Final Thesis

A Scalable Run-Time System for NestStep on Cluster Supercomputers

by

Joar Sohl

LITH-IDA-EX--06/011--SE

2006-03-13

Linköpings universitet
Department of Computer and Information Science

Final Thesis

A Scalable Run-Time System for NestStep on Cluster Supercomputers

by

Joar Sohl

LITH-IDA-EX--06/011--SE

2006-03-13

Supervisor: Christoph Kessler
Examiner: Christoph Kessler

Table of Contents

1 Introduction

1.1 Why parallel programming?

Many applications, especially scientific ones, today require a vast amount of computational power, e.g. weather predictions and particle simulations. As uniprocessors at this point are too slow to solve these problems at an acceptable speed, we need to utilize a larger amount of processors that work together in order to solve these problems. The rate by which performance is increasing for uniprocessors is also on a decline, largely because of increasing leakage currents, and increased power consumption due to the higher clock frequencies. This makes it highly unlikely that uniprocessors ever will be able to reach such speeds that is demanded. This is realized by chip manufacturers like Intel and AMD and most new processors developed by them today use dual cores, and even more cores are being planned for. That is, as we know that parallel computing is needed today and in future, there is a need for tools and programming languages that enables programmers to easily utilize the power of parallel computers.

1.2 What is NestStep?

NestStep[1, 2] is a collection of parallel programming extensions to existing programming languages, developed by Christoph Kessler. NestStep is based on the bulk-synchronous programming model, introduced by Valiant[3]. The most notable features supported by NestStep is its shared memory model and nested parallelism. The runtime system developed during this project is for the extensions for the C programming language.

The NestStep-C runtime system is designed to work on top of parallel computers supporting MPI 1.2 (Message Passing Interface, the interested reader may find more information at www.mpi-forum.org), since the supercomputer available only supports version 1.2. It uses Tlib[4], a library for group management on top of MPI developed by Rauber and Rünger for controlling processors groups. It has been tested on a distributed memory system at the National Supercomputer Centre in Linköping, Sweden.

1.3 Thesis outline

This thesis is outlined as follows:

- Background reading: In this section we will go through some fundamental knowledge that is required in order to benefit the most from this thesis.
- NestStep constructs: A brief overview covering the most important NestStep extensions.

- Communication structure: A discussion of how the NestStep system handles the task of communicating data.
- Project requirements.
- Interface: A set of guidelines to those interested in writing a front end for NestStep. This section is of no importance to any other section, and can thus be skipped.
- Example applications and implementations: In this section the reader will be introduced to some example applications and their implementations in NestStep.
- Evaluation: Benchmarks of the example applications on the NestStep system with a discussion on the performance.
- Related work: A small discussion on other similar projects.
- Conclusions and future work: A highlight of the most important things to note about the NestStep system, followed by some ideas on what directions the future development of NestStep could take.

2 Background reading

NestStep is based on the Bulk-Synchronous Programming (BSP) model that was introduced by Valiant[3].

Programs created using this model are modeled as a series of supersteps, where the computing nodes are synchronized at the beginning and end of each step.

When the processors are synchronized, local computation with cached versions of the shared data is performed.

When the local computation phase is finished a communication phase begins. During this phase each processor sends any data that other processors may need for the next superstep.

Please see Figure 1 for an illustration of a superstep.

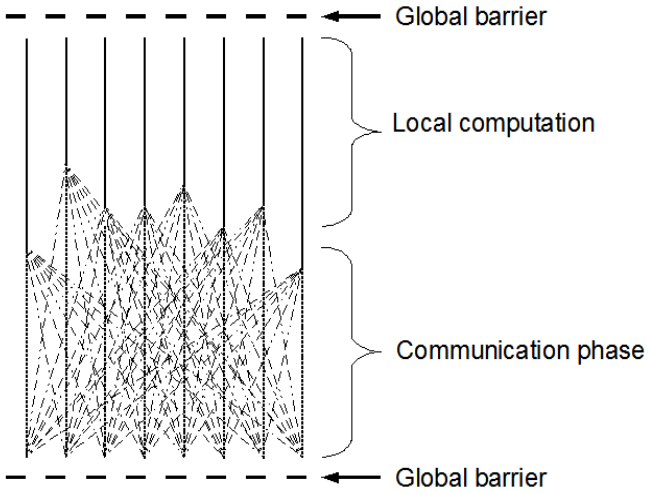

Figure 1: A BSP-superstep.
Recreation of an illustration in [1].

In order to get the most out of this thesis it is suggested to the reader to read Kessler's papers on NestStep[1, 2].

To better appreciate the performance measurements it is useful to understand what kind of performance can be expected from parallel systems. Appendix A contains a brief introduction to this topic.

3 NestStep constructs

This section contains a brief overview of the most important NestStep constructs. For a more detailed description the reader is recommended to read Kessler's papers on NestStep [1,2]. For examples of these constructs please see section 7 of this document.

3.1 Parallel extensions

NestStep enables code sections to be run in parallel. Each processor that is running a parallel section together with other processors is assigned a rank, that ranges between 0 and the number of processors – 1. This collection of processors is called a group, and the processor with rank 0 is the leader.

Memory consistency is ensured outside the supersteps, which means that all shared data is identical on all processors outside the supersteps.

step *statement*
executes the statement in parallel, and keeps memory consistency at the beginning and end of *statement*.

neststep(*k*) *statement*
splits the current group (of size p) in k subgroups of size $\lceil p/k \rceil$ or $\lfloor p/k \rfloor$, with the subgroups indexed from 0 to k-1. This group id can be accessed by the @ symbol while in *statement.*

neststep(*k*, weight) *statement*
Here weight must be a replicated shared array, of k or more non-negative floats where the sum of the elements equals one. The k subgroups that are created are chosen so that each group i, $0 \le i \le k-1$, gets a fraction of the available processors that is as close as possible as the fraction in *weight[i].* Each subgroup gets at least one processor.

neststep(*k*, @=*intexpr*) *statement*
creates k subgroups. *intexpr* is evaluated on each processor and needs to be in the range [0...k-1]. The processor then joins the group with an id identical to *intexpr*. If *intexpr* is outside of the range [0...k-1] it skips *statement*.

seq *statement*
statement is executed by the group leader only(rank 0). No synchronization is implied by this construct, if that is necessary it needs to be wrapped inside a **step**.

In Figure 2 we can see an illustration of the way the nested parallelism works in NestStep.

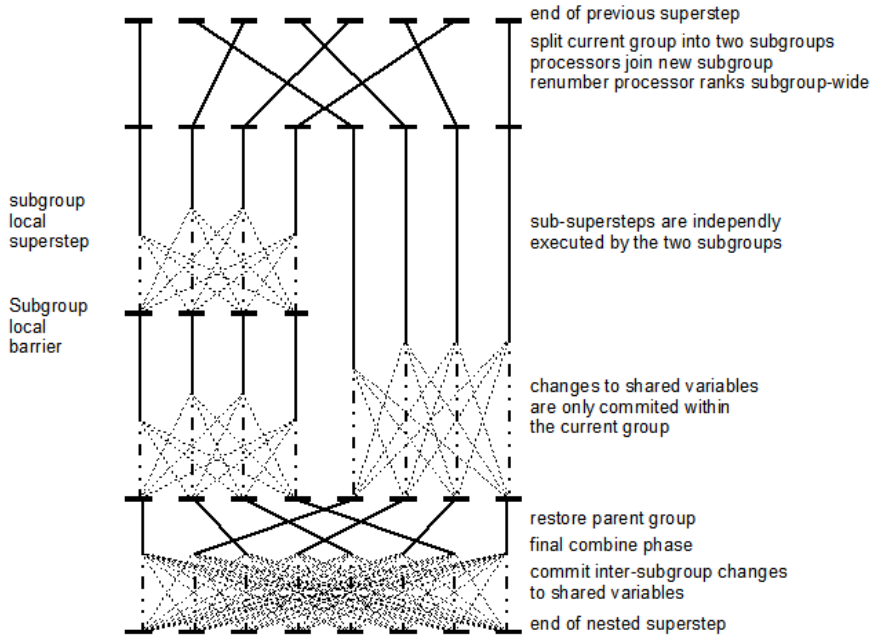

Figure 2: Nested supersteps.
Recreation of an illustration in [1].

3.2 Symbols

#: The number of processors in the current group. It is also accessible by thisgroup.size().

$: The processor rank in the current group. It is also accessible by thisgroup.rank().

@: The id of the group the processor is a part of. This id is assigned after each split. It is also accessible by thisgroup.gid().

3.3 Shared data

Data can be shared in two ways in NestStep. The first way is to use replication, i.e. each processor has it own version of the data, and the shared data is to be made consistent at the beginning and end of supersteps. The other way is by distributing an array's elements across the processors. It is only arrays that may be distributed.

Combining

At the end of steps there is a group-wide combine phase, where the shared data is combined using different combine strategies. The shared data that can be combined is shared variables and replicated arrays, which we will discuss more later. All combine strategies are applied element wise in the case of replicated arrays. The available strategies are:

<0>: The leader's value is broadcast at synchronization.

<?>: An arbitrary updated copy is chosen and broadcast.

<=>: No combining, the programmer is responsible for ensuring that the processors write an equal value.

<+>: All local copies are added together and is broadcast to the processors of the group. This also works with multiplication and bitwise AND/OR.

<foo>: A user defined method is used when combining. In NestStep-C this function need to be declared as

```
void foo(void * src1, void * src2, void * dest);
```

<strategy:variable>: This can be used to store the prefix sums of the shared variable that is used with this combine strategy in the **local** *variable*. As an example, let us look at how it works when using addition. The prefix sum is defined as $variable_i = \sum_{j=0}^{i-1} shared_j, \forall i$, where the indices refer to the processor they belong to. That is, *variable* is the sum of all updates of *shared* that belong to processors with lower rank than oneself. Here is an example of this for three processors:

	Processor 1	Processor 2	Processor 3
shared before	1	2	3
variable after	0	1	3
shared after	6	6	6

These combine strategies are used in conjunction with the *combine* keyword.

```
sh int a, b;
int c;
...
step {
...
} combine(a<+>, b<*:c>);
```

If we have a shared array that needs to be combined, but only need to combine a part of the array, we can specify the range by using array notation when using combine.

```
Step {
...
} combine(array[lower:upper]<+>);
```

Replicated data

Replicated shared variables

Shared variables are declared with the keyword **sh**. For example,

```
sh int a;
```

One may also specify which combine strategy the variables should be combined with at synchronization between processors at the declaration.

Here is an example of this:

```
sh<+> int a;
sh<?> int b;
```

Replicated arrays

Replicated arrays are used similarly as replicated variables. What is worth noting is that combine strategies work element-wise on arrays.

Distributed arrays

Only arrays may be shared in a distributed manner. The arrays elements may be distributed in blocks or cyclically.

Block arrays

When the array is distributed in blocks the elements are distributed evenly across the nodes. For example, a size of 1000 on four nodes would be distributed as 250 elements on each node. A size of 1013 would be distributed with 254, 253, 253, 253 elements respectively.
A block distributed array is declared by adding **</>** after the type. Example:

```
sh int b[N]</>;
```

Cyclic arrays

Cyclic arrays distributes the elements of the array cyclically in blocks of a specified block size. Cyclic arrays are declared using **<%>**. The block size is the number of elements to the right of **<%>**, which if omitted is considered one. For example, we show how to declare two arrays:

```
sh int c[5]<%>[10];
sh int d[9]<%>;
```

These arrays on four processors would be distributed as:

- node 0: c[0][0-9], c[4][0-9], d[0], d[4], d[8].
- node 1: c[1][0-9], d[1], d[5].
- node 2: c[2][0-9], d[2], d[6].
- node 3: c[3][0-9], d[3], d[7].

Reading/updating remote array elements

To read a range of values from a distributed array into a local array one can use *mirror*. In the following example *pre* is the local array.

```
sh int b[N]</>;
mirror(pre, b.range(lower_index, upper_index));
```

To write to a distributed array one uses *update*. Example:

```
sh int c[M]<%>[N];
update(c.range(lower_index, upper_index), pre);
```

To move elements between distributed arrays in a parent and a child group, one uses *importArray/exportArray* similarly to *mirror/update*.

4 Communication structure

All communication happens in a combine phase, except for remote reads/writes to distributed arrays.

Reading/writing to remote parts of distributed arrays is done after the combine phase. The problem of knowing how many read/write requests one node is going to receive from other nodes is solved by using an array that holds one integer for each node. For each request that a node sends to another, it adds one to the array element with the receiving node's index. This array is then combined using addition in the combine phase. We then read the element with the same index as the nodes rank, which will contain the number of read/write requests sent to it.

The way combining works is illustrated in Figure 3. The broad gray lines are distributed reads/writes. The tree used in the figure is a binomial tree of order 3.

