Issues in using Heterogeneous HPC Systems for Embedded Real Time Signal Processing Applications*

Prashanth B. Bhat Young W. Lim† Viktor K. Prasanna‡

Department of EE-Systems, EEB-244
University of Southern California
Los Angeles, CA 90089-2562
http://www.usc.edu/dept/ceng/prasanna/home.html
{prabhat + lim + prasanna}@halcyon.usc.edu

Abstract

Embedded signal processing systems have traditionally been built using custom VLSI to meet real-time requirements. This leads to limited programmability and restricted flexibility. With recent technological advances in high performance computing, scalable systems based on heterogeneous "off the shelf" modules are attractive as computing platforms in real-time embedded environments, leading to an emerging class of Scalable Heterogeneous High Performance Embedded (SHHiPE) systems. These systems offer advantages of low-cost, scalability, easy programmability, software portability, and the ability to incorporate evolving hardware technology. In order to satisfy the timing and predictability requirements that arise in embedded environments, several issues must be considered. These issues arise at the hardware level - such as choice of processing element architecture, and also at the software level - issues related to operating system and communication libraries. We propose an integrated methodology to develop efficient parallel solutions for signal processing applications on the SHHiPE platforms. Our approach is to develop scalable portable algorithms based on accurate computational models of the hardware platforms. We present preliminary performance results of such an approach applied to a Radar signal processing problem.

*The implementations were performed on the SP-2 at the Maui High Performance Computing Center. This research is sponsored in part by the Phillips Laboratory, Air Force Material Command, USAF, under cooperative agreement number F29601-93-2-0001.

†Student author supported by a scholarship from Hyundai Corporation.

‡Work supported by NSF under grant CCR-9317301 and in part by ARPA under contract no. DABT63-95-C-0092.

1 Introduction

Embedded systems in real-time environments have traditionally been based on custom VLSI. Examples include single-board computers(SBCs), custom-built boards, and attached processors in Radar, Sonar and other signal processing applications. Although these systems satisfy embedded system requirements of small size, light weight, etc., the flexibility offered by them is highly restricted. Timing requirements are incorporated into the hardware design phase, and critical regions of software are programmed using a low-level language. Hence, porting solutions across different application environments often requires significant design modifications. Further, the application-specific nature of the developed hardware leads to higher overall system costs.

From the application perspective, the demand for system flexibility and higher computing power points to a different approach to embedded system design. The signal processing area has evolved significantly in recent years. New theories have been developed in statistical modeling, estimation methods, transformation theory, among others. Many Radar applications such as Space Time Adaptive Processing (STAP) require performance levels of teraflops [20]. Currently, SBCs and other custom VLSI based embedded systems do not provide adequate processing power to perform these computations in real-time.

Recent advances in microsystems research are making it extremely attractive to utilize Commercial-Off-The-Shelf(COTS) components in embedded systems [2]. Currently, commercial High Performance Computing(HPC) systems are developed by integrating COTS processor and interconnect components in a 'plug-and-play' fashion. Most COTS processors offer over 1 Gops and 400 MFlops performance. For

0-8186-7106-8/95 $04.00 © 1995 IEEE

message transfer between the processors, the typical startup times are in the range of tens of μsec and the transfer time is in the range of a few hundredths of a μsec/byte. With advanced implementation and systems packaging technologies, it is feasible to manufacture low power, light weight, and small sized versions of these components for embedded systems. Embedded systems can easily be designed using these components as building blocks. We refer to such systems as Scalable Heterogeneous High Performance Embedded(SHHiPE) systems.

SHHiPE systems have a similar high level architecture consisting of three main components: (a)**A heterogeneous collection of computing nodes,** each consisting of COTS processors (DSPs, general purpose processors) and local memory, (b) **An intelligent network interface** consisting of processors and support circuitry that performs message handling and DMA access to the local memory, and (c) **A low latency high bandwidth message passing network.**

In comparison with the custom VLSI based systems, SHHiPE systems offer advantages of architectural flexibility, software programmability, and portability. Systems can be easily scaled in size and modified in architecture to suit application requirements. Algorithms can be specified using high level programming languages. This feature allows for easy and architecture independent programming. New algorithms can also be incorporated into the design relatively easily. This leads to increased system life-cycle time, and thereby lower equipment costs. Figure 1 compares the key features of custom VLSI and SHHiPE technologies.

