APPLICATION ORIENTED AND COMPUTER SCIENCE CENTERED HPCC RESEARCH

Laxmikant V. Kalé
Department of Computer Science
University of Illinois
Urbana, IL 61801
E-mail: kale@cs.uiuc.edu

Abstract

At this time, there is a perception of a backlash against the HPCC program, and even the idea of massively parallel computing itself. In preparation to defining an agenda for HPCC, this paper first analyzes the reasons for this backlash. Although beset with unrealistic expectations, parallel processing will be a beneficial technology with a broad impact, beyond applications in science. However, this will require significant advances and work in computer science in addition to parallel hardware and end-applications which are emphasized currently. The paper presents a possible agenda that could lead to a successful HPCC program in the future.

1 Introduction

It is clear that amid the excitement about the emerging high performance computing technology, a backlash of sorts is developing. This backlash is against the HPCC program as well as the idea of massively parallel computing itself. Ken Kennedy, a leading researcher in parallel computing, wrote an article recently, titled "High Performance Computing in Trouble" [6] in which he alluded to the funding difficulties of the HPCC program, the skepticism about its goals in the Senate and Congress, the critical and negative report by the Congressional Budget Office, etc. An article by Fred Weingarten [8] discusses this report as well as the report by GAO on ARPA's management of the HPC architecture research. All of these indicate the backlash against the HPCC program. The backlash against parallel computing itself comes in part from users who have tried to use these computers, and find that the continually improving uniprocessor workstations give them a better return on their investment at the moment.

To formulate an agenda for HPCC in this context, one must first understand the reasons for this backlash, assess the current status of the technology, and understand trends in the base technologies – computer architecture and hardware. Section 2 and 3 of this paper elaborate our view on this. Next, a possible agenda for the HPCC program is described in Section 4. The suggested agenda is divided in two parts: strategic and technical.

The strategic agenda suggests:

1. The HPCC community, including vendors, must *project realistic expectations* of the benefits this exciting and important technology.
2. The HPCC program currently emphasizes development and deployment of massively parallel machines on one hand, and specialized end-user applications on the other. If the HPCC program is to enable the adoption of the parallel technology across a broad range of applications, and thus help the national economy and competitiveness, it is necessary to *equally emphasize the middle layers* that include research on languages, tools, environments, algorithms, and libraries.
3. The parallel machines provide a potential for high performance, but it remains difficult to realize this potential for a wide variety of applications. As the principles in harnessing this technology are better understood, they must be taught via a *strong educational initiative* to the next generation of researchers and developers who must develop inter-disciplinary skills.

The technical agenda presents our view of what research directions should be pursued to effectively harness the power of parallel computers. The directions include efficient portability, message driven execution (as distinct from "message passing"), specific parallel programming abstractions and constructs, and intelligent performance analysis. This agenda underscores a meta-point: Although I am convinced of the validity and significance of this approach, it is clearly not a mainstream approach. As the parallel technology is quite immature, and has not been explored for many classes of potential applications yet, it is important that we avoid standardizing and committing too early. A diverse set of approaches need to be supported at this stage, as long as they stay relevant to applications.

*The author's research was supported in part by the National Institute of Health PHF-2-P41-RR-05969-03 and National Science Foundation grants ASC-181593, CCR-91-06608, and CCR-90-07195.

2 Current Status

The current state of affairs can be characterized by an artificially heightened set of expectations from the potential users of parallel computing, coupled with the reality of the difficulty of obtaining good performance on a wide range of applications. These difficulties present opportunities for computer science (CS) research, yet the participation of the CS community in the HPCC program remains low.

2.1 Heightened Expectations

It must be admitted that the expectations of the user community for performance of parallel machines on real programs have been elevated to unrealistic levels in recent years. The reasons for these heightened expectations include vendors overselling the capabilities of their machines, unwarranted extrapolation of some impressive-looking performance results achieved on regular computations to applications in general, and the availability of MPP machines at federally funded centers.

Vendors raise unrealistic expectations by using raw performance figures to describe the capabilities of their machines. The typical approach is to multiply "guaranteed-not-to-exceed" performance figures for a single processor by the number of processors to yield a "peak performance" rate for their machine. However, these figures are difficult to approach with even the most carefully selected benchmarks because their achievement depends on sustaining a fortuitous combination of operations which match the characteristics of the hardware throughout most of the execution of the program. A vendor can then compound the misconceptions that result by insinuating that these numbers might apply to ordinary applications. A person might thereby be completely misled by such figures unless they are aware of the nature of these claims or have noticed the pervasive cynicism toward vendor specifications that exists among experienced users.