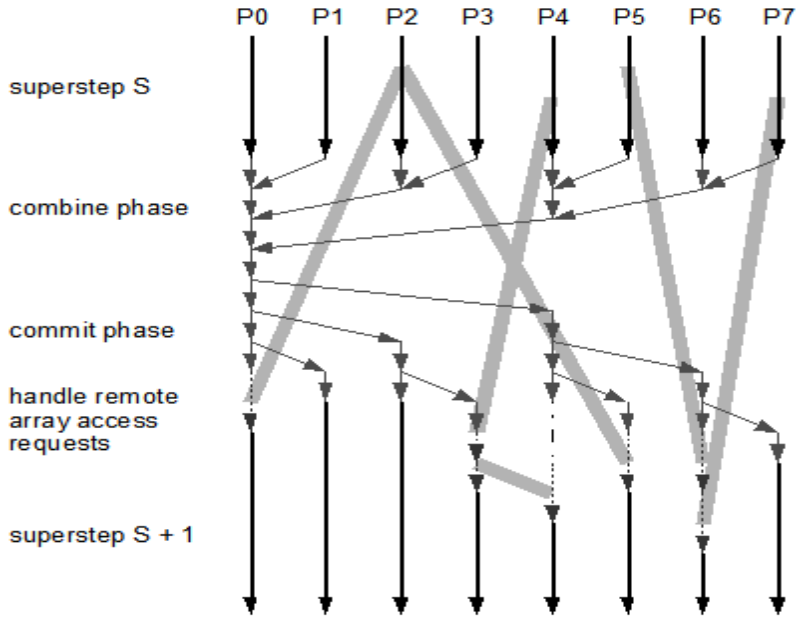

Figure 3: Combining in the NestStep system
Recreation of an illustration in [2].

The following definition of a binomial tree is taken from [5].

The binomial tree of order $k \geq 0$ with root *R* is the tree B_k defined as follows:

1. if $k = 0$, $B_k = B_0 = \{R\}$. That is, the binomial tree of order zero consists of a single node, *R*.
2. If $k > 0$, $B_k = \{R, B_0, B_1, \ldots, B_{k-1}\}$. That is, the binomial tree of order $k > 0$ comprises the root *R*, and *k* binomial subtrees, $B_0, B_1, \ldots, B_{k-1}$.

An illustration of a few binomial trees can be seen in Figure 4.

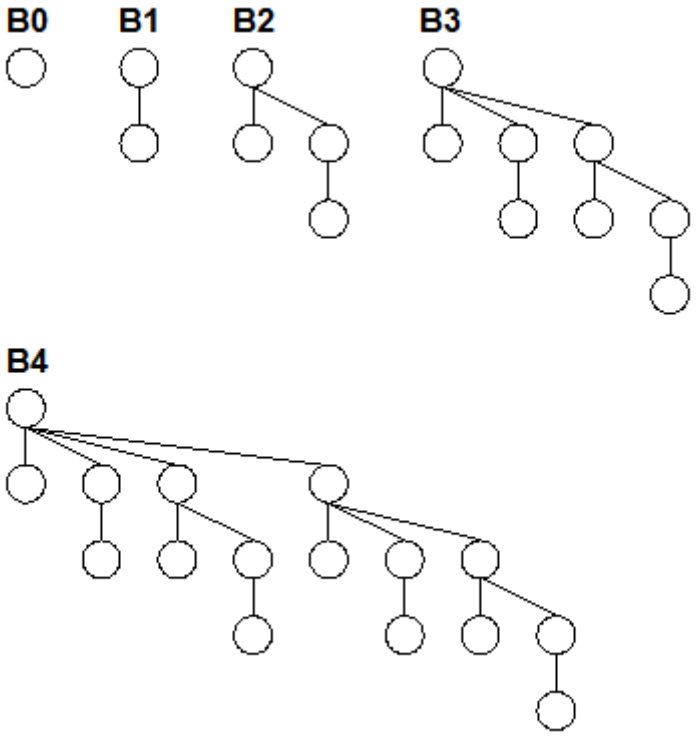

Figure 4: The first five binomial trees.
Recreation of an illustration in [5].

If we look at the the binomial trees, we can observe that if were to cut the branch from any node to its rightmost child, the trees below the node and the child would be identical. That is, if we would send a message to a child, afterwards the child and the parent would have an equal amount of nodes below them to distribute data to. This is, at least in the author's opinion, the most intuitive way to broadcast data fast.

There are trees similar to binomial trees that may be used in NestStep. These trees are associated with a fraction α that is less than one. A tree like this is represented in Figure 5, where T1 is number of nodes in the subtrees below the parent(excluding the rightmost tree with the child), and T2 is the number of nodes in the subtrees below the rightmost child. Then $\alpha = \frac{T2}{T1 + T2}$.

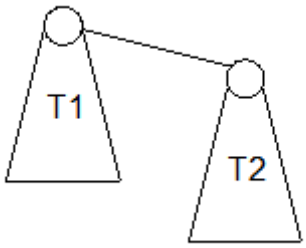

Figure 5

These trees are referred to as binomial trees with a fraction α. A normal binomial tree has fraction 0.5.

The NestStep-C system also support flat and D-ary trees. Flat trees are trees where one node is the root node and the rest of the nodes are children to that node.

A D-ary tree is a tree where an inner node has D children, where D is an integer. See Figure 6 for an illustration of a D-ary tree where D = 2.

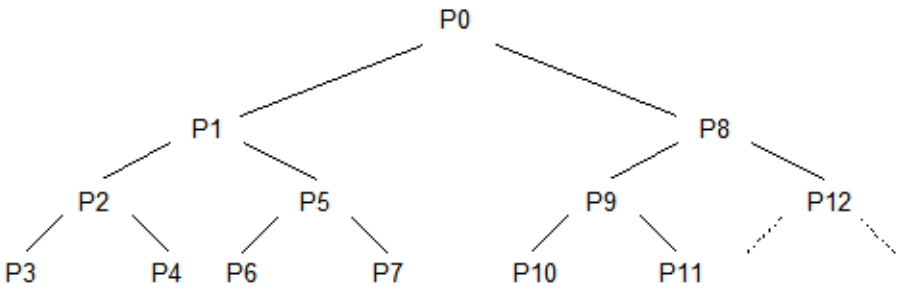

Figure 6: D-ary tree with D=2.
Recreation of an illustration in [1].

Analytically determining an optimal tree structure for a system is quite hard. One well known model for communication costs is the LogP model, that is presented in [6]. The model uses four parameters:

- L: L is the communication delay for propagating the message over the network.
- o: o is the communication overhead experienced when sending or receiving a message.
- g: g represents the communication bandwidth. This is modeled as the minimum time between two consecutive sends by a node, i.e. from the beginning of sending the first message to the beginning of sending the second message.
- P: P is the number of processors.

All of these parameters must be known to be able to use this model to predict the performance.

[7] discusses on how one might be able to find optimal trees using this

communication model.

We will not discuss this in length here, but will use two examples to illustrate that under different circumstances some tree structures may be better than others, and that it is worth investigating which yields the best performance. The objective is to broadcast a message from node 0 to all other nodes. To relate to NestStep, combining is a reduction, which is essentially a backwards broadcast except that every internal node perform some additional computation for each element transmitted(e.g. adding elements together). Since the time for sending an element over the network is much larger than processing it, we can ignore the processing time for each element when modeling the reduction phase. The commit phase is a broadcast. That is, we need a tree that is suitable for broadcasting.

As the first example, consider four nodes(i.e. P = 4) where L is five time units, and g and o is 1 time unit. If we use a flat tree, node 0 will send three messages, to one node after one time unit(the o parameter), to a second node two time units, and finally to a third node after three time units. The first node to receive its message will receive it after 7 time units.(five for the latency and two for the overhead from sending/receiving it.) That is, if we were not using a flat tree, first child would be ready to transmit to any eventual child nodes after 7 time units. As the message sent to node 3 from the root node occurred after 3 time units, there is no other tree that could do this faster than a flat tree. The flat tree for this situation is illustrated in Figure 7. It is clear that node 3 in the figure can not send a message to node 1 before node 0, so a different tree structure would perform worse.

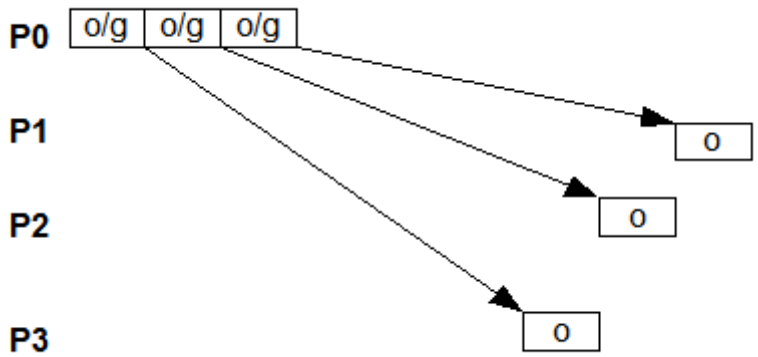

Figure 7: Broadcast with a flat tree using parameters from example one.

For a second example, let us change L and o to one time unit, and g to five time units. In this scenario, node 3 would be ready to send the original message to its eventual children after 3 time units.(2o + L.) At that time node 0 will have two time units left before it can start sending to node 2. (g – 3.) In this case it will clearly be better if node 3 sends the message to node 2 instead of node 0, that is, a binomial tree would perform better than a flat tree. Figure 8 illustrates this scenario with at binomial tree.

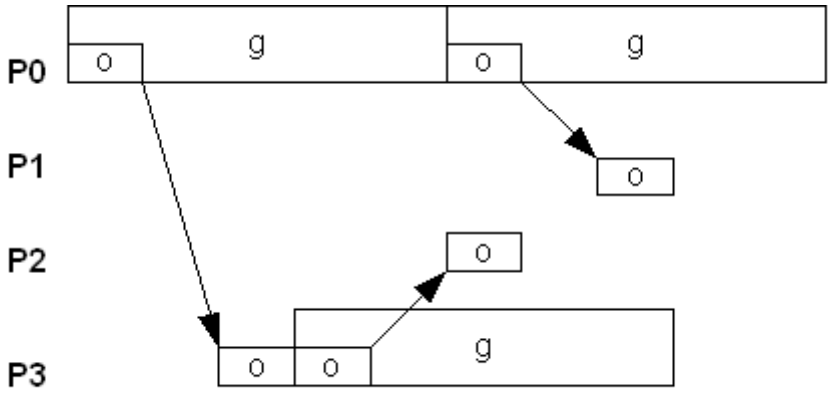

Figure 8: Broadcast with a binomial tree using parameters from example two.

A problem with the LogP model is that some of the parameters will depend on the message size. That is, we can not use the same values for different message sizes if we want a good approximation from the model. [8] introduces the LogGP model, which also takes into account the amount of data to be transmitted.

The default tree structure used in NestStep is a binomial tree structure. We will later show that this tree structure seems to have the most potential of a few selected candidate structures.

5 Project requirements

The aim for this project was to develop a runtime system for NestStep-C. As for most projects a set of primary requirements was set up in order to determine when the project can be considered to be finished. In addition to the primary requirements there was also two secondary requirements. These secondary requirements were to be met if time allowed.

5.1 Primary requirements

- The runtime system must support all of the language features mentioned in [1] and [2] except shared volatile variables(that has been dropped from NestStep in [2]).
  The features to be supported are:
  1. The step statement.
  2. The neststep statement.
  3. Shared variables and arrays. Shared arrays can be replicated or distributed.
  4. Combining for shared variables, arrays and objects.
- The communication system must support D-ary trees, where D is to be specified at compilation time or runtime.
- Accessing remote sections of distributed arrays must be handled as specified in [2], i.e. as in section 4.

- The system must not have any known bugs that prevent the use of the system.

5.2 Secondary requirements

- The best tree structure among the tree structures mentioned in section 4 should be obtained, empirically and/or by analysis.
- The runtime system should take advantage of hybrid architectures, where the nodes in the clustered supercomputer may contain SMPs.

We were only able to find the best tree structure empirically among these secondary requirements before the time allocated to the project ended.

6 Interface

This section describes how to interface with the NestStep runtime system, if one were to write a front end for NestStep-C. It is not a description of how one writes actual NestStep programs. It is primarily written for someone interested in writing a front end for NestStep, while others may skip it at no loss.

