses the constructs of the final implementation. For example, the first goal requires algorithm decomposition using various partitioning strategies to be compared and analyzed with emphasis on efficient execution, minimal processor coupling and synchronization, and balanced task loading. The second goal suggests the use of a parallel programming system that can express the decomposed model, including the required message passing and process synchronization, at the highest level of abstraction possible without compromising the required execution throughput.

In this paper, a detailed approach to meeting the primary goals of the parallel system is presented followed by a specification of the final implementation for development. Two different parallel programming systems with significantly different development techniques are used to implement the final system. A description of each implementation is given. Finally, the two development systems are used as a basis of analysis for the comparison of the ease of development, speed of execution and other performance issues, and any difficulties encountered as a limitation of the programming system.

2 Approach

Several methods of parallel decomposition from previous research are considered in this section as candidate partitioning strategies.

2.1 Classes of Parallelization

Carriero and Gelernter describe three conceptual classes of parallelization: result, agenda, and specialist [4]. In result parallelism, a unique process is created to compute one element of the final output. In general, programs structured around result parallelism tend to be fine grained. In agenda parallelism, a program is divided into a list of tasks to be sequentially performed in a specific order. Programs using agenda parallelism are normally structured in a master-worker configuration where the master process initiates the computation and assigns subsets of the computation to a collection of identical worker processes. Agenda parallelism is naturally load balancing and tend to be coarser grained than result parallelism. In specialist parallelism, a program is divided into task types to be performed and processes are created to compute a single task type. Therefore, a specialist program is comprised of several types of processes each doing a single task in the overall computation. Specialist parallelism is naturally expressed in a pipelined structure.

An additional relevant and perhaps more traditional method of decomposition is data parallelism. In data parallelism, each process executes the same algorithm on different portions of the total data structure concurrently with other processes [5]. Here, typically a large kernel of the computation can be executed independently on different parts of the data with possibly a common computation preceding the large kernel and a collation process following it [6]. Data parallelism is a natural representation for engineering and scientific computations, such as the computations considered here [7].

2.2 Parallel Partitioning of Beamformer Coefficient Algorithm

At the highest level, the complete set of filter coefficient computations has near total parallelism. In fact, at the lowest level, the algorithm presented in figure 2 has inherent parallelism. However, in a large grained context, the computational structure can be thought of as a three dimensional matrix (figure 3). Each cell in the matrix consists of one instantiation of the algorithm described above. The matrix paradigm provides a basis for considering the alternatives in partitioning the problem. It also aids in understanding parallelism, granularity of the solution, and any system-level or procedural-level constraints. This decomposition will also lead to a definition of overall dataflow for an efficient implementation. In this prototypical system, it is assumed that communication costs are expensive and therefore, to find an efficient implementation requires minimizing message passing, or equivalently, increasing the granularity of the problem.

As stated earlier, there are many possible parallel solutions to the algorithm. Decomposing the problem as result parallelism is unsatisfactory, because the final

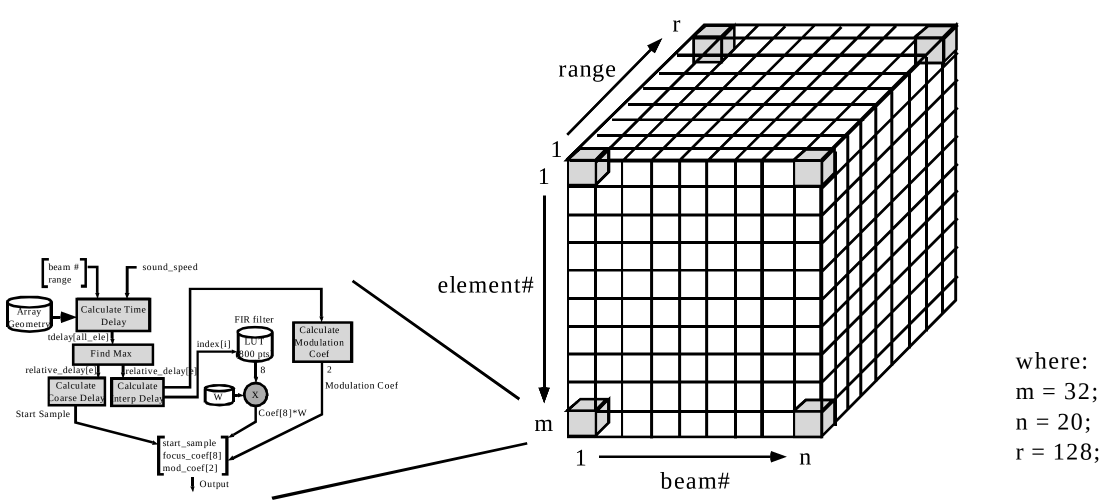


Fig. 3: Coefficient Computation Matrix

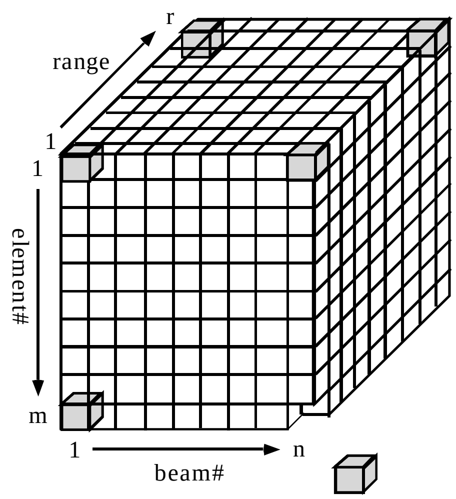
program would be too finely grained. This method is equivalent to decomposing at the algorithm level, where one process is created for each coefficient value to be computed. In this case and in the strictest definition of agenda parallelism, nearly one million processes would be required for all outputs. This clearly would result in tremendous overhead due to process synchronization, data passing, and context switching and would not produce an efficient solution.

Therefore, to keep the grain of the final program within reason, only decompositions along the axes of the three dimensional matrix will be considered. These are variations of data parallelism. Three alternatives range from medium grain to coarse grain. A medium grain partitioning strategy involves assigning a process to calculate one cell of the matrix. In other words, each process would compute one coefficient set for one element, for one beam at one focus range (figure 4a). This technique also has a tremendous amount of parallelism, requiring nearly 82,000 processes. In addition, this partitioning scheme requires close process coupling and synchronization since every coefficient calculation requires knowledge of time delays across all elements. This communication adds complexity to the design and is therefore, error prone.

