
A Matlab-Based Parallel Computation Engine for Sound Localization

Steven J. Rennie
M.A.Sc Candidate
The Edward S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto

1 Introduction

This paper presents the design, implementation and performance analysis of a Matlab-based parallel computation engine for performing multi-speaker speech localization. Utilizing a collection of high performance personal computers connected by a high speed Local Area Network (LAN), the goal of our work is twofold:

- 1) To Develop a set of communication primitives that may be used to facilitate parallel programming in Matlab
- 2) To employ these communication primitives to the development of a Matlab-based parallel computation engine for performing multi-speaker speech localization.

This paper is organized as follows. Section 2 gives a brief view of some fundamentals of sound localization. Section 3 presents the Temporal Power Fusion (TPF) algorithm for multi-speaker speech localization that will be employed by our localization engine. Section 4 describes in detail the implementation of our sound localization engine. Section 5 presents the performance results obtained. In section 6 we close with a brief discussion of our results, and possible directions of future work.

2 Sound Localization Fundamentals

All localization techniques- acoustic or otherwise- are based fundamentally on the use of multiple diverse observations to constrain the location of the underlying target(s). In the case of sound localization systems, diversity in both the intensity and time of arrival of sound at a set of observation points(microphones) can potentially be used to locate the underlying acoustic source(s).

In the case of two microphones and one dominant sound source, for example, estimated knowledge of interaural level difference(ILD) or relative intensity of the two microphone readings gives information as to the relative distances of each microphone from the underlying sound source. If the environment is non-reverberant, the power of the acoustic signal will decay proportionally to inverse of the square distance from the source [1,2], and the ratio of the distance of each microphone from the sound source can reasonably estimated. Using a collection of microphones with known location and sufficient diversity in intensity collectively then, accurate estimation of the location of the sound source becomes plausible.

Alternatively, estimated knowledge of the time difference of arrival (TDOA) of the sound source at one microphone relative to the other- utilizing knowledge of the speed of sound- gives us information about the *difference* in the distance of each microphone from the underlying source. In two dimensions this difference defines a hyperbola on which the sound source is estimated to lie. In three dimensions, this difference defines a hyperboloid. Once again, using a collection of microphones with sufficient diversity (a sufficient number of pairwise non-zero TDOAs) accurate estimation of the location of the sound source becomes plausible.

Over the years several techniques built upon the use of ILD information, TDOA information or some combination thereof have been developed, and have demonstrated good success in localizing sound under certain conditions [1-3]. The next section introduces a TDOA-based speech localization algorithm known as Temporal Power Fusion (TPF), which stands at the leading edge of speech localization technology today, and is the basis of our sound localization engine.

3 The Algorithm: Temporal Power Fusion

TPF is a TDOA-based speech localization technique that is able to localize multiple speech sources simultaneously by taking advantage of the bursty nature of speech.

The TPF algorithm in short, consists of the following steps:

- 1) TDOA Calculation- Computing the most likely TDOA of each 10-20ms segment of microphone data as it becomes available, for all microphone pairings that yield useful TDOA information (within a meter of one another)
- 2) TDOA Histogramming- Updating a histogram of the last *WindowLength* TDOA estimations made, for each useful microphone pairing.
- 3) Spatial Likelihood Function (SLF) Construction- Assigning to each point in space a value proportional to the likelihood of a sound source being located there, based on the what TDOA maps to that point, the current value of the histogram for that TDOA, and the observability of the location, for each useful microphone pairing.
- 4) SLF Fusion- Integrating the SLFs obtained from each valid microphone pairing, to form an overall SLF.

Because TDOA computation is done over short segments and histogrammed, the algorithm is able to maintain the location of several speech sources simultaneously, utilizing the silence periods of the stronger source(s) to identify the TDOAs and hence the locations of the weaker ones.

Figure 1 illustrates a typical TPF-Built SLF for the case of 24 element microphone array applied to a room containing two active speech sources. For a detailed discussion of TPF and its relation to other current speech localization techniques, see [1].