Similar misconceptions might stem from early reports of results obtained on MPP machines. Since performance is often measured in MFLOPS or MIPS, the higher number the better, it is natural to mistake better numbers for more important results. The fallacy once again is that the higher numbers that are reported tend to represent the "low-hanging fruit" of parallel computing. As an example, the Gordon Bell Award for high performance on a real application was awarded a couple of years ago to a very regular stencil-oriented program which achieved remarkable performance but on a computation unusually well-suited to parallelization. We would not dispute the worthiness of this effort or the importance of the application. But it certainly seems to be the case that these figures are not being repeated for a wide range of applications. Yet the widespread publicity surrounding this award naturally catalyzes hopes that such performance is at last within reach for many applications.

Federally funded centers with MPP machines have also contributed to heighten expectations among users. When MPP machines were first becoming available, the users of those machines tended to be persons experienced in high-performance computing and willing to expend the resources to achieve high performance in their applications. And again, these early applications were often based on regular computations that were relatively easy to execute in parallel. As parallel machines began to proliferate, especially as they found homes in federally funded centers, the user base expanded to include many users less experienced with parallelizing programs and not especially interested in diverting resources from development of their applications to coping with new programming paradigms. Furthermore, these new users brought a wider variety of applications, including many irregular computations, for which the commonly-used parallelization approaches might not apply. The availability of parallel machines to this broad set of new users brought the heightened expectation in contact with the reality.

2.2 Difficulty of Efficient Parallelism

It is difficult to get good performance on complete applications using parallel computers, except for a small class of regular applications. The difficulties of writing parallel programs are not just due to the amount of inter-processor communication, but include :

- Asynchrony : MIMD parallel computers have independently running processes, and events in the parallel program can potentially occur in different orders on different executions. Moreover, communication latencies can vary widely, and most message passing systems do not even ensure in-order message delivery. Ensuring correctness in the face of this non-determinism is a difficult problem.
- Complexity : exploiting parallelism efficiently requires consideration of partitioning (how to split the computation into parallel parts), mapping (how to assign the parallel parts to processors) and scheduling (in what order to execute the parallel parts of a computation on a processor).
- Idle time : causes wasted processing resources, and should be minimized. Idle time is caused by remote communication network latency, remote processing time (delays because the remote processor does not process the request immediately), transient load imbalances, and long critical paths.
- Smarter algorithms tend to be more irregular. E.g., for n-body problems arising in molecular dynamics and stellar dynamics applications, the simplest algorithm having an $O(n^2)$ complexity for n particles is regular and can be easily parallelized. However, the fast multipole algorithm having $O(n)$ complexity has highly irregular patterns of communication.
- Irregular applications are difficult to parallelize because they aggravate all the problems due to load imbalance, latency, and scheduling.

While massively parallel machines were being deployed at supercomputer centers across the country, single-processor workstations were gaining in performance in spurts. Currently, workstations in excess of 100 MFLOPS are readily available. What is more, one can achieve a significant fraction of this peak performance in a sustained manner on real applications. The performance of these workstations almost equals that of supercomputers which application scientists used to employ only a few years ago, and it is immediately available on their desktops.

In the face of the problems of parallelization and the availability of workstation networks, why do we need parallel machines? First, we must recognize that simple-minded parallelization often gets bad results in terms of speedups and other performance metrics. We should not blame or give up the technology of massively parallel computing. Instead, we must try to appreciate the complexities of obtaining effective parallelization. Second, the potential of massively parallel computers is enormous, if we can solve the parallelization problem. Third, the trends in microprocessor technology suggest that desktop workstations will have between 8 and 64 processors in them within the next three to five years. This is partly because because microprocessors, which are a commodity product, constitute a small fraction of the cost of a workstation. Therefore, widespread availability of parallel computing technology is inevitable and must be dealt with.

This motivates research in techniques and algorithms to exploit parallelism effectively.