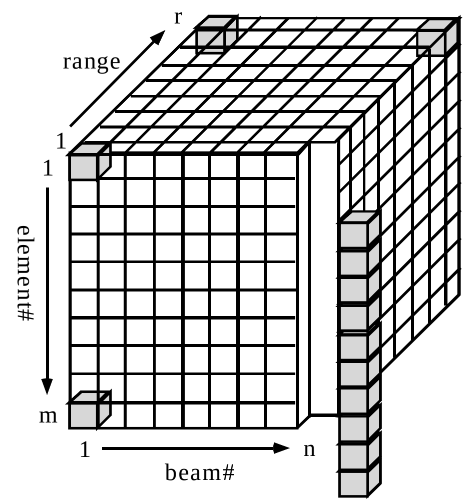
Because knowledge of all time delays is required to compute the relative time delay, the problem can be significantly simplified by partitioning such that all time delays for a given beam and range are computed by a single processor (figure 4b). This technique reduces the number of processes to 2560. It increases the coarseness of granularity and greatly reduces the amount of process coupling and synchronization. No intermediate results are passed between processes in order to complete the computations.

To reduce complexity further, another possible partitioning strategy involves assigning each processor to calculate the output structure for one beam at all focusing ranges (figure 4c). This technique requires only 20 processes (number of beams) and the coarseness of the granularity has been greatly increased. Processor coupling is also greatly reduced. Even though the number of processes has been greatly reduced from the first method, this method still requires more processes than processors available. Therefore, this method requires context switching between processes which increases overhead. In addition, synchronism between processes and processors is required to compute the full data set. Thus, inherent complexity can be eliminated with the following method.

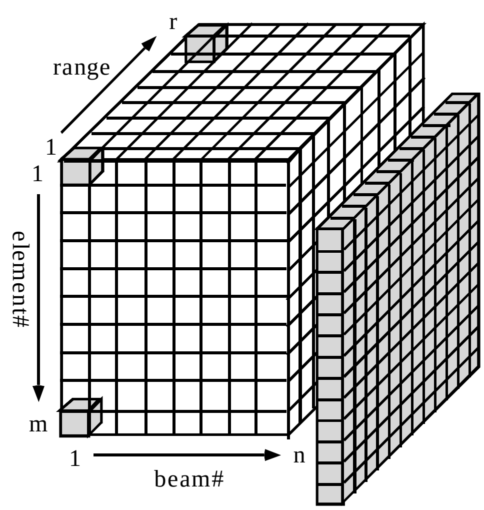
Figure 5 presents an attractive alternative. In this solution, the beam indices are divided into groups and the groups are distributed to each processor in the network to compute the coefficient sets. Thus, each processor computes coefficient sets for multiple beams, all elements, and all ranges. The number of beam index groups is chosen to match the number of available processors. Thus, each processor executes one thread of control, and therefore, no overhead due to context switching is incurred. This method also reduces overhead by minimizing interprocessor communication and synchronization. This method is naturally load balancing, since the task loading is equally divided. By design, the algorithm can execute on the available number of processors and is directly scalable for a variable number of elements, focusing ranges, beams, or even processors. It allows identical code to be written, debugged, and executed for multiple processors,



a) Each processor calculates one cell of the matrix



b) Each processor calculates one "pillar" of the matrix



c) Each processor calculates one vertical plane of the matrix

Fig. 4: Possible parallel Decomposition Strategies

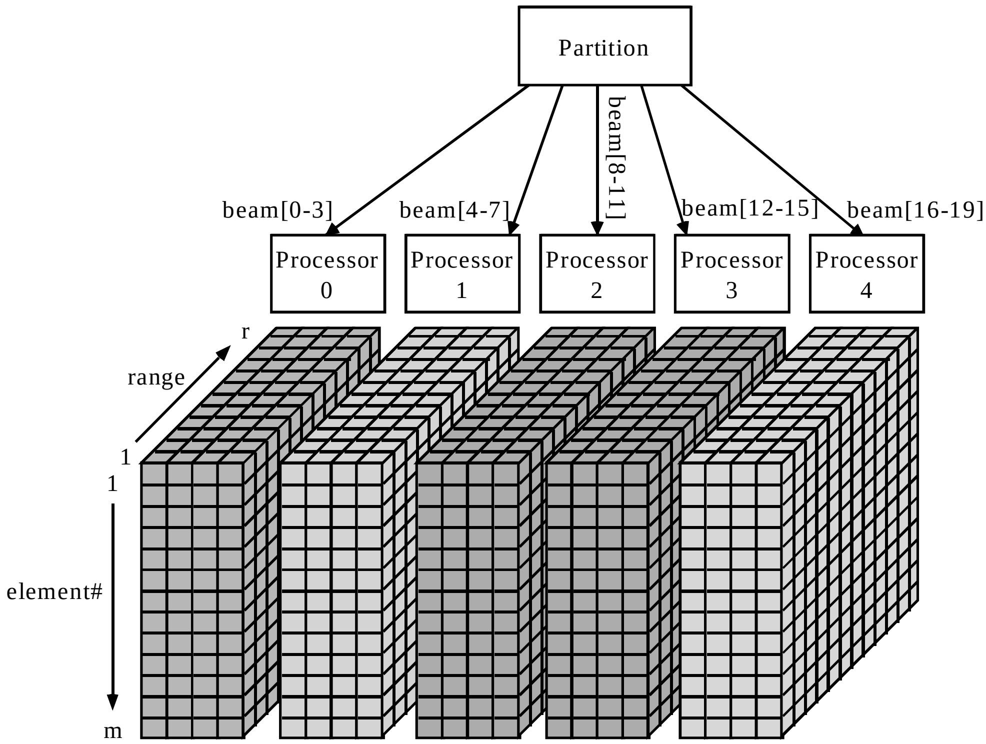


Fig. 5: Preferred Data Partitioning Solution

thus greatly simplifying development. This is the decomposition method we will evaluate.

3 Research Plan

Two considerably different parallel programming systems have been studied to implement the parallel coefficient computation and evaluate their respective performance and developmental efficiency. These systems include Computationally-Oriented Display Environment (CODE) and Message Passing Interface (MPI).