4 The Task at Hand

Using a 24 element microphone array and the TPF algorithm for multi-source speech localization, once again our goal is to develop a Matlab-based parallel computation engine for performing multi-speaker speech localization that makes maximal use of the available computing resources- 12 high performance personal computers connected by a high-speed LAN- to achieve an SLF update rate as high as possible.

Each high performance computer is a Dell Optiplex 2.0 GHz machine with 1GHz of SDRAM, with 512KB and 8KB level 2 and level 1 caches, respectively. Each node runs Matlab version 6.1.0.450 Rel. 12.1 on Windows 2000 Professional Edition, service pack 3.

Acoustic data from all 24 microphones is acquired by a custom high speed data acquisition interface (DAQ) resident at the primary host, at a rate of 20kHz for each microphone in 20 ms chunks, which

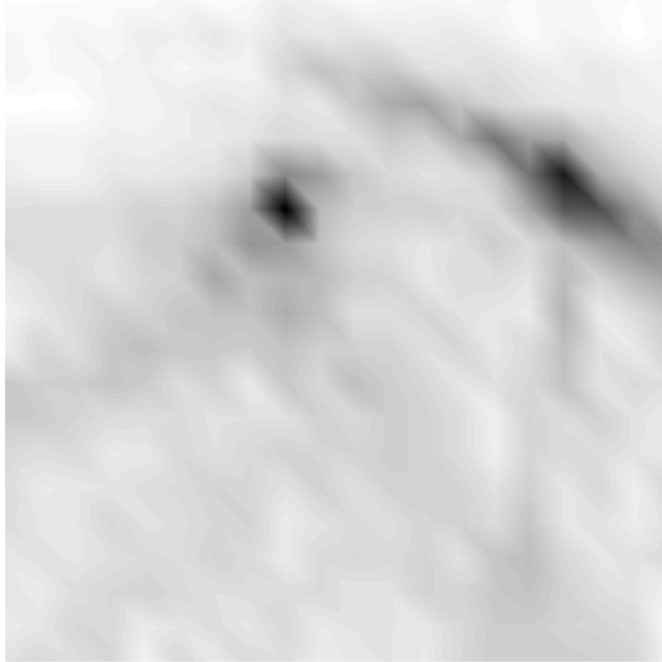


Figure 1: Example of a TPF-Built SLF for the case of 24 element microphone array applied to a room containing two active speech sources. The SLF peaks are taken as the estimated locations of the two speech sources. In this case the TPF peaks are within 0.15 meters of the actual source locations.

are processed immediately to update the overall SLF estimate. Ideally we would like our sound localization engine to operate at the rate of frame acquisition (20ms per cycle), but in reality all we really need get somewhere close.

As we shall see, with a sequential cycle time over over 6.2 seconds, we have a long way to go, but with some ingenuity, we get there.

5 Going for Speedup

5.1 Step 1: Implementation Optimization

The first step in our development of a parallel computation engine for speech localization is to optimize the existing sequential code. In doing so, our aim is to minimize the real-time workload by:

- 1) Eliminating unnecessary computation
- 2) Computing data-independent computation offline
- 3) Maximizing the temporal and spatial locality of operations where ever possible.

In doing so, our end result is a program that is a solid baseline for parallelization, and as we shall see shortly, takes the goal of near real-time SLF updates from seemingly far-fetched, to attainable through paralelization.

Figure 1(a) depicts the computation profile of the unoptimized uniprocessor version of our speech