6.1 Identifiers of shared data

In order to be able to identify shared data among different nodes each shared variable/array is given a name. This name is a struct named Name and has two fields; procedure and relative.

procedure should be set to 0 if it is a global variable, otherwise it should be set to a value that is unique for the function it is declared in, i.e. all names in a function share the same procedure value. Relative is an integer that is to be unique within the function. It should be noted that there is a path field in the Name struct, it is however handled by the runtime system and is of no concern to the front end developer. Example:

```
void someFunc(void) {
    Name name;
    name.procedure = 4; // unique for this function
    name.relative = 0;
    ...
    one = new_ShVar(name, IVAR, pointer_one);
    name.relative++;
    two = new_ShVar(name, IVAR, pointer_two);
    name.relative++;
    ...
}
```

6.2 Shared variables

Shared variables are implemented using a normal variable for the data storage and a wrapper that holds it. To create the wrapper one uses the function

```
ShVar * new_ShVar(Name name, int type, void * base);
```

The arguments are:

- name: a name as discussed above.
- type: An integer that specifies the type of the variable. Possible entries are IVAR(int), FVAR(float) and DVAR(double).
- base: a pointer to the normal variable.

The memory used by the wrapper should later be reclaimed by using

```
void free_ShVar(ShVar * var);
```

For example, when a shared integer is declared it would be translated as:

```
...
int shared;
ShVar * _shared;
...
_shared = new_ShVar(unique_name, IVAR, &shared);
...
free_ShVar(_shared); /* free the memory the wrapper consumes */
...
```

It is possible to use pointers with this shared memory construct, by using the base member of the ShVar struct. Obviously the data that the pointer points to is what is shared.

```
double one, two;
ShVar * pointer;
...
/* initialize the pointer before use */
pointer = new_ShVar(unique_name, DVAR, NULL);
...
pointer->base = &one;
...
pointer->base = &two;
...
free_ShVar(pointer);
...
```

6.3 Replicated arrays

Replicated arrays are used in a similar manner to shared variables. Replicated arrays needs to have an existing array. To create and free a replicated array one uses

```
RepArray * new_RepArray(Name name, int type, int size, void * base);
```

and

```
void free_RepArray(RepArray * array);
```

The arguments are the same as for a shared variable, except that the size of the size of the array needs to be specified. Example:

```
int array[1000];
RepArray * _array;
...
_array = new_RepArray(unique_name, IVAR, 1000, array);
...
free_RepArray(_array);
...
```

To set all elements in a replicated array to zero one can use

```
void set_RepArray_zero(RepArray * _array);
```

6.4 Distributed arrays

Distributed arrays do not use preexisting data structures, they create their own memory on the heap.

Block arrays distribute the memory evenly across the nodes. For example, a size of 1000 on for nodes would be distributed as 250 on each node. A size of 1013 would be distributed as 254, 253, 253, 253 elements respectively. To create and free a block array one uses

```
BlockDistArray * new_BlockArray(Name name, int type, int size);
```

and

```
void free_BlockArray(BlockDistArray * array);
```

An example on how to create/free a block array:

```
...
BlockDistArray * block_array;
block_array = new_BlockArray(unique_name, IVAR, 1000);
...
free_BlockArray(block_array);
...
```

To set the local elements of a block array to zero,

```
void set_BlockArray_zero(BlockDistArray * _array);
```

can be used. To take a global index and turn into a local index,

```
int global_local_index_b(BlockDistArray * _array, int _index);
```

is supplied. If the global index is not among the local elements -1 is returned. To take a local index and turn into a global index,

```
int local_global_index_b(BlockDistArray * _array, int _index);
```

can be used.

Cyclic arrays distribute the elements cyclically in blocks of a minimum block size. A size of 50 on four processors with a block size of 10 would be distributed as:

- node 0: 0-9, 40-49
- node 1: 10-19
- node 2: 20-29
- node 3: 30-39

The size needs to be able to be divided by the block size without leaving a remainder.

To create and free a cyclic array one uses

```
CyclicDistArray * new_CyclicArray(Name name, int type, int size, int
blocksize);
```

and

```
void free_CyclicArray(CyclicDistArray * array);
```

An example on how to create/free a cyclic array:

```
...
CyclicDistArray * cyclic_array;
cyclic_array = new_CyclicArray(unique_name, IVAR, 100, 10);
...
free_CyclicArray(cyclic_array);
...
```

To set the local elements of a cyclic array to zero,

```
void set_CyclicArray_zero(CyclicDistArray * _array);
```

can be used. To take a global index and turn into a local index,

```
int global_local_index_c(CyclicDistArray * _array, int _index);
```

is supplied. If the global index is not among the local elements, -1 is returned. To take a local index and turn into a global index

```
int local_global_index_c(CyclicDistArray * _array, int _index);
```

can be used.

6.5 Group management

Before running **any** function related to NestStep(including the ones creating the data structures), a call to

```
void NestStep_init(int * pargc, char *** pargv);
```

needs to be made. The arguments are the addresses of the arguments to your main function. Before the program ends a call to the following function needs to be made in order to clean up. After it has been called no more functions related to NestStep are to be called.

```
void NestStep_finalize(void);
```

```
int NestStep_get_rank(void);
```

Retrieves the node's rank in the current group.

```
int NestStep_get_size(void);
```

Retrieves the size of the node's current group.

At the beginning of a step the following function needs to be called.

```
void NestStep_step(void);
```

To end a step the following function needs to be called.

```
void NestStep_step_end(void);
```

```
void NestStep_neststep(int k);
```

splits the current group into k groups.

To split the current group into k groups where the amount of nodes in each group is different

```
void NestStep_neststep_w(int k, float weight[]);
```

is used. weight is an array with k floats where the sum of whole array is 1.

To split the current group into k groups where the value of an integer determines the final subgroup for a node

```
void NestStep_neststep_c(int k, int color);
```

is used. A node ends up in the group that has the same value as color.

```
void NestStep_neststep_end(void);
```

merges groups that has been split previously.

To examine which group a node ended up in, the following function can be used.

```
int NestStep_get_group_id(void);
```

6.6 Combining

Before a step is ended the following function needs to be run if any variables or arrays are to be combined, or there is a possibility that any remote read/write request have been made.

```
int NestStep_combine(CombineList * _clist, ArrayHolder * _holder);
```

The parameters should be NULL. As an example:

```
...
NestStep_step();
{
...
}
NestStep_combine(NULL, NULL);
NestStep_step_end();
```

NestStep_combine uses the current group for combining, so it should be used after *NestStep_neststep_end* instead of before.

A few macros has been provided to choose how to combine items. A few functions below use these so they will be listed here:

- ADD: Standard addition of items.
- MULT: Standard multiplication of items
- AND: The elements are AND:ed with each other. Only for integers.
- OR: The elements are OR:ed with each other. Only for integers.
- FUNC: A special function is to be provided.
- LEADER: The value(s) of the node with rank 0 in the the group are used.
- UPDATED: The values of an updated element are used. It is not specified from which node with updated values they are taken from.

If a shared variable is to be combined at the end of a step it needs to be attached to the CombineList that is to be used in the combine phase. The function that does this is:

```
int NestStep_shvar_attach(ShVar * var,
    int combine_strategy,
    void (*combineFn)(void *, void *, void *),
    int prefix_tag,
    void * prefix_base,
    int updated,
    CombineList * clist);
```

The parameters to the function are:

- var: The pointer returned by new ShVar.
- combine strategy: An integer that determines how the variable should be combined.
- combineFn: If an own function is to be used for combining the function pointer should be passed here. If no special function is to be used NULL should be passed.
- prefix_tag: If a prefix computation such as the one described in section 3.3 is to be performed the macro PREFIX needs to be supplied for this value. If no prefix computation is necessary the macro NO_PREFIX should be supplied.
- prefix_base: a pointer to the memory location where the prefix sum elements should be stored.
- updated: This parameter is a remain from an early version of the NestStep system. The value supplied here does not matter.
- clist: the CombineList that is to be used. This should be NULL to use the default combine list that belongs to the group, which is what should be used.

Example:

```
...
/* add the variables and let local_var hold the prefix sum element */
NestStep_shvar_attach(shared, ADD, NULL, PREFIX, &local_var, 0,
      NULL);
...
```

Replicated arrays need to be attached in the same way as shared variables. For this

```
int NestStep_RepArray_attach(RepArray * array, int _combine_strategy,
    void (*_combineFn)(void *, void *, void *), int prefix_tag,
    void * prefix_base, int _updated, CombineList * _clist);
```

is used. The arguments are the same as for a shared variable, except of course that an replicated array is to be passed as the first parameter. To specify the range to be combine,

```
int NestStep_RepArray_range(RepArray * _array, int _lower, int _higher);
```

is used. The range is reset to normal after the next combining.

To gather/scatter the shared elements of a replicated array to a

distributed array the following functions are provided. The sizes of the arrays have to match.

```
/* scatters a replicated array to a block array */
int NestStep_scatter_rep_to_b(RepArray * reparray, BlockDistArray *
barray);

/* scatters a replicated array to a cyclic array */
int NestStep_scatter_rep_to_c(RepArray * reparray, CyclicDistArray *
carray);

/* gathers a block array to a replicated array */
int NestStep_gather_b_to_rep(BlockDistArray * barray, RepArray *
reparray);

/* gathers a cyclic array to a replicated array */
int NestStep_gather_c_to_rep(CyclicDistArray * carray, RepArray *
reparray);
```

To issue read/write requests the following functions are provided:

```
/* reads all elements with indexes lower to higher into buffer */
int NestStep_read_request_b(BlockDistArray * array, int lower, int higher,
      void * buffer, CombineList * clist);

int NestStep_read_request_c(CyclicDistArray * array, int lower, int higher,
      void * buffer, CombineList * clist);

/* writes to all elements with indexes lower to higher from buffer */
int NestStep_write_request_b(BlockDistArray * array, int lower, int higher,
      void * buffer, CombineList * clist);

int NestStep_write_request_c(CyclicDistArray * array, int lower, int higher,
      void * buffer, CombineList * clist);
```

The clist parameter should be NULL.

To import/export arrays into a subgroup the following functions can be used. The arrays that are imported from or exported to need to be available in the parent group. If *active* is non-zero elements are imported/exported, otherwise the processor just supplies/receives values.

```
int importArray_rep_to_rep(RepArray * from, RepArray * to,
      int lower, int higher, int active);

int exportArray_rep_to_rep(RepArray * from, RepArray * to,
      int _lower, int higher, int active);

int importArray_b_to_b(BlockDistArray * from, BlockDistArray * to,
      int _lower, int higher, int active);

int exportArray_b_to_b(BlockDistArray * from, BlockDistArray * to,
      int _lower, int higher, int active);

int importArray_b_to_rep(BlockDistArray * barray, RepArray * rarray,
      int lower, int higher, int active);

int exportArray_rep_to_b(BlockDistArray * rarray, RepArray * barray,
      int lower, int higher, int active);
```

```
int importArray_c_to_rep(CyclicDistArray * carray, RepArray * rarray,
    int lower, int higher, int active);

int exportArray_rep_to_c(CyclicDistArray * rarray, RepArray * carray,
    int lower, int higher, int active);
```

In order to change the trees used for combining, calls to any of the following three functions can be made:

```
void change_tree_bin(double fraction);
void change_tree_dary(int D);
void change_tree_flat(void);
```

These calls needs to be followed by

```
void rebuild_tree();
```

7 Example applications and implementations

We are going to look at three example applications, parallel prefix sums, mergesort and Gaussian elimination. In appendix D all code presented here is supplied without comments.

7.1 Parallel prefix

Application

The parallel prefix sums that our example is going to calculate is of the following form:
An input in form of an array of integers is going to be summed together as $answer[i] = \sum_{j=0}^{i} input[j], \forall i$.

Example:

Input = [1, 2, 3, 4], Output = [1, 3, 6, 10]

Implementation

Implementing a parallel prefix summation is almost as trivial as implementing it in a sequential language.

The function *parprefix* takes a distributed block array as an argument. We will assume that the number of values to be computed will be evenly divisible by the number of processors.

```
void parprefix(sh int</> a[]) {
    int i, j;                   // loop counters
    int pre[a->size()/#];       // for storage of temporary local sums
    int myoffset;               // prefix offset for this processor
    sh int sum;
    ...
}
```

Initially we declare a few variables and an array, which should be pretty self-explanatory together with the comments. The rest of the function is written using two steps, that should be located after each other

instead of the three dots above.

Step 1:

```
step {
      sum = 0;
      j = 0;
      forall(i, a) {
         sum += a[i];
         pre[j++] = sum;
      }
} combine(sum<+:myoffset>);   /* combine sum and store */
                              /* prefixes in myoffset */
```

In the first step we calculate a prefix sum into the local array. We specify *myoffset* to be used for storing a prefix sum value after combining.

Step 2:

```
step {
      j = 0;
      forall(i, a) {
         a[i] += pre[j++] + myoffset;
}
```

Finally, in the second step, we add the prefix variable *myoffset* and the local array with the prefix values to *a*.

