Programowanie równoległe

Laboratorium 10

Sprawozdanie

Blazhyievskyi Maksym

Kody źródłowe:

Kod źródłowy rozwiązania z wykorzystaniem nowego typu danych MPI:

```
#include <iostream>
#include <mpi.h>
#include <cstring>

#define TESTS 100

struct Data {
    int n;
    double d;
    char s[30];
};

std::ostream& operator<<(std::ostream& os, const Data& dt) {
    os << "{n: " << dt.n << ", d: " << dt.d << ", s: \"" << dt.s
<< "\"}";

    return os;
}

int main(int argc, char *argv[]) {
    int npes;
    int myrank;
    double start, end;

    MPI_Init(&argc, &argv);

    start = MPI_Wtime();
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    MPI_Datatype dtData;
    MPI_Type_contiguous(1, MPI_INT, &dtData);
    MPI_Type_contiguous(1, MPI_DOUBLE, &dtData);
    MPI_Type_contiguous(30, MPI_CHAR, &dtData);
    MPI_Type_commit(&dtData);
```

```
    for (unsigned short i = 0; i < TESTS; ++i) {
      if (myrank == 0) {
           Data x;
           x.n = i;
           x.d = 3.14;
           strcpy(x.s, "sample string");

           MPI_Send(&x, 1, dtData, 1, 13, MPI_COMM_WORLD);
      } else if (myrank == 1) {
           MPI_Status status;
           Data y;

           MPI_Recv(&y, 1, dtData, 0, 13, MPI_COMM_WORLD,
&status);
           std::cout << "process 1 received data " << y << "
from process 0" << std::endl;
      }
    }

    end = MPI_Wtime();
    MPI_Finalize();

    if (myrank == 0) {
      std::cout << "end of process " << end - start <<
std::endl;
      std::cout << "average time per struct sent is " << (end -
start) / TESTS << std::endl;
    }

    return 0;
}
```

Kod źródłowy rozwiązania z wykorzystaniem typu spakowanego MPI:

```
#include <iostream>
#include <mpi.h>
#include <cstring>

#define TESTS 100

int main(int argc, char *argv[]) {
    int npes;
    int myrank;
    double start, end;
```

```
    MPI_Init(&argc, &argv);

    start = MPI_Wtime();
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    for (unsigned short i = 0; i < TESTS; ++i) {
      char buffer[42];
      int position = 0;

      if (myrank == 0) {
           int n = i;
           double d = 3.14;
           char s[30]; strcpy(s, "sample string");

           MPI_Pack(&n, 1, MPI_INT, buffer, 42, &position,
MPI_COMM_WORLD);
           MPI_Pack(&d, 1, MPI_DOUBLE, buffer, 42, &position,
MPI_COMM_WORLD);
           MPI_Pack(s, 30, MPI_CHAR, buffer, 42, &position,
MPI_COMM_WORLD);
           MPI_Send(buffer, position, MPI_PACKED, 1, 13,
MPI_COMM_WORLD);
      } else if (myrank == 1) {
           MPI_Status status;
           int n;
           double d;
           char s[30];

           MPI_Recv(buffer, 42, MPI_PACKED, 0, 13,
MPI_COMM_WORLD, &status);
           MPI_Unpack(buffer, 42, &position, &n, 1, MPI_INT,
MPI_COMM_WORLD);
           MPI_Unpack(buffer, 42, &position, &d, 1, MPI_DOUBLE,
MPI_COMM_WORLD);
           MPI_Unpack(buffer, 42, &position, s, 30, MPI_CHAR,
MPI_COMM_WORLD);

           std::cout << "process 1 received packed data [" << n
<< ", " << d << ", \"" << s << "\"] from process 0" <<
std::endl;
      }
    }
```

```
    end = MPI_Wtime();
    MPI_Finalize();

    if (myrank == 0) {
      std::cout << "end of process " << end - start <<
std::endl;
      std::cout << "average time per struct sent is " << (end -
start) / TESTS << std::endl;
    }

    return 0;
}
```

Kod źródłowy rozwiązania z wykorzystaniem nowego typu danych MPI w trybie buforowanym:

```
#include <iostream>
#include <mpi.h>
#include <cstring>

#define TESTS 100

struct Data {
    int n;
    double d;
    char s[30];
};

std::ostream& operator<<(std::ostream& os, const Data& dt) {
    os << "{n: " << dt.n << ", d: " << dt.d << ", s: \"" << dt.s
<< "\"}";

    return os;
}

int main(int argc, char *argv[]) {
    int npes;
    int myrank;
    double start, end;

    MPI_Init(&argc, &argv);

    start = MPI_Wtime();
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    MPI_Datatype dtData;
    MPI_Type_contiguous(1, MPI_INT, &dtData);
    MPI_Type_contiguous(1, MPI_DOUBLE, &dtData);
    MPI_Type_contiguous(30, MPI_CHAR, &dtData);
    MPI_Type_commit(&dtData);

    char buf[42 * TESTS];
    MPI_Buffer_attach(&buf, 42 * TESTS);

    for (unsigned short i = 0; i < TESTS; ++i) {
      if (myrank == 0) {
           Data x;
           x.n = i;
           x.d = 3.14;
           strcpy(x.s, "sample string");

           MPI_Bsend(&x, 1, dtData, 1, 13, MPI_COMM_WORLD);
      } else if (myrank == 1) {
           MPI_Status status;
           Data y;

           MPI_Recv(&y, 1, dtData, 0, 13, MPI_COMM_WORLD,
&status);
           std::cout << "process 1 received data " << y << "
from process 0" << std::endl;
      }
    }

    end = MPI_Wtime();
    MPI_Finalize();

    if (myrank == 0) {
      std::cout << "end of process " << end - start <<
std::endl;
      std::cout << "average time per struct sent is " << (end -
start) / TESTS << std::endl;
    }

    return 0;
}
```