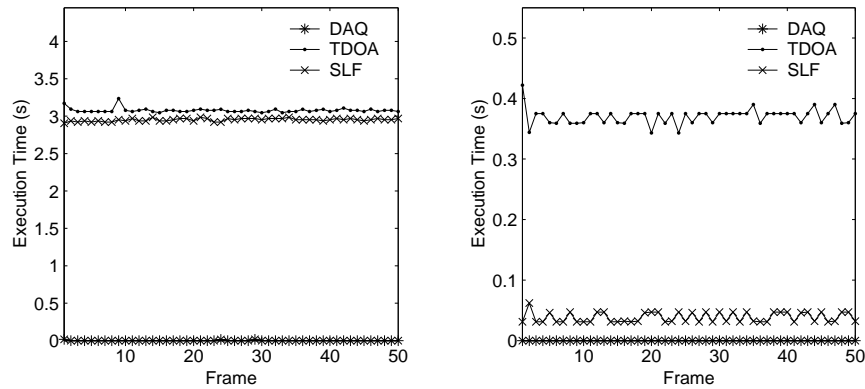


Figure 2: Computation profiles for the uni-processor version of our speech localization engine. (a) and (b) depict plots of the main computation components- data acquisition, tdoa histogram updating, and slf generation - before and after code optimization, respectively. Here via sequential code optimization we are able to increase the computation speed of the algorithm by more than 20 times, bringing the total computation time down to approximately 0.3 seconds.

localization engine. Each processing cycle consists of three main parts:

- 1) Microphone data acquisition
- 2) TDOA computation and histogram updating
- 3) SLF generation and fusion

From the plot, it is clear that the TDOA and SLF computations dominate the processing cycle, each taking approximately 3 seconds to compute. The total computation time is around 6.2 seconds, or over 300 times the ideal goal of a 20ms processing cycle. The situation looks grim, but fortunately upon analysis, several optimizations revealed themselves.

The first realization was that the SLF generation process is largely data independent, and thus can be computed offline. As described in section 2, the SLF generation process consists of two major steps: Mapping each discretized point to a TDOA for each employed microphone pairing, and then assigning the associated TDOA histogram derived likelihood to that point. The first step is data independent, and therefore can be implemented via a lookup table, computed offline.

The second major optimization discovered was that the TDOA computation- achieved through a brute force search of all possible TDOAs for the one yielding a maximal correlation between microphone signals for each microphone pair- was not range optimized. The physical distance between microphone pairs defines precisely the physically possible TDOAs, and therefore the TDOA range for every employable microphone pairing can be tabulating precisely.

Figure 1(b) depicts the computation profile of our optimized uniprocessor speech localization engine. As can be seen readily by comparing the results of Figure 1(a), the processing cycle time of both the TDOA and SLF computation components has been reduced dramatically. The TDOA computation cycle time has been reduced by almost 10 times to approximately 0.37 seconds, while the SLF computation time has been reduced by 100 times, to 0.03 seconds. The overall computation time is now approximately 0.41 seconds, or more than 15 times faster than our unoptimized computation engine.

It is worth mentioning that in addition to the fundamental algorithm-related optimizations mentioned earlier, every conceivable more 'standard' optimization technique was applied to fullest extent possible to achieve speedup.

Converting computation from loop based to matrix based wherever possible for example, decreased the computation cycle time substantially, owing to the fact that this gives the matlab interpreter the opportunity to optimize these operations through the exploitation of spatial and temporal data locality. Separating the TDOA and SLF computational loops over all microphone pairings actually decreased the computation time by almost 10 due to the exploitation of temporal locality.

6 Parallelizing our Sound Localization Engine

6.1 Decomposition and Assignment

The first step in parallelizing our sound localization engine is to decompose the computation cycle into logical units of work. Since the TDOA calculation/histogramming and SLF generation computations for any microphone pairing are independent of all other computation, we designate this as our logical unit of work.

Because our logical units of work are independent, interprocess communication is only required between the master processor and the slave processors to send the raw microphone data, and to retrieve the SLF results. No interprocess communication between slave processors is required.

To minimize interprocess communication then, we must assign microphone-pair workloads to each processor so as minimize the total amount of data overlap between processing slaves.

Since only microphone pairings of sufficient physical locality yield useful TDOA information (typically no greater than 1 meter for speech applications [1,2]) this is accomplished by maximizing the physical locality of microphone pairing assignments. This also automatically leads to balanced workloads, since the average microphone pair physical distance for each processor assignment is naturally maximally balanced (recall that doubling the distance between microphones in a pairing quadruples the TDOA space that has to be searched by brute force, and that TDOA computation dominates the computation cycle).

