Proceedings of Workshop on *Developing Computer Science Education – How Can It Be Done?*, March 10, 2006, Linköping university, Linköping, Sweden.
http://www.ida.liu.se/~chrke/neststep/dcse06.pdf

Teaching parallel programming early

Christoph W. Kessler
Institute for Computer and Information Science (IDA)
Linköping university, S-58183 Linköping, Sweden
chrke@ida.liu.se

***Abstract*— In this position paper, we point out the importance of teaching a basic understanding of parallel computations and parallel programming early in computer science education, in order to give students the necessary expertise to cope with future computer architectures that will exhibit an explicitly parallel programming model.**

We elaborate on a programming model, namely shared-memory bulk-synchronous parallel programming with support for nested parallelism, as it is both flexible (can be mapped to many different parallel architectures) and simple (offers a shared address space, structured parallelism, deterministic computation, and is deadlock-free).

We also suggest taking up parallel algorithmic paradigms such as parallel divide-and-conquer together with their sequential counterparts in the standard CS course on data structures and algorithms, in order to anchor thinking in terms of parallel data and control structures early in the students' learning process.

I. Introduction

For 50 years we have been teaching students programming, algorithms and data structures with a programming model dominated by the sequential von-Neumann architecture. This model is popular because of its simplicity of control flow and memory consistency and its resemblance to the functionality of early computer architectures.

However, recent trends in computer architecture show that the performance potential of von-Neumann programming has finally reached its limits. Computer architectures even for the desktop computing domain are, at least under the hood, increasingly parallel, e.g. in the form of multithreaded processors and multi-core architectures. The efforts for grafting the von-Neumann model on top of thread-level parallel and instruction-level parallel processor hardware, using techniques such as dynamic instruction dispatch, branch prediction or speculative execution, are hitting their limits, in the form of high design complexity, limited exploitable instruction-level parallelism in applications, and power and heat management problems. We foresee that explicitly parallel computing models will emerge in the near future to directly address the massive parallelism available in upcoming processor architectures. In order to fully exploit their performance potential, applications will have to be parallelized, that is, be (re)written to exhibit explicit parallelism. However, most programmers are reluctant to adopting a parallel programming model, (1) because parallel programming is notoriously more complex and error-prone than sequential programming, at least with the parallel programming systems that are in use today, and (2) because most programmers were never trained in *thinking* in terms of parallel algorithms and data structures.

For the years to come, explicitly parallel programming paradigms will have to be adopted by more and more programmers. In this position paper, we suggest preparing students in time for this paradigm shift. A first step could be to take up parallel computing issues relatively early in existing standard courses, e.g. in the undergraduate course on data structures and algorithms. Many fundamental algorithmic concepts such as divide-and-conquer have an immediate parallel counterpart, which may be considered together. The goal is to anchor thinking in parallel structures early in the education process.

In particular, we advocate *simple* parallel programming models, such as the *bulk-synchronous parallel* (BSP) model, because they are (a) still flexible enough to be mapped to a wide range of parallel architectures, and (b) simple enough to provide a good basis for the design and analysis of parallel algorithms and data structures that offers a compatible extension of the existing theory.

The remainder of this paper is organized as follows. In Section II we summarize the most important parallel programming models and discuss their suitability as a platform for teaching parallel programming early. Section III elaborates on one particular parallel programming model and language, NestStep. We discuss more teaching issues in Section IV, and Section V concludes.

II. Survey of parallel programming models

With the need to exploit explicit parallelism at the application programming level, programmers will have

TABLE I
SURVEY OF PARALLEL PROGRAMMING MODELS

Progr. Model	Control Structure	Data View	Consistency	Main Restriction	Examples
Message Passing	Asynchronous, MIMD	Local	N.a.	None	MPI
Shared Memory	Asynchronous, MIMD	Shared	Weak	None	Pthreads, OpenMP, UPC, Cilk
Data-Parallel	Synchronous, SIMD	Shared	Strict	SIMD-like Control	HPF, C^*
PRAM	Synchronous, MIMD	Shared	Strict	None	Fork
BSP	Bulk-synchr., MIMD	Local	N.a.	Superstep Structure	BSPlib, PUB
NestStep-BSP	Nested BSP	Shared	Superstep	Superstep Structure	NestStep

to adopt a parallel programming model. In this section, we briefly review the most important ones.