Kod źródłowy rozwiązania z wykorzystaniem typu spakowanego MPI w trybie buforowanym:

```
#include <iostream>
#include <mpi.h>
#include <cstring>

#define TESTS 100

int main(int argc, char *argv[]) {
    int npes;
    int myrank;
    double start, end;

    MPI_Init(&argc, &argv);

    start = MPI_Wtime();
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    char buf[42 * TESTS];
    MPI_Buffer_attach(&buf, 42 * TESTS);

    for (unsigned short i = 0; i < TESTS; ++i) {
      char buffer[42];
      int position = 0;

      if (myrank == 0) {
           int n = i;
           double d = 3.14;
           char s[30]; strcpy(s, "sample string");

           MPI_Pack(&n, 1, MPI_INT, buffer, 42, &position,
MPI_COMM_WORLD);
           MPI_Pack(&d, 1, MPI_DOUBLE, buffer, 42, &position,
MPI_COMM_WORLD);
           MPI_Pack(s, 30, MPI_CHAR, buffer, 42, &position,
MPI_COMM_WORLD);


           MPI_Bsend(buffer, position, MPI_PACKED, 1, 13,
MPI_COMM_WORLD);
      } else if (myrank == 1) {
           MPI_Status status;
           int n;
           double d;
```

```
            char s[30];

            MPI_Recv(buffer, 42, MPI_PACKED, 0, 13,
MPI_COMM_WORLD, &status);
            MPI_Unpack(buffer, 42, &position, &n, 1, MPI_INT,
MPI_COMM_WORLD);
            MPI_Unpack(buffer, 42, &position, &d, 1, MPI_DOUBLE,
MPI_COMM_WORLD);
            MPI_Unpack(buffer, 42, &position, s, 30, MPI_CHAR,
MPI_COMM_WORLD);

            std::cout << "process 1 received packed data [" << n
<< ", " << d << ", \"" << s << "\"] from process 0" <<
std::endl;
      }
    }

    end = MPI_Wtime();
    MPI_Finalize();

    if (myrank == 0) {
      std::cout << "end of process " << end - start <<
std::endl;
      std::cout << "average time per struct sent is " << (end -
start) / TESTS << std::endl;
    }

    return 0;
}
```

Kod źródłowy rozwiązania z wykorzystaniem nowego typu danych MPI w trybie synchronicznym:

```
#include <iostream>
#include <mpi.h>
#include <cstring>

#define TESTS 100

struct Data {
    int n;
    double d;
    char s[30];
};
```

```
std::ostream& operator<<(std::ostream& os, const Data& dt) {
    os << "{n: " << dt.n << ", d: " << dt.d << ", s: \"" << dt.s
<< "\"}";

    return os;
}

int main(int argc, char *argv[]) {
    int npes;
    int myrank;
    double start, end;

    MPI_Init(&argc, &argv);

    start = MPI_Wtime();
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    MPI_Datatype dtData;
    MPI_Type_contiguous(1, MPI_INT, &dtData);
    MPI_Type_contiguous(1, MPI_DOUBLE, &dtData);
    MPI_Type_contiguous(30, MPI_CHAR, &dtData);
    MPI_Type_commit(&dtData);

    for (unsigned short i = 0; i < TESTS; ++i) {
      if (myrank == 0) {
           Data x;
           x.n = i;
           x.d = 3.14;
           strcpy(x.s, "sample string");

           MPI_Ssend(&x, 1, dtData, 1, 13, MPI_COMM_WORLD);
      } else if (myrank == 1) {
           MPI_Status status;
           Data y;

           MPI_Recv(&y, 1, dtData, 0, 13, MPI_COMM_WORLD,
&status);
           std::cout << "process 1 received data " << y << "
from process 0" << std::endl;
      }
    }

    end = MPI_Wtime();
    MPI_Finalize();
```