2.3 Role of Computer Science

What role are computer science and computer scientists playing in the HPCC program? Obviously, computer science played an important role in enabling the technology that is at the base of the HPCC program itself. Advancements in the microprocessor technology via the RISC methodology, interconnection networks (such as the fat-trees used in the CM-5), switching technology (e.g. wormhole routing), compilers, parallel algorithms, and portability libraries have all contributed to making fast parallel computers possible and usable. Yet, one finds that the participation of the computer science research community, as measured by the number of faculty, graduate students, and postdoctoral associates supported directly by the HPCC program, is quite low. I was unable to obtain objective data to support this belief, but many of my colleagues in computer science have expressed it. They do not see a significant increase in the human-resource budgets within their CS departments that can be attributed to the HPCC program. (This should be a number that the federal agencies could identify and monitor.)

What are the reasons for this lower-than-expected participation? It is sometimes said that computer science has become quite an inward-focussed discipline. Indeed, this is a danger the computer science community must avoid, although some fraction of the research must be devoted to basic research directions that may not appear relevant from the outside, to maintain the vitality of the field. However, this is not the reason for the low participation. There are many computer scientists who see their discipline as an engineering one, and are eager to contribute in an application-oriented direction. The reason is more likely to be the funding patterns that have emerged, by conscious design or otherwise, in the HPCC program. The program has supported development of parallel machines, their deployment, and end-application development projects, but has not supported efforts on languages, systems, compilers, generic algorithms and their implementation with equal emphasis. For example, the recent report by the NSF Blue Ribbon Panel on HPC [1] states the following in justifying its Recommendation B-1 for a challenge program in computer science within CISE:

> There is a consensus that the absence of sufficient funding for systems and algorithms work which is not mission-oriented is the primary barrier to lower cost, more widely accessible, and more usable massively parallel systems.

What role *should* computer science and computer scientists play in the HPCC program?

When I asked this question to an eminent physical scientist recently, he told me that he takes a dim view of the role of computer scientists. The computer scientists have their own agenda, he said, and they tend to take off on work tangential to the development of application programs. I believe we computer scientists should have our agendas, because we would like the principles and techniques we develop to be applicable to a broad variety of applications, rather than only the one at hand. However, we need to stay application oriented, to avoid the danger of developing techniques that are irrelevant to any significant class of applications.

But the broader question of our role is answered by the difficulty (and importance) of effective parallel programming discussed above. A computer scientist's knowledge of algorithms, architectures, compilers, analysis techniques, software engineering concepts, etc. is important to obtain good performance and productivity on parallel machines. We must contribute by studying individual applications, developing core techniques, and tools, and testing their applicability to many applications.

3 Answers to Focal Questions

The organizers of the workshop on HPCC agendas posed three specific questions to the contributors of position papers, which will be briefly addressed now.

Transition to parallel computation: Overall, I expect the transition to be gradual, as individual applications face competitive challenges based on speeds, and migrate to parallel machines. Availability of desktop parallelism will accelerate the transition considerably. However, if we are able to develop tools that can simplify development of parallel programs, provide efficient portability, allow reuse of parallel modules without sacrificing efficiency, and thereby protect the investment of independent software developers in writing parallel applications, it would provide a catalyst that can lead to explosive growth.

Parallel Machines: strategically planned progression toward the "ultimate" machine, or a series of attempts? As the trends in technology and the utility of machine features cannot be effectively predicted for the long-term future, I expect machines to be designed to be the best ones at the moment, as they should be. However, the strategic plans, such as the Touchstone project at Intel SSD, have their value and should be pursued to jump-start technological development.

Is High Performance computing a means to an end or end in itself? It clearly is a means to an end – actually to many "ends". However, to enable all these end applications, one must develop the core technology in a "non mission-oriented" manner. This point is further elaborated in the next section.

4 Suggestions for an Agenda

My suggestions for the future of the HPCC program are divided in two parts: the strategic agenda and the technical agenda.

4.1 Strategic Agenda

The broad policy and emphasis I would like to see followed by the policy-makers and the HPCC community include an attempt to project realistic expectations about the new technology, to emphasize "middle layers" of computer science based research equally with the end-layers of parallel machine development, and end-user applications, and to invest in education for training a new generation of parallel programmers.

4.1.1 Project Realistic Expectations

The promise of high performance computing is exhilarating. The ability to compute at a teraflops rate routinely will open up application areas not imagined yet. Thus it is no wonder that the HPCC community, including the researchers, vendors, and the funding agencies are enthusiastic about it. But the community, and the vendors especially, must act responsibly, and project realistic expectations. In particular, the complexities of parallel programming and hurdles in getting good performance should be readily acknowledged. Support for the HPCC program should be sought because it is a promising and important agenda, not because it is an easy problem to solve.