3.1 CODE Parallel Programming System

CODE is a graphical parallel programming system that targets multiple-instruction, multiple data (MIMD) style parallel computers at a coarse level of granularity (at the subroutine level) [8]. In CODE, the programmer specifies parallelism explicitly through graph topologies at a high level of abstraction [9]. These graphs are then automatically translated by CODE into architecture specific source code.

The "programming language" syntax of CODE is a multigraph [9]. A multigraph consists of nodes and arcs. Nodes represent atomic computations, computations that can only be interacted with at the beginning and the end of execution. These nodes encapsulate sequential computations on local data, firing rules that determine when a computation can execute, and routing rules that specify the recipients of data produced by the computation. These sequential computations are written in a conventional programming language such as C, just as any typical sequential subroutine would be written. Data is input and output to a node through named ports. Arcs bind port names between nodes. Arcs represent unbounded FIFO queues that send data from one node to another. Arcs specify the communication structure of the program. Nodes with no dataflow dependencies can be executed in parallel.

Figure 6 shows the graphical user interface of CODE. The interface includes a number of pre-defined node types, various input tools, several compile option buttons, a graph annotation button, and a large drawing area.

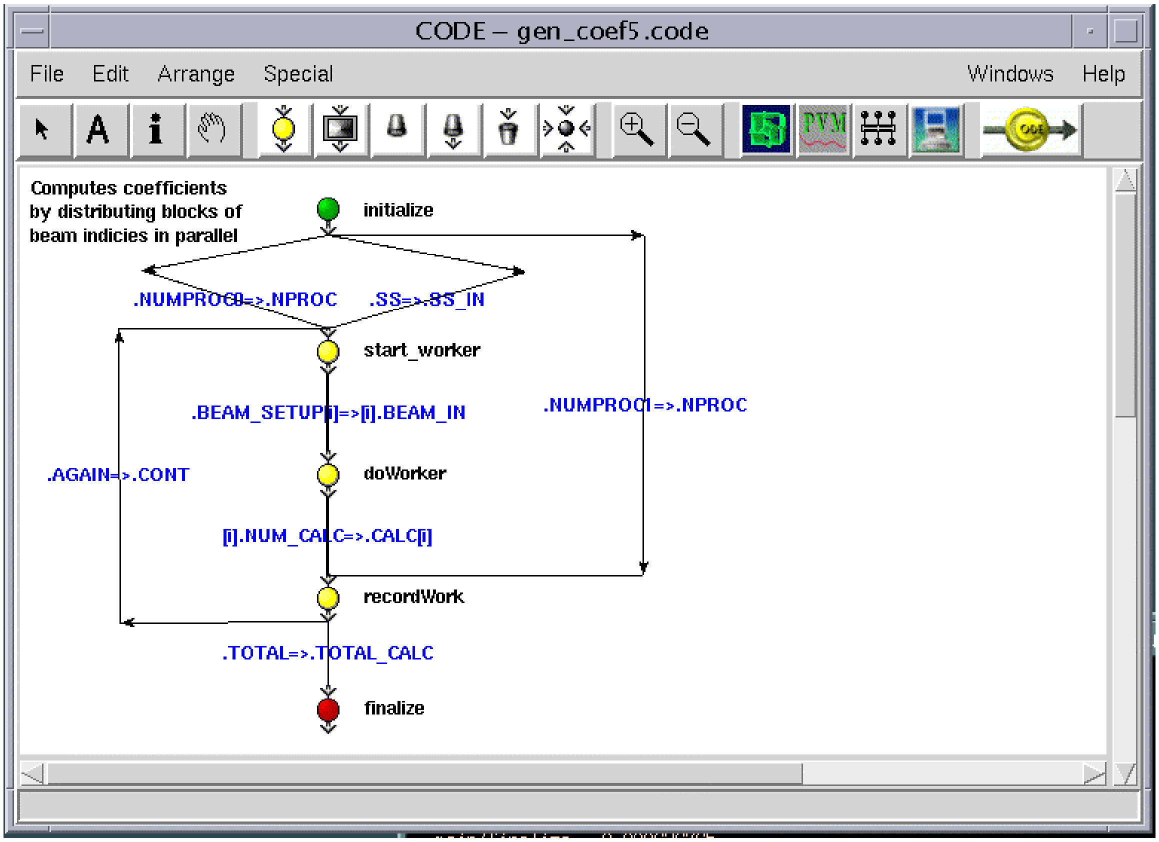


Fig. 6: CODE Graphical User Interface

The pre-defined node types include the unit of computation node, call node, container node, input and output nodes, and name sharing node (figure 7). The most important node in CODE is the unit of computation node. This node represents an atomic, sequential computation. These nodes can be instantiated multiple times through arc topologies, thus, representing parallel execution of these nodes. CODE supports hierarchical structuring of programs through the use of the call node. This node allows one graph to call another graph. The container, input, and output nodes support the use of call nodes. In particular, the container node is used to “broadcast” constant data to all nodes in the called graph. The input and output nodes are used to route data into and out of the called graph, respectively. CODE also supports shared variables through the use of

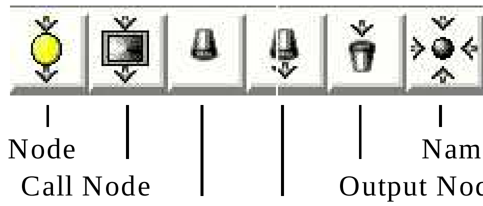


Fig. 7: CODE Node Types

the name sharing node. Nodes that access the name sharing nodes must specify themselves as a reader or writer of the shared memory.

When a parallel program is created with CODE, the program is fully and explicitly specified through graph, node, and arc annotations. For instance, when a unit of computation node is placed, the programmer specifies the local data, input ports, firing rules, output ports, routing rules, initialization code, and code that is executed each time the node is fired.

Firing rules serve two purposes. First, they define the conditions under which a node is permitted to execute [10]. Second, they describe the bindings that will occur when the node fires. Bindings specify the data that will be removed from an input port and placed into local variable. Firing rules are very flexible. There can be firing rules for more than one arc input. Also, nodes can have many firing rules and can fire when any of the rules is satisfied. If more than one firing rule is satisfied, one rule is chosen arbitrarily. CODE has syntax to replicate guards, bindings, or entire rules. For instance, rules can be written to fire a node when all input ports have data or when any of the ports have data.