A. Message passing

The currently predominant model for programming supercomputers is message passing with MPI, the message-passing interface [13], which is a portable but low-level standard for interprocessor communication. Message passing may be considered a least common denominator of parallel computing, which is available on almost every parallel computer system that is in use today. Message passing code is generally unstructured and hard to understand, maintain and debug. It only supports a local address space (starting from location 0 on each processor) and requires the programmer to place explicit send and receive operations in the code, and maybe even handle buffering explicitly, in order to exchange data between processors. Modest improvements such as one-sided communication (automatic receive), nested parallelism (by processor group splitting), and collective communication operations (such as reductions, scatter and gather) help to somewhat reduce complexity—usually at the expense of some performance loss—but are not enforced, such that unstructured parallelism is still the default.

B. Shared memory and shared address space

Shared-memory parallel computing is, in practice, often realized by asynchronously operating threads or processes that share memory. Synchronization between these parallel activities (here referred to as processors, for simplicity) can be in the form of asynchronous signals or mutual exclusion in various forms (semaphores, locks, monitors etc.). A global address space provides to the programmer a more natural interface to access data that is compliant with sequential computing models. Unstructured shared-memory parallel programming platforms such as *pthreads* have been complemented by more structured languages such as *OpenMP* [12], which supports work-sharing constructs to schedule parallel tasks such as parallel loop iterations etc. onto a fixed set of processors. Additionally, shared-address-space languages such as UPC [2] that emulate a shared memory view on top of a message passing platform have emerged recently. On the other hand, memory consistency must increasingly be handled explicitly by the programmer at a fairly low level, e.g. by *flush* operations that reconcile the local value of a cached memory location with its main memory value. Finally, none of these platforms really supports nested parallelism.

C. Data-parallel computing

Data-parallel computing is the software equivalent of SIMD (single instruction stream, multiple data streams) architectures. Processors share a single program control, that is, execute at the same time the same operation on maybe different data (or do nothing). Data-parallel computing was very popular in the 1980's and early 1990's because it mapped directly to the vector processors and array computers of that period. Many data-parallel programming languages have been developed, most notably High-Performance Fortran (HPF) [4]. Generally, data-parallel languages offer a shared address space with a global view of large data structures such as matrices, vectors etc. Data-parallel computing is suitable for regular computations on large arrays but suffers from inflexible control and synchronization structure in irregular applications, for which a MIMD (multiple instruction streams, multiple data streams) based model is more appropriate. Although HPF is not widely used in practice, many of its concepts have found their way into standard Fortran and into certain shared-address-space languages.

D. PRAM model

In the design and analysis of parallel algorithms, we mainly work with three theoretical models that are all extensions of the sequential RAM model (Random Access Machine, also known as the von-Neumann model): the PRAM, the BSP, and systolic arrays.

The PRAM (Parallel Random Access Machine) model, see the book by Keller, Kessler and Träff [6] for an introduction, connects a set of P processors to a single shared memory module and a common clock signal. In a very idealistic simplification, shared memory access time is assumed to be uniform and take one clock cycle, exactly as long as arithmetic and branch operations. Hence, the entire machine operates synchronously at the instruction level. If simultaneous accesses to the same memory location by multiple processors are resolved in a deterministic way, the resulting parallel computation will be deterministic. Memory will always be consistent. Synchronization can be done via signals, barriers, or mutual exclusion; deadlocks are possible. As the shared memory becomes a performance bottleneck without special architectural support, the PRAM model has not been realized in hardware, with one notable exception, the SB-PRAM research prototype at Saarbrücken university in the 1990s [6].