4.1.2 Computer Science Based Research

The efforts required to enable the use of parallel computing for end-applications can be stratified into layers that build upon each other as shown in Figure 1. At the base of the hierarchy are the different parallel machines. Parallel programming languages allow one to abstract away the machine details. Different families of languages distinguish themselves by their application areas, and programming paradigms they support. Hoping for a single (or even a few) language standards at this stage is not realistic for two reasons:

1. Different computations need different languages. E.g., HPF is better suited for data parallel computations, whereas MPI or PVM are more general purpose.
2. There isn't an agreement on what a good parallel language should be like. Some of us believe in object-based message driven languages, others in a generic message passing language, such as PVM or MPI, functional languages, logic languages, coordination languages such as Linda, or shared memory languages. Each approach has its merit, and it is probably too early to decide which one is "right", if such a decision is possible. It is also likely that multiple paradigms and languages will have to survive and co-exist in single applications, due to the merits of each in individual situations.

First, to develop programs effectively using these languages, tools for debugging and performance analysis are needed which must be integrated into programming environments. To be effective, the tools (such as performance analysis tools) must exploit the information provided by the language rather than just the underlying machine, and so will have to be language specific.

Pragmatic parallel algorithms represent the next layer in the hierarchy. For sequential programming, a wide body of knowledge, accumulated over several decades, is available in the literature. Such is not the case in parallel programming. Given a particular problem and context (such as machine characteristics, size of the problem, performance criteria), one should be able select an algorithm from the many available. To bring this about, extensive research on algorithmic development is needed. Deciding when a parallel algorithm is better than another is more difficult than in the sequential case–where a comparison of operation counts suffices. Also, asymptotic analysis, which is a very effective tool for sequential algorithm analysis, is less effective for parallel algorithm analysis which involves a "game of constants". Therefore, empirical studies are needed to map beneficial regions for each algorithm. The resultant algorithms should be embodied in generic libraries whenever possible.

Individual application domains may require development of specialized libraries. These may be specialized versions of the generic algorithms, or new special purpose algorithms that happen to be relevant in a particular field only. An interesting possibility here is that of customizable modules: Many parallel algorithms have variants that are better in different circumstances. They may also have subcomponents where alternative algorithms can be used. Today such variants are often dealt with by rewriting the programs from scratch. A customizable module allows the user to select the variants and the subcomponents. For example, a CFD researcher may use such a module for a 3-D grid based calculation of turbulent flow. They can then specify what linear system solvers to use in its various phases, and what data decomposition to use.

The next layer in the hierarchy is that of end-user application programs. If the HPCC technology is to have a broad impact on society, a broad spectrum of application areas must be addressed. In particular, scientific applications which will enable discoveries and help fight diseases, for example, are important but so are applications in the manufacturing area (in engineering and operations research, for example). The end users in the layer above these are the ones who will utilize the parallel computing cycles to run the application programs, and who need the application programs as well large-time access to parallel machines.

In the context of theses layers, the current HPCC program as implemented appears to focus at the bottom and top extremes of the picture: development and deployment of parallel machines, availability to some end users, and development of a few specialized end-user applications. The middle layers have received some attention, but it is clearly not on par with these. This is a dangerous situation. If this perception is accurate, the HPCC program may end up producing a collection of highly specialized application programs, without advancing the state of the art of parallel programming significantly. This will not help the programmers from industry who will follow to use this new technology. What technology can we transfer to them? Support for the middle layers is crucial to the adoption of the HPC technology to a broad class of applications.

The ASTA (Advanced Software Technology and Algorithms) component of the HPCC program, as articulated