Routing rules determine what local data will be placed on outgoing arcs. All bindings within a rule are performed from left to right. Routing rules can also be replicated to place local data from an array onto multiple output ports.

One of the primary design goals of CODE is to produce efficient executable parallel structures. To this end, CODE has many inherent features. CODE has the notion of lazy creation and state retention [11]. Lazy creation of element instances means no element is instantiated at runtime until data first appears on its input port. State retention means once an element is created, it is retained throughout the life of the program and may be used many times. By not creating all instances at start-up, execution time and memory are saved. By not destroying and re-creating each instance throughout the life of the program, time is saved.

CODE translation process also provides opportunities to produce efficient code. CODE has an object-oriented software architecture that allows translation processes to be independently written for different architectures [11]. When a translation process is written, optimization techniques can be applied specifically for the targeted architecture. For instance, if the targeted machine is a shared memory multiprocessor (SMP) machine, then to pass data from one process to another, a pointer can be sent instead of a traditional message which can be very expensive.

All translation processes use runtime libraries that implement an efficient scheduler for CODE nodes. This scheduler exploits the fact that nodes are atomic in nature and therefore, uses a non-preemptive scheduling technique.

Also, translation methods use algorithms to recognize the most common constructs in parallel programming. Once a common construct is found, it can then apply optimizations to these constructs to produce efficient code. An example of a very commonly used structure is the fan construct (figure 8) [9]. In this construct, a single source node (A) equally distributes data to other nodes (B nodes). Each B node performs identical tasks on their portion of the data and then passes the result to a collector node (C node). CODE translation methods have algorithms to “look for” the fan structure and then apply optimizations to this structure. These optimizations include avoid queuing data on the arc, avoid general firing rule checks for B nodes and C node, and avoid general routing rule evaluation for B nodes.

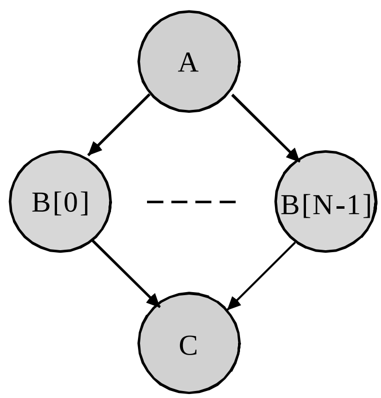


Fig. 8: Fan Structure

3.2 Standard Message Passing Interface

Message Passing Interface, MPI, is a proposed standard for explicit message passing on MIMD parallel computers [12]. MPI is a library, either C or Fortran, that contains a large number of message passing constructs and related methods. MPI supports point-to-point communication and collective communication across multiple process groups. A typical message includes the rank of the sender and receiver, a message tag, the size of the message, the datatype, the scope of the message, and the actual message content [13]. Communication can be blocking or non-blocking and a mixture of blocking and non-blocking communication protocols can be used.

The MPI library is massive. However, it has several methods that are useful in this context which will briefly be described here. As indicated earlier, MPI has blocking send and receive methods called **mpi_send**() and **mpi_recv**() [14]. These methods only complete when both the sender and receiver have exchanged data and are used in point-to-point communications. To synchronize execution of multiple processes, MPI has the barrier method called **mpi_barrier**(). This method blocks until all processes have entered this call. MPI also has collective communication methods. All collective routines are blocking in nature. The broadcast method, called **mpi_bcast**(), distributes common data to all processors in a group. The scatter method, called **mpi_scatter**(), divides array data into equal segments and distributes the segments among the processors.

4 Description of Beamformer Coefficient Computation Solutions

The specific solutions involving the use of CODE and MPI are presented here. The resulting structure of the code for a sequential implementation, CODE parallel implementation, and MPI parallel implementation is described. The sequential implementation provides a basis for comparison with respect to execution time and speedup to the parallel solutions.

4.1 Sequential Implementation of Beamformer Coefficient Computation

The full compliment of computations was written for the sequential algorithm. This sequential implementation provides a baseline for comparison to the parallel solutions and is briefly introduced here. A portion of the code listing is given in figure 9. First, the sound speed is converted from ft / sec into units needed by subsequent calculations, sec / inch. Then, a loop of 30 iterations is entered. The number of iterations is completely arbitrary and was chosen for performance comparison purposes only. After initializing the beam index and the total number of bytes calculated, a second "processing" loop is entered. This loop is based on the number of beam indices to be calculated. With each iteration, a call to the function, **worker_main**() is made, passing into the function the beam index and the sound speed in sec / inch. This function returns the number of bytes calculated. The total number of bytes calculated is computed. This total will be used in the final system to monitor the hardware and the communication systems.

```
void main(argc,argv)
int argc;
char *argv[];
{
  sound_speed = (1.0/(12.0 * sound_speed));
  for (j=0; j<30; j++) {
    beam_index = 0;
    total_calc = 0;
    for (i=0; i<NUM_BEAMS; i++) {
      num_calc = worker_main(beam_index, sound_speed);
      beam_index++;
      total_calc += num_calc;
    }
  }
  return;
}
```