E. BSP model

In contrast, the BSP (Bulk-Synchronous Parallel) model [15] is an abstraction of a restricted message passing architecture and charges a cost for communication. The machine is characterized by only three parameters: the number of processors P, the byte transfer rate g in point-to-point communication, and the overhead L for a global barrier synchronization (where g and L may be functions of P). The BSP programmer must organize his/her parallel computation as a sequence of *supersteps* that are conceptually separated by global barriers. Hence, the cost of a BSP computation is the sum over the cost of every superstep. A superstep (see also Figure 1) consists of a *computation phase*, where processors only can access local memory, and a subsequent *communication phase*, where processors exchange values by point-to-point message passing. Hence, BSP programs have to use local addresses for data, and the programmer needs to write explicit send and receive statements for communication. Routines for communication and barrier synchronization are provided in BSP libraries such as BSPlib [5] and PUB [1]. Nested parallelism is not supported in classical BSP; in order to exploit nested parallelism in programs, it must be flattened explicitly by the programmer, which is generally difficult for MIMD computations. Hence, although announced as a "bridging" model (*i.e.*, more realistic than PRAM but simpler to program and analyze than completely unstructured message passing), some of the problems of message passing programming are inherited. We will show in the next section how to relax these constraints.

F. Which model is most suitable?

Table I summarizes the described parallel programming models, with their main strenghts and weaknesses.

From a technical point of view, a model that allows to control the underlying hardware architecture efficiently is most important in high-performance computing. This explains the popularity of low-level models such as MPI and pthreads.

From the educational point of view, a simple, deterministic, shared memory model should be taught as a first parallel programming model. While we would opt for the PRAM as model of choice for this purpose [10], we are aware that its overabstraction from existing parallel computer systems may cause motivation problems for many students. We therefore consider BSP as a good compromise, as it is simple, deterministic, semi-structured and relatively easy to use as a basis for quantitative analysis, while it can be implemented on a wide range of parallel platforms with decent efficiency. However, to make it accessible to masses of programmers, it needs to equipped with a shared address space and better support for structured parallelism, which we will elaborate on in the next section.

The issues of data locality, memory consistency, and performance tuning remain to be relevant for high-performance computing in practice, hence interested students should also be exposed to more complex parallel programming models at a later stage of education, once the fundamentals of parallel computing are well understood.

III. NESTSTEP

NestStep [8], [7] is a parallel programming language based on the BSP model. It is defined as a set of language extensions that may be added, with minor modifications, to any imperative programming language, be it procedural or object oriented. The sequential aspect of computation is inherited from the basis language. The new NestStep language constructs provide shared variables and process coordination. The basis language need not be object oriented, as parallelism is not implicit by distributed objects communicating via remote method invocation, but expressed by separate language constructs.

NestStep processes run, in general, on different machines that are coupled by the NestStep language extensions and runtime system to a virtual parallel computer. Each processor executes one process with the same program (SPMD), and the number of processes remains constant throughout program execution.

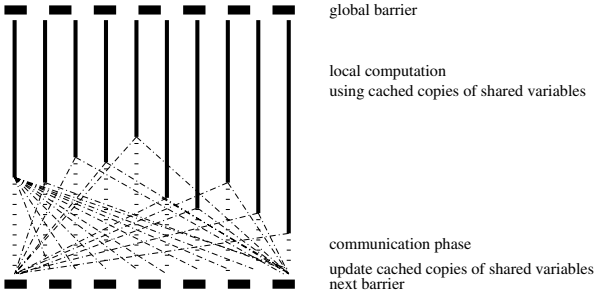

Fig. 1. A BSP superstep. — In NestStep, supersteps form the units of synchronization and shared memory consistency.