7.2 Mergesort

Application

Mergesort works as follows:

1. Divide phase: First there is a divide phase, when the array of data is recursively divided into two smaller arrays, until there only remains one element in each array.
2. Conquer phase: When we after the divide phase merge two arrays of data together, we keep the arrays sorted at all times. This is easily done by always taking the smallest values one at at time while moving the elements to the final array.

Figure 9 illustrates this process.

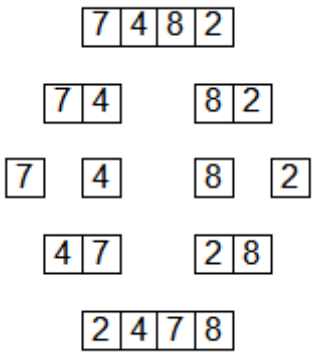

Figure 9: The principle behind mergesort.

A more thorough description of mergesort is available in [9].

The mergesort algorithm that we are going to use sorts a distributed block array. The algorithm only uses some limited parallelism. In this case we are going to let each processor sort a subsection of an input array with unordered integers, and then perform the conquer phase on the ordered subsections. We omit the initial divide phase as the distributed block array already is divided into suitable parts to be sorted. We will use the standard quicksort function in the C library to sort these initial subsections.

Figure 10 shows how the algorithm will work for four processors. The lines represents which parts that are sorted. When everything is sorted we omit the lines.

1. P0 and P2 form a subgroup that imports the array the original group.
2. P0 and P2 then merge these arrays.
3. P0 forms a subgroup that imports the array.
4. P0 merges the sorted parts.
5. P0 exports the array to parent group.
6. P0 and P2 export the array to the parent group.

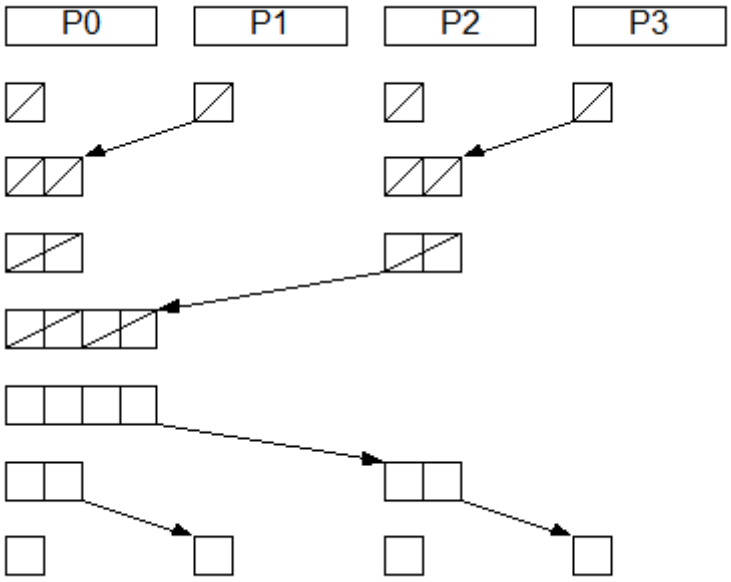

Figure 10: Merging of arrays using version one.

The reason for using this algorithm is for testing that nested parallelism and importing/exporting arrays works.

We assume that the number of elements to be sorted is able to be distributed evenly, i.e. all processors starts with the same number of elements.

Implementation

The function that calls mergesort is assumed to have sorted the local parts of the array before calling mergesort.

```
...
qsort(a->base, a->local_elements, sizeof(int), compare_ints);
mergesort(a);
...
```

a is a block array, and *local_elements* is the number of elements of the array that is located on the processor.

```
/* assume that local parts are already sorted */
void mergesort(sh int a[]</>)
{
      if(# == 1) // nothing left to merge
          return;

   neststep(2, @ = $ % 2) {
          sh int temp_array[a.size()]</>;

          /* The following statement needs to be executed by all
          processors that have some elements of the original array.
          The last parameters is a boolean flag that tells if the processor
          should import the values, or simply provide its values to
          other processors of another subgroup */

          importArray(a, temp_array, 0, a.size() - 1, 1-@);

          if(@ == 0)
          {
              merger(temp_array); // sort the imported array
              if(# > 1) // if there are more than one processor in the group
                  mergesort(temp_array);
          }

          exportArray(temp_array, a, 0, a->size – 1, 1-@);
   }
}
```

The reason for modulo 2 in the *neststep*-statement for determining group is because then half of the array to be imported is already on the processors.(See Figure 10.) *merger* merges the lower and upper local parts of *temp_array*. If there are more than one processor in the group left we need to call the *mergesort*-function recursively until we end up with a single processors that has the whole sorted array, after which of course the recursion unravels and we export the sorted parts back.

The actual merging of the sub arrays is done by the *merger* function.

```
void merger(BlockDistArray * a)
{
	int i;
	int l_c;
	int u_c;
	int buffer[a->local_size];

	l_c = a->lower; // counter for the lower section of the local array
	u_c = l_c + a->local_elements / 2; // counter for the upper section

	for(i = 0; i < a->local_size; i++) {
	    if(a[l_c] < a[u_c] && l_c < a->local_size / 2) {
	        buffer[i] = a[l_c];
	        l_c++;
	    }
	    else if(u_c < a->local_size) {
	        buffer[i] = a[u_c];
	        u_c++;
	    }
	    else {
	        buffer[i] = a[l_c];
	        l_c++;
	    }
	}
	a[a->lower:a->lower + a->local_size – 1] = buffer[:];
}
```

7.3 Gaussian elimination

Application

In this application we will solve an equation system using Gaussian elimination with pivoting, in order to avoid numerical errors in case the row used for elimination would have a small value in the column that is eliminated. The equation system is given as $A*x=b$, where A has the dimensions $N*N$, while x and b are vectors of length N.

We will also run some tests using using a simplified algorithm, where we will skip the pivoting and back substitution in order to calculate x.

In Figure 11 we can see how the matrix A and vector b are transformed by the Gaussian elimination.

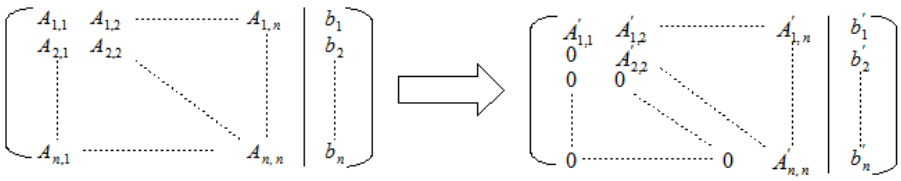

Figure 11: Gaussian elimination performed on A and b.

Implementation

The equation system to be solved is as mentioned previously given by the matrix A, and the vectors x and b, so that $A*x=b$. The result will be stored in x.

A, b and x are originally replicated arrays given by the function call.

We will later scatter them to cyclic arrays.

```
int gaussian(sh double A[], sh double b[], sh double x[], int N) {
```

First we declare some variables that will be used later.

```
double max = 0;
int largest = 0;
int row;
sh double[N]<%>[N] _A;
sh double[N]<%>[1] _b;
sh double b_tmp;
sh double b_elim;
sh double[N] tmp;
sh double[#] sizes;
sh double[N] elim;

int i, j, k, l;
```

The first thing that is done is to scatter the values in A and b to the cyclic arrays _A and _b. The Gaussian elimination will be performed on the cyclic arrays.

```
// scatter A and b
step {
    scatter(A, _A);
    scatter(b, _b);
}
```

Before we start we see which local values are the largest in the first column, in order to be able to pivot the rows in _A and _b later. We use the forall2 statement to see which row has the largest value in column 0. We store the locally largest values in a shared array called sizes. This will be used in the beginning of the main loop. This calculation will be repeated in the end of the main loop for the column *k* for the remaining rows of the matrix. We do this in two places in the code to avoid unnecessary communication.

```
// find largest element among own kind
max = 0;
step {
    forall2(i, _A) {
        if(fabs(_A[i][0]) > max) {
            max = fabs(_A[i][0]);
            row = i;
        }
    }
    sizes[$] = max;
}
```

We loop for N – 1 times, once for each column that needs to have elements eliminated.

```
for(k = 0; k < N - 1; k++) {
```

In the first step of the main loop each node checks whether it has the largest value in the active row/column. If it is true, then the row is copied to the shared array elim and the corresponding value in _b to the shared variable b_elim. If the maximum value occurs more than once, the processor with the lowest rank is considered to have the largest value. This array and variable will later be used in the elimination step.

```
// move largest array to elim or zero
step {
    largest = 1;
    for(l = 0; l < #; l++) {
        if(fabs(sizes[l]) > max || (fabs(sizes[l]) == max && l < $)) {
          largest = 0;
          break;
        }
    }
    if(largest == 1) {
        elim[:] = _A[row][:];
        b_elim = _b[row];
    }
```

If the processor owns the active row, we need to store it in the shared structures tmp and b_tmp, in order to be able to swap it with the row that has the largest element.

```
    // move swapped row to tmp or zero
    if(owned(_A[k][0])) {
        tmp[:] = _A[k][:];
        b_tmp = _b[k];
    }
}
step {
```

If we own the active row, we copy the elimination row into its place.

```
    // copy elim to correct row
    if(owned(_A[k][0])) {
        _A[k][:] = elim[:];
        _b[k] = b_elim;
    }
```

If we had the largest value, we copy tmp and b_temp to the row that had the largest value.

```
    // copy tmp to correct row
    if(largest == 1) {
        _A[row][:] = tmp[:];
        _b[row] = b_tmp;
    }
```

We eliminate the values below row *k* in column *k*.

```
    // eliminate using elim
    forall2(i, _A, k + 1, _A.size() - 1, 1) {
          double modifier = -_A[i][k] / elim[k];
          _b[i] += modifier * b_elim;
          _A[i][k:N - 1] += modifier * elim[k:N – 1];
    }
```

After the elimination we locate which row has the largest value in column *k*+1. We do this here instead of the beginning of the loop, because then we only have to combine twice in the main loop, and thus avoid unnecessary communication.

```
            // find largest element among own kind
            max = 0;
            forall2(i, _A, k + 1, _A.size() - 1, 1) {
                if(fabs(_A[i][k + 1]) > max) {
                  max = fabs(_A[i][k + 1]);
                  row = i;
                }
            }
            sizes[$] = max;
        }
    }
```

When we have finished the Gaussian elimination we gather the results.

```
    // gather
    step {
        gather(_A, A);
        gather(_b, b);
    }
```

Finally, each node calculates the values of x, since doing this in parallel is hardly worth the effort. Doing this in serial requires about $2*(N^2+N)$ floating point instructions.(Excluding the cost of looping.) Calculating one value and broadcasting it to other nodes so that they may begin their calculation of other values would relate to about $2*(N^2+N)*\frac{cost\ of\ N\ combines}{p}$ floating point instructions, where p is the number of processors. This is undesirable since $\frac{cost\ of\ N\ combines}{p} > 1$ by far. Trying to decrease the cost of combining by calculating several values between each combine is futile as well, since the amount of parallelism will decrease by the same amount and the expression will remain the same.

```
    // calculate x
    for(i = N - 1; i >= 0; i--)
    {
        double sum = 0;
        for(j = N - 1; j > i; j--)
            sum += x[j] * A[i][j];
        x[j] = (b[i] - sum) / A[i][j];
    }
}
```

The simplified version is very similar except that we do not pivot and calculate x. A range for which values to update when sharing data by using shared arrays is added. The source code for that can be found in Appendix D.

8 Evaluation

All applications were benchmarked using a driver program that ran the respective functions a number of times. In all test runs, an initial call to each respective function was made to set up the distributed system. This is because there is an initial cost for initializing the nodes, and the initialization is not made before the MPI processes are launched on the distributed system. This initial run is not weighed in to the average execution time. The reason for excluding it is that for real scientific

computations the initial cost is negligible, but for short test runs it can sway the average quite a bit.

The Monolith cluster at the National Supercomputer Centre in Linköping, Sweden, was used to evaluate the NestStep runtime system. Monolith has two Intel Xeon processors at 2.2 Ghz and 2 Gigabyte primary memory per node. The MPI library used is ScaMPI, and the network is a high bandwidth/low latency SCI network. The operating system used is a Red Hat distribution with Linux kernel 2.4.

Additional software used is Tlib[4] for the group management of the NestStep runtime system.

All programs were compiled using Intel's C compiler with the optimization flag -O2.

Unfortunately, when both Xeon processors are used on a node cache conflicts may occur if two data sections worked upon have “wrong” offset from each other. The author was unaware of this fact when the testing of the system began, and two processors per node were used. Therefore, some of the tests give poor results. We will see a significant performance boost when only using one processor per node, and twice the amount of nodes instead. It is normally possible to get around this issue by making sure that data has suitable offsets from each other. The runtime system does not offset any data at this point. However, even though some results are affected by this cache conflict, the relative performance of different tree structures should be valid, since it is unlikely that these cache problems would affect the time for sending data over the network. It is quite probable that the differences would be larger, as the computation time would decrease and the communication time remain relatively the same.(There could be some speedup in the handling of data, i.e. putting data in the send buffer, committing the data to variables/arrays.)