Currently, there are many development efforts directed towards building prototype SHHiPE systems. Figure 2 shows a typical system. An example is Martin Marietta (Lockheed Martin) Labs' High Performance Scalable Computing System (HPSCS) [6]. These systems provide features such as low latency messaging, support for user defined communication primitives, and support for light weight protocols. These features enable efficient implementation of message passing standards such as MPI [11]. These systems also provide hardware support for asynchronous communication, techniques to hide network latency, support for integrating message passing with shared memory, and for distributed computation using object oriented languages [6].

In our opinion, the SHHiPE systems signify a shift in the design methodology of embedded systems. With the ready availability of COTS components, the hardware design phase is considerably simplified, in comparison with the custom VLSI approach. Instead, the emphasis is on development of efficient software. Scalable and portable algorithms are a critical component in achieving real-time performance on these systems. Scalable algorithms can realize increasing speed-ups as larger systems are utilized. Therefore, real-time performance requirements can be met by suitably scaling up the hardware. To ensure portability, algorithms can be specified in a hardware independent fashion using standard software libraries such as MPI. The functionality to be supported by communication libraries for real-time performance is still an open issue, and is the focus of the ongoing real-time MPI standardization efforts [12].

	Custom VLSI	SHHiPE
Computation	Fine grain	Coarse grain
Operation Type	Synchronous	Asynchronous
Hardware	Custom VLSI Technology	COTS components
Communication	Systolic: very little overhead($\sim$ *nsec*)	General communication: large overheads($\sim$ μ*sec*)
Algorithm Design	Built into hardware	Parallel algorithms to be designed
Flexibility	Inflexible	Highly flexible
Programmability	Limited	Highly programmable
Scalability	Limited	Highly scalable
Portability	Limited	Highly Portable

Figure 1: **Comparison between custom VLSI and SHHiPE technologies**

The rest of the paper is organized as follows – Section 2 discusses the issues in achieving real-time performance on SHHiPE systems. Section 3 describes our algorithmic approach and techniques for designing scalable solutions. Section 4 considers the computational characteristics of STAP, and also presents our preliminary performance results in developing scalable and portable STAP algorithms for SHHiPE systems. Section 5 is the concluding section.

2 Requirements and Issues

Considerable research work in real-time hardware and systems software has focused on ensuring correct timing operation of real-time systems. Much of this effort is specific to custom VLSI based systems. The use of SHHiPE systems brings up several issues that must be addressed to meet application requirements.

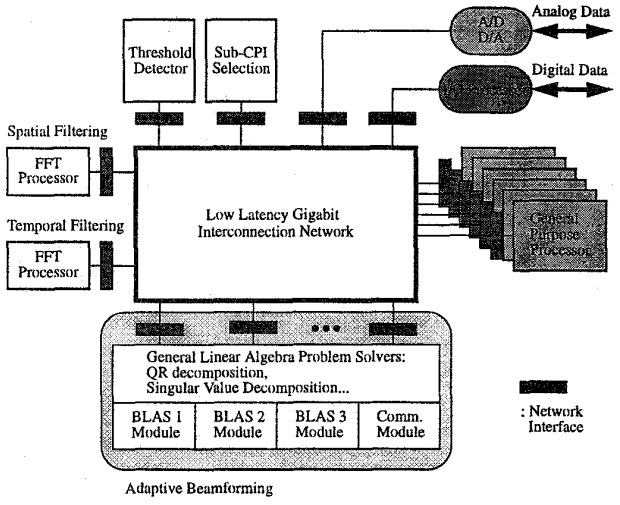

Figure 2: **An example SHHiPE architecture for Automatic Target Recognition in a typical airborne application**

2.1 Application Requirements

The most important requirement from the application perspective is that tasks must meet specified timing deadlines. *Predictability* refers to the ability of the system to accurately estimate the execution time of tasks, and schedule them so that all timing constraints are satisfied [17]. Typically, estimates are made of *worst case execution times*. Many application environments also require support for task *priorities*. Appropriate *task scheduling strategies* that take these deadlines and task priorities into consideration are necessary for correct system operation. There is also need for communication protocols that can guarantee certain minimum bandwidth and buffer space, and thereby assure *bounded message communication delays*.

Many of the embedded systems are safety-critical, and must therefore continue to operate in the presence of faults. *Fault tolerance* techniques must encompass both hardware and software, and must be integrated with the timing requirements [18].