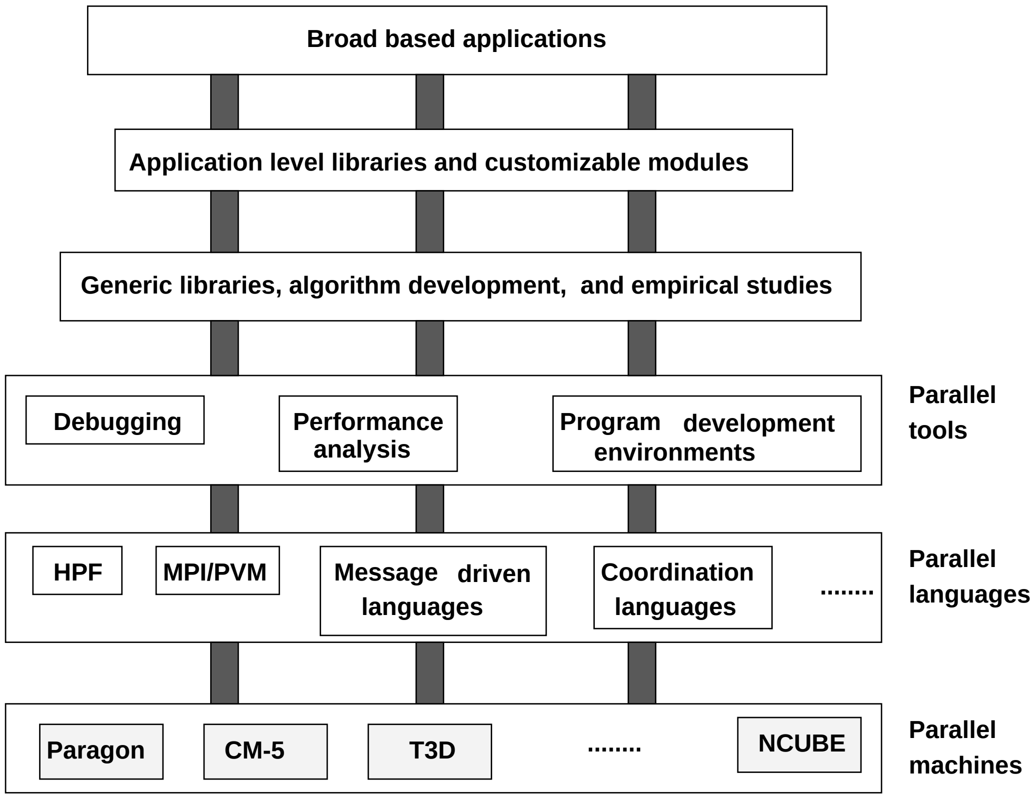

Figure 1: **Hierarchy of research efforts in parallel processing.**

since its beginning, seemed to address these middle layers; yet its implementation seems to have de-emphasized them. I call for a reassessment of the roles of various components, allocation of resources to individual layers, and objective and public monitoring of the support levels.

Fortunately, the research on the middle layers isn't resource-intensive. Developing parallel software, languages, and tools does not require extensive production runs on massively parallel supercomputers. Access provided through national centers is probably adequate for this purpose, along with a few inexpensive workstations. It mainly requires expenditures on human resources. One needs to support many teams with a faculty member, one or more postdoctoral researchers or research programmers, and several graduate research assistants. This will also have a positive impact on the human resource development for this crucial area.

There may be a concern that supporting only mid-layer work may generate irrelevant systems. This is a real danger that must be avoided. However, this can be accomplished by requiring demonstrations of the work using a few sample end-applications or their components.

My suggestion here is fully consistent with recommendation B-1 of the Branscomb report [1], which recommends establishing a number of major projects in computational science and mathematics.

4.1.3 CSE Education

As parallel computers, massive or otherwise, become commonly available, there will be a need to train or retrain a large community of programmers to use them. The current generation of parallel programmers learned by doing, in an ad-hoc manner. For the broader audience, the training must be systematized. Techniques and algorithms must be collected, categorized, and pedagogically organized into textbooks. Appropriate courses must be developed at the universities. For applications in science and engineering, interdisciplinary academic programs need to be developed. Although this is an obvious item for the agenda, it requires consolidation of knowledge in parallel programming, and one may have to wait a while for the dust to settle before the proper teaching (and programming) paradigms emerge.

4.2 Technical Agenda: Personal View

For widespread use of parallel computing, it is necessary to design programming systems that will allow programmers to:

1. Port their program from one machine to another without change, and without a significant performance penalty.
2. Obtain good performance even on irregular classes of applications
3. Simplify the process of developing parallel programs
4. Reuse parallel modules in different applications.

These objectives motivate the agenda pursued by my research group at the University of Illinois. The approach involves an object-based language that encourages locality for efficient portability, message driven execution, dynamic load balancing for dealing with irregular computations, design of parallel abstractions and implementation of tools to simplify programming, and explicit support for modularity. Some aspects of this agenda are described below.