Fig. 9: Sequential Implementation Source Code Segment

Code listing for **worker_main**() is given in figure 10. The function, **worker_main**(), is a sequential program that performs the bulk of the coefficient computations. In particular, the function is divided into two main sections. The first section is only executed if it is the first time **worker_main**() has been called. In this section, all of the system level constants are read from files. The system level constants include the array geometry (3-dimensional location on each element in the array), the focussing points in space in 3-dimensions, the FIR filter coefficients, and the weighting coefficients. The second section is executed each time the function is called. This section consists of a loop that is based on the number of focussing ranges for which coefficients sets are to be calculated. Once in the loop, it calls the function, **calc_tdelays**(), which computes the relative time delays across each element in the array to a point in space given by the current focussing range and beam index. Using these time delays, it then calls three different functions to compute all coefficients as described in section 2. In particular, the function, **calc_start_index**(), computes the starting sample number for each element. The function, **calc_focus_coef**(), computes the offsets into the FIR filter coefficient table necessary to induce the desired time delay, retrieves the appropriate set of coefficients from this table, and applies the weighting coefficients accordingly. The function, **calc_modu_coef**(), computes the real and imaginary modulation coefficients given the time delays and sampling frequency. After all coefficient sets have been computed for the given beam index, **worker_main**() returns the number of bytes calculated.

```
int worker_main(int beam_index, float sound_speed)
{
  if (isfirst_time == TRUE) {
    /* read array geometry, focus points, FIR filter coef
       from files */
  }
  for (rng_i=0; rng_i<NUM_RNG; rng_i++) {
    range = focus_loc.foc_seg[rng_i].distance;
    steer = focus_loc.foc_seg[rng_i].steerdist[beam_index];

    /* time delay calculations */
    fine_time_delay = calc_time_delays(sound_speed, sensor,
                                       steer, range);

    /* coefficient calculations */
    calc_start_index(fine_time_delay, &rng_set[rng_i]);
    calc_focus_coef(fine_time_delay, weights_ptr,
                    &rng_set[rng_i], coef_table_ptr);
    calc_modu_coef(fine_time_delay, &rng_set[rng_i]);
  }
  num_bytes_calc = NUM_RNG * sizeof(elem_set);
  return num_bytes_calc;
}
```

Fig. 10: Function worker_main() Source Code Segment

4.2 CODE 2.0 Beamformer Coefficient Computation Implementation

The CODE 2 graph that computes a full compliment of beamformer coefficients for all elements, all beams, and all ranges is shown in figure 11. This graph computes the coefficients by distributing blocks of beam indices in parallel. The arc annotations depict the parallelism in this graph. In particular, **doWorker** node is instantiated multiple times in the classic fan construct described earlier. The **initialize** node is the start node. It simply reads the sound speed and the number of processors. It translates the units of the sound speed from ft / sec to sec / inch for subsequent processing. It passes the number of processors onto the **start_worker** node and the **recordWork** node. In addition, it sends the converted sound speed to the **start_worker** node. The **start_worker** node, in general, sets processing parameters for all of the **doWorker** nodes and distributes these parameters accordingly. The **doWorker** nodes actually compute the coefficients and then pass the number of bytes computed to the **recordWork** node. This node simply collects and sums the total number of bytes computed by all **doWorker** nodes and maintains a count of the number of times the overall loop in the graph has been executed. If the graph loop has been executed less than 30 times, then it sends a message to the **start_worker** node to begin the next iteration. If the graph has executed 30 times, it sends a "finished" message to the **finalize** node. The **finalize** node simply writes a message to the screen and then terminates the program.

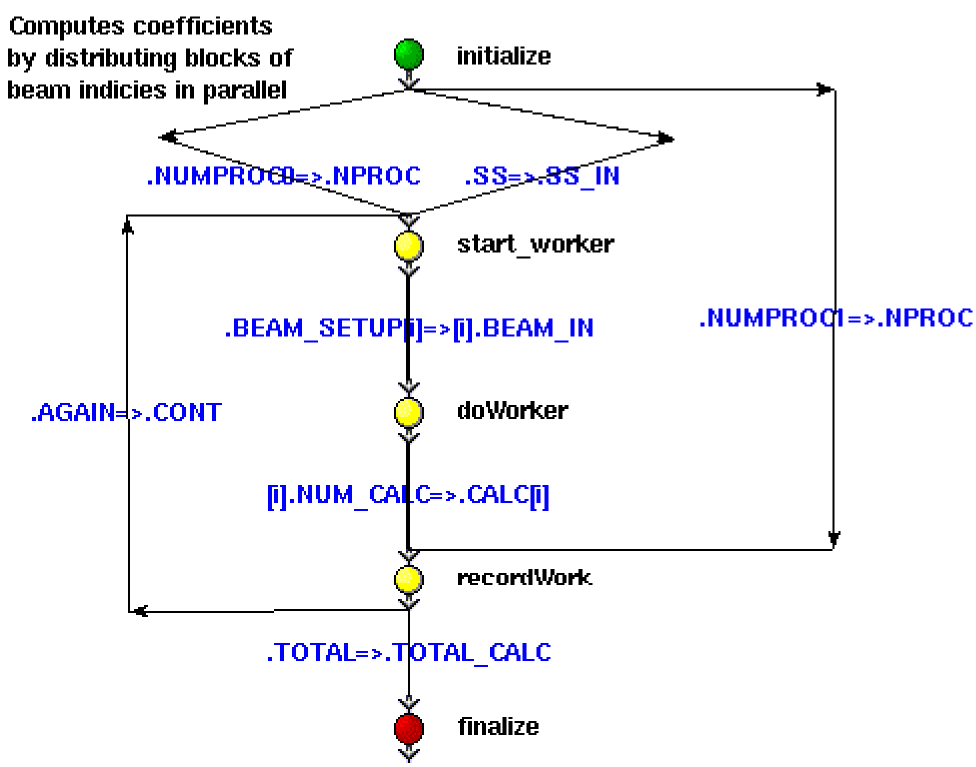


Fig. 11: CODE Parallel Coefficient Computation Solution

The details of the **start_worker** node are given in figure 12. As shown by the firing rules, the **start_worker** node can fire when it receives sound speed and number of processors from the **initialize** node or when it gets a continue message from **recordWork** node. Once it fires, it computes the number of beam indices that each **doWorker** node will compute coefficient sets for. It then constructs an array of structures that setup each **doWorker**. The structure includes the start beam index, which is the first beam index a particular **doWorker** node will compute coefficient sets, the number of coefficients to be computed, and the sound speed. At the completion of the execution, the **start_worker** node distributes the array of setup structures to each of the **doWorker** nodes. As described in the routing rule, the number of **doWorker** nodes that are instantiated is the number of processors sent from the **initialize** node.