Typical Radar and Sonar applications involve assimilating and processing large quantities of data. A fast I/O subsystem with *scalable I/O bandwidth* is needed. *Size constraints* are also encountered in embedded system environments. Memory systems are therefore constrained to have small sizes.

For cost-effective operation of the system, high levels of processor *utilization* and *throughput* must be sustained.

2.2 System Design Issues

Various issues arise in satisfying real-time application requirements on SHHiPE systems. The use of general purpose commercial processors in real-time processing gives rise to several sources of unpredictability. For example, the presence of a cache memory could lead to improved *average* memory access performance, but it introduces a source of uncertainty in the execution time. The difference in execution time between a cache hit and a cache miss is often a factor of 10. Accurately predicting the worst case execution time is therefore complicated in such situations. Similar uncertainties arise due to page faults in virtual memory management [18, 15].

Other sources of uncertainty in execution time are external interrupts, DMA transfers, and operating system overheads. DMA transfers result in bus arbitration overheads, even in cycle-stealing mode [7]. Operating system overheads consist mainly of context switching times in multitasking operating systems.

An architectural solution to address this unpredictability is to use separate processors for application tasks and for system related tasks. The system processors perform task scheduling, handle interrupts, etc. The application processors are dedicated to perform user computations, and therefore can have predictable execution times. Such a solution is adopted in the SpringNet architecture [18] and also in Myricom's 2-level multicomputer [13]. Each node in the 2-level multicomputer employs a "primary" processor for message handling and other runtime tasks, and one or more "secondary" processors for performing user computations. The secondary processors are not burdened by a runtime system or operating system. Instead, tasks are scheduled onto them by the runtime system executed by the primary (network) processor. The secondary processors can be simple, low-cost, performance oriented devices such as DSP chips, without cache memories. Since the secondary processors can be dedicated for the currently assigned task, predictability is enhanced.

The requirements imposed by real-time applications suggest enhancements to general purpose processor architectures. For example, consider an enhancement which allows the processor to lock a line in the cache. With this feature, once a line is loaded into cache, it will not be replaced until explicitly allowed. This is particularly useful in instruction caches, wherein a code, if small enough, can be loaded into cache entirely, and then locked. Since real-time application programs often tend to be small, this feature can provide significant benefits. The issue lies in de-

ciding what enhancements can be incorporated without sacrificing the generality of the processor architecture. The low cost of COTS processors is primarily due to economies of scale, since the same processors are used in many other applications.

Another issue is the partitioning of critical system functionality into software and hardware. Tradeoffs must often be made between system complexity and achievable performance benefits. For example, providing efficient and predictable communication support in point-to-point message-passing systems is crucial. By migrating communication functions to special-purpose hardware, it is possible to reduce the communication overheads incurred by the processor while simultaneously increasing communication performance. On the other hand, this functionality can also be provided by software communication scheduling schemes [5] at a lower hardware cost. Similarly, hardware support for scheduling can be provided in the form of a co-processor. The Spring scheduling co-processor is an example of such hardware support [14].

On the software side, the development of high level programs for the SHHiPE systems requires programming language support. An object oriented programming language allows for direct mapping of the problem domain semantics onto the programming model. This feature allows for easy representation of complex interactions between the various entities in a typical real-time application. Programming language functionality should include provisions for specifying timing constraints. Extensions to popular high level languages that incorporate this feature would be convenient, rather than a completely new language [3]. The partitioning of functionality between the programming language and a standard software library is also an issue to be considered.

The definition of a standard software library enables development of application code that can be easily ported across different computing platforms. In the HPC domain, the MPI standard has been recently defined for this purpose [11]. However, existing techniques for programming real-time embedded processors are inadequate as a basis for a standard. Real-time MPI is currently in its early stages of formalization [12]. The functionality supported by a real-time software library should allow the application programmer to exploit features available at the hardware level, such as priority, preemption, etc. Since the features provided by different hardware platforms vary widely, the functionality to be included in a standard library is a point of debate. Operations supported by the standard library must provide efficient and predictable performance, even for operations with irregular computational profiles. For example, [19] presents an efficient algorithm for many-to-many personalized communication with message length variance.