4.2.1 Message Driven Execution

The predominant paradigm used for programming parallel machines is provided by the *traditional SPMD* model, which is supported by vendors of parallel machines in their operating systems. Even data parallel and functional languages are often implemented using the traditional SPMD model as their back end.

The SPMD — single program multiple data — model simplifies program development by using a simple model for internal synchronization and scheduling. In the SPMD model, as used in this paper, there is one process per processor (usually all processes are executing the same program). Communication among processes (hence processors) is usually accomplished with blocking primitives. Messages have tags, and the *receive* primitive blocks the processor until a message with a specified tag arrives (of course, it is possible to use non-blocking communication occasionally if it does not complicate the code). Moreover, we use "traditional SPMD model" to mean strict use of blocking receives.

A single thread of control and blocking receives makes the programming of these machines relatively easy. The simplicity of the flow-of-control attained in SPMD is at the expense of idling processors. After issuing a blocking receive, the processor must wait idly for the specified message to arrive. This wait may not always be dictated by the algorithm, *i.e.*, the algorithm may have more relaxed synchronization requirements. Yet the use of blocking primitives forces unnecessary synchronization and may cause idle time.

This idle time can be decreased by rearranging the send and receive operations. This involves moving the sends earlier and postponing the receives as much as possible in the code. Such local rearrangement of communication can in many cases achieve the desired objective–increasing the utilization of processors. However, this strategy cannot handle cases with more complex dependences and unpredictable latencies.

Further, even though the SPMD model can achieve limited performance improvements as discussed before, it cannot overlap computation and communication across modules and libraries. In the SPMD style, invocation of another module passes the flow of control to that module. Until that module returns control, the calling program cannot do anything. Therefore, the idle times that a module experiences cannot be overlapped with computation from another module.

Message-driven execution, in contrast to the SPMD model, supports many small processes (or objects) per processor. These processes are activated by the availability of messages that are directed to them. At this level of description, it suffices to say that each process has a state and a set of functions (methods) for dealing with incoming messages. When a message arrives for a particular process, the system eventually activates the process. Then the process, depending on the content and type of the message, executes the appropriate method.

Message-driven execution overcomes the two difficulties experienced by the SPMD model. It can effectively overlap latency with useful computation adaptively:

- within a module
- across modules

Message-driven execution could process whichever message arrived first (or whichever message is made available for processing by the message-scheduler), hence, adapting itself to the runtime conditions. This permits greater latency tolerance within a module.

In addition, the message-driven paradigm allows different modules that might have some concurrent computations to share processor time. Consider the computation shown in Figure 2 (figures taken from a recent Ph.D. thesis of my student Attila Gursoy): module `A` invokes two other modules `B` and `C`. In the SPMD model, module `A` cannot activate `B` and `C` concurrently even if the computations in `B` and `C` are independent of each other. As a result, the processor time is not fully utilized, as illustrated in the same figure. In a message-driven paradigm, the idle times on a processor can be utilized by another module if it has some work to do. Such a scenario is illustrated in Figure 3. Module `C` gets processor time (by virtue of having its message selected by the scheduler) while `B` waits for some data, and vice versa, thus achieving a better overlap than the SPMD program.

Libraries constitute an important part of the software development process. They provide reusable, portable code, and they hide details from application programmers. There are many SPMD parallel libraries for commonly used kernel operations, such as numerical solvers, FFT, etc. The SPMD style does not encourage use of multiple concurrent libraries. When faced with performance loss in a situation, such as in Figure 2, an SPMD programmer typically breaks the library abstraction, combines modules B and C with A, and then tries to achieve better overlap by static movement of code. On the other hand, message-driven style encourages creation of smaller and more reusable modules. Therefore, we expect libraries to be a major strength of message-driven systems in the future.

However, developing message-driven libraries requires a different set of assumptions. The organization of the code and the nature of the interfaces across modules is different. The issues involved in message-driven inter-module interfaces must therefore be understood before one can undertake development of large sets of message-driven libraries. The main reason why message-driven libraries have to be treated differently from SPMD libraries is the split-phase nature of inter-module exchanges. When a module calls another module, it must simply deposit the data it wishes to deposit, and then continue or suspend. Eventually, the other module will return data and control to the calling module in the form of a message. Substantial research on message driven protocols for exchange of data among modules and on development of message-driven libraries is still needed.