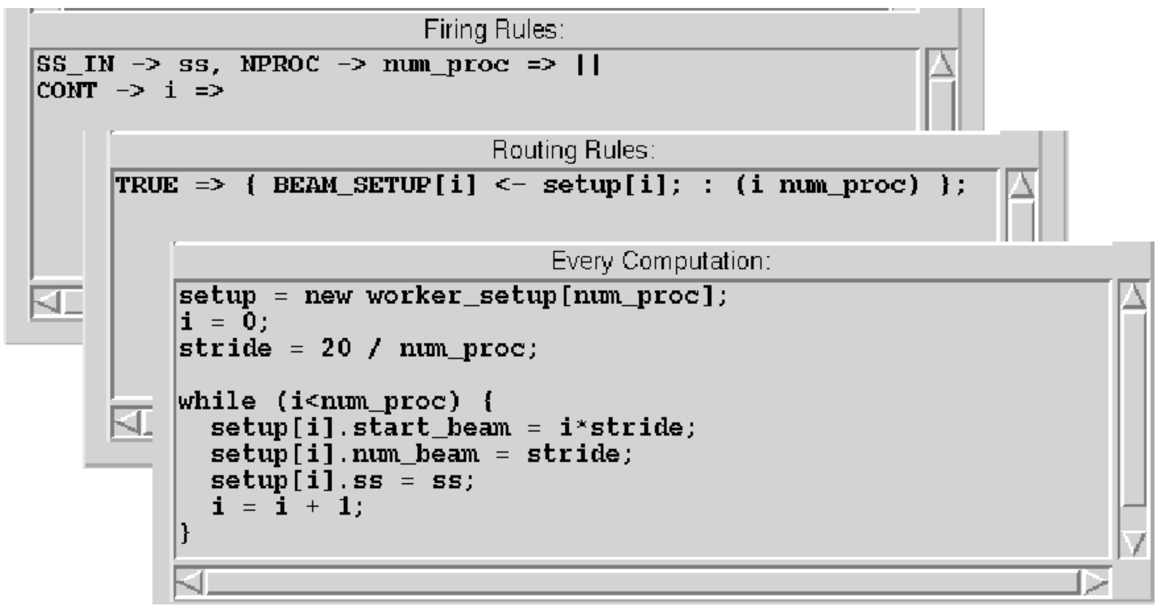


Fig. 12: start_worker Node Definition

The details of the **doWorker** node are shown in figure 13. The firing rule (not shown) simply states that the **doWorker** node can fire when setup structure appears at the only input node. Again, this is the behavior of the classic fan structure. Once the node fires, it enters a loop based on the number of beam indices to be computed specified in the structure. It calls **worker_main**() routine described above, passing in the first beam index, again specified in the structure, and the sound speed. It maintains a running sum of the number of bytes computed during each loop iteration. For each loop iteration, the beam index is incremented before calling **worker_main**() again. After all beamformer coefficient sets have been computed by the particular **doWorker** node, the node terminates execution by sending the number of bytes computed to the **recordWork** node.

The details of the **recordWork** node are shown in figure 14. As specified by the firing rules, this node can fire when it has received the number of processors from the

Every Computation:

```
i = 0;
part_sum = 0;
stride = setup.num_beam;
beam = setup.start_beam;

while (i<stride) {
  num_calc = worker_main(beam, setup.ss);
  part_sum = part_sum + num_calc;
  beam = beam + 1;
  i = i + 1;
}
```

Fig. 13: doWorker Node Definition

initialize node or when it has received summation data from all **doWorker** nodes. The second firing rule states that the number of ports created for this node is set by the number of number processors received by the **initialize** node. The first time the node fires is when it has received the number of processors from the initialize node. It does no computations and it does not output any data on any port. All subsequent times the node fires, it sums all the partial sums received by all of the **doWorker** nodes and stores this

Firing Rules:

```
NPROC -> num_proc => ||
{ CALC[i] -> calc[i] : (i num_proc) } =>
```

Routing Rules:

```
(count > 0) && (count < 30) => AGAIN <- i; &&
(count == 30) => TOTAL <- total;
```

Every Computation:

```
i = 0;
total = 0;

if (count >= 0) {
  while (i<num_proc) {
    total = total + calc[i];
    i = i + 1;
  }
}
count = count + 1;
```

Fig. 14: recordWork Node Definition

sum in the internal variable, total. It also maintains a count of the number of times the loop in the graph has been executed. If the graph has been executed less than 30 times, it sends a simple continue message to the **start_worker** node. As described above, this results in another full set of beamformer coefficients being computed. If the graph has executed 30 times, the **recordWork** node outputs the total number of bytes computed to the **finalize** node. As stated above, this node simply terminates execution of the graph.

4.3 MPI Beamformer Coefficient Computation Implementation

The MPI methods described in section 3 "naturally" construct the desired fan structure. Therefore, the implementation using MPI is very compact and straightforward. This is primarily due to the use of the scatter method. Here, the scatter method is used to divide a vector of beam indices into equal segments based on the number of available processors (figure 15). Then one group of beam indices is distributed to each processor to compute coefficient sets. This construct closely emulates the CODE graph.

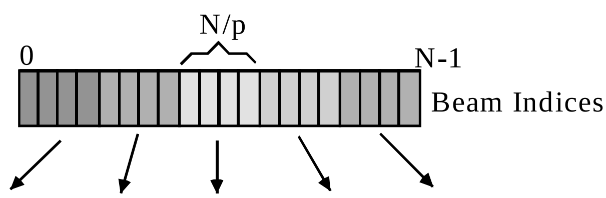


Fig. 15: Scattering Beam Indices

The MPI code listing is shown in figure 16. First, the number of beam indices each processor will compute coefficient sets for is calculated. Then the overall loop of 30 iterations is entered. The barrier method is used to ensure all processors are "ready" to start the next set of computations. Then the vector of beam indices is constructed. The sound speed is converted from ft / sec to sec / inch. This sound speed is then broadcast to all processors in the network. A group of beam indices is distributed to each processor using the scatter method. Each processor proceeds to compute their full compliment of coefficient sets, keeping a sum of the number of bytes computed. Up to this point in the execution of the program, all processors are executing the exact same code. At this point, however, all processors except the "root" processor must send their total number of bytes computed to the "root" processor and the "root" processor must receive each of these sums. Once the root processor has received all partial sums, it computes the total sum. This process continues for 30 times through the loop. After 30 iterations, the program terminates, calling the MPI finalize method as required by the standard.