The heterogeneity among the computing nodes in SHHiPE systems raises issues of task *partitioning, mapping, and scheduling.* Each of the computing nodes efficiently supports tasks with a particular kind of processing requirement. The application algorithms must first be partitioned into subtasks that each have a homogeneous computational structure. Multiple subtasks could then be combined into larger modules. These program modules must be mapped onto the computing nodes that best suit their processing requirements. Suitable scheduling strategies are necessary to share the computing resources among the program modules so that timing constraints specified by the application are satisfied. These scheduling algorithms must also focus on achieving high system throughput and utilization. Further, the heterogeneous nodes could have different data input/output formats. Data originating from one node must be efficiently converted into the format of the receiving node [8].

The partitioning and mapping of the subtasks can be done statically, while the scheduling must be typically done online by the operating system. Microkernel based operating systems are necessary to provide low overhead context switching and interrupt latency characteristics. A microkernel operating system consists of a relatively small kernel providing minimal services, surrounded by a set of processes providing higher level operating system services such as file and device I/O and networking. The POSIX 1003.4 real-time operating system standard specifies functionality for prioritized process scheduling, high-resolution timer control, and enhanced IPC primitives, among others [4].

All the processors within a SHHiPE system must have a common view of time. Events must be time-stamped, so that deadline based scheduling can be performed. The use of a single central clock leads to low fault tolerance. With multiple clocks, synchronization between the clocks is necessary to avoid clock drifts [18].

3 A Unified Approach to Developing Efficient Parallel Solutions

From the discussion in Section 2, it is clear that the design approach on the SHHiPE systems should incorporate many issues not addressed in custom VLSI based design. As shown in Figure 3, we propose an integrated framework to develop efficient solutions on

the SHHiPE systems. The framework considers hardware aspects of the systems, as well as parallel algorithmic techniques to efficiently exploit the hardware features.

3.1 The Computational Model

An important component of our approach is a realistic computational model of the SHHiPE systems. The model enables us to choose appropriate data partitioning schemes, and to schedule computational subtasks. It also allows us to analyze the computation and communication overheads, and hence the scalability of a particular algorithm-architecture pair.

Most SHHiPE systems have a uniform high-level architecture as described in Section 1. The communication operation in such a system can be modeled with two parameters. One is *startup time* which occurs in every message transfer. This includes software and communication protocol processing overheads. The other is *unit transmission time* which is the cost of transferring a message of unit length. Let T_d denote the startup time and τ_d denote the transmission time. Then, sending a message containing m units of data from a module to another module takes $T_d + m\tau_d$ time. This is the case when "blocking" send is used and no special hardware for communication is available. During $T_d + m\tau_d$ time, the sending processor is busy handling the communication. However, if there is a communication processor and nonblocking send is used, the sending processor has to spend only the startup time to activate the communication processor. After this initial startup time, the processor is free and can continue its computations, while actual transfer of messages is handled by the communication processor. Such a model has been used for algorithm design on homogeneous distributed memory parallel systems in [19]. It has been called the General Purpose Distributed Memory(GDM) model.

The GDM model accurately represents the communication features of SHHiPE architectures. The model also represents features for overlapping computation with communication and latency hiding(for example, this feature represents the DMA access capability of the LANai network interface chip in Myricom systems [1]). Since many SHHiPE platforms are under development, architecture independent algorithms and portability of the developed code are very important. We specify our algorithms using the MPI standard [11] to ensure portability of our code.

3.2 Data Layout and Remapping

In general, the data layout problem is to design a distribution of data among the processors such that the overall communication time is minimized, and the workload is balanced. In typical signal processing applications, the data is accessed and processed in different ways in each subproblem. The goal of data remapping is to rearrange the data between the steps of the algorithm so as to reduce the communication time. In some scenarios, the overhead incurred in performing the remapping operation might be larger than the benefit due to remapping. To estimate the tradeoffs, we use the model to evaluate the benefits and overheads due to data remapping.

3.3 Mapping and Scheduling of Computations

Most signal processing problems have regular and known computation and communication characteristics. This permits mapping and scheduling of computations to be performed at compile time. Indeed, this feature has been exploited in systolic arrays to transform the computations into a form that can be executed with localized communication. The scheduling problem for SHHiPE platforms is different from that of custom VLSI systems on account of the larger granularity of the computations, heterogeneity of the architecture, large overheads in performing communication, and the capability to overlap computation with communication.

In the next section, we demonstrate the application of our techniques, using a generic problem in Space Time Adaptive Processing (STAP) as an example [10, 9].