Regardless of whether one or two processor per node were used, each processor is in charge of one MPI process. The NestStep-C runtime system only considers MPI processes, which means that using one or two processors per node is not different from the NestStep runtime system's point of view.

For all algorithms the parallel versions were used to measure the time consumption for one processor, i.e. all speedup values are relative.

In the graphs in this section, abbreviations such as D-X, B-X and Flat are used. D-X stand for a D-ary tree where X equals the child nodes per inner node, B-X stands for a binomial tree where X is the fraction as discussed in section 4. Flat stands for a flat tree structure.

For all gathered data, Appendix C is supplied.

8.1 Parallel prefix

The parallel prefix sum is not that interesting as a performance benchmark. The amount of communication is about as little as one can get in a parallel program, and all tree structures should be able to scale

well, at least as long as a reasonable amount of processors is used. (I.e. $p << N$.) This algorithm was only tested with three tree structures, and as can be seen in Figure 12, it scales as expected. It does not quite reach a perfect speedup, but the execution times are so low(about 0.14 seconds for 32 processors) for parallel prefix sums that one can not expect the required communication to go unnoticed.

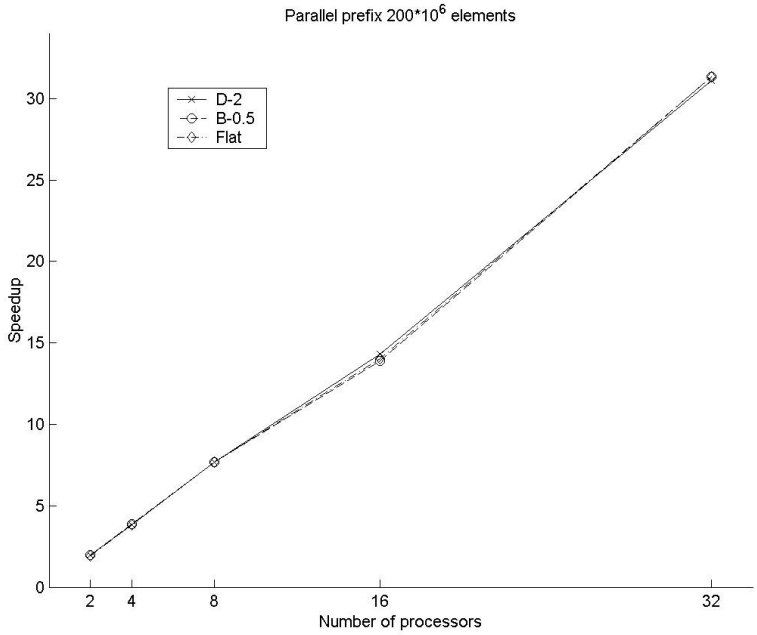

Figure 12: Speedup for the parallel prefix algorithm.
An average of ten runs is used.
One processor per node were used for these measurements.

8.2 Mergesort

The mergesort program was mainly written to test that certain features of NestStep works, namely nested parallelism and importing/exporting distributed arrays between parent and child.

The programs send very large amounts of data while importing/exporting data, and the amount of data increases with increasing number of processors, so it is expected that good speedups will only be achieved when each node has a lot of local elements to sort, compared to the global merge work. As can be seen in Figure 13 and 14, this is exactly the case. The speedup is initially good for a low number of processors. The speedup is also improved when the amount of elements is increased.

These tests were run using two processors per node, and probably suffered from the cache conflict, as both the parallel prefix computation and Gaussian elimination suffered from this when using two processors per node with them. This is not certain as no test runs were made with only one processor per node for mergesort.

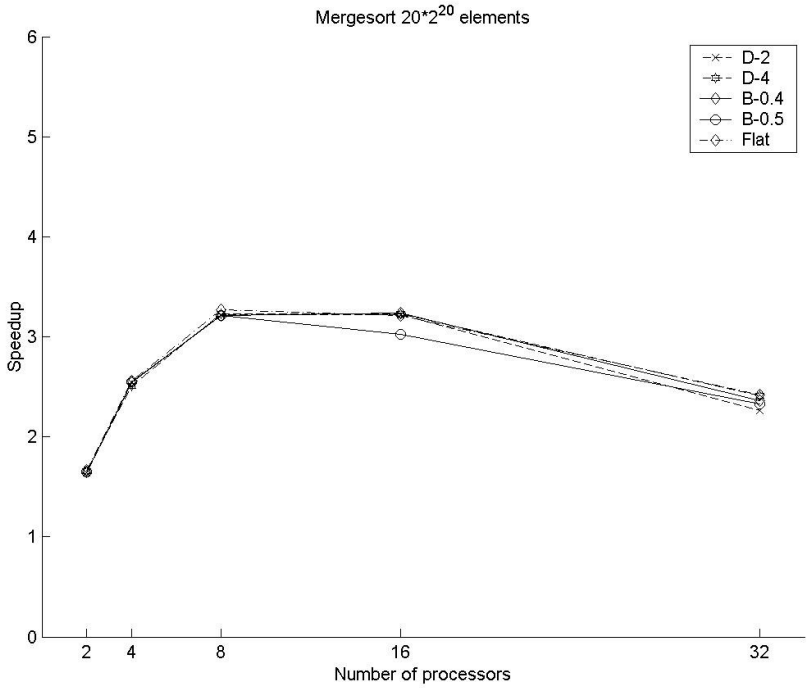

Figure 13: Speedup for the mergesort.
An average of 20 runs is used.

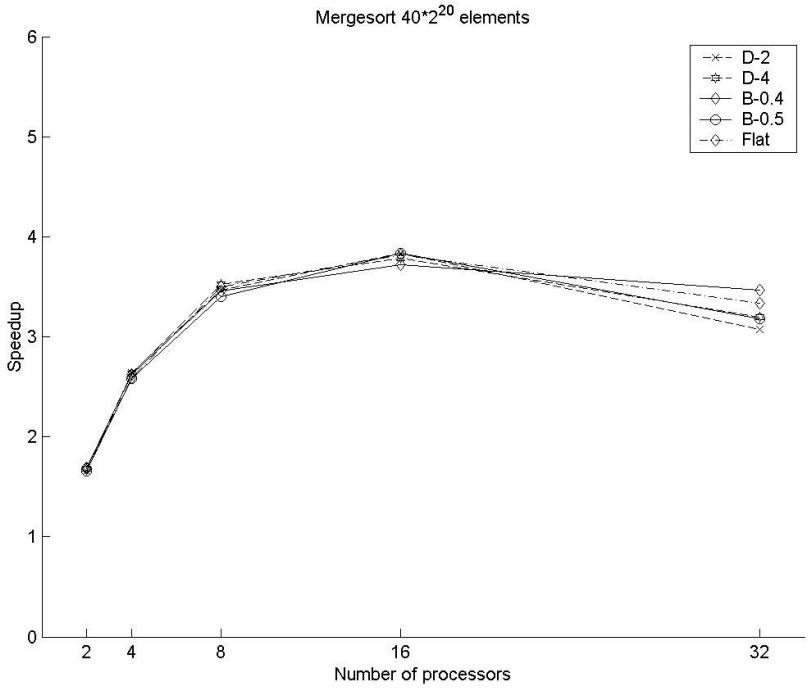

Figure 14: Speedup for mergesort.
An average of 10 runs is used.

Since most of the overhead is from point-to-point communication it is hard to determine the efficiency of the NestStep runtime system from this, or to determine performance of different tree structures.

8.3 Gaussian elimination

Gaussian elimination with pivoting

The Gaussian elimination application makes heavily use of the NestStep runtime system, as the algorithm combines twice in each iteration of the outer loop, sending about 2*N elements each time(since we do not specify a range in this version). With this amount of data and frequency of combines we should be able to detect some significant differences in performance among different tree structures, if there are any.

The test runs for this version of Gaussian elimination were run on two processors per node, and it is expected that the speedup will be higher on a system without cache conflicts.

Figure 15 shows the speedup of an equation system where N=2000 was solved, that is, *A* had the dimensions 2000*2000. We can see that the binomial trees perform a little better than the D-ary trees, as one might have suspected.(As discussed in section 4.) The flat tree performs poorly, which is not surprising, considering we have one node as a bottleneck.

Increasing N to 4000 gives a better speedup than for N = 2000. The results from this can be seen in Figure 16. The gap between the D-ary trees increases a bit, and gives further indication of that binomial trees are better. For N = 4000 only two runs was averaged. This is because of the quite long computation time for this value, around 200 seconds/run for one processor. This is a problem due to the limited CPU time for this project on the cluster, and it is hard to get the jobs scheduled for many processors when a lot of time is requested.

However, as the execution time is quite long, and we combine around eight thousand times, the variance caused by the network should not be that large.

In Figure 17 we can see the speedup for some more trees for 8, 16 and 32 processors. The binomial trees group together, with the fraction 0.5 being the best. The D-ary trees are quite close to each other, with D=4 being the best among those.

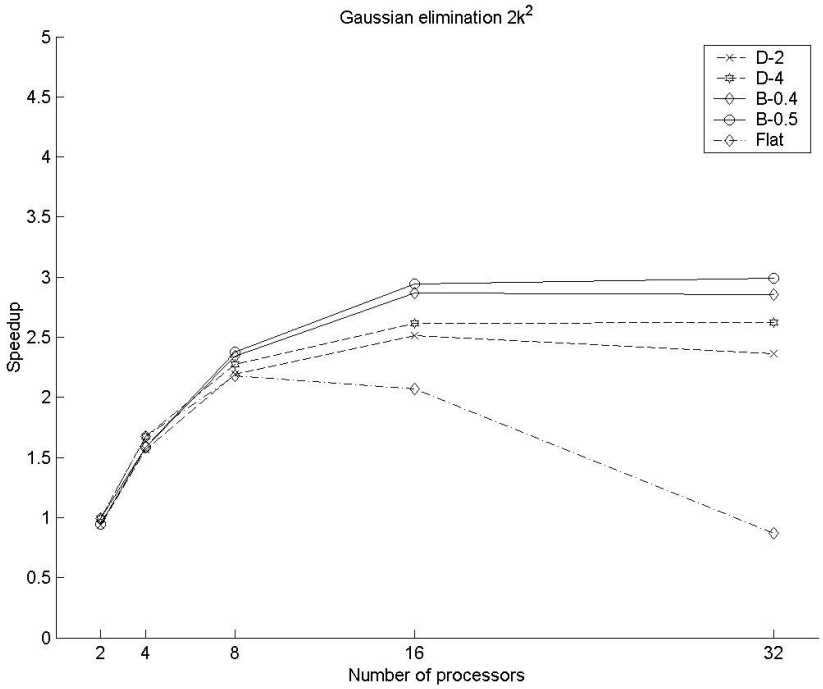

Figure 15: Speedup for Gaussian elimination. An average of 20 runs is used, except for the flat tree, which was an average of five.

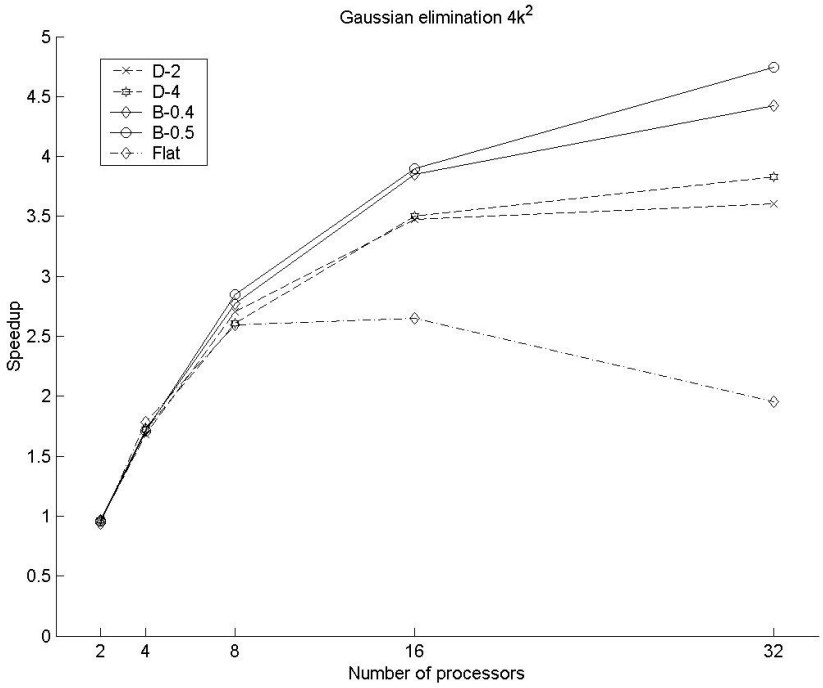

Figure 16: Speedup for Gaussian elimination. An average of 2 runs is used.