Figure 2 illustrates optimal microphone pair assignment for a nine element microphone array, for the case where only those microphone pairings within 0.4 meters of one another are considered in constructing the overall SLF. In this particular example four processors are employed, and we can see that the first three processors are perfectly load balanced, while the last processor is less loaded, both in terms of number of pairings assigned and average distance per pairing, due to the termination of the array. We take advantage of this natural occurrence by assigning this lighter workload to the master processor who, in addition to its assigned computation, must manage the distribution of data and collection of results.

Processor workload assignment for our twenty-four microphone sound localization engine is done in the exact same manner, except that for our particular case, there are twenty four microphones, all microphones within 1 meter of each other are considered as valid pairings, and the number of processors ranges from 2 to all 12 for a particular sound localization engine instance.

Figure 3 summarizes the basic structure of our sound localization engine. Before entering the data processing loop, the master processor initializes the data acquisition interfaces, computes the optimum overall workload assignment, and connects to and initializes all slave processors. Once inside the data processing loop, the master retrieves the raw microphone data, sends the data to each slave processor as appropriate, computes its portion of the workload, retrieves and fuses the SLF results returned by each slave, and saves to file, or displays the overall SLF.

6.2 Design Implementation

Because the computation associated with each unit of work (microphone pairing) is independent of all other work, implementing a parallel version of our sound localization engine is quite straightforward, with one major exception: communication primitive development.

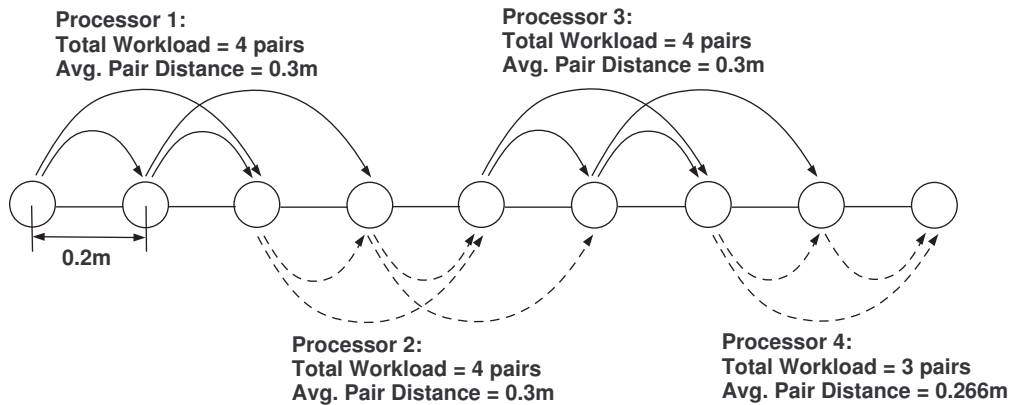


Figure 3: Optimal microphone pair assignment for a nine element microphone array, for the case of four available processors, and only those microphone pairings within 0.4 meters of one another considered in constructing the over SLF. The processors are maximally load balanced, with the exception of the fourth processor assignment, whose workload is conveniently assigned to the master processor who, in addition to it's assigned computation, must manage the distribution of data and collection of results.

Since both the data sent to initiate each processing cycle and the SLF contribution returned at the end of the cycle are of fixed size once specified during initialization, receipt of the expected amount of data can be used as synchronization for both initiating the processing cycle, and signaling

it's completion. Therefore there is no need for any synchronization primitives beyond a robust message passing mechanism.

So why is a message passing between computers on common LAN an issue? One word: Matlab. Our need for a message passing interface in Matlab (recall that the development of an interface for parallelizing Matlab code is one of the fundamental goals of this project) presents problems.

We searched thouroughly, and although the are people doing research in parallel matlab programming [3,4], there exists no readily available Matlab support for network communication between Matlab instances running on different machines- until now.

Matlab has an external interfaces API that can be used to bind programs written in other programming languages such as Fortran and C, so that they may executed within the Matlab environment just like a regular Matlab function.