4 An Example Application: STAP

STAP techniques are needed in many applications such as Automatic Target Recognition, Airborne Surveillance Systems, etc. STAP is a multidimensional adaptive filtering process, performing computations over spatial samples from the elements of an antenna array and temporal samples from multiple pulses of a coherent waveform [20]. Due to the computationally intensive nature of the application, tradeoffs between accuracy and real-time constraints must be made. Higher-Order Post-Doppler processing (HOPD) is one possible method for maintaining performance while reducing computational complexity [20].

4.1 Computational Characteristics

Typically, HOPD processing proceeds as follows [16]: The input data can be conceptually represented as a three-dimensional data cube as shown in Figure 4(a). The dimensions represent the delay M, the element N, and the rangegate L. A cross section of the data cube orthogonal to the rangegate dimension is called a snapshot. Typical values of these parameters

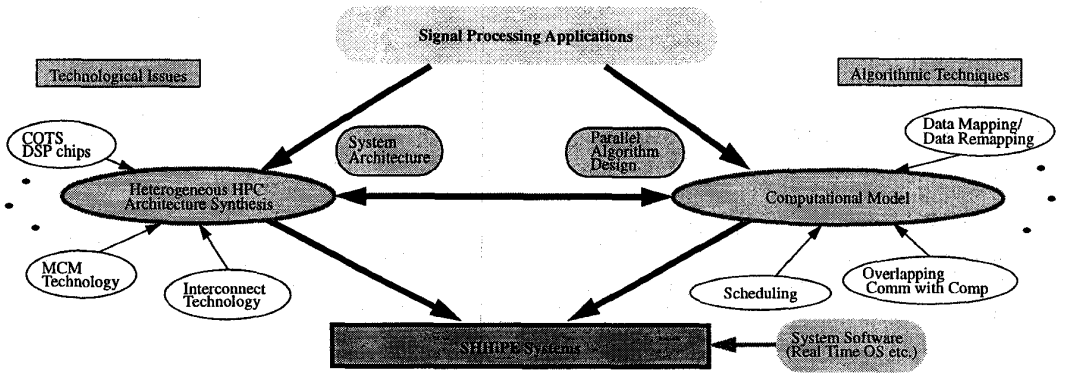

Figure 3: **An integrated framework for the design of efficient parallel solutions**

are shown in Figure 4(a). The data cube is first transformed to the Doppler domain, which requires NL M-point FFT's. Next, the transformed data cube is divided into least squares problems: The data cube is divided into M slabs along the rangegate dimension, which are referred to as Doppler bins as shown in Figure 4(b). There are $(M-2)$ problems, where for each problem i, the problem matrix is obtained by concatenating the bins $i-1, i$ and $i+1$. The resulting matrix has a size of $L \times 3N$. Each matrix is divided into two parts, along the rangegate dimension. The first part of size $L_{ls} \times 3N$ defines the least square problem that is solved by QR decomposition; the solution of the least square problem (or weight vector) is applied to the remaining part $(L_{wa} \times 3N)$. L_{ls} will typically vary from $2(3N) = 6N$ to $5(3N) = 15N$[1].

The primary operations are FFT and QR decomposition. The sequential m-point FFT problem requires $m \log m$ complex multiplications and additions. Sequential QR decomposition of a $m \times n$ matrix needs $n^2(m - n/3)$ floating point operations. We estimate the computational requirement of HOPD STAP as 3.24 GFlops, assuming that the entire process should be done in less than 0.5 seconds for real-time computation.

4.2 Scalable Portable Algorithm

We illustrate our integrated approach with a scalable solution for HOPD STAP. Results are presented from an IBM SP-2 implementation of the scalable algorithm. While our final target platform is a SHHiPE system such as Martin Marietta's HPSCS [6], a prototype of such a system is not currently available. Since we analyze the scalability of the algorithm using the GDM model, similar scalable performance can be expected to be achieved on the SHHiPE platforms.