5 Analysis of Results of Study

The CODE graph was compiled for SMP architecture. The resulting source code was compiled with the SUN C++ compiler under Solaris 2.6. The MPI implementation, as well as the sequential implementation, was compiled with the SUN C compiler, again under Solaris 2.6. In the case of the MPI implementation, the SUN High Performance Computing (SUNWhpc) package was used for its MPI library. This library is optimized for SUN multiprocessor workstations and servers. Thus, by using this optimized package

```
void main(argc,argv)
{
  stride = NUM_BEAMS/nproc;
  for (j=0; j<30; j++) {
    MPI_Barrier(MPI_COMM_WORLD);
    total_calc = 0, part_sum = 0, num_rcvd = 0;
    for (i=0; i<NUM_BEAMS; i++) {
      send_buf[i] = i;
    }

    sound_speed = (1.0/(12.0 * sound_speed));
    MPI_Bcast(&sound_speed, 1, MPI_FLOAT, root, MPI_COMM_WORLD);

    /* construct packet and start all workers */
    MPI_Scatter(send_buf, stride, MPI_INT, recv_buf,
                     stride, MPI_INT, root, MPI_COMM_WORLD);

    for (i=0; i<stride; i++) {
      beam_index = recv_buf[i];
      num_calc = worker_main(beam_index, sound_speed);
      part_sum += num_calc;
    }

    if (my_rank == 0) { /* master definition */
      total_calc += part_sum;

      /* get any data from workers */
      while (num_rcvd < (nproc - 1)) {
        MPI_Recv(&work_sum, 1, MPI_INT, MPI_ANY_SOURCE,
                     DATA_MSG, MPI_COMM_WORLD, &status);
        total_calc += work_sum;
        num_rcvd++;
      }
    } else {   /* worker definition */
      MPI_Send(&part_sum, 1, MPI_INT, root,
                     DATA_MSG, MPI_COMM_WORLD);
    }
  }
  MPI_Finalize();
  return;
}
```

Fig. 16: MPI Parallel Coefficient Computation Solution

and specifying the architecture during the compilation process as shared-memory machine, the performance comparison between CODE and MPI is better matched.

The sequential implementation execution time was measured as a performance metric. The execution times for the parallel CODE and MPI implementations were also measured and recorded. The number of processors specified in the parallel executions was 1, 2, 4, and 5. These numbers were chosen primarily as a requirement to maintain very good, if not ideal, load balance and secondarily as a system constraint for the number of available processors. All timing measurements were made on a SUN E5000, 8 processor server, running Solaris 2.6.

Figure 17 is a graph of the speedup for the CODE implementation and the MPI implementation. The ideal speedup is plotted for comparison purposes. Speedup is computed as the ratio of the sequential execution time to the parallel execution time. As shown on this graph, both CODE and MPI demonstrated linear speedups. However, CODE exhibited a more rapid increase in the linear rate than the MPI implementation. In addition, both CODE and MPI had slow down when one processor was specified. However, CODE nearly matched the execution speed of the sequential program. Thus, CODE execution times were less than MPI execution times regardless of the number of processors chosen.

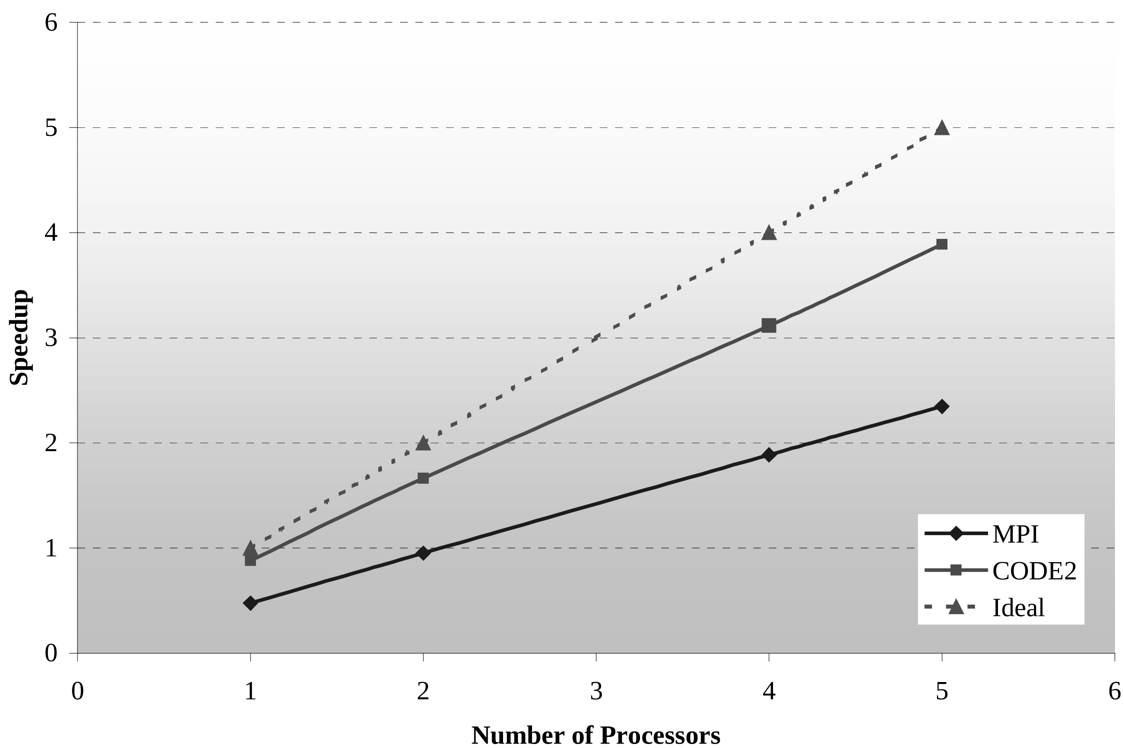


Fig. 17: Processing Speedup for Two Parallel Systems

5.1 Technical Comparison and Evaluation

The CODE parallel program outperformed the MPI parallel solution with respect to speedup and execution time. CODE produced a more efficient solution than MPI by applying optimization techniques described in section 3. First, CODE recognized the fan structure that dominated the overall structure of the graph. Thus, CODE reduced overhead by avoiding data queuing on the arcs, firing rule checks, and routing rule evaluations. Second, CODE translation process optimized for the particular architecture. The CODE parallel implementation was compiled for SMP architecture. The optimization techniques applied for this architecture include the use of light-weight threads employed by the runtime library and pointer passing from one process to another instead of message passing from one process to another. Third, CODE uses a non-preemptive scheduler that exploits the atomic nature of nodes to reduce overhead. Finally, CODE employs the notion of lazy creation and state retention to further reduce overhead. Each of these optimizations contributes in a positive way to produce a more efficient implementation to compute beamformer coefficient sets. This efficiency is of primary importance in a real-time system as is considered here.