Using the Matlab external interfaces API, a wrapper function that takes care of all the data and type conversion necessary to bind the desired program to Matlab interface (written native language of the program being connected to) must first be written. Then using the built-in matlab executable generation tool 'mex', a Matlab executable version of the program can be produced [5].

In Windows, this executable is implemented as a Windows dynamically linked library(DLL), which is loaded dynamically into Matlab process memory each time the function is called, with the wrapper function acting as the DLL entry point (from the programmer's perspective). This implementation is worthy of note since it means that compiled functionality often ends up executing MORE SLOWLY than equivalent interpreted matlab code. In attempting to optimize our sequential program for example, the mex tool was used to convert the TDOA computation function to C and then to a mex executable(Windows Dll). This made the execution loop of the optimized version of our sound localization 18 percent *slower*.

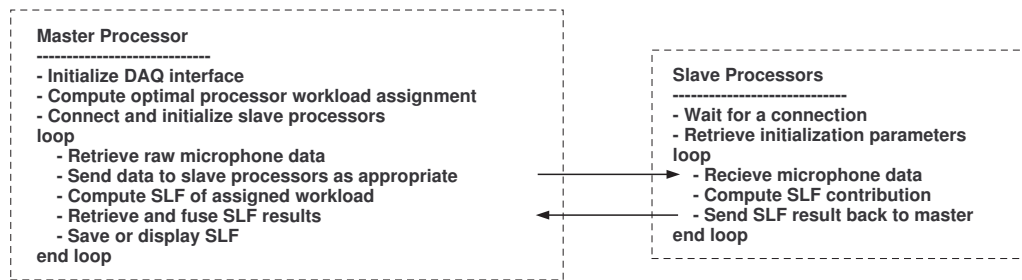


Figure 4: The basic structure of our sound localization engine

Nevertheless, the existence of the Matlab external interfaces API gives us a mechanism to potentially bind to existing network communication libraries such as the WinSock. The question is, is it possible?

Programming inside the Matlab interpreter imposes several restrictions. Multi-threaded programming is not supported, and neither is any dynamic memory allocation that does not go through the provided external interfaces API. The official stance of Mathworks is that use of the Windows API inside mex files is not supported- but you can try.

And so we did. Using the WinSock interface (for programming sockets in windows) a client/server network interface for matlab based on synchronous messaging was developed. Synchronous messaging was chosen for its simplicity in terms of matlab interface and implementation. A synopsis of each network communication function, from the point of view of the Matlab programmer, is given below; the implementation for these functions is given in the appendix.

Client Functions:

socketHandle = openconnection(string serverAddress)- creates a stream socket connects to the specified server

Server Functions:

socketHandle = createlister(string pcAddress)- creates a stream socket server connection
 acceptconnection(socketHandle) - accepts a connection request on the specified socket

Shared Functions:

senddata(socketHandle, matrix) - sends the specified matrix out on the specified socket in row major order
 recievedata(socketHandle, numElements) - retrieves the specified amount of data from the specified socket (use in conjunction with the the Matlab reshape command to receive matrices of arbitrary dimensions)
 closeconnection(socketHandle) - closes the specified socket connection

6.3 Performance

Figure 4(a) depicts a plot of the mean computation profile of our sound localization engine as a function of the number of processors. From the plot it is clear that we are able to reduce the overall processing cycle time substantially by adding additional processors. The total communication time associated with sending and receiving the data is seen to be stable over the processor range, increasing only marginally with the number of processors. The computation per processor conversely, falls as the number of processors is increased as expected, but only approximately proportional 1/N where N is the number of processors. Why does the benefit of adding additional processors die off