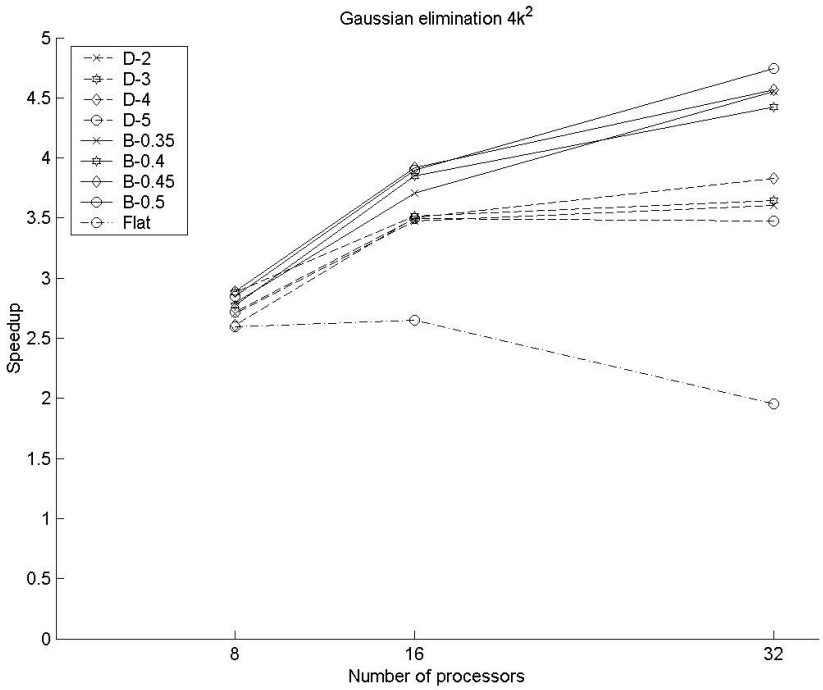

Figure 17: Speedup for Gaussian elimination. An average of 2 runs is used.

Simplified Gaussian elimination

In Figure 18 we can see the results from the simplified Gaussian elimination.

The speedup is quite high compared to the first Gaussian elimination, which is expected since pivoting and backward substitution is removed, and for combining we added a range for the shared array in order to avoid sending data that is not used. The tests for the simplified version were also run on one processors per node, compared to two processors per node for the non-simplified version.

The maximum speedup is slightly over 17 for 32 processors, which might be considered slow for such a simple Gaussian elimination. This is probably caused by three things:

- It is not possible to overlap communication and computation in NestStep. By using MPI for example, one could send the elements asynchronously while beginning the elimination.
- When we get near the end of the elimination, each processor has very few rows to do calculations for. That is, we get a lot of communication but very little computation(just touching the network costs several thousands of cycles). Normally one may specify larger block sizes, and not use all processors for the end, but since the entire group is involved in every combine, this is not likely to increase the performance for NestStep. One possibility might be to create a subgroup that take over the computation when the remaining matrix gets to small.

- Only the commit phase when combining is really needed. In other implementations one might just broadcast the elements needed for the elimination from the node where these elements are local.

However, if we consider how short and easy the code for the simplified version is, the performance is not to bad.

The difference in performance between trees is reduced for the simplified version, as is quite natural since the amount of supersteps is cut in half.(We only have to combine half as many times as well, which means less communication.)

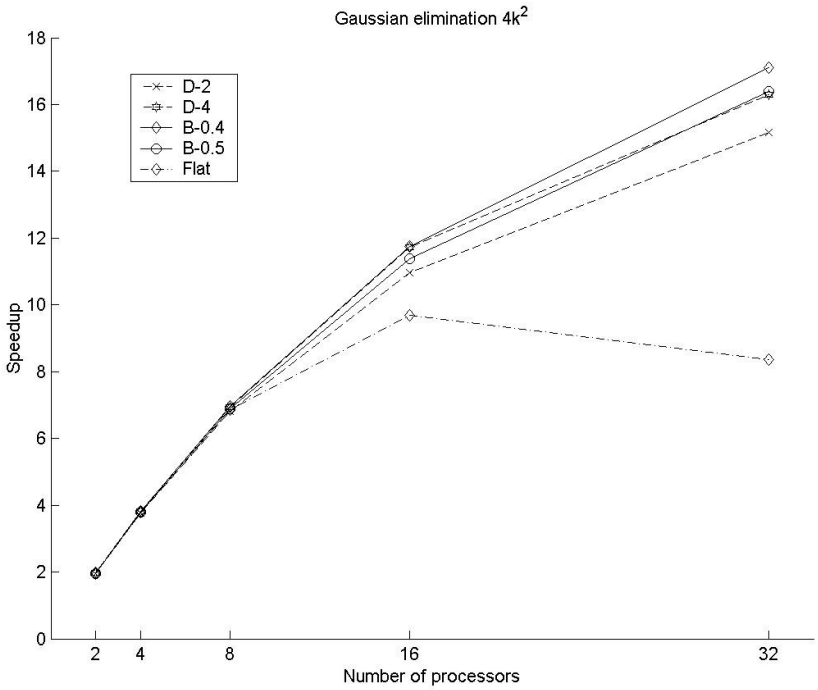

Figure 18: Simplified Gaussian elimination.
An average of 2 runs is used.

In Figure 19 we can see the difference between using one processor per node compared to using two processors per node for a binomial tree with the fraction 0.5. By examining the figure, we can see that using one processor yields better performance even if we were to use half the amount of processors. That is, it is better to ignore the second processor on each node, because using it will not give any speedup, instead it will make the computation run slower.

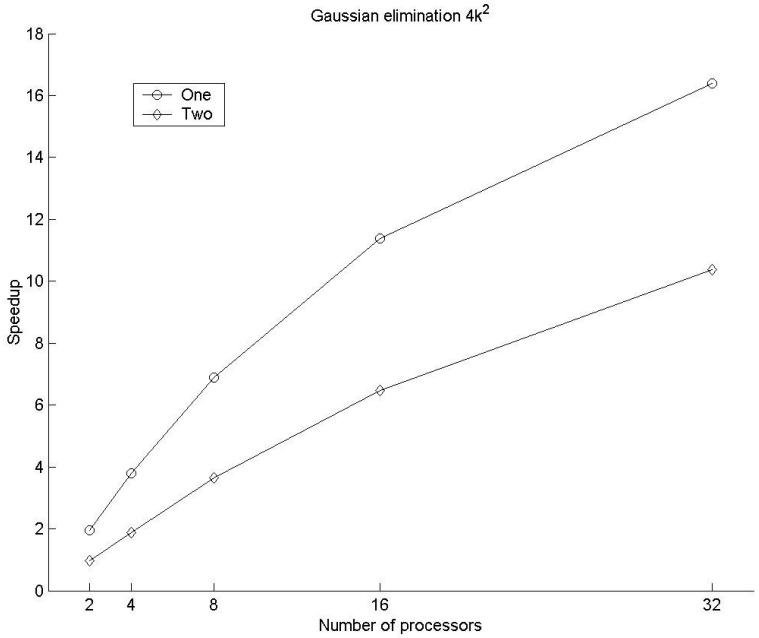

Figure 19: Simplified Gaussian elimination
An average of 2 runs is used.

Simulation

Interested in what speedup can be expected from a runtime system like this, a program that simulates the simplified Gaussian elimination was written. The goal was to get an upper bound on the relative speedup. This simulation makes a few assumptions:

- According to NSC, Monolith can transmit a small message between two nodes in 4,5 microseconds. The bandwidth for large messages is 2 Gbit/s. For each transmission, we calculate the message size and divide it by the bandwidth. If this value is larger than 4,5 microseconds we use the calculated value as transmission time, otherwise we use 4,5 microseconds. This yields a higher performance for the simulated network than the real network. We do not add any time for the runtime system to set up the transmissions.
- The time to modify a value in the matrix was modeled as a constant * the time for one clock cycle. This constant was chosen so that the simulation and the real benchmarks got the same value for one processor. This means that the simulated system does not model any additional overhead for using more processors, and that the overhead present when using one processor can be parallelized, which it can not be in the real world.
- We assume that we have a binomial tree and that there are no waiting periods, that is, we only count the transmission time from the node furthest from the root to the root and back. This is very unlikely to happen in the real system, so again we improve the performance of the simulated one compared to any real system.

The code used for this simulation can be found in Appendix D.

In Figure 20 we can see the speedup for the simulated system. For this simulation, a simplified Gaussian elimination where *A* is a 4000*4000 matrix was simulated. The real system is not that far behind the simulated one. It should be remembered that the simulated system's transmission times is smaller than can be expected of a real system. The simulated runtime system is only considering the overhead that is present when using one processor(and quite unrealistically parallelizes it when more processors are used), while using more processors generate more overhead in real life. The delays where processors waits for each other is ignored. Considering this, the performance of the real system is pretty good.

One thing to note is that while the simulated version almost reaches a speedup of 23 in Figure 20, if we remove the combine phase and only keep the commit phase, we will get a speedup of 26.74. (See Appendix C for the measurements of the simulated version without combine phase.)

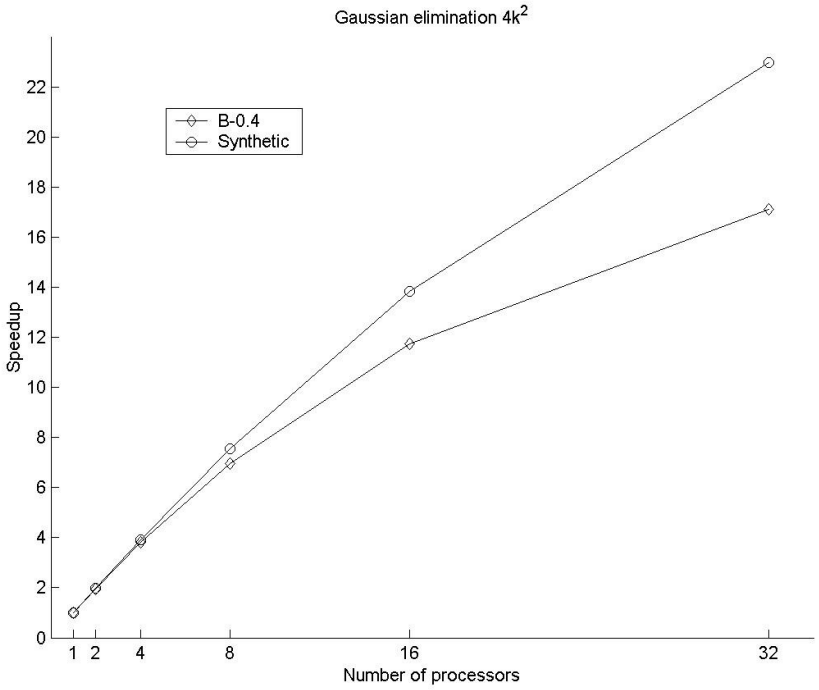

Figure 20: Simulated system

OpenMP

Open Multi Processing, OpenMP, is an open standard for a shared memory model. We will discuss OpenMP a little more in the following section, related work.

Two programs were written using OpenMP, doing essentially the same as simplified Gaussian elimination. The KAI OpenMP library was used, which is a commercial implementation. The code for these programs can be found in Appendix D.

In Figure 21 we can see the execution time for the OpenMP versions

and the simplified NestStep version for for a 2000*2000 matrix. The OpenMP versions perform poorly, which is probably because it is not possible to specify where data is located in OpenMP, and only access local elements. This could probably be solved by simulating a large shared array using non-shared arrays, and performing the shared memory management ourselves in the program. This would however remove the point of using a shared memory system, and we might as well use MPI.

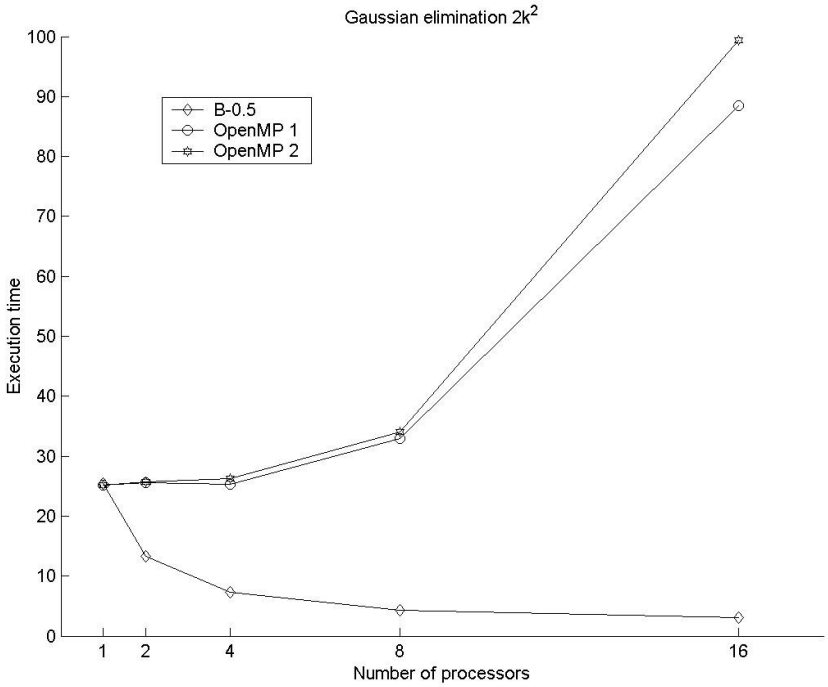

Figure 21: Simplified Gaussian elimination and OpenMP versions. An average of 2 runs is used.

Figure 22 shows the speedup gained from switching from the fastest OpenMP version to the NestStep version for a given amount of processors.(E.g., the speedup for 8 processors is the processing time for the OpenMP version using 8 processors divided by the processing time for the NestStep version using 8 processors.) As can be seen, the speedup is quite significant.

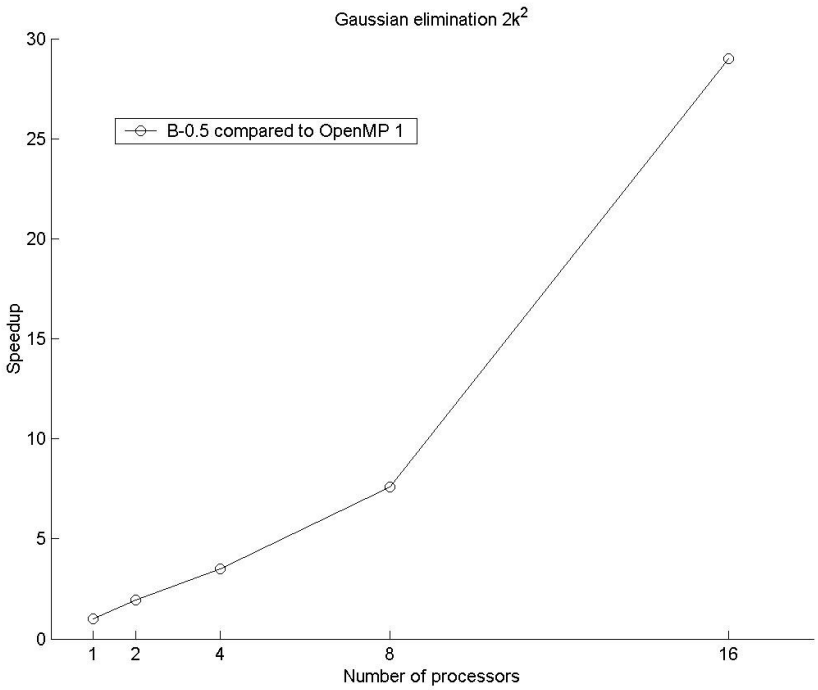

Figure 22: Speedup when moving from OpenMP to NestStep.

9 Related work

9.1 UPC

Unified Parallel C, UPC, is very similar to NestStep. UPC is an extension to C, and gives the programmer access to a shared address space, regardless of whether the hardware has shared or distributed memory.

UPC is not based on the BSP model as NestStep is, and writing/reading to a shared variable or an element in a shared array is possible at any time. However, to be certain that previous writes to an variable/array element has taken effect when reading a value, a barrier must have been encountered after the write.

Shared arrays and variables are distributed in UPC. It is called that shared data has an affinity to certain processors. That is, certain elements in an array that have an affinity to the processor(i.e. are local) are faster to access than elements that have an affinity for another processor. This also applies to shared variables. Shared variables always have an affinity for thread 0.(The leader in NestStep.) UPC allows the programmer to specify the affinity for elements in a shared array in ways similar to NestStep's block and cyclic arrays. UPC does not have anything similar to NestStep's replicated arrays.

We refer the interested reader to [10] for an introduction of UPC. [11] is currently the latest specifications of UPC. [12] provides a more detailed interpretation of the specification and shows how UPC is to be used.

9.2 OpenMP

Open Multi Processing, OpenMP, is an open standard for multi processing and is also similar to NestStep. It supports a shared memory model and is probably the most common specification that commercial vendors use for their solutions for shared memory programming.

OpenMP views data in a way similar to UPC. All shared data may be read/written to at any time, however a barrier or a flush operation must come between a write and a read to ensure that the write has taken effect. There is no way to specify how the data will be stored. Whether the data is distributed across the processors or replicated is up to the implementation.

For more information about OpenMP, please visit www.openmp.org.

10 Conclusions and future work

According to the results gained, a binomial tree or similar structure seems to yield the best performance for the combine/commit phase. However, it would be good to test the system more rigorously with more different applications.

We also found that for some applications may take a performance hit when using NestStep, due to the combine phase. These are problems where there is always one node that has all information needed in the next superstep and needs to broadcast this information. This is because NestStep always requires combine/commit, that is, always performs one reduction and one broadcast. When using other ways to implement these applications, i.e. using MPI, the combine phase can be omitted. However, considering that it is considerably easier to implement parallel algorithms in NestStep, at least those that were developed during this project, NestStep's performance is still acceptable.

NestStep has very good performance compared to OpenMP, which is probably the most popular shared memory system. Although only Gaussian elimination was compared, the speedup when using NestStep instead of OpenMP is so remarkable that it is likely that NestStep will outperform OpenMP for any algorithm that makes somewhat heavy use of shared data.

As there is a possibility that there may be different tree structures that perform better under certain circumstances, developing a model which would determine the best tree at runtime given the parameters of a BSP step would be of great use. That would enable the NestStep runtime system to dynamically determine which tree structure to use for each step, and always use the best one.

It would be very beneficial if a front end for NestStep was developed. That is, to develop a program that translates NestStep code into C code with the calls to the NestStep-C runtime system. Currently it is very cumbersome developing NestStep programs, as all calls to the runtime system have to be manually translated.

11 Acknowledgments

The author would like to thank Christoph Kessler for being both the examiner and supervisor of this project, Thomas Rauber and Gudula Rünger for developing Tlib[4], and finally the National Supercomputer Centre in Linköping, Sweden, for allowing the use of their resources.

12 Appendix A – Performance predictions/measurements

This chapter is largely based on Christoph Kessler's compendium[13] for the programming of parallel computers course at Linköping's university.

12.1 Speedup and efficiency

In order to be able to discuss the performance of parallel computing it is necessary to define some notions. We use speedup as a measurement on how much faster the program runs on *p* processors.

Let T_s be the time it takes to complete the task using the best serial algorithm(on one processor of course). We also let $T(p)$ be the time it takes to finish using a parallel algorithm on *p* processors. We define the *absolute speedup* for *p* processors as

$$S_a(p)=\frac{T_s}{T(p)} \quad (1)$$

and the *relative speedup* for *p* processors as

$$S_r(p)=\frac{T(1)}{T(p)} \quad . \quad (2)$$

Efficiency is a measurement that tells us how well we utilize the processors.
We define the *absolute efficiency* as

$$E_a(p)=\frac{S_a(p)}{p}=\frac{T_s}{pT(p)} \quad (3)$$

and the *relative efficiency* as

$$E_r(p)=S_r\frac{(p)}{p}=\frac{T(1)}{pT(p)} \quad . \quad (4)$$

12.2 Amdahl's law

Lets assume that for a parallel algorithm A , there is a sequential part A^s that cannot be parallelized and a parallel part A^p that can be perfectly parallelized by *p* processors. With a problem size of *n* this gives us the total work of

$$w_A(n)=w_{A^s}(n)+w_{A^p}(n) \quad . \quad (5)$$

The time to complete this work on one processor is

$$T(1)=T_{A^s}+T_{A^p} \quad (6)$$

and on *p* processors it is

$$T(p)=T_{A^s}+\frac{T_{A^p}}{p} \quad . \quad (7)$$

Amdahl's law:

If the sequential part of the algorithm is a *fixed* fraction of the total amount of work, then regardless of the problem size *n*, there is a

constant $\beta = \frac{w_{A^s}}{w_A} \leq 1$, and the relative speedup of A with p processors is limited by $S_r(p) = \frac{p}{\beta p + (1-\beta)} \leq \frac{1}{\beta}$.

Proof:

We have:

$$S_r(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{T_{A^s} + \frac{T_{A^p}}{p}} \qquad (8)\ (2\&7)$$

$$T_{A^s} = \beta T(1) \qquad (9)$$

$$T_{A^p} = (1-\beta)T(1) \qquad (10)$$

Substitution with (9) & (10) into (8) gives

$$S_r(p) = \frac{T(1)}{\beta T(1) + \frac{(1-\beta)T(1)}{p}} = \frac{p}{\beta p + (1-\beta)} \leq \frac{1}{\beta} \quad .$$

In Figure 23 to the left there is an illustration of what causes this limit to occur.

It should be noted that most parallel algorithms do not have a fixed sequential fraction, and that Amdahl's law is quite pessimistic. It does however serve a useful purpose in showing that one cannot expect parallel system to be p times faster.

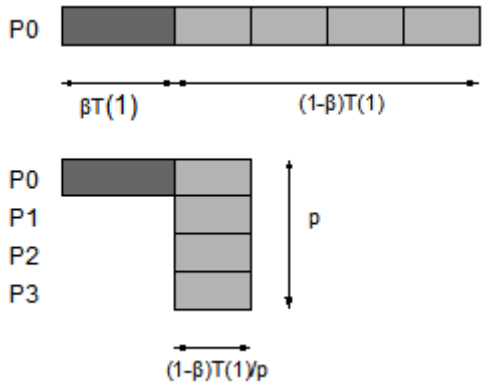

Figure 23: An illustration of Amdahl's law. Recreation of an illustration in [13].

13 Appendix C – Benchmark tables

13.1 Parallel prefix

Average time of 10 runs in seconds – $200 * 10^6$ elements.
Average time for one processor: 4.325 seconds.
One processor per node were used.

# procs	Dary–2	Binomial–0.5	Flat
2	2.213	2.214	2.229
4	1.125	1.111	1.126
8	0.564	0.564	0.564
16	0.302	0.311	0.308
32	0.139	0.138	0.138

13.2 Mergesort

Average time of 20 runs in seconds - $20 * 2^{20}$ elements.
Average time for one processor: 12.182 seconds.
Two processors per node were used.

# procs	Dary-2	Dary-3	Dary–4	Dary-5	Dary–6	Dary-7	Flat
2	7.405	7.371	7.355	7.403	7.399	7.376	7.307
4	4.794	4.806	4.857	4.787	4.731	4.794	4.755
8	3.783	3.742	3.769	3.821	3.785	3.734	3.722
16	3.781	3.724	3.773	3.771	3.760	3.853	3.788
32	5.383	5.265	5.045	5.116	5.153	5.014	5.039

# procs	Binomial-0.3	Binomial-0.35	Binomial–0.4	Binomial-0.45	Binomial–0.5	Binomial–0.6
2	7.395	7.326	7.374	7.411	7.382	7.437
4	4.792	4.758	4.797	4.799	4.777	4.740
8	3.788	3.781	3.788	3.754	3.788	3.769
16	3.944	3.952	3.763	3.894	4.025	3.914
32	5.386	5.424	5.164	5.390	5.228	5.317

Average time of 10 runs in seconds - $40 * 2^{20}$ elements.
Average time for one processor: 24.999 seconds.
Two processors per node were used.

# procs	Dary-2	Dary-4	Binomial-0.4	Binomial-0.5	Flat
2	15.052	14.862	14.798	15.082	14.833
4	9.474	9.509	9.516	9.683	9.647
8	7.219	7.088	7.224	7.345	7.143
16	6.515	6.606	6.724	6.524	6.549
32	8.144	7.817	7.205	7.867	7.493

13.3 Gaussian elimination

Average time of 20 runs* in seconds – 2000 * 2000 matrix.
Average time for one processor: 27.672 seconds.
Two processors per node were used.

# procs	Dary-2	Dary-3	Dary–4	Dary-5	Dary–6	Dary-7	Flat
2	29.533	27.643	27.908	27.728	27.720	27.989	27.830
4	17.646	16.982	16.497	16.997	16.818	16.890	16.514
8	12.615	12.284	12.166	11.802	12.730	12.240	12.690
16	11.021	10.580	10.580	10.689	10.349	10.754	13.384
32	11.715	11.553	10.554	11.810	12.134	11.519	31.839

*For 32 processors and a flat tree only five values were used to calculate the average due to its excessive time requirements.