4.2.2 Charm

Charm [5] is a portable, object-based, and message-driven parallel programming language developed at the Parallel Programming Laboratory at the University of Illinois.

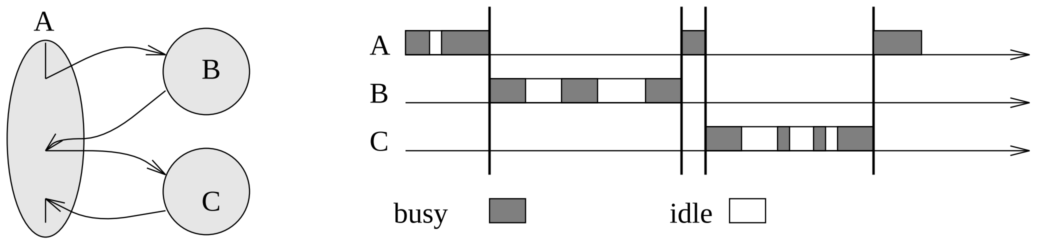

Figure 2: SPMD modules cannot share the processor time.

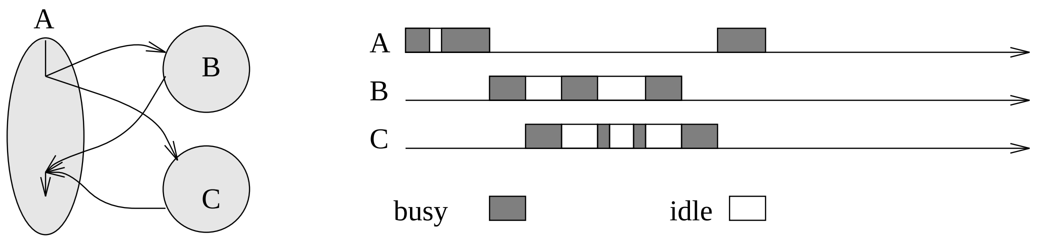

Figure 3: Message-driven modules share the processor time.

Charm programs run unchanged on a variety of shared and private memory parallel machines. The basic unit of computation in Charm is a *chare* (which is a concurrent object). A chare's *definition* consists of an encapsulated data area and entry functions that can access the data area. Chare *instances* can be created dynamically, such that each chare instance has a unique address. Charm provides a second type of process called a *branch office chare.* A copy (branch) of the chare executes on each processor. Branch office chares provide a convenient abstraction for the implementation of various distributed strategies, such as load balancing, quiescence detection, and distributed data structures.

4.2.3 Information Sharing Abstractions

A parallel computation can be characterized as a collection of processes running on multiple processors. Depending on the programming model and language, it may have just one or many processes on each processor. As the processes are part of a single computation, they often have to exchange data with each other.

One of the most popular information sharing mechanisms is a shared variable. Two or more processes may exchange information by setting and reading the same shared variable. This model offers great simplicity as it appears to extend the sequential programming model in a natural manner. However information exchange through shared variables suffers from one major drawback: the difficulty of efficient implementation on large parallel machines. Shared variables can be implemented efficiently on small parallel machines, which physically share memory across a bus and can provide hardware support for a single global address space. However, many large-scale machines available today, such as Intel iPSC/860 and Paragon, NCUBE/2, and CM-5, include hundreds of processors. Implementing shared variables on such machines is difficult and inefficient.

Messages provide another important means of exchanging information between processes in systems such as PVM [7], Express [3], and Actors [4]. Messages containing necessary information can be sent from a "sender" process to a known "receiver" process[1]. Most commercial distributed memory machines provide hardware support for message passing, so this mechanism to exchange information can be easily implemented. However message passing as the sole means of exchanging information may not be adequate, or may not be expressive enough to easily represent many different modes of information exchange. For example, in order to send a message, the sending process must know the identity of the receiving process. In many applications, such information may not be easily available. Message passing can also prove to be a cumbersome, if not an inefficient, mechanism to express information sharing between multiple processes. For example, read-only information[2] *can be* exchanged via messages in a language with message passing as the universal information sharing mechanism. But the cost of accessing the information is substantial. Access to the information can be optimized by replicating the read-only information on each processor. However the user needs to go to considerable effort in order to implement (with messages) a replicated variable, which is accessed through a unique identifier.

There exist other mechanisms to exchange information among parallel processes. The information sharing mecha-

[1]In most current message-passing models, information can be exchanged only on a point-to-point basis. However, collective communication primitives are being designed by the message passing interface (MPI) standardization committee [2].