Despite the fact that an optimal MPI library was used in the MPI solution, the execution times of MPI lagged those of the CODE solution. Because MPI is simply a library of standard message passing functions, MPI has no inherent optimization techniques. Any optimizations within the MPI code comes from the choice of message passing functions, the efficiency of those functions, and optimizations within the compiler itself. CODE, on the other hand, has many optimization techniques in its arsenal to produce efficient code. In addition, MPI uses processes to execute the work in parallel whereas CODE when compiled for SMP uses light-weight threads. It is well known that processes are expensive as far as execution time is concerned and that considerable savings can be realized by using threads. The result of all the above is simply faster code for the CODE implementation.

5.2 Code Development Experience

The code development processes using CODE and MPI were vastly different. The overall experience of code development of each of these environments will be compared and contrasted.

There are many advantages to developing parallel programs using a graphical programming system such as CODE rather than the traditional text based systems. These advantages quickly became apparent during the development phase. The graphical representation of the dataflow of the program, data dependencies, and control flow is a "natural" means of describing a program, especially a parallel program. Visual representation of the structure helps to identify parallelism and granularity in the overall program. In fact, the data decomposition of the beamformer coefficient computations was restructured early in the development phase of the program because the graph structure clearly showed the inherent bottlenecks and poor load balancing. With each decomposition refinement, the graphs quickly portrayed any intrinsic performance problems. This restructuring process greatly improved the performance of the final program. In addition, this process greatly influenced and accelerated the development process using traditional text based system. For instance, once this final structure was decided using CODE, the MPI implementation development was extremely easy to code and debug. Recall, this structure "fans out" the beam indices to all of the processes. Knowing this desired structure prompted the search for MPI methods to emulate this structure exactly. By following the structure of the graph, the MPI version was developed very quickly.

One attribute to any programming system that directly effects the readability and therefore, the maintainability, of the code is its support for functional flow. In other words, does a language require the use of for-loops to output information or can it "broadcast" the information to those that need it? The CODE firing and routing rule constructs along with the arc annotations allow information to be sent to any number of nodes without the use of for-loops. A single line of text can route a set of data to multiple nodes. MPI also has constructs to send information to multiple processes without the use of for-loops. These include all collective communication routines. The built-in communication structures provided by MPI produce code that is intuitive, easy to read, and can be compact.

It is much easier to produce an efficient solution with CODE than with MPI. During the development of the CODE implementation, there was no effort to optimize the design once the structure was decided. All optimizations are internal to CODE as described above. To produce an efficient solution using MPI requires considerable effort on the part of the programmer. The programmer must carefully choose the appropriate message-passing paradigm that guarantees performance without the possibility of deadlock. There is nothing to prevent the programmer from writing and building a program that will deadlock. In fact, it is quite easy to write a program that will deadlock using the multitude of methods offered by MPI.

CODE automatically and dynamically schedules all event executions. This makes code development much easier. It also allows dynamic runtime process and object creation. MPI is very static in nature. It is strictly a library. Therefore, all scheduling of events is through program control flow, which again can easily lead to deadlock.

Finally, CODE supports various standard parallel message passing protocols such as MPI, Parallel Virtual Machine (PVM), and SMP. CODE can run on various platforms as well. MPI is also supported on a multitude of platforms and can be run across these platforms.

In general, the development process using CODE was very straightforward and provided valuable insight into the structure of the program at each stage. The development process using MPI was also very straightforward. However, this experience was greatly enhanced by applying the knowledge gained through the CODE development.

References

1. R. A. Mucci, “A Comparison of Efficient Beamforming Algorithms”, *IEEE Trans. on Acoust., Speech, Signal Processing,* vol. ASSP-32, no. 3, pp. 548-557, June 1984.

2. R. G. Pridham and R. A. Mucci, “A Novel Approach to Digital Beamforming”, *Journal of Acoust. Soc. Amer.*, vol. 63, no. 2, pp. 425-434, February 1978.

3. D. C. Horvat, J. S. Bird, and M. M. Goulding, “True Time-Delay Bandpass Beamforming”, *IEEE Journal of Oceanic Engineering*, vol. 17, no. 2, pp. 185-192, April 1992.

4. N. Carriero and D. Gelernter, “How to Write Parallel Programs: A Guide to the Perplexed”, *ACM Computing Surveys*, vol. 21, no. 3, pp. 323-357, September 1989.

5. S. Ramaswamy, S. Sapatnekar, and P. Banerjee, “A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 11, pp. 1098-1116, November 1997.

6. R. Agrawal and H. V. Jagadish, “Partitioning Techniques for Large Grained Parallelism”, *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1627-1633, December 1988.

7. J. C. Browne and D. Banerjee, “Complete Parallelization of Computations: Integration of Data Partitioning and Functional Parallelism for Dynamic Data Structures”, *Intl Parallel Processing Symposium*, April 1996.

8. J. C. Browne, J. Dongarra, S. I. Hyder, K. Moore, and P. Newton, “Visual Programming and Debugging for Parallel Computing”, *IEEE Parallel and Distributed Technology*, vol. 3, no. 1, Spring 1995.

9. P. Newton, “A Graphical Retargetable Parallel Programming Environment and Its Efficient Implementation”, PhD thesis, Technical Report TR93-28, Dept. of Computer Sciences, University of Texas at Austin, December 1993.

10. P. Newton and S. Y. Khedekar, *CODE 2.0 User and Reference Manual*, March 1993. Available at www.cs.utexas.edu/users/code.

11. P. Newton and J. C. Browne, “The CODE 2.0 Graphical Parallel Programming Language”, *Proc. Of the Intl. Conf. on Supercomputing*, pp. 167-177, July 1992.

12. D. W. Walker, “MPI: From Fundamentals to Applications”, Oak Ridge National Laboratories, Oak Ridge, TN, April 1995. Available at www.epm.ornl.gov/~walker/mpi/papers/talk90min.ps.Z.

13. P. Pacheco, *Parallel Programming with MPI*, Morgan Kaufman Publishers, San Francisco, CA, 1997.

14. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, version 1.1, June 1995. Available by anonymous ftp from ftp.mcs.anl.gov.