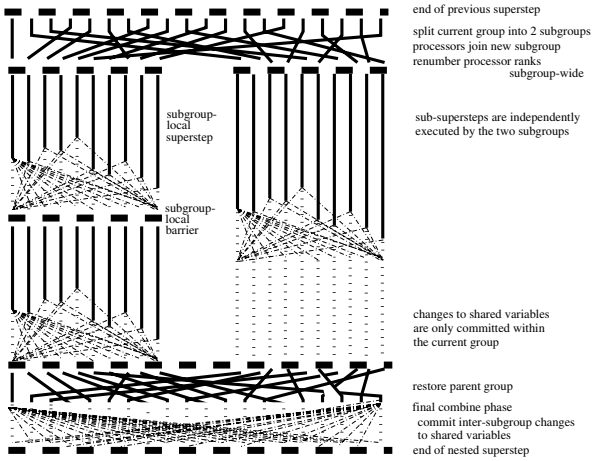

Fig. 2. Nesting of supersteps, here visualized for a `neststep(2,...) ...` statement splitting the current group into two subgroups. Dashed horizontal lines represent implicit barriers.

The NestStep processors are organized in *groups*. The processors in a group are automatically ranked from 0 to the group size minus one. The `main` method of a NestStep program is executed by the *root group* containing all available processors of the partition of the parallel computer the program is running on. A processor group executes a BSP superstep as a whole. Ordinary, flat supersteps are denoted in NestStep by the `step` statement

`step` *statement*

Groups can be dynamically subdivided during program execution, following the static nesting structure of the supersteps. The current processor group is split by the `neststep` statement, as in

```
neststep(2; @=(cond)?1:0) // split group
  if (@==1) stmt1();
  else      stmt2();
```

into several subgroups, which can execute supersteps (with local communication and barriers) independent of each other. At the end of the neststep statement, the subgroups are merged again, and the parent group is restored. See Figure 2 for an illustration. Note that this corresponds to forking an explicitly parallel process into two parallel subprocesses, and joining them again.

Group splitting can be used immediately for expressing parallel divide-and-conquer algorithms.

Variables in NestStep are declared to be either *shared* (`sh`) or *private*. A private variable exists once on each processor and is only accessible locally. A shared variable, such as `sum` in Figure 3, is generally replicated: one copy of it exists on each processor. The NestStep runtime system guarantees the *superstep consistency* invariant, which says that at entry and exit of a superstep, the values of all copies of a replicated shared variable will be equal. Of course, a processor may change the value of its local copy in the computation phase of a superstep. Then, the runtime system will take special action to automatically make all copies of that variable consistent again, during the communication phase of the superstep. The conflict resolution strategy for such concurrent writes can be programmed individually for each variable (and even for each superstep, using the `combine` clause). For instance, we could specify that an arbitrary updated value will be broadcast and committed to all copies, or that a reduction such as the global sum of all written values will be broadcast and committed. As the runtime system uses a communication tree to implement this combining of values, parallel reduction and even prefix computations can be performed on-the-fly without additional overhead [8]. Exploiting this feature, parallel prefix computations, which are a basic building block of many parallel algorithms, can be written in a very compact and readable way, see Figure 3.

Shared arrays can be either replicated as a whole, or distributed (each processor owns an equally large part of it), as `a` in the example program in Figure 3. Distributed shared arrays are complemented by appropriate iterator constructs. For instance, the forall loop

```
forall ( i, a )
   stmt(i, a[i]);
```

scans over the entire array `a` and assigns to the private iteration counter `i` of each processor exactly those indices of `a` that are locally stored on this processor.

As superstep computations only allow access to locally available elements, values of remote elements needed by a processor must be fetched before entry to a superstep, and written elements will be shipped (and combined) in the communication phase at the end of a superstep. Fetching array elements beforehand can be

```
void parprefix( sh int a[]</> )
{
 int *pre;          // priv. prefix array
 int p=#, Ndp=N/p; // assume p divides N
 int myoffset;      // my prefix offset
 sh int sum = 0;
 int i, j = 0;
 step {
   pre = new_Array( Ndp, Type_int );
   forall ( i, a ) {  // owned elements
     pre[j++] = sum;
     sum += a[i];
   }
 } combine( sum<+:myoffset> );
 j = 0;
 step
   forall ( i, a )
     a[i] = pre[j++] + myoffset;
}
```

Fig. 3. Computing parallel prefix sums in NestStep-C.