[2]Read-only information is data that is initialized once and not altered thereafter.

nisms provided by Linda and Strand suffer from the same problem: each provides only a single information exchange mechanism. Compilers for languages with a universal information sharing mechanism often attempt to detect various modes of information sharing in order to produce more efficient object code. However the detection of a particular mode of information sharing can be imperfect and conservative at best. It would be more intuitive and convenient for the programmer to specify a mode of information exchange, rather than trying to fit all information sharing modes into the single mode of information exchange.

The problems with a single universal sharing mechanism suggest that a parallel language must provide multiple mechanisms to share information. Also, for portability, there must be a separation between the implementation of a particular mode of information sharing and its abstraction available to the user. Empirical observation of parallel programs suggests that processes share data in a few *distinct* and *specific* modes. We argue that such modes should be identified and explicitly supported in parallel languages and their associated models. We have identified and implemented some of these specific modes of information exchange in Charm: *readonly*, *accumulator*, *monotonic*, *write-once*, and *distributed table*. These abstractions lead to improved clarity and expressiveness of user programs. The abstractions have been implemented in the most efficient manner on the underlying architectures.

4.2.4 "Intelligent" Performance Analysis

There exist a large and diverse collection of parallel programming tools. The primary emphasis of these tools has been to present generic information about the execution of the program. The analysis of the information about a program's execution has been left largely to the user. Automatic performance analysis of a program's execution is possible if sufficient information about the characteristic behavior of a parallel program is available to the performance analysis tool. In most parallel languages, such information is not easily available. However, for Charm, based on the stratification provided by objects and specificity provided by multiple information sharing abstractions, a performance analysis system can combine run-time traces with compile-time information to provide intelligent performance analysis. In fact, we are pursuing development of an analysis system that goes further in exploiting this information, and will provide the user with suggestions on improving the performance.

4.2.5 Under-represented Application Areas

Application areas, such as computational biology and computational fluid dynamics, are quite important and are being addressed by the HPCC program. However there are other areas, which I expect to benefit significantly from parallel computing and to have a significant impact on society, that are currently under-represented. These include operations research for industry, discrete event simulation for electronics and defense applications, combinatorial search and AI for its future application potential.

4.3 Technical Agenda: Broader View

We believe that message driven execution, for the reasons outlined above, will be the "right" way of writing parallel programs. Yet, the currently dominant paradigm is the SPMD based message passing one. In addition, other competing approaches, with their own merits, are also being pursued — such as functional languages, Linda, etc. Our views on specific information sharing abstractions compete with those that believe message passing, or shared variables, or tuple spaces, or weak models of consistency, provide an adequate and appropriate abstraction. This is symptomatic of the diverse needs of different applications, different machine characteristics, and the need for further experimentation. A danger to avoid, then, is that of early standardization, which might stunt creativity and progress on the "right" paths.

Acknowledgements: I am grateful to Attila Gursoy, Ed Kornkven, Sanjeev Krishnan, and Amitabh Sinha for their help in preparing this manuscript.

References

[1] L. Branscomb, T. Belytschko, P. Bridenbaugh, T. Chay, J. Dozier, G. S. Grest, E. F. Hayes, B. Honig, N. Lane, J. W. Lester, G. J. McRae, J. A. Sethian, B. Smith, and M. Vernon. Recommendations to implement this goal. *From Desktop to teraflop: exploiting the U.S. lead in high performance computing*, pages 13–14, August 1993.

[2] J. Dongarra, M. Snir, W. Gropp, E. Lusk, A. Geist, and et. al. Document for a standard message-passing interface. 1993.

[3] J. Flower, A. Kolawa, and S. Bharadwaj. The Express way to distributed processing. In *Supercomputing Review*, pages 54–55, May 1991.

[4] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT press, 1986.

[5] L. V. Kale. Parallel programming with Charm: an overview. Parallel Programming Laboratory, Technical Report PPL-TR-93-8, University of Illinois, Urbana-Champaign, Department of Computer Science, July 1993.

[6] K. Kennedy. High performance computing in trouble. *Parallel computing research*, 1(4):2, October 1993.

[7] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2, 4:315–339, December 1990.

[8] F. W. Weingarten. HPCC research questioned. *Communications of the ACM*, 36(11):27–29, November 1993.