[1] $L = L_{ls} + L_{wa}$

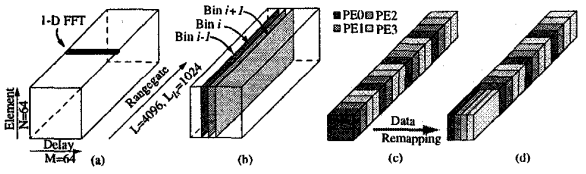

Figure 4: **(a) Doppler Processing (b) Bins (c) Snapshot mapping (d) Rangegate mapping**

The above STAP computation contains NL FFT problems and $(M-2)$ QR decomposition problems. We develop a scalable algorithm that exploits this high degree of parallelism. The algorithm incorporates efficient data distribution and data remapping schemes that significantly reduce interprocessor communication overhead. Let p be the number of Processing Elements(PEs). In the initial stage of our algorithm, the data cube is partitioned along the rangegate dimension into slabs, each slab containing L_{ls}/p snapshots. The slabs are distributed among the p PEs in a cyclic way. We refer to this as snapshot mapping. By this mapping scheme, the FFT computations need no interprocessor communication because all the point data is contained within a PE. The other mapping is shown in Figure 4(d), which we name as rangegate mapping. Here, only the L_{ls} portion of the data cube is partitioned along the delay dimension into p blocks. Note that only the L_{ls} portion of the data cube is used for finding weight vectors. Thus, this part is block mapped along the rangegate dimension. There

are $(M/p-2)$ QR decomposition problems which can be solved without any communication and two more problems to be solved by communicating between adjacent processors.

During the remapping process, every processor distributes its data to all other PEs. This is a collective communication and can be converted into $(p-1)$ all-to-all personalized communications (i.e., perfect permutations) by a simple scheduling. This exploits the fact that many point-to-point communications can be done in parallel. As another nice feature in this algorithm, it is possible to overlap all the communication with computation.

Using asynchronous nonblocking communication primitives (such as `MPI_Isend`, `MPI_Irecv`, `MPI_Wait` in MPI), we can overlap the remapping communication with the remaining FFT computations, (which are performed over L_{wa} part of the data cube), and the subsequent QR decomposition computations. Upon receiving at least three consecutive slabs from the remapping process, one QR decomposition problem can be solved by using the sequential algorithm. Thus, there are no more communication overheads in solving this QR decomposition problem.

4.3 Scalability Analysis

In our approach, the remapping process consists of $(p-1)$ permutations. During a permutation, each PE sends and receives $\frac{L_{ls}MN}{p^2}$ size data simultaneously. Therefore, the communication time for the remapping process without overlapping computation is:

$$T_{remap} = (p-1) \cdot \{T_d + \tau_d \cdot \frac{L_{ls}MN}{p^2}\}$$

For the values taken by p, L_{ls}, M, and N in typical STAP applications, $T_d(p-1) << \tau_d \cdot \frac{L_{ls}MN}{p}$. Then, T_{remap} can be approximated by $\tau_d \cdot \frac{LMN}{4p}$.

It has been reported that the row wrap mapping can be used to avoid the data redistribution [16]. With this scheme, FFT is performed without interprocessor communication, and QR decomposition is done in parallel. However, parallel QR decomposition comprises most of the communication time as well as the computation. Furthermore, the communication time increases rapidly as the number of processors is increased. For example, when a 32 node SP-1 is employed, the communication time is more than 50% of the total execution time.

In contrast, the communication time of our algorithm is inversely proportional to p. If more processors are used in the range $(p \leq M)$, the communication time becomes less. Additional data communication is necessary for the boundary data. This can be done either by two permutation operations or by sending redundant data during remapping. The boundary data is used for solving two more QR decomposition problems. Each of these two data transfers have a small size of $L_{ls}N$. For the weight application step, the solutions of QR decomposition problems, i.e., weight vectors should also be permuted. But in this permutation, the message size is very small. Therefore, these communication times are not considered here.

Due to the remapping process, the load on the PEs is well balanced. Therefore, the parallel computation time is [10]:

$$T_p = \frac{9(M-2)N^2(L/4+N) + 5LMN\log M}{p} = \frac{T_s}{p},$$

where T_s is sequential execution time. The time for data remapping is

$$T_{remap} = \tau_d \cdot \frac{LMN}{4p}$$

Thus, our algorithm is scalable since the execution time of the algorithm on a machine with p processors varies as $\frac{1}{p}$.

4.4 Experimental Results

We have performed our implementations using C and MPI on the IBM SP-2 at the Maui High Performance Computing Center. The use of MPI ensures portability of the code, and we can therefore port our solutions to SHHiPE platforms such as the HPSCS platform at Martin Marietta Labs. The SP-2 was configured in the dedicated mode to obtain the results shown in Table 1. For the sake of comparison, an earlier implementation [16] of the same technique using the same parameters runs in about 30 seconds on a 32 node SP-1.