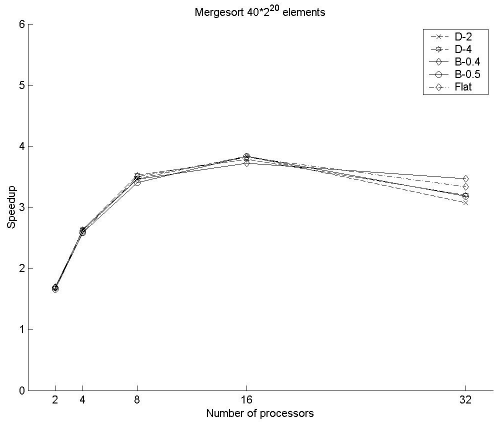

Fig. 4. Relative speedup of a parallel mergesort implementation in NestStep where the merge function is not parallelized. Measurements were done on a Linux cluster; the different speedup curves correspond to different configurations of the NestStep runtime system [14].

a problem in irregular computations where the actual elements to be accessed are not statically known. A technique for scheduling the necessary two-sided communication operations on top of NestStep's combining mechanism for replicated variables is described in [7].

NestStep does not support mutual exclusion, and is thus deadlock-free. The main synchronization primitive is the barrier synchronization included in the `step` and `neststep` statements. In many cases, the need for mutual exclusion disappears as it can be expressed by suitably programming the concurrent write conflict resolution of shared variables. Otherwise, the design pattern for serializing computation is to determine the next processor to do the critical computation, usually by a prefix combine operation at the end of a superstep, and then masking out all but that processor in the computation phase of the following superstep.

At the time of writing, the run-time system of NestStep, implemented on top of MPI, is operational. In measurements on a Linux cluster supercomputer, NestStep outperformed OpenMP (running on top of a distributed-shared memory emulation layer) by a factor of up to 30 [14]. A front end (compiler) for NestStep is in preparation.

IV. Teaching parallel programming

A. Parallel algorithmic paradigms

Many modern textbooks about (sequential) algorithms teach algorithmic concepts and then present one or several algorithms as incarnation of that concept. Many of these concepts indeed have also a direct parallel counterpart.

For instance, *divide-and-conquer* (DC) is an important algorithmic problem solving strategy in sequential computing. A problem is split into one or more independent subproblems of smaller size (divide phase); each subproblem is solved recursively (or directly if it is trivial) (conquer phase), and finally the subsolutions are combined to a solution of the original problem instance (combine phase). Examples for DC computations are mergesort, FFT, Strassen matrix multiplication, Quicksort or Quickhull.

In parallel computing, the *parallel DC* strategy allows for a simultaneous solution of all subproblems, because the subproblems are independent of each other. Parallel DC requires language and system support for nested parallelism, because new parallel activities are created in each recursion step. However, a significant speedup can generally be obtained only if also the work-intensive parts of the divide and the combine phase can be parallelized. This effect is often underestimated by students. For instance, Figure 4 shows the relative speed-up curve for a student's implementation of parallel mergesort that exploits the independence of subproblems in the DC structure of mergesort but uses a sequential merge function for combining the sorted subarrays, which leads to parallel time $O(n)$ with n processors and elements, rather than $O(\log^2 n)$ which can be achieved if the merge function is parallelized as well to run in logarithmic parallel time. If this phenomenon is not made transparent to students in the form of a *work-time-cost analysis framework* (see e.g. [6]), they wonder why their "fully parallel" code does not scale to large numbers of processors but shows saturation effects, as in Figure 4.

B. Textbooks and curricular issues

At Linköping university, about 300 undergraduate students in computer science and closely related study programs per year take one of our fundamental courses on data structures and algorithms, usually in the second year. However, only few of these (ca. 40 per year) find their way into the optional, final-year course on parallel programming, and only 4–8 PhD students every second year sign up for a PhD-level course in parallel programming. Beyond unavoidable diversification and specialization in the late study phases, this may also be caused by a lack of anchoring a fundamental understanding of parallel computing in undergraduate education and, as a consequence, in the mind of the students.