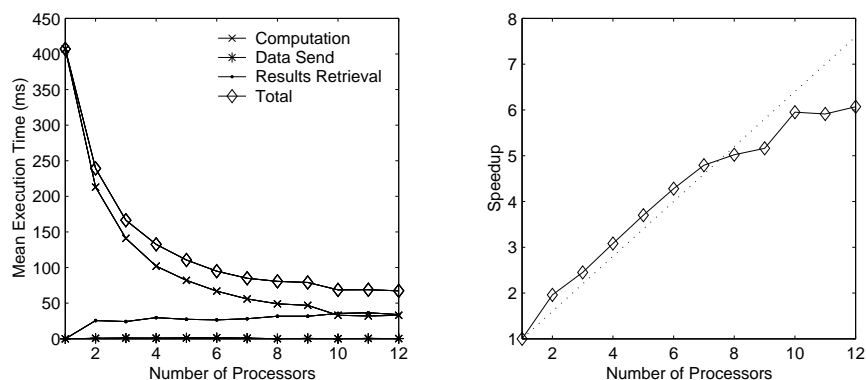


Figure 5: Assessing the performance of our sound localization engine. (a) depicts the computation profile of our sound localization engine as a function of the number of employed processors, and (b) the corresponding speedup curve.

so quickly?

This occurs because the processing time required for TDOA and SLF computation is pure computation dominated. Because the entire dataset for each processing cycle is small even compared the L2 cache on each processing node (400 data pts/mic x 24 mics*8 bytes/point = 75 KB versus an L2 cache size of 512KB), parallelizing the program yields virtually no data locality benefit during computation, since even with one processor, the entire primary working set of the program can easily fit in the L2 cache.

As a result, the computational benefit falls off directly proportionally to the size of the per processor workload, which decreases proportionally to the inverse of the number of processors, or $1/N$. Looking at the Figure 4(a) however, we can see one notable exception. A substantial decrease in average computation time- comparable to that observed when going from 4 to 5 processors- occurs when going from nine to ten processors. This occurs presumably because when we transition to ten processors, the per-processor working set (7.5KB) is able to fit entirely in the L1 cache of each processing node (8KB). As a result the decrease in cycle computation time in going from 9 to 10 processors is over 15ms, or more than 3 times what would be predicted by $1/N$ proportionality. Just a small example of the power of increasing data locality through parallelization.

Figure 4(b) depicts a plot of overall speedup obtained versus number of processors employed in our sound localization engine. The dotted line represents linear speedup at 0.6 units per processor. Once again, we see a dramatic increase in speedup between 9 and 10 processors, corresponding to the threshold where the entire primary working set of each processor is able to fit in the L1 cache. Using all twelve available processors, we are able to obtain a speedup of just over six.

6.4 Discussion

We have succeeded in developing a set of primitives for facilitating blocked message passing-based parallel programming in Matlab, and through sequential code optimization and the use of these primitives, we have succeeded in reducing the average SLF update rate from 6.2 seconds to 75 milliseconds.

The upgraded speed of the computation engine can be and will be used to facilitate the research and development of online algorithms for sound localization in the University of Toronto's Artificial Perception Lab(APL). The developed Matlab sockets interface will be made available on the APL webpage shortly for anyone and everyone who wants to make thier Matlab simulation go faster. All and all we achieved everything that we had hoped for, but of course as always, more can be done.

A matlab sockets interface supporting asynchronous message passing would be a worthwhile endeavour. As we saw in the results obtained for our sound localization engine, as the number of processors increases, so does the potential for large spikes in the results retrieval time. Due to the synchronous (and hence sequential) collection of the results from the processing slaves, a delay in the retrieval of one set of results slows down the entire process- this can be minimized with an interface that supports asynchronous socket handling in Matlab.

References

[1] P.Aarabi, Robust multi-source sound localization using temporal power fusion, in: Proceedings of Sensor Fusion: Architectures, Algorithms, and Applications V (AeroSense'01), Orlando, FL, April 2001.

[2] P.Aarabi, S. Zaky, Robust sound localization using multi-source audiovisual information fusion, in: Information Fusion 2(2001),209-233.

[3] Cornell MultiTask Toolbox for Matlab:

<http://www.tc.cornell.edu/Services/Software/CMTM/>

[4] Otter Compiler- translates matlab into C programs targeting architectures that support MPI, Oregon State University :

<http://www.cs.orst.edu/~quinn/ai-parallel-quinn.html>

[5] Matlab External Interfaces API :

http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/matlab_external.shtml