Table 1: Execution Times on SP-2 for $M = N = 64$, $L = 4096$

$p=4$	$p=8$	$p=16$	$p=32$	$p=64$
15 *sec*	7.8 *sec*	3.9 *sec*	2.0 *sec*	0.96 *sec*

5 Conclusion

Design methodology for real-time embedded systems is undergoing a shift from custom VLSI based solutions to COTS based heterogeneous architectures. The emerging SHHiPE systems offer advantages of architectural scalability, flexibility, software portability

and lower overall system costs. This paper has identified several issues that must be addressed in the context of this new design paradigm. We have proposed an integrated approach that incorporates hardware aspects, as well as development of scalable algorithms to efficiently exploit the hardware features. Application of this approach to a STAP algorithm has yielded promising performance results.

Further research is necessary before the SHHiPE platforms can be widely used for real-time applications. In particular, research on techniques and methodologies that efficiently exploit the hardware platforms is needed. Accurate computational models which represent the benefits due to task partitioning and mapping, and which allow for task migration are needed. As further experience is gained with this approach, new standards such as Real-time MPI must evolve to provide portability and high performance in real-time signal processing systems.

References

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet - A Gigabit-per-Second Local-Area Network," *IEEE Micro,* Feb. 1995.

[2] J. A. Caruso, "The Challenge of the Increased Use of COTS: A Developer's Perspective," *Proc. of the Third Workshop on Parallel and Distributed Real-Time Systems,* 1995.

[3] T. M. Chung and H. G. Dietz, "Language Constructs and Transformation for Hard Real-Time Systems," *Proc. of ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems,* 1994.

[4] B. O. Gallmeister, *POSIX.4 - Programming for the Real World,* O' Reilley and Associates, Inc., 1995.

[5] R. A. Games, A. Kanevsky, P. C. Krupp, and L. G. Monk, "Real-Time Embedded High Performance Computing: Communications Scheduling," *MITRE Technical Report, MTR 94B0000146,* October 1994.

[6] R. Graybill, Presentation at ARPA Embedded Systems PI Meeting, March 1995.

[7] T.-Y. Huang and J. W.-S. Liu, "Predicting the Worst-Case Execution Time of the Concurrent Execution of Instructions and Cycle-Stealing DMA I/O Operations," *Proc. of ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems,* 1995.

[8] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang, "Heterogeneous Computing: Challenges and Opportunities," *IEEE Computer,* June 1993.

[9] Y. W. Lim, "Scalable Portable Parallel Algorithms for Radar Signal Processing," *Ph.D. thesis proposal, Department of EE-Systems, USC,* 1995.

[10] Y. W. Lim and V. K. Prasanna, "Scalable Portable Algorithms for Space Time Adaptive Processing," *Manuscript, Department of EE-Systems, USC,* April 1995.

[11] Message Passing Interface Forum. "MPI: A Message-Passing Interface Standard," *Technical Report CS-94-230, University of Tennessee, Knoxville, TN,* May 5 1994.

[12] *MPI Forum discussion list for real-time extensions,* majordomo.mcs.anl.gov.

[13] Myricom Home Page, at URL http://www.myri.com

[14] D. Niehaus, K. Ramamritham, J. A. Stankovic, G. Wallace, and C. Weems, "The Spring Scheduling Co-Processor: Design, Use, and Performance," *Proc. of IEEE Real-Time Systems Symposium,* 1993.

[15] K. D. Nilsen and B. Rygg, "Worst-Case Execution Time Analysis on Modern Processors," *Proc. of ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems,* 1995.

[16] S. J. Olszanskyj, J. M. Lebak, and A. W. Bojanczyk, "Parallel Algorithms for Space-Time Adaptive Processing," *IPPS '95,* pp. 77-81, 1995.

[17] J. Stankovic and K. Ramamritham, "What is Predictability for Real-Time Systems," *Real-Time Systems Journal,* Dec. 1990.

[18] J. Stankovic, "Distributed Real-Time Computing: The Next Generation," *Special issue of Journal of the Society of Instrument and Control Engineers of Japan,* Vol. 31, No. 7, 1992.

[19] C.-L. Wang, "High Performance Computing for Vision on Distributed Memory Machines," *Ph.D. Thesis, University of Southern California,* Aug. 1995.

[20] J. Ward, "Space-Time Adaptive Processing for Airborne Radar," *MIT Lincoln Lab., Technical Report 1015,* Dec 1994.