Part of this effect may be attributed to an underrepresentation of parallel computing issues in the established course literature on algorithms. Standard textbooks on algorithms generally focus on sequential algorithms. Some also contain a chapter or two on parallel algorithms, such as Cormen et al. [3]. Up to now, we only know of a single attempt to provide a unified treatment of both sequential and parallel algorithms in a single textbook, by Miller and Boxer [11]. However, their text, albeit fairly short, covers many different parallel programming models and interconnection network topologies, making the topic unnecessarily complex for a second-year student.

Instead, we propose to take up parallel algorithmic paradigms and some example parallel algorithms in a second-year algorithms course, but based on a *simple parallel programming model*, for instance an enhanced BSP model as supported by NestStep.

C. Experiences in special courses on parallel computing

In a graduate-level course on parallel programming models, languages and algorithms, we used the PRAM model and the C-based PRAM programming language Fork [6], which is syntactically a predecessor of NestStep. That course contained a small programming project realizing a bitonic-sort algorithm as presented in theory in Cormen et al. [3] in Fork and running and evaluating it on the SBPRAM simulator [9]. The goal was that the students should understand the structure of the algorithm, analyze and experimentally verify the time complexity ($O(\log^2 N)$ with N processors to sort N elements). Our experiences [10] indicate that the complementation of the theoretical description of the parallel algorithm by experimental work was appreciated, and that the available tools (such as a trace file visualizer showing the computation and group structure) helped in developing and debugging the program. We expect similar results for NestStep once a frontend will be available.

V. Conclusion

We have motivated why technical and teaching support for explicit parallel programming gets more and more important in the coming years. We have reviewed the most important parallel programming models, elaborated on an enhanced version of the BSP model, and described the NestStep parallel programming language supporting that model. Overall, we advocate a simple, structured parallel programming model with deterministic consistency and synchronization mechanism, which should be accessible to a larger number of students and presented earlier in the curriculum to create a fundamental understanding of parallel control and data structures. We suggested a scenario how parallel programming concepts could be added into a standard course on algorithms, that is, relatively early in computer science education.

References

[1] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) Library. *Parallel Computing*, 29:187–207, 2003.

[2] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, second printing, IDA Center for Computing Sciences, May 1999.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[4] High Performance Fortran Forum HPFF. High Performance Fortran Language Specification. *Sci. Progr.*, 2, 1993.

[5] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. BSPlib: the BSP Programming Library. *Parallel Computing*, 24(14):1947–1980, 1998.

[6] Jörg Keller, Christoph Kessler, and Jesper Träff. *Practical PRAM Programming*. Wiley, New York, 2000.

[7] Christoph Kessler. Managing distributed shared arrays in a bulk-synchronous parallel environment. *Concurrency – Pract. Exp.*, 16:133–153, 2004.

[8] Christoph W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. *The J. of Supercomputing*, 17:245–262, 2000.

[9] Christoph W. Keßler. Fork homepage, with compiler, SBPRAM simulator, system software, tools, and documentation. www.ida.liu.se/∼chrke/fork/, 2001.

[10] Christoph W. Kessler. A practical access to the theory of parallel algorithms. In *Proc. ACM SIGCSE'04 Symposium on Computer Science Education*, March 2004.

[11] Russ Miller and Laurence Boxer. *Algorithms sequential & parallel: A unified approach*. Prentice Hall, 2000.

[12] OpenMP Architecture Review Board. OpenMP: a Proposed Industry Standard API for Shared Memory Programming. White Paper, `http://www.openmp.org/`, October 1997.

[13] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[14] Joar Sohl. A scalable run-time system for NestStep on cluster supercomputers. Master thesis LITH-IDA-EX-06/011-SE, IDA, Linköpings universitet, 58183 Linköping, Sweden, March 2006.

[15] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Comm. ACM*, 33(8), August 1990.