# procs	Binomial-0.3	Binomial-0.35	Binomial–0.4	Binomial-0.45	Binomial–0.5	Binomial–0.6
2	27.710	30.007	27.954	29.783	29.395	27.704
4	17.787	17.690	17.473	17.794	17.435	17.714
8	11.821	12.026	11.798	11.737	11.656	12.736
16	10.022	9.805	9.649	9.612	9.402	10.641
32	10.071	9.686	9.688	9.035	9.251	10.389

Average time of 2 runs in seconds – 4000*4000 matrix.
Average time for one processor: 204.51 seconds.
Two processors per node were used.

# procs	Dary-2	Dary-4	Binomial-0.4	Binomial-0.5	Flat
2	213.08	211.92	211.60	213.63	218.78
4	121.59	118.85	118.54	119.48	114.84
8	75.52	78.36	73.77	71.81	78.83
16	58.83	58.40	53.15	52.48	77.14
32	56.74	53.40	46.24	43.12	104.47

Additional values for 8, 16 and 32 processors.

# procs	Dary-3	Dary-5	Binomial-0.35	Binomial-0.45
8	71.00	75.26	73.15	70.84
16	58.13	58.45	55.20	52.16
32	56.07	58.79	44.93	44.78

Average time of 2 runs in seconds – 4000*4000 matrix – simplified version. Average time for one processor: 199.22 seconds.
One processors per node were used.

# procs	Dary-2	Dary-4	Binomial-0.4	Binomial-0.5	Flat
2	102.32	101.36	101.95	101.81	100.98
4	52.75	52.08	52.26	52.58	52.56
8	29.19	28.67	28.60	28.96	29.07
16	18.15	16.98	16.96	17.51	20.59
32	13.13	12.22	11.65	12.16	23.85

Average time of 2 runs in seconds – 4000*4000 matrix – simplified version. Average time for one processor: 199.22 seconds.
Two processors per node were used.

# procs	Binomial-0.5
2	204.26
4	105.55
8	54.76
16	30.81
32	19.22

Simulated Gaussian elimination – 4000*4000 matrix.

# procs	Simulated	Simulated without combine phase
1	203.64	203.64
2	102.32	102.07
4	51.91	51.41
8	26.96	21.21
16	14.73	13.73
32	8.87	7.92

Average time of 2 runs in seconds – 2000*2000 matrix – OpenMP and simplified version.
One processor per node were used.

# procs	OpenMP one	OpenMP two	B-0.5
1	25.17	25.16	25.45
2	25.59	25.76	13.24
4	25.37	26.29	7.25
8	32.98	34.03	4.34
16	88.43	99.33	3.05

14 Appendix D – Source code listings

14.1 Parallel prefix

```
void parprefix(sh int</> a[]) {
    int i, j;                   // loop counters
    int pre[a->size()/#];       // for storage of temporary local sums
    int myoffset;               // prefix offset for this processor
    sh int sum;

    step {
        sum = 0;
        j = 0;
        forall(i, a) {
           sum += a[i];
        pre[j++] = sum;
        }
    } combine(sum<+:myoffset>);     /* combine sum and store */
                                    /* prefixes in myoffset */
    step {
        j = 0;
        forall(i, a) {
            a[i] += pre[j++] + myoffset;
    }
}
```

14.2 Mergesort

```
/* assume that local parts are already sorted */
void mergesort(sh int a[]</>)
{
    if(# == 1) // nothing left to merge
        return;

  neststep(2, @ = $ % 2) {
    sh int temp_array[a.size()]</>;

        /* The following statement needs to be executed by all
        processors that have some elements of the original array.
        The last parameters is a boolean flag that tells if the processor
        should import the values, or simply provide its values to
        other processors of another subgroup */

        importArray(a, temp_array, 0, a.size() - 1, 1-@);

        if(@ == 0) {
            merger(temp_array); // sort the imported array
            if(# > 1) // if there are more than one processor in the group
                mergesort(temp_array);
        }

        exportArray(temp_array, a, 0, a->size – 1, 1-@);
  }
}
```

```
void merger(BlockDistArray * a)
{
	int i;
	int l_c;
	int u_c;
	int buffer[a->local_size];

	l_c = a->lower; // counter for the lower section of the local array
	u_c = l_c + a->local_elements / 2; // counter for the upper section

	for(i = 0; i < a->local_size; i++) {
	    if(a[l_c] < a[u_c] && l_c < a->local_size / 2)
	    {
	        buffer[i] = a[l_c];
	        l_c++;
	    }
	    else if(u_c < a->local_size) {
	        buffer[i] = a[u_c];
	        u_c++;
	    }
	    else {
	        buffer[i] = a[l_c];
	        l_c++;
	    }
	}
	a[a->lower:a->lower + a->local_size – 1] = buffer[:];
}
```

14.3 Gaussian elimination

Gaussian elimination with pivoting

```
int gaussian(sh double A[], sh doubleb[], sh double x[], int N) {
    double max = 0;
    int largest = 0;
    int row;
    sh double[N]<%>[N] _A;
    sh double[N]<%>[1] _b;
    sh double b_tmp;
    sh double b_elim;
    sh double[N] tmp;
    sh double[#] sizes;
    sh double[N] elim;

    int i, j, k, l;

    // scatter A and b
    step {
        scatter(A, _A);
        scatter(b, _b);
    }
    // find largest element among own kind
    max = 0;
    step {
        forall2(i, _A) {
            if(fabs(_A[i][0]) > max) {
                max = fabs(_A[i][0]);
                row = i;
            }
        }
        sizes[$] = max;
    }
    for(k = 0; k < N - 1; k++) {
        // move largest array to elim or zero
        step {
            largest = 1;
            for(l = 0; l < #; l++) {
                if(fabs(sizes[l]) > max || (fabs(sizes[l]) == max && l < $)) {
                  largest = 0;
                  break;
                }
            }
            if(largest == 1) {
                elim[:] = _A[row][:];
                b_elim = _b[row];
            }
            // move swapped row to tmp or zero
            if(owned(_A[k][0])) {
                tmp[:] = _A[k][:];
                b_tmp = _b[k];
            }
        }
        step {
            // copy elim to correct row
            if(owned(_A[k][0])) {
                _A[k][:] = elim[:];
```

```
                _b[k] = b_elim;
            }
            // copy tmp to correct row
            if(largest == 1) {
                _A[row][:] = tmp[:];
                _b[row] = b_tmp;
            }
            // eliminate using elim
            forall2(i, _A, k + 1, _A.size() - 1, 1) {
                double modifier = -_A[i][k] / elim[k];
                _b[i] += modifier * b_elim;
                _A[i][k:N - 1] += modifier * elim[k:N – 1];
            }
            // find largest element among own kind
            max = 0;
            forall2(i, _A, k + 1, _A.size() - 1, 1) {
                if(fabs(_A[i][k + 1]) > max) {
                  max = fabs(_A[i][k + 1]);
                  row = i;
                }
            }
            sizes[$] = max;
        }
    }
    // gather
    step {
        gather(_A, A);
        gather(_b, b);
    }
    // calculate x
    for(i = N - 1; i >= 0; i--) {
        double sum = 0;
        for(j = N - 1; j > i; j--)
            sum += x[j] * A[i][j];
        x[j] = (b[i] - sum) / A[i][j];
    }
}
```

Simplified Gaussian elimination

```
int solve(CyclicDistArray * _A, CyclicDistArray * _b, int N) {
    int i, j, k, l;

    sh<?> double b_elim;
    sh double[N] elim;

    // move first row to elim
    step {
        if(owned(_A[0][0])) {
            elim[:] = _A[0][:];
            b_elim = _b[0];
        }

    for(k = 0; k < N - 1; k++) {
        step {
            // eliminate using elim
            forall2(l, _A, k +1, N -1, 1) {
                double modifier = -(_A[l][k] / elim[k]);
                _b[i] += modifier * b_elim;
                _A[l][:]+= modifier * elim[:];
            }
            // move row k+1 to elim
            if(owned(_A[k+1][0])) {
                elim[:] = _A[k+1][:];
                b_elim = _b[k+1];
            }
        } combine(elim[k+1:N-1]);
    }
}
```

OpenMP versions of Gaussian elimination

```
double A[N][N];
double b[N];
double elim[N];
#pragma omp threadprivate(elim)

void gaussian_one(void)
{
    int i, j, k;
    double modifier;

    for(i = 0; i < N - 1; i++)
    {
        #pragma omp parallel for private(k, modifier) schedule(guided)
        for(j = i + 1; j < N; j++)
        {
            modifier = -A[j][i]/A[i][i];
            for(k = i; k < N; k++)
            {
                A[j][k] += modifier * A[i][k];
            }
            b[j] += modifier * b[i];
        }
    }
}
```

```
void gaussian_two(void)
{
    int i, j, k;
    double modifier;

    for(i = 0; i < N - 1; i++)
    {
        #pragma omp parallel private(k, modifier)
        {
            #pragma omp single copyprivate(elim)
            {
                for(j = i; j < N; j++)
                  elim[j] = A[i][j];
            }
            #pragma omp for schedule(guided)
            for(j = i + 1; j < N; j++)
            {
                modifier = -A[j][i]/elim[i];
                for(k = i; k < N; k++)
                {
                  A[j][k] += modifier * elim[k];
                }
                b[j] += modifier * b[i];
            }
        }
    }
}
```

Simulated Gaussian elimination

```
#include <iostream>

int main(int argc, char ** argv) {
	double time = 0.0;
	double N_t = (8*8)/(2048e6); // time per element for large messages
	double min_t = 4.5e-6;	// Transfer time for small messages
	double cycle_t = 1/(2.2e9); // time for one clock cycle
	unsigned int cycles_req;	// used for simulating the time required for
				// one elimination
	unsigned int p;		// # processors
	double times[33];

	for(cycles_req = 1; cycles_req <= 35; cycles_req++) {
	    std::cout << "cycles_req: " << cycles_req << std::endl;
	    for(p = 1; p <= 32; p *= 2) {
		time = 0.0;
		std::cout << "p: " << p;
		for(int N = 4000; N >= 2; N--) {
		    double t_time = (min_t > (N*N_t)) ? min_t : N*N_t;
		    switch(p) {
		      case 1:
				t_time = 0;
				break;
		      case 2:
				t_time *=2;
				break;
		      case 4:
				t_time *= 4;
				break;
		      case 8:
				t_time *= 6;
				break;
		      case 16:
				t_time *= 8;
				break;
		      case 32:
				t_time *= 10;
				break;
		    }
		    time += t_time;
		    time += (N*(N-1)*cycle_t*(double)cycles_req)/(double)p;
		}
		times[p] = time;
		std::cout << " time: " << time << std::endl;
	    }
	    std::cout << "Speedup:";
	    for(int i = 1; i <= 32; i *=2)
		std::cout << " " << times[1]/times[i];
	    std::cout << std::endl;
	    std::cout << "--------------------------" <<std::endl;
	}
	system("pause");
	return 0;
}
```

15 References

(1) Keßler, C. W. 2000. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model. *J. Supercomput.* 17, 3 (Nov. 2000), 245-262.

(2) Keßler, C. W. Managing distributed shared arrays in a bulk-synchronous parallel programming environment. CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE, 16(1):133–153, 2004.

(3) Valiant, L. G. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103-111.

(4) Thomas Rauber and Gudula Rünger. Library support for hierarchical multiprocessor tasks. In Supercomputing ’02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, pages 1–10, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

(5) Bruno R. Preiss. Data structures and Algorithms with Object-Oriented Design Patterns in Java. Wiley, 1999. ISBN 0-471-34613-6.

(6) Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. 1993. LogP: towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, United States, May 19 - 22, 1993). PPOPP '93. ACM Press, New York, NY, 1-12.

(7) Richard M. Karp, Abhijit Sahay, Eunice E. Santos, and Klaus Erik Schauser. Optimal broadcast and summation in the logp model. In SPAA ’93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures, pages 142–153, New York, NY, USA, 1993. ACM Press.

(8) Alexandrov, A., Ionescu, M. F., Schauser, K. E., and Scheiman, C. 1995. LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures* (Santa Barbara, California, United States, June 24 - 26, 1995). SPAA '95. ACM Press, New York, NY, 95-105.

(9) Mark Allen Weiss. Data Structures and Problem Solving Using C++. Addison-Wesley 2000. ISBN 0-201-61250-X.

(10) W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

(11) UPC Consortium. UPC Language Specifications, v1.2. Lawrence Berkeley National Lab Tech Report LBNL-59208, 2005.

(12)T. El-Ghazawi, W. Carlson, T. Sterling and K. Yelick. UPC – Distributed Shared Memory Programming. Wiley, 2005. ISBN 0-471-22048-5.

(13)Keßler, C. W. Programming of Parallel Computers - Compendium OHs edition 2005. Linköpings university.