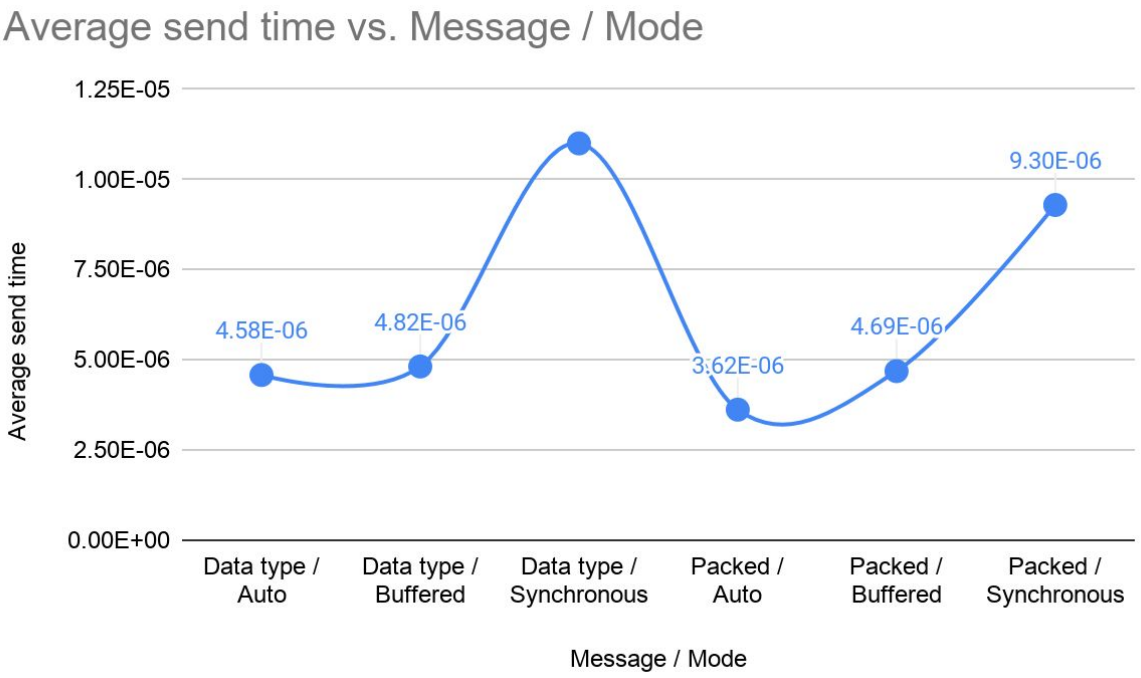
```
    if (myrank == 0) {
      std::cout << "end of process " << end - start <<
std::endl;
      std::cout << "average time per struct sent is " << (end -
start) / TESTS << std::endl;
    }

    return 0;
}
```

Kod źródłowy rozwiązania z wykorzystaniem typu spakowanego MPI w trybie synchronicznym:

```
#include <iostream>
#include <mpi.h>
#include <cstring>

#define TESTS 100

int main(int argc, char *argv[]) {
    int npes;
    int myrank;
    double start, end;

    MPI_Init(&argc, &argv);

    start = MPI_Wtime();
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    for (unsigned short i = 0; i < TESTS; ++i) {
      char buffer[42];
      int position = 0;

      if (myrank == 0) {
           int n = i;
           double d = 3.14;
           char s[30]; strcpy(s, "sample string");

           MPI_Pack(&n, 1, MPI_INT, buffer, 42, &position,
MPI_COMM_WORLD);
           MPI_Pack(&d, 1, MPI_DOUBLE, buffer, 42, &position,
MPI_COMM_WORLD);
           MPI_Pack(s, 30, MPI_CHAR, buffer, 42, &position,
```

```
MPI_COMM_WORLD);
            MPI_Ssend(buffer, position, MPI_PACKED, 1, 13,
MPI_COMM_WORLD);
      } else if (myrank == 1) {
            MPI_Status status;
            int n;
            double d;
            char s[30];

            MPI_Recv(buffer, 42, MPI_PACKED, 0, 13,
MPI_COMM_WORLD, &status);
            MPI_Unpack(buffer, 42, &position, &n, 1, MPI_INT,
MPI_COMM_WORLD);
            MPI_Unpack(buffer, 42, &position, &d, 1, MPI_DOUBLE,
MPI_COMM_WORLD);
            MPI_Unpack(buffer, 42, &position, s, 30, MPI_CHAR,
MPI_COMM_WORLD);

            std::cout << "process 1 received packed data [" << n
<< ", " << d << ", \"" << s << "\"] from process 0" <<
std::endl;
      }
    }

    end = MPI_Wtime();
    MPI_Finalize();

    if (myrank == 0) {
      std::cout << "end of process " << end - start <<
std::endl;
      std::cout << "average time per struct sent is " << (end -
start) / TESTS << std::endl;
    }

    return 0;
}
```

Wykres czasu średniego przesyłania danych:



Wnioski:

- Tryb synchroniczny najbardziej spowalnia przesyłanie komunikatów.
- Przesyłanie danych spakowanych jest nieco szybsze od przesyłania struktur.
- Szybkość przesyłania danych w trybie domyślnym i w trybie buforowanym wygląda podobnie, więc najprawdopodobniej tryb buforowany na danym sprzęcie jest wykorzystywany